baSnake

A simple Snake-like videogame for the ZX Spectrum Next, coded in BASIC by Marco Varesio.

Document history

Version	Date	Notes
1.0.0	2019-07-20	First document release.

Introduction

baSnake is a simple Snake/Nibbles-like videogame, written in BASIC language, for the ZX Spectrum Next computer.

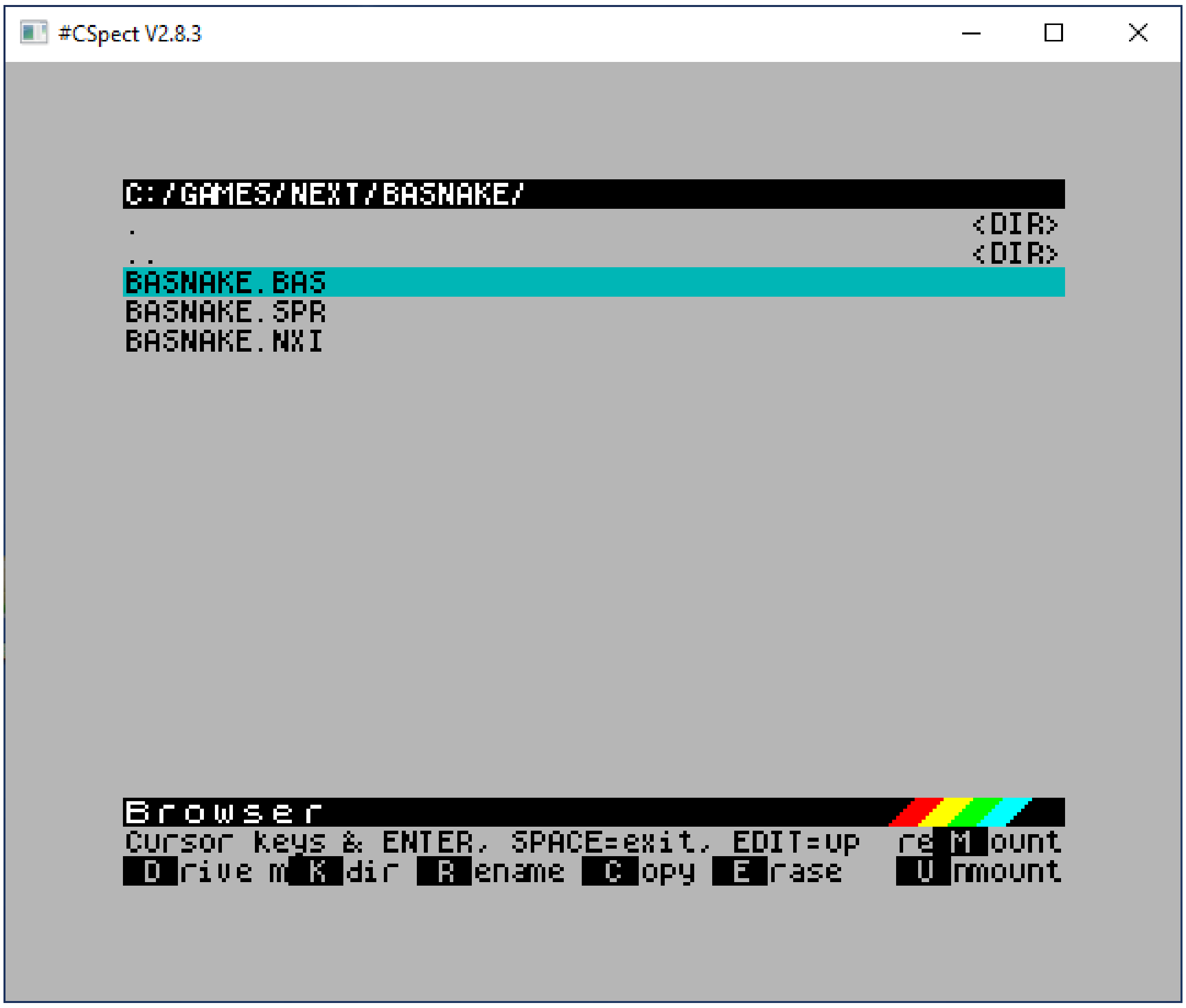
I originally coded *baSnake* for the classic ZX Spectrum computer, during the ZX Spectrum BASIC Jam, hosted on itch.io in June 2017. Some months later, I decided make it more enjoyable by taking advantage of the ZX Spectrum Next turbo modes and so released a second version. Finally, I made a third release exclusively for the ZX Spectrum Next, using some of its peculiar features, such as Layer 2, Sprites and Turbo Sound Next.

This document is aimed at explaining *baSnake* source code and is intended for people willing to learn how to program games in BASIC on the ZX Spectrum Next; however, it does not claim to be a BASIC / ZX Spectrum Next / Video games development reference.
The program suffers from the fact that it was quickly coded during my coffee breaks, without a real software engineering process and its features were added progressively, as soon as I found the time to implement them, so do not take it as a good programming example. In fact, by writing this document, I realized that the code could be heavily refactored in order to improve its readability, for example by reducing the number of `GO TO` statements and making the flow more linear.
You are encouraged to tinker with the program and improve it!

You can also take a look at my other creations by browsing my retrogaming projects page on itch.io, my retro web site/blog and my blog in Italian language or by visiting my YouTube channel.

Playing baSnake

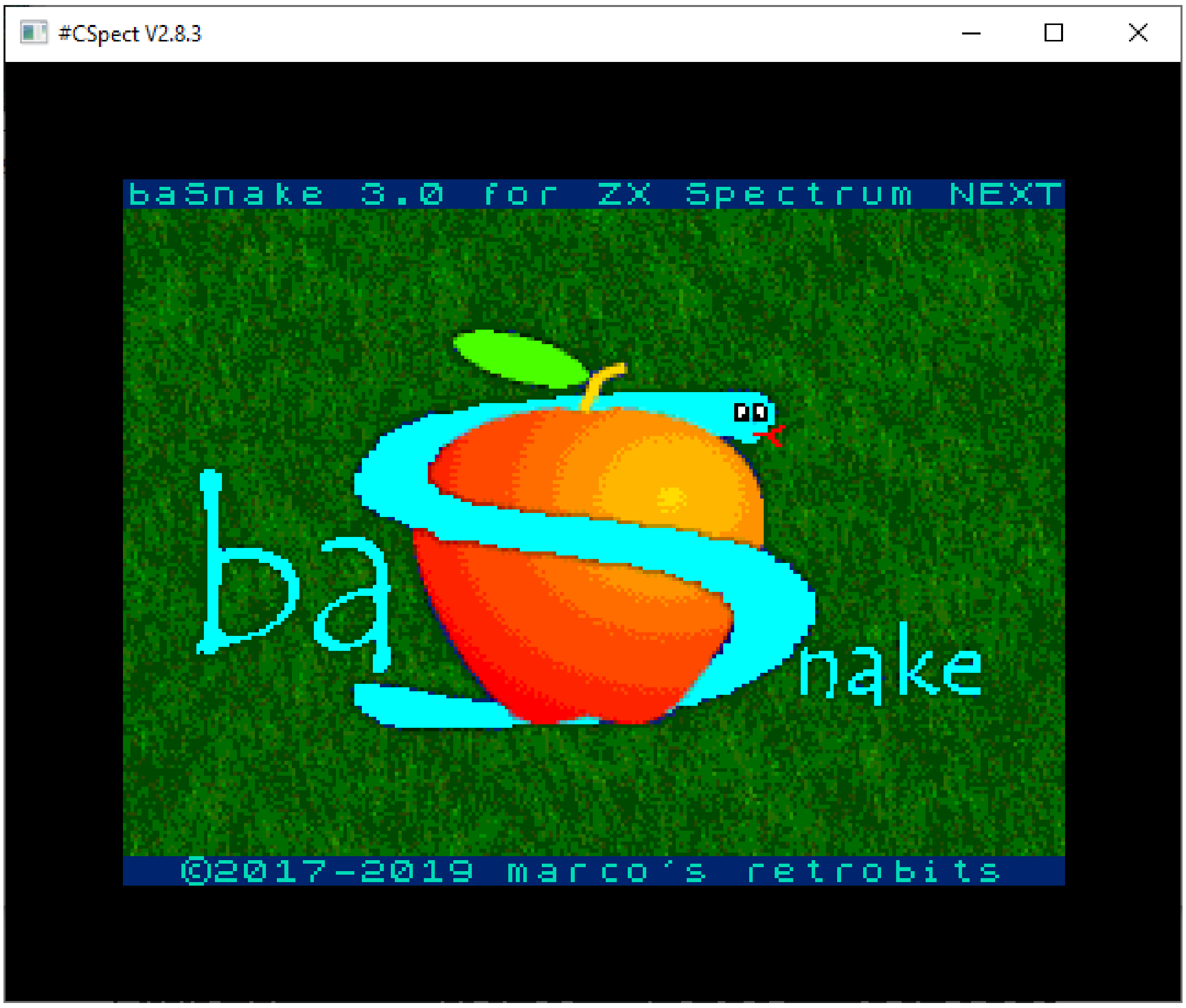
The best way to understand how a game works is to play it, so, before diving into code, let's have a game!
Assuming *baSnake* is already in the SD card in your Spectrum Next, simply locate `BASNAKE.BAS` file in NextZXOS Browser, as depicted in the following screenshot:


```
#CSpect V2.8.3

C:/GAMES/NEXT/BASNAKE/
.                                             <DIR>
..                                            <DIR>
BASNAKE.BAS
BASNAKE.SPR
BASNAKE.NXI

Browser
Cursor keys & ENTER, SPACE=exit, EDIT=up   reMount
 D rive mK dir  R ename  C opy  E rase     U nmount
```

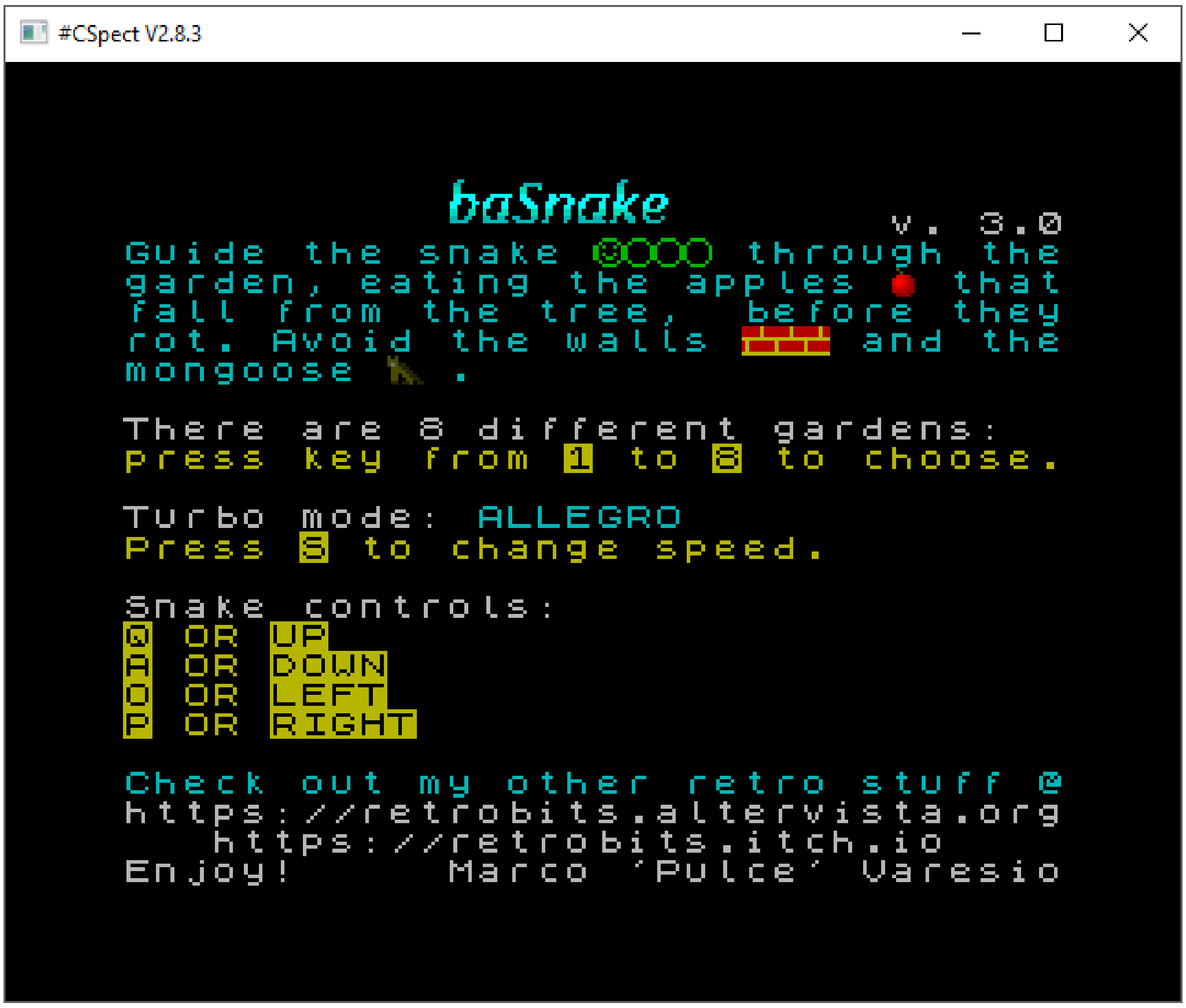
and then and tap `ENTER` key to load and run it.

> The screenshots included in this document have not been taken on a real ZX Spectrum Next, but on a PC running the excellent #CSpect emulator by Mike Dailly. Phoebus R Dokos kindly maintains the updated ZX Spectrum Next SD card distribution images for CSpect and real machines.
> The other available emulator currently supporting the ZX Spectrum Next is ZEsarUX by César Hernández Bañó, so, if you don't (yet) have a real ZX Spectrum Next, you can get started with the aforementioned emulators.

On successful load, you will be welcomed with a short string quartet musical intro, adapted from Pachelbel's Canon in D, and with the game logo screen:


#CSpect V2.8.3
baSnake 3.0 for ZX Spectrum NEXT
baSnake
©2017-2019 marco's retrobits

After some seconds, when the music stops, you will be allowed to enter the main menu screen by pressing any key:

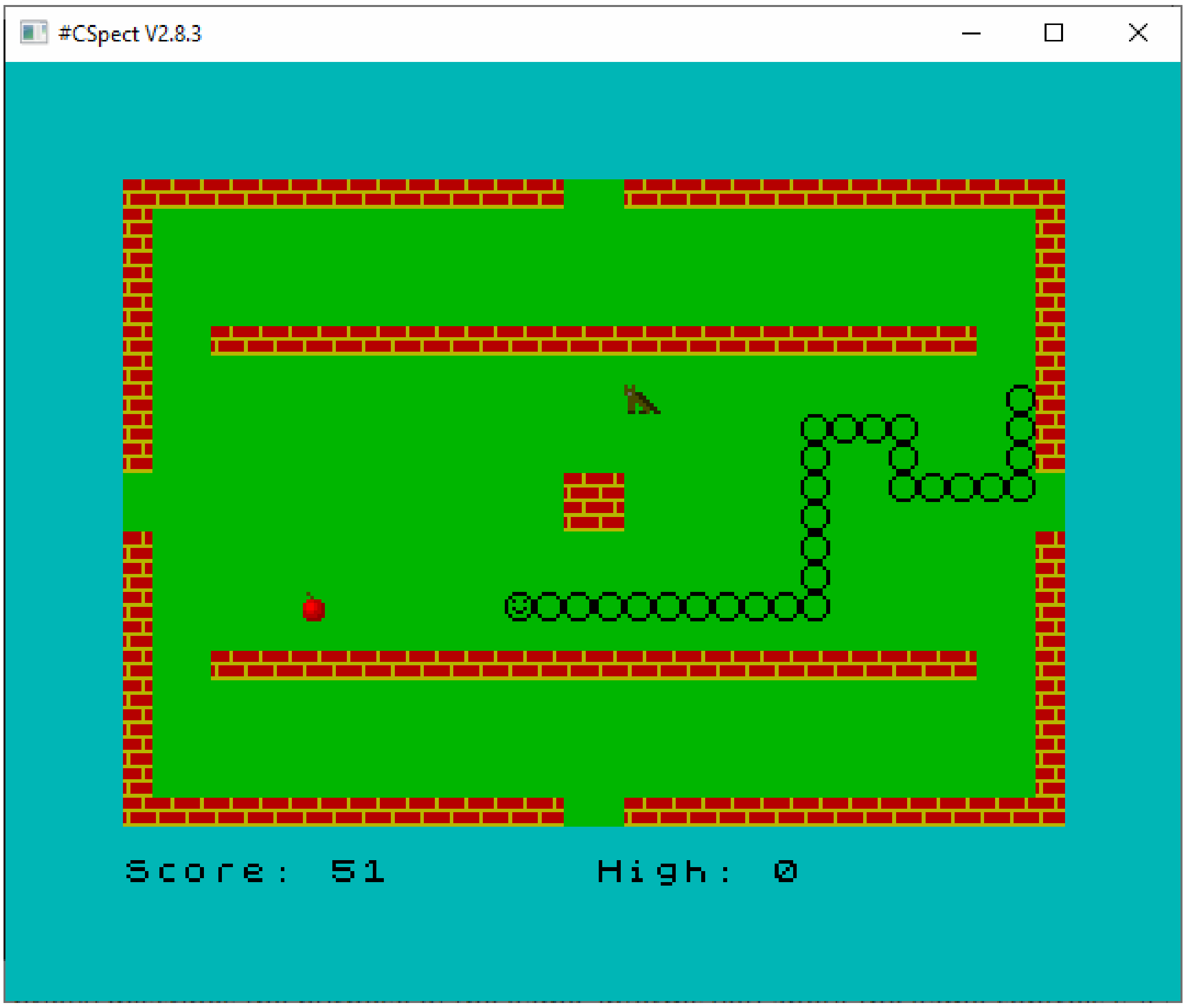


```
baSnake
                                  v. 3.0
Guide the snake through the
garden, eating the apples that
fall from the tree, before they
rot. Avoid the walls and the
mongoose .

There are 8 different gardens:
press key from 1 to 8 to choose.

Turbo mode: ALLEGRO
Press S to change speed.

Snake controls:
Q OR UP
A OR DOWN
O OR LEFT
P OR RIGHT

Check out my other retro stuff @
https://retrobits.altervista.org
   https://retrobits.itch.io
Enjoy!      Marco 'Pulce' Varesio
```

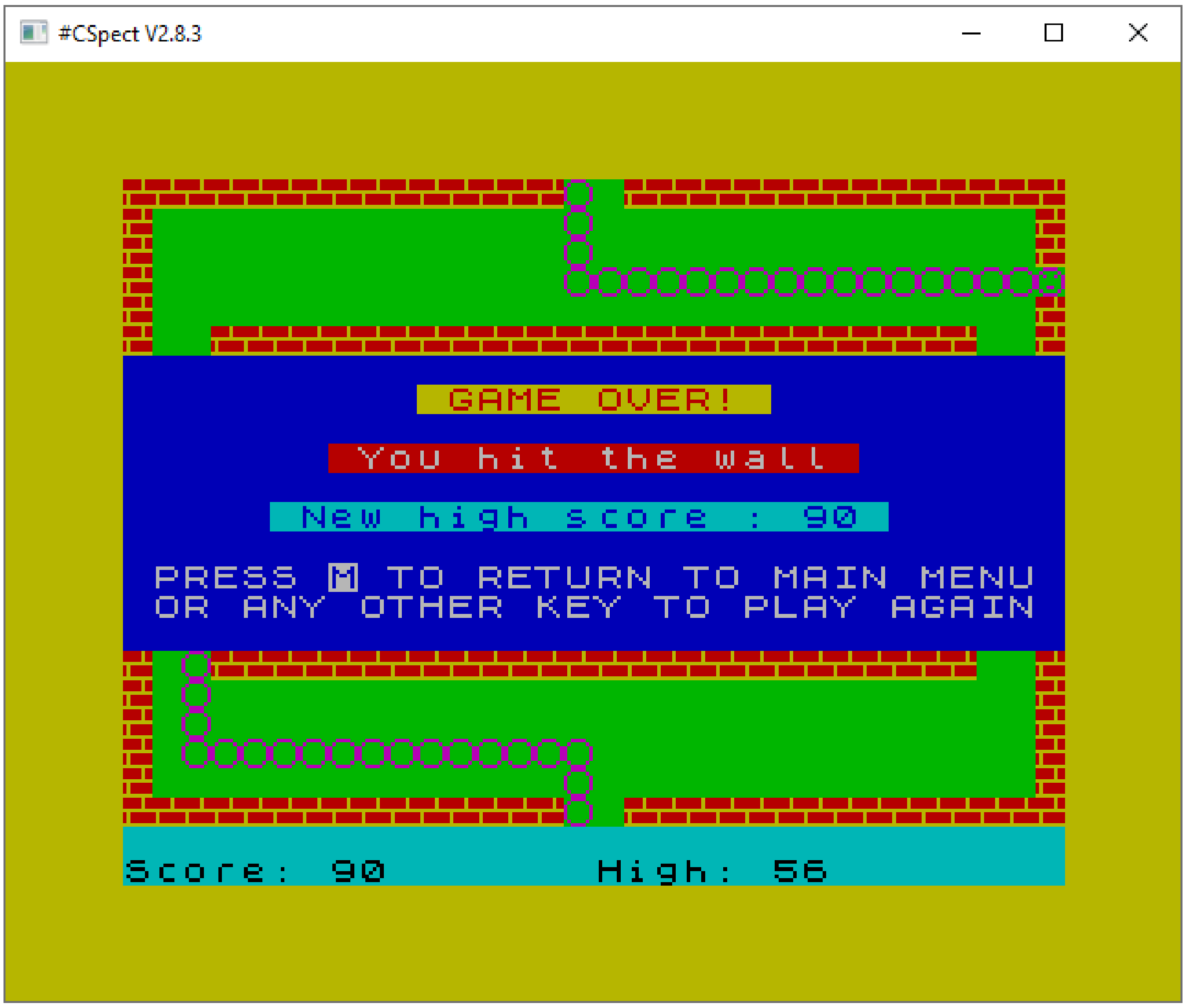
This screen briefly describes the purpose of the game, informs you about the game controls (`Q`, `A`, `O`, `P` keys or cursor keys), allows you to choose game speed through `S` key (there are 5 modes, from slowest to fastest, called: "Andante", "Moderato", "Allegro", "Vivace", "Presto") and finally allows you to start the game, by choosing one of the eight available scenarios, by pressing a numeric key between `1` and `8`.

The game's objective is to guide the snake trough the garden and gain score by eating the apples that randomly appear on the screen, while avoiding collisions with the walls, the mongoose and the snake itself. Each time an apple is consumed, the snake length increases, and so the game difficulty.

The screenshot below depicts garden n. 8: I am heading the snake to a succulent red apple, while a threatening mongoose appeared. Current score and high score are reported at the bottom of the screen.



When the snake hits itself, one of the walls or the mongoose, the game ends and you can choose to go back to the main menu screen and change some settings, by pressing the `M` key, or to play again with current settings, by pressing any other key:



```
GAME OVER!
You hit the wall
New high score : 90
PRESS M TO RETURN TO MAIN MENU
OR ANY OTHER KEY TO PLAY AGAIN
Score: 90          High: 56
```

You can access the program source code by pressing the `BREAK` key; then you can execute again by typing `RUN`, followed by the `ENTER` key.

Program description

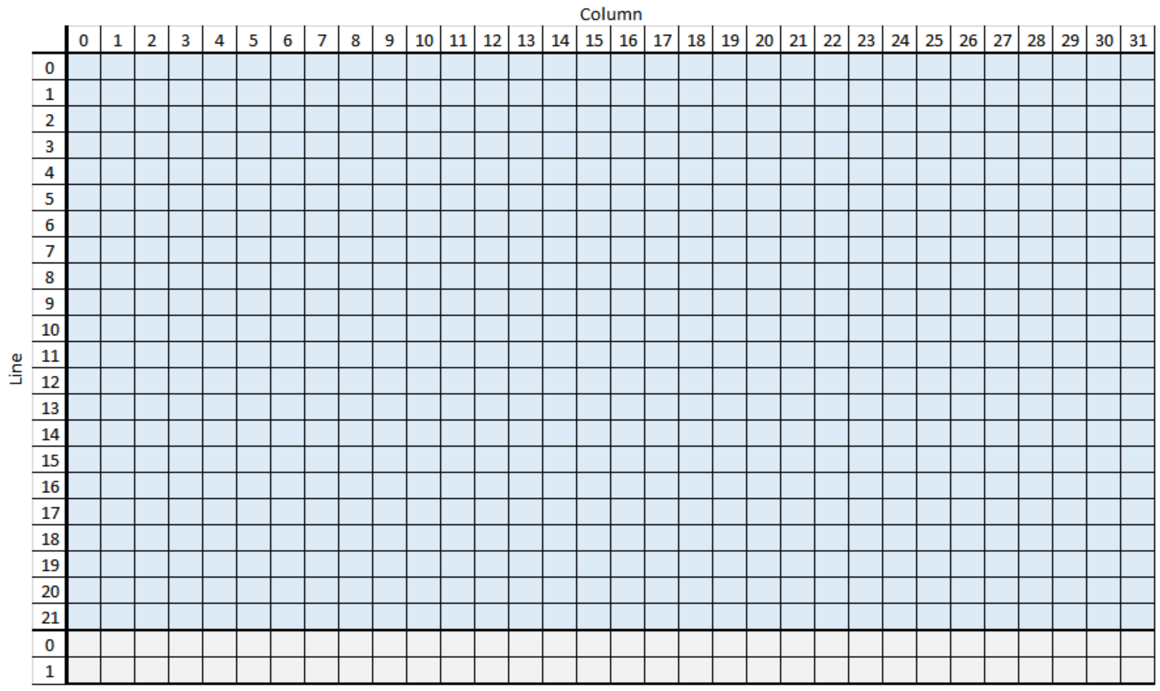
The *baSnake* program is made up of a main program and some support subroutines, called by the main program. The most important part of *baSnake* program is the game loop, i.e. a sequence of instructions that is repeatedly executed; we can imagine each iteration of the loop as a snapshot of the game at a given instant, or tick.

baSnake game screen

In order to understand how the program works, it's useful to see how the ZX Spectrum Next screen is accessed by *baSnake* to draw the game elements such as the snake, the walls, the apples and the mongoose.

The ZX Spectrum Next provides various graphic modes (see NextBASIC new commands and features); among these, *baSnake* mostly uses *Layer 0*, which is the standard Spectrum mode: 256x192 pixels, which correspond to 32x24 characters, each character position being 8x8 pixels. The actual game screen (the green area, including the red walls, in the previous picture) is made up of 22 lines (numbered from 0 to 21 from top to bottom) and 32 columns (numbered from 0 to 31 from left to right), for a total of 704 character positions. *baSnake* uses the `PRINT AT line, column` statement to draw the walls and the snake. At each game tick, the snake is moved by one character position.
Two more lines (numbered 0 and 1) are available at the bottom of the screen, below the game area; you can access these lines by printing to *stream* #1, associated to the *keyboard channel*. *baSnake* uses the second of these lines for printing current score and high score information. For

more information about channels and streams, please read Channels & Streams on the comp.sys.sinclair FAQ.



The *sprite system*, another feature of the ZX Spectrum Next, provides a 320×256 pixels wide surface on top of Layer 0, for drawing colorful *sprites*. *baSnake* uses the sprite system to draw the apples and the mongoose.

The new *Layer 2* mode, which provides a 256-color screen at the full 256x192 resolution, in which every pixel is individually colored, is used for the game logo screen.

Variables and data structures

The snake queue

One of the most important variables in *baSnake* is the bidimensional array `S`, declared as `DIM S(704, 2)`, which stores the positions (line and column numbers) of each character element constituting the snake's body. The size of `S` is 704; we have already seen this number before: in fact, it is the maximum number of characters that can be printed in the game area. The other dimension is 2, since for each character we need two numbers, one for the line number and the other for the column number.
Two numeric variables, `H` and `T`, are used as indices to the `S` array and respectively refer to the position of the snake's *head* (first element) and *tail* (last element). Numbers from `T+1` to `H-1` are the indices of all other snake elements. As a matter of fact, the snake is modeled with a *queue*, implemented as a *circular buffer* through `S`, `H` and `T`.
At each game tick, the snake movement is implemented by "adding" a new head, whose position is calculated based on current head position and direction, in `S` at index `H+1`, and "removing" the old tail (unless the snake is growing) by incrementing `T`; the positions of all other snake body elements do not change.

For example, let's assume that, at instant I, the snake is horizontally laid, moving from left to right; current head position, at index 41, is: (line 10, column 20) and current tail position, at index 38, is: (line 10, column 17). At instant I+1, the new head position will be (line 10, column 21): since the snake is moving horizontally, the line number does not change, and since the movement is from left to right, the column number is incremented by 1. The new head position will be saved in

`S` at index `H` = 42. The tail index `T` will be incremented to 39, reflecting the fact that the last element before the tail becomes the tail itself:

Instant I

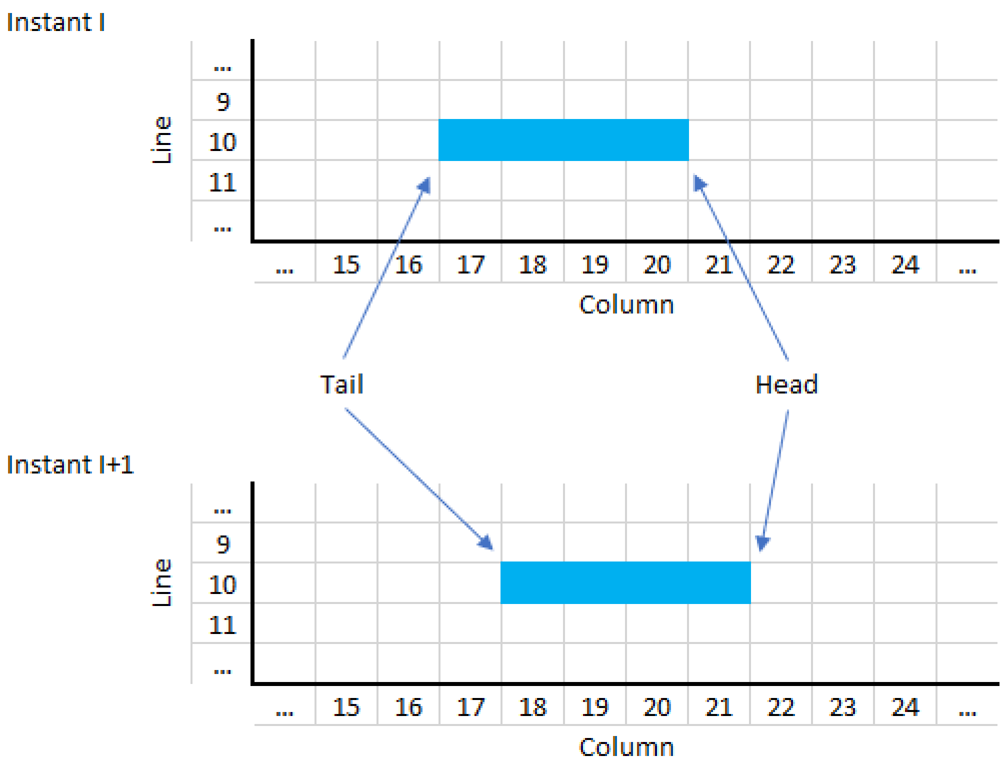
S array

	index	line	column
	...	...	...
	35	XX	YY
	36	XX	YY
	37	XX	YY
T=38	38	10	17
	39	10	18
	40	10	19
H=41	41	10	20
	42	XX	YY
	43	XX	YY
	44	XX	YY
	45	XX	YY
	...	...	...

Instant I+1

S array

	index	line	column
	...	...	...
	35	XX	YY
	36	XX	YY
	37	XX	YY
	38	10	17
T=39	39	10	18
	40	10	19
	41	10	20
H=42	42	10	21
	43	XX	YY
	44	XX	YY
	45	XX	YY
	...	...	...



The game area map

Another bidimensional array we have to deal with is `M`, declared as `DIM M(22, 32)`. As you can guess by looking at the sizes of `M`, it models a map of the game area (22 lines x 32 columns). Each cell of the matrix `M` can assume one of the following values:

- 0: the cell is empty,
- 1: the cell is occupied by the snake,
- 2: the cell is occupied by a wall,
- 3: the cell is occupied the mongoose.

`M` is used for detecting collisions among the snake and itself, the walls or the mongoose.

Variables quick reference

The following table summarizes the most important variables in *baSnake*:

Variable name	Purpose
`S`	Queue containing line and column numbers of each snake element
`H`	Index of head element in snake queue `S`
`X`	Used for calculating the next head column number, based on current head column number and direction
`Y`	Used for calculating the next head line number, based on current head line number and direction
`T`	Index of tail element in snake queue `S`
`TY`	Used for storing the tail line number
`TX`	Used for storing the tail column number
`M`	Game area map
`MV`	Used to access `M` values
`SC`	Current score
`HI`	High score
`FX`	Apple position column number, when the apple is visible; -1 otherwise
`FY`	Apple position line number, when the apple is visible; -1 otherwise
`FD`	Apple lifecyle countdown: is initialized to a random value when the apple is drawn and decremented at each game tick; when it reaches 0, the apple is deleted
`MX`	Mongoose position column number, when the mongoose is visible; -1 otherwise
`MY`	Mongoose positoin line number, when the mongoose is visible; -1 otherwise
`MR`	Random number, used to determine whether the mongoose should be visible
`D`	Snake direction: 1: up; 2: down; 3: left; 4: right
`DT`	Temporary variable used for calculating snake direction `D`
`K`	Code of the currently pressed key; snake direction `D` is set according to currently pressed key code `K`
`G`	Whether the snake is growing: when the snake eats some food, `G` is set to a positive number and decremented at each game tick until it becomes 0. While `G` is greater than 0, the snake body length grows
`F`	Points to the FRAMES system variable; is used for synchronization and to determine game ticks duration
`TRB`	Selected game speed; together with `F`, is used to determine game ticks duration

Commented baSnake code

BASIC programs are made up of numbered lines; each line can contain one or more statements, separated by the `:` character. This section explains in detail *baSnake* code line by line.

Program and author info

Let's start easy with 5 `REM` lines. `REM` stands for "remark" and can be followed by any text; this text will be ignored by the computer, but can be used to improve program readability by humans. You can, for example, use `REM` to provide author information, to specify what a variable means, what a routine does or whatever you like:

```
1 REM ***********************
2 REM *     baSnake 3.0.0    *
3 REM *   ZX Spectrum Next   *
4 REM * Marco Varesio 2019   *
5 REM ***********************
```

Initializations

Line 7 contains 2 commands: as stated before, in BASIC a single program line can contain multiple commands and `:` is used as separator.
The first statement, `LET TRB = 7`, is a variable assignment: it sets `TRB` variable, which represents the game speed, to value 7, which is medium speed.
The second statement, `RUN AT 2`, is another new feature of the ZX Spectrum Next and changes the computer speed to 14 MHz:

```
7 LET TRB = 7 : RUN AT 2
```

The `GO SUB` instruction at line 8 makes the computer jump to the sprites loading and initialization routine, which starts at line 5000, and executes it. When the routine execution completes (i.e. when the `RETURN` statement at line 5050 is reached), the computer returns immediately after the `GO SUB` statement and executes the next instruction. In this case, the next instruction is another `GO SUB`, in particular to the welcome routine at line 2800, which displays the game logo screen, plays the musical intro and waits for a key press to continue.

```
8 GO SUB 5000: GO SUB 2800
```

When the program returns from the welcome routine, execution continues with line 9; here, once more, a `GO SUB` instructions changes the flow of the program, this time to the user defined graphics initialization routine at line 900:

```
9 GO SUB 900
```

Main menu

This section describes the code portion that is executed at startup and every time the player chooses to return to main menu, by pressing the `M` key when the game is over.

```
10 REM *** MAIN MENU ***
```

Variable assignment at line 11 initializes the high score to 0. A nice improvement to the game could be saving high scores to SD and loading them at startup, instead of initializing to 0, so that you could keep track of your achievements across different sessions.

```
11 LET HI = 0
```

Line 12 jumps to main menu management routine, which renders the main menu screen and handles user choices, at line 1500:

```
12 GO SUB 1500
```

Game setup

The main menu management routine ends when the user selects the desired scenario, or garden; the following code completes game initialization and renders the selected garden.

`RANDOMIZE` instruction at line 15 initializes the sequence of pseudorandom numbers that will be used in the game main loop for randomly placing the apples and the mongoose in the game area:

```
15 RANDOMIZE
```

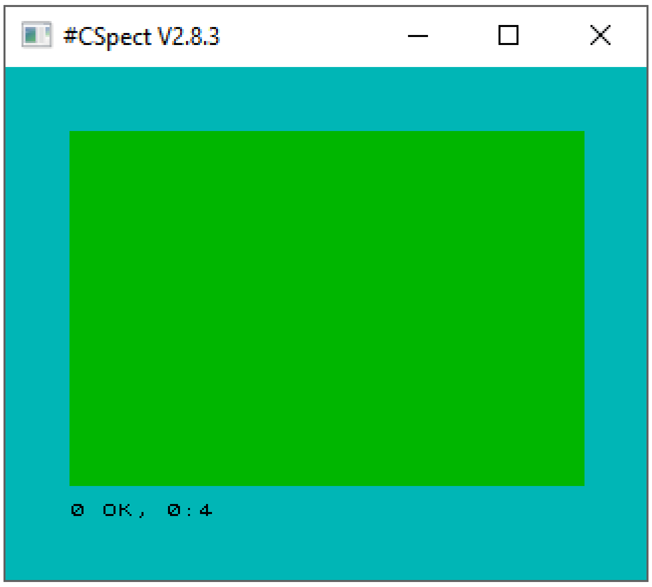
Line 20 defines the game area matrix `M` and the snake queue `S`.
The dimensions of `M` are the lines count (22) and the columns count (32).
The dimensions of `S` are the maximum number of snake elements (704) and 2, because for each snake element we need to know the coordinates (line and column number) in a two-dimensional space:

```
20 DIM M(22, 32): DIM S(704, 2)
```

Line 22 tells the computer to use the colors set with low brightness (`BRIGHT 0`), sets the border (i.e. the frame around the game screen) to cyan color (`BORDER 5`) and sets the paper (i.e. the background of the screen) to green color (`PAPER 4`). Then, the `CLS` command, which clears the screen to the specified paper color, is invoked.

```
22 BRIGHT 0: BORDER 5: PAPER 4: CLS
```

This wikipedia image shows the available colors in Layer 0 mode, with both low (on the left) and high (on the right) brightness levels.
The effects of the statements at line 22 on the ZX Spectrum Next screen are shown in the following picture:



```
0 OK, 0:4
```

The `GO SUB` statement at line 25 jumps to the routine which draws selected garden on screen:

```
25 GO SUB 1600
```

Line 30 resets current player score:

```
30 LET SC = 0
```

Lines 35 and 40 initialize the snake queue: the snake will be initially made up of only one element, so, at line 35, the head index `H` and the tail index `T` are set to the same element in array `S`; in particular, the first:

```
35 LET T = 1: LET H = 1
```

Line 40 sets the position of this element to line = 11, column = 8:

```
40 LET S(1, 1) = 11: LET S(1, 2) = 8
```

Line 60 saves the snake element position to variables X and Y; these variables will be used a bit later for updating the new snake head position at each game tick:

```
60 LET Y = S(1, 1): LET X = S(1, 2)
```

Line 70 sets the snake direction `D` to 4 (which means from left to right) and the growing amount `G` to 4 (which means that for the first 4 game ticks, the snake will grow).
So, if the player does not change the direction, the snake will move from left to right and in the first 4 game ticks, only the head position will be updated; besides, the snake is growing, the tail position will remain the same. Starting from the next game tick, the tail index will be incremented.

```
70 LET D = 4: LET G = 4
```

Line 80 sets the apple column (`FX`) and line (`FY`) numbers to -1. -1 is neither a valid line nor a valid column number and it means no apple has fallen into the game area. The apple lifecycle countdown `FD` is defined and set to 0:

```
80 LET FX = -1: LET FY = -1: LET FD = 0
```

Similarly, when the game starts, the mongoose is hidden in its den and not displayed in the game area, so we set its column (`MX`) and line (`MY`) numbers to -1. The variable used for determining whether the mongoose should appear is defined and initialized to 0:

```
81 LET MX = -1: LET MY = -1: LET MR = 0
```

Line 90 prints current score (which is 0) and high score to the last line of the screen. The two lines at the bottom of the screen are accessed by printing to stream `#1`.
Simply stated, you can use `PRINT AT line, column` (where 0 <= line <= 21 and 0 <= column <= 31) to move the print position to one of the first 22 lines and `PRINT #1; AT line, column` (where 0 <= line <= 1 and 0 <= column <= 31) to move the print position to one of the 2 lines at the bottom of the screen:

```
90 PRINT #1;  AT 1, 0; "Score: "; SC, "High: "; HI
```

Main game loop

The essence of *baSnake* program is the main game loop, from line 100 to line 600.
It is a loop because it keeps doing the same sequence of actions at each game tick over and over, until the games ends. Basically, these actions are:

- update the snake position based on current direction;
- manage the appearance of the apple and the mongoose;
- check collision between the snake and the apple (good), or with the walls, the mongoose or the snake itself (bad);
- update snake direction based on the key pressed;
- wait for the necessary amount of time until the next game tick.

```
100 REM *** MAIN GAME LOOP ***
```

The first half of line 110 sets `F` to the least significant byte of *FRAMES* system variable. This looks rather complicated, so let me try to explain.
The computers uses a portion of its memory for some specific purposes, and one of these is the frames counter. It is a 3 bytes long value, which is automatically incremented by the computer at a fixed frequency (between 50 and 60 Hz, based on configuration). For our purpose, we consider only the least significant byte of this counter, which is located at byte 23672. It is useful to save this value at the beginning of the game loop for later usage; in fact, at the end of the loop, it will be used to determine the time elapsed since the loop started and so the time we still have to wait for the next game tick to occur. By doing so, we make the game run at a constant defined speed.
The `PEEK` function, which returns the value in memory at the provided address, is used for accessing the value of the *FRAMES* counter.

In the second half of line 110, the `IF...THEN...` control statement instructs the computer to execute the apple management routine at line 1000 only if `G` equals 0; instead, if `G` is different from 0, the program flow will continue normally to the next line. This means that no apple will fall into the garden while the snake is growing:

```
110 LET F = PEEK 23672: IF G = 0  THEN GO SUB 1000
```

The next program lines calculate the new snake head position based on current head position (which is stored in `S` array at index `H` and was previously copied to variables `X` and `Y`, at line 60) and snake movement direction (represented by variable `D`). If the snake is moving vertically, only the head line number `Y` is updated; conversely, if the snake is moving horizontally, only the column number `X` will be updated.
So, for example, if the snake head is currently at line 10 (`Y` = 10), column 5 (`X` = 5) and the snake is going up (`D` = 1), the head column number will not change, since the snake is moving horizontally, and the head line number will be decremented by 1, since the screen lines numbering is ascending from top to bottom. This situation is managed at line 250.
After `X` or `Y` value is updated, the program will jump (GO TO statement) to line 270:

```
250 IF D = 1  THEN LET Y = Y - 1 : GO TO 270
255 IF D = 2  THEN LET Y = Y + 1 : GO TO 270
260 IF D = 3  THEN LET X = X - 1 : GO TO 270
265 IF D = 4  THEN LET X = X + 1 : GO TO 270
```

The following lines make the game area behave as if its opposite edges (top and bottom; left and right) were connected, like in a planisphere.
So, for example, if the snake is going from right to left (`D` = 3) and current head column number `X` is 0, if there is no wall obstructing its way, the new head position will be at the right end of the screen (`X` = 21).
After `X` or `Y` value is updated, the program will jump to line 290:

```
270 IF Y < 0  THEN LET Y = 21 : GO TO 290
275 IF Y > 21  THEN LET Y = 0 : GO TO 290
280 IF X < 0  THEN LET X = 31 : GO TO 290
285 IF X > 31  THEN LET X = 0 : GO TO 290
```

Then, the computer will execute the mongoose management routine, at line 2500:

```
290 GO SUB 2500
```

In line 300, we check whether the snake has reached the apple. If either the snake column number `X` is different from the apple column number `FX` or the snake line number `Y` is different from the apple line number `FY`, the snake has not reached the apple and the computer skips the next instructions, by jumping to line 400:

```
300 IF X <>FX  OR Y <>FY  THEN GO TO 400
```

Instead, if the snake has reached the apple, it will eat it and for the next 3 iterations of the game loop, the snake body will grow. This is modeled by incrementing the `G` variable by 3 in the first statement at line 310.

The SPRITE statement, in the same line, resets all properties of sprite number 0, which corresponds to the apple picture. In particular, its visible flag is set to 0 and this makes the apple disappear from the screen, since the apple has been eaten by the snake:

```
310 LET G = G + 3: SPRITE 0, 0, 0, 0, 0: LET FX = -1: REM HIDE APPLE
```

Line 320 rewards the player for having fed the snake, by increasing the score. The score is increased based on the apple lifecycle: the player will gain 5 points for eating a red apple, 10 for eating a golden apple and only 1 point for an apple which is about to rot.
In this line, we make use of the `ELSE` clause in the `IF` command. It was not available in the original ZX Spectrum BASIC and is one of the many improvements provided by NextBASIC:

```
 320 IF FD > 16 THEN LET SC = SC + 5: ELSE IF FD > 8 THEN LET SC = SC + 10: ELSE
LET SC = SC + 1
```

Line 330 notifies the player about the new score, by printing it to the bottom of the screen:

```
 330 PRINT #1;  AT 1, 0; "Score: "; SC, "High: "; HI
```

The program continues at line 400, which is executed regardless of whether the snake is growing or not.
In particular, we prepare for printing the new snake head, whose position was previously calculated, by setting the foreground (`INK`) color to black (color code 0):

```
 400 INK 0
 405 REM *** Draw new snake head ***
```

Before updating the snake head to the new position, we have to reflect the fact that the position previously occupied by the snake head is now occupied by an element of the snake body, since the snake is moving.
So we print the snake body graphic character (whose code is 144) to the position previously occupied by the snake head, overwriting it:

```
 410 PRINT AT S(H, 1), S(H, 2);  CHR$ 144
```

Then, before drawing the head at the new position, we save it into the snake queue `S`.
To do so, we increment the head index `H` by 1 and will use it to store the new head position in the snake queue `S`. Since `S` is managed as a circular buffer, if we exceed its capacity by going beyond the last position, we'll start writing again at the first position. The cleverly chosen size of `S` ensures that by doing so, no useful data is overwritten:

```
 420 LET H = H + 1: IF H = 705  THEN LET H = 1
```

We can now save the new head line and column numbers in `S` at the new index `H`:

```
 430 LET S(H, 1) = Y: LET S(H, 2) = X
```

We can finally print the snake head character, whose code is 145, at the new head position:

```
 440 PRINT AT S(H, 1), S(H, 2);  CHR$ 145
```

Now that we have managed the snake's head, by updating its position and printing it to the new position, it's time to take care of the snake's tail. If the snake is growing, its tail position will not change; instead, position currently occupied by the snake's tail will become empty and the last snake body character before the tail will become the tail itself.
So, the next line checks if the snake is growing; if so, it jumps to line 490 and skips tail deletion:

```
450 IF G > 0  THEN GO TO 490
```

If the snake is not growing, we delete the character currently drawn to the tail position, by overwriting it with a blank space `" "`:

```
452 REM *** Delete snake tail ***
455 LET TY = S(T, 1): LET TX = S(T, 2)
460 PRINT AT TY, TX; " "
```

Then, we mark, in the game map `M`, the the position previously occupied by the snake tail as free, by setting the corresponding value to 0.
The use of `TY+1` and `TX+1`, instead of `TY` and `TX`, as indices to access M might be confusing; it is done this way because screen lines and columns number starts from 0, while indexing of arrays starts from 1:

```
465 LET M(TY + 1, TX + 1) = 0
```

We reflect he fact that the penultimate character of the snake body becomes the tail by incrementing the tail index `T`, with the usual circular buffer management:

```
470 LET T = T + 1: IF T = 705  THEN LET T = 1
```

Finally, we jump to collision detection code at line 500, where we'll check for collisions between the snake and the walls or the mongoose:

```
480 GO TO 500
```

Line 490 is the line we jumped to, from line 450, because the snake was growing after eating an apple (`G` > 0). Here we decrement `G` (when the value of `G` will be 0, the snake will stop growing):

```
490 LET G = G - 1
```

Based on current value of `G`, we play a different note, by means of the BEEP statement:

```
495 IF G = 2  THEN BEEP .04, -10
496 IF G = 1  THEN BEEP .04, -20
497 IF G = 0  THEN BEEP .04, -5
```

Collision detection

The following code determines whether the snake collides with itself, the mongoose, or one of the walls. This is done by inspecting the value in game map `M` at the position corresponding to the updated snake head position (still identified by `X` and `Y`). If this position is empty, the game can continue and the snake head can actually advance to that position; otherwise, if the position is already occupied, there is a collision and the game is over.
First, we set `MV` variable to the value of game map `M` corresponding to the new head position (remember that array indexing starts from 1, so we have to add 1 to `X` and `Y`):

```
500 LET MV = M(Y+1, X+1)
```

Then, we check `MV` value and in case of collision, we set the `M$` string variable to a message describing the collision and then make the program jump to the game over code at line 700:

```
501 IF MV = 1  THEN LET M$ = " You bit yourself ": GO TO 700
502 IF MV = 2  THEN LET M$ = " You hit the wall ": GO TO 700
503 IF MV = 3  THEN LET M$ = " The mongoose bit you ": GO TO 700
```

If we arrived here, it means that the collision did not occur, because the position in the game map `M` corresponding to the snake's head position is free.
So, we mark this position as occupied by the snake, by setting it to 1 (with usual array index displacement by 1):

```
510 LET M(Y+1, X+1) = 1
```

So far, we have updated the snake head and tail positions, managed snake eating growing and checked against collision, which did not occur (otherwise we would not be here, but inside the game over code).
In the last phase of the game loop, the program checks for player input, updates snake direction based on key pressed and waits for the necessary amount of time until the next game tick. This is done by the input routine at line 4000.

```
590 GO SUB 4000
```

Now we are at the end of the game loop, and the next instruction is a jump to the start of the loop, for the next iteration:

```
600 GO TO 100
```

Game over

If we've come this far, it means that a collision occurred and so the game is over.

```
700 REM *** GAME OVER ***
```

First, we hide from the screen both the apple (number 0) and the mongoose (number 1) sprites, by setting all their properties (ad in particular the visible flag) to 0. This is done with a FOR loop, iterating 2 times over the sprite number `I`, which assumes values 0 and 1:

```
701 FOR I=0 TO 1: SPRITE I, 0, 0, 0, 0: NEXT I: REM HIDE APPLE AND MONGOOSE
```

We need to hide sprites because in *baSnake* sprites are on top of the Layer 0 image and we want to prevent them from overlapping the game over message "window" that we will draw later.

Snake color change effect

Then, with the help of the new `REPEAT ... REPEAT UNTIL` looping structure, introduced in NextBASIC, we implement a simple special effect: starting with the snake head and ending with the snake tail, we progressively change the color of each snake character element from black to magenta, by overwriting it. Moreover, we replace the smiling snake head character with the disappointed one.

First, the foreground color is set to magenta (color code 3):

```
705 INK 3
```

Then, the loop iterator `I` is set to the snake head index `H`, because we will start from the head character:

```
710 LET I=H
```

Now the loop on each snake character actually starts:

```
715 REPEAT
```

We set `C` to the code of the character to print, that is 144 for the snake body and 147 for the snake disappointed head:

```
717  LET C=144: IF I = H THEN LET C=147
```

The following line is a trick for overwriting the snake head only once, with the disappointed head character, and not overwriting it again with the snake body character, in case the snake bit itself. The `IF` condition makes the computer print the character with code `C` either at the first iteration, when we are printing the head (`I = H`), or at subsequent iterations, when we are printing the other snake body parts whose positions are different from the snake head position (`S(I, 1) <> S(H, 1) OR S(I, 2) <> S(H, 2)`):

```
720  IF I = H OR S(I, 1) <> S(H, 1) OR S(I, 2) <> S(H, 2) THEN PRINT AT S(I,
1), S(I, 2); CHR$ C
```

The following line makes the loop terminate when `I = T`, i.e. when we have replaced all black snake characters until the last (which is the tail, identified by `T`), with the magenta ones.
If `I` is different from `T`, the loop execution continues with the next line:

```
725  WHILE I <> T
```

In line 730, loop iterator `I` is decremented by one in order to index the next snake character position in `S`. Line 735 implements the usual circular buffer behavior:

```
730  LET I=I-1
735  IF I = 0 THEN LET I = 704
```

The `PAUSE` statement at line 737 adds a small delay at each iteration. Without the pause, the loop would execute too fast and we would see the snake color immediately change from black to magenta.
You could change the pause value and see by yourself what happens with different values:

```
737  PAUSE 1
```

The next line is the end of the loop. Technically speaking, `REPEAT UNTIL 0` denotes an endless loop, but here we do not run the risk of infinitely looping because of the loop end condition specified by the `WHILE` statement at line 725.

```
740 REPEAT UNTIL 0
```

Game over message window

After the magenta snake effect, we display a blue "window" showing some game over message. The window background is displayed by printing 10 lines, from number 6 to 15, of blank characters on a blue (color code: 1) background, or `PAPER`:

```
810 PAPER 1
811 FOR I = 6  TO 15
812 PRINT AT I, 0; "                                "
813 NEXT I
```

The game over reason message, i.e. the description of the collision previously stored in variable `M$`, is printed at screen line 9, with white (color code: 7) `INK` on red (color code: 2) `PAPER`.
The screen `BORDER` is set to yellow (color code: 6).
The message will be horizontally centered by calculating its column number from the screen width (32) and message length (see `LEN` function), using the following formula: `(32 -  LEN (M$)) / 2`:

```
814 BORDER 6: PAPER 2: INK 7
815 PRINT AT 9, (32 -  LEN (M$)) / 2; M$
```

Line 820 sets the `INK` color to yellow and activates the `FLASH` effect, which is used in next program line for printing the flashing "GAME OVER" window title at screen line 7:

```
820 INK 6: FLASH 1
821 PRINT AT 7, 10; " GAME OVER! "
```

Then, `FLASH` effect is turned off for subsequent prints and colors are set to blue `INK` on cyan `PAPER`:

```
825 FLASH 0: PAPER 5: INK 1
```

High score update

Line 830 checks whether high score is unbeaten; if so, execution jumps to line 850; otherwise, program flow continues with next line:

```
830 IF SC <=HI  THEN GO TO 850
```

High score has been beaten, so current score becomes the new high score, and the player is informed with a message, which is printed to screen line 11 and centered horizontally:

```
840 LET HI = SC
841 LET M$ = " New high score : "+  STR$ (HI) + " "
845 PRINT AT 11,  INT ((32- LEN (M$))/2); M$
```

Afterwards, colors are set to white `INK` on blue `PAPER`:

```
850 PAPER 1: INK 7
```

The following code is a small loop in which the computer quickly plays a sequence of notes with decreasing pitch:

```
870 FOR I = 15  TO -30  STEP -2
871 BEEP .05, I
872 NEXT I
```

Then, the program informs the player that he can either return to main menu, by pressing the `M` key, or play again with current settings, by pressing any other key.

```
875 PRINT AT 13, 1; "PRESS "; : INVERSE 1 : PRINT "M"; : INVERSE 0 : PRINT " TO RETURN TO MAIN MENU"
876 PRINT AT 14, 1; "OR ANY OTHER KEY TO PLAY AGAIN"
```

The following code implements what is stated in the previous message by means of a loop in which the program repeatedly checks whether a key is pressed and whether the potentially pressed key is the `M` key. The `INKEY$` function is used in line 880 to read the keyboard status and determine the pressed key; if exactly one key is pressed, the corresponding character is stored in `K$`; otherwise, `K$` will contain an empty string. In the latter case (line 885), the program will jump back to line 880 to read the keyboard status. The program flow will continue to line 890 only when a single key will be pressed by the player.
Line 890 checks whether the key pressed is the `M` key; if so, the program jumps to the main manu at line 10; otherwise (line 895), it jumps to the game setup code at line 15 to set up a new game with current settings:

```
880 LET K$ =  INKEY$
885 IF K$ = "" THEN GO TO 880
890 IF K$ = "M" OR K$ = "m" THEN GO TO 10
895 GO TO 15
```

User Defined Graphics initialization

In computers, the letters, digits, punctuation marks, the white space and so on are called characters and each character is identified by a numeric code. In the ZX Spectrum family of computers, for example, code 97 identifies character "a" and code 49 identifies character "1". Moreover, there are some special characters (from code 144 onwards) whose grapheme can be defined by the user. These are called user-defined graphics or *UDG*s and we will use them for defining the appearance of:

- snake body elements (code 144),
- snake head (code 145),
- wall bricks (code 146),
- sad/disappointed snake head (code 147), used in the game over routine.

Since characters are 8x8 pixels wide, the shape of each UDG is defined through a sequence of 8 bytes, in which each byte represents a line and each bit in a line defines whether for the corresponding dot, the computer should show the `PAPER` (0) or `INK` (1) color. These patterns must me loaded into ZX Spectrum memory, at byte 65368 onwards; so, data for UDG with code 144 will be stored in memory from byte 65368 to byte 65375, data for UDG with code 145 will be stored in memory from byte 65376 to byte 65383, etc.

So, for example, let's consider the first two UDGs defined in *baSnake*, which are:

- the character used for printing all snake body elements, except its head, and

- the character used for printing the snake head

The following pictures represent the characters shapes and the corresponding pattern values:

Snake body - code 144								Binary value	Decimal value
		■	■	■	■			00111100	60
	■					■		01000010	66
■							■	10000001	129
■							■	10000001	129
■							■	10000001	129
■							■	10000001	129
	■					■		01000010	66
		■	■	■	■			00111100	60

Snake head - code 145								Binary value	Decimal value
		■	■	■	■			00111100	60
	■					■		01000010	66
■		■			■		■	10100101	165
■							■	10000001	129
■		■			■		■	10100101	165
■			■	■			■	10011001	153
	■					■		01000010	66
		■	■	■	■			00111100	60

Now, let's take a look at the code. The UDGs loading routine starts at line 900.
The RESTORE statement, located at line 901, defines line 903 as the starting point for UDGs pattern DATA bytes, which will be fetched byte by byte at each READ execution. Line 903 provides snake's body elements data, line 904 provides snake's head data, line 905 provides walls bricks and line 906 provides sad snake's head data.
Line 902 uses a FOR loop for reading all these patterns byte by byte and storing them into memory, by means of the POKE statement. We have 4 UDGs, for a total of 32 pattern data bytes, so 32 iterations (from 0 to 31) are required:

```
900 REM *** UDGs ***
901 RESTORE 903
902 FOR I = 0  TO 31: READ L: POKE 65368+I, L: NEXT I
903 DATA 60, 66, 129, 129, 129, 129, 66, 60: REM SNAKE BODY 144
904 DATA 60, 66, 165, 129, 165, 153, 66, 60: REM SNAKE HEAD 145
905 DATA 4, 4, 4, 255, 64, 64, 64, 255: REM WALL 146
906 DATA 60, 66, 165, 129, 129, 153, 66, 60: REM SNAKE SAD 147
910 RETURN
```

Apple management routine

The apple management routine is quite simple: if there is no apple in the game area, it calculates a random position and, if the position is not occupied by any other game item (snake, mongoose, walls), it draws the apple at that position and sets the apple lifecycle counter to a random positive value.
Otherwise, if the apple is already on screen, its lifecycle counter is decremented and based on its value, the apple sprite image is updated. When the lifecycle counter reaches 0, the apple is deleted.

```
1000 REM *** APPLE ROUTINE ***
```

First, we check whether the apple is already in the game area, by looking at its column number `FX`. The -1 value (which is an invalid column number) denotes the fact that the apple is currently hidden; otherwise, the value is valid and it means that the apple is currently shown, and the program jumps to line 1100:

```
1001 REM APPLE NOT YET FALLEN
1005 IF FX <>-1  THEN GO TO 1100
```

Apple not yet fallen

If we are still here, the apple is currently hidden, so we calculate the random column (`FX`) and line (`FY`) numbers where we will display the apple and the (adjusted) random apple lifecycle duration `FD`:

```
1010 LET FX =  INT ( RND *32): LET FY =  INT ( RND *22): LET FD = 40+ INT ( RND
*30)
```

Then we inspect the game map `M` to check if position at line `FY`, column `FX` is already occupied; if so, we set again `FX` to -1 and return to the game loop without drawing the apple. We will be luckier on next game iteration, maybe:

```
1015 IF M(FY + 1, FX + 1) <>0  THEN LET FX = -1 : LET FD = 0: RETURN
```

Please note that we increment indices `FY` and `FX` by 1 before accessing `M` matrix, because array indexing starts from 1, while line and column numbering starts from 0.

Instead, if the position is actually free, we draw the red apple at the position corresponding to line `FY`, column `FX` by assigning image 0 (red apple) to sprite 0 (apple sprite) by means of the `SPRITE` command.
The `SPRITE` command requires 5 parameters:

1. The sprite number (0 is the apple sprite)
2. The sprite x coordinate
3. The sprite y coordinate
4. The sprite image (0 is the red apple)
5. The sprite flags (1 means that the sprite must be visible).

Since the character positions surface is 32x22 characters, each character is 8x8 pixels and the display surface is 320×256 pixels, overlapping the Layer 0 / ULA screen by 32 pixels, we can calculate the sprite x and y coordinates by multiplying the column and line numbers by 8 and displacing them by 32 pixels:

```
1030 SPRITE 0, FX*8+32, FY*8+32, 0, 1: REM RED APPLE
```

Finally, we return to the game loop:

```
1040 RETURN
```

Apple already fallen

If the apple is already in the game area, the apple lifecycle counter `FD` is decremented:

```
1100 REM APPLE ALREADY FALLEN
1105 LET FD = FD - 1
```

and then change the apple image to golden (sprite image 1) or rotting (sprite image 2) based on the apple age (please note that the only difference in the parameters to the `SPRITE` command is the fourth, i.e. the sprite image):

```
1110 IF FD = 8  THEN SPRITE 0, FX*8+32, FY*8+32, 2, 1 : RETURN : REM ROTTEN
APPLE
1111 IF FD = 16  THEN SPRITE 0, FX*8+32, FY*8+32, 1, 1 : RETURN : REM YELLOW
APPLE
```

If the apple is too old (`FD` = 0), we make it disappear, by resetting its properties. In particular, the sprite is made invisible by setting the sprite flags to 0. We also set the apple column number `FX` to -1:

```
1115 IF FD = 0  THEN SPRITE 0, 0, 0, 0, 0: LET FX = -1: REM HIDE APPLE
```

Finally, we return to the game loop:

```
1130 RETURN
```

Main menu management routine

This routine displays the main menu screen and allows the player to change the game speed and to select a game scenario and consequently start a new game.
The code for checking keypresses, printing characters to screen and displaying sprites is better covered in other sections, so don't worry if you cannot find detailed explanations here.

First, the screen is painted all black:

```
1500 BORDER 0: PAPER 0: CLS
```

Then, a `FOR` loop iterates over the 4 sprites making the game title and displays them:

```
1505 FOR I=0 TO 3: SPRITE I+2,120+16*I,32,I+4,1: NEXT I
```

Next, the title screen is rendered, with a mixture of multicolored text and sprite images:

```
1507 PRINT "Guide the snake "; : INK 4: PRINT CHR$ 145;  CHR$ 144;  CHR$ 144;
 CHR$ 144; : INK 5: PRINT " through the"
1508 PRINT "garden, eating the apples   that": SPRITE 0,240,56,0,1
1509 PRINT "fall from the tree,  before they"
1510 PRINT "rot. Avoid the walls "; : PAPER 2: INK 6: PRINT CHR$ 146;  CHR$ 146;
 CHR$ 146; : PAPER 0: INK 5: PRINT " and the"
1511 PRINT "mongoose   .": SPRITE 1, 104, 80, 3, 1
1512 PRINT
1513 INK 7:PRINT "There are 8 different gardens:"
1514 INK 6: PRINT "press key from "; : INVERSE 1 : PRINT "1";
1515 INVERSE 0: PRINT " to "; : INVERSE 1: PRINT "8";: INVERSE 0
1516 PRINT " to choose."
1517 PRINT
```

Then the routine for printing currently selected game speed is executed:

```
1518 GO SUB 3050
```

And the screen rendering continues:

```
1519 INK 6: PRINT "Press "; : INVERSE 1: PRINT "S"; : INVERSE 0 : PRINT " to
change speed."
1520 INK 7: PRINT : PRINT "Snake controls:": INK 6: INVERSE 1
1521 PRINT "Q"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "UP"
1522 PRINT "A"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "DOWN"
1523 PRINT "O"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "LEFT"
1524 PRINT "P"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "RIGHT"
1525 INVERSE 0: PRINT
1527 INK 5: PRINT "Check out my other retro stuff @": INK 7
1528 PRINT "https://retrobits.altervista.org"
1529 PRINT #1;  AT 0, 0; "   https://retrobits.itch.io    "
1530 PRINT #1; "Enjoy!     Marco 'Pulce' Varesio"
```

Then, there is a loop, starting at line 1550, in which we check for the currently pressed key. If the pressed key is a numeric key between 1 and 8, the program jumps out of the loop at line 1570; if the pressed key is S, the program cycles through game speeds and selects the next one (see line 3100) and remains inside the loop; if none of the above keys is pressed, the program stays in the loops and keeps checking for key presses, by jumping back to line 1550:

```
1550 LET L$ =  INKEY$
1555 IF L$ >= "1" AND L$ <= "8" THEN GO TO 1570
1557 IF L$ = "S" OR L$ = "s" THEN GO SUB 3100
1560 GO TO 1550
```

If we've come this far, we have pressed a key between 1 and 8 and are ready to start a new game with the selected scenario. Before returning to the main program, we clear all drawn sprites, by making them not visble:

```
1570 FOR I=0 TO 63: SPRITE I,0,0,0,0: NEXT I
1580 RETURN
```

Selected garden drawing routine

The routine staring at line 1600 draws the scenario selected by the player in the menu screen, by pressing a numeric key between `1` and `8`. The chosen garden is stored in string variable `L$`.

```
1600 REM *** DRAW SELECTED GARDEN ***
```

The first garden is simpler than the others, in fact it has no walls; so, in case it has been selected, the program skips the walls drawing code and jumps directly to the end of the routine, at line 1700:

```
1601 IF L$ = "1" THEN GO TO 1700
```

All other gardens have some walls, that must be rendered and taken into account during the game; in fact, the collision with a wall will make current game end.
The next lines specify, for each garden, where to find the walls data. The `RESTORE` statement, in fact, specifies at which line the data starts, the `DATA` statement defines the actual data and the `READ` statement reads this data into the provided variable, as described in the ZX Spectrum manual.

```
1602 IF L$ = "2" THEN RESTORE 2000
1603 IF L$ = "3" THEN RESTORE 2050
1604 IF L$ = "4" THEN RESTORE 2010
1605 IF L$ = "5" THEN RESTORE 2020
1606 IF L$ = "6" THEN RESTORE 2030
1607 IF L$ = "7" THEN RESTORE 2080
1608 IF L$ = "8" THEN RESTORE 2090
```

For each garden, walls data is organized in the following way:

- the first `DATA` statement is followed by the number of walls
- all subsequent `DATA` statements, one for each wall, are followed by 4 numbers, which specify the line and column numbers of the two wall ends, i.e. the minimum line number, the minimum column number, the maximum line number and the maximum column number.

For example, let's consider walls data for garden 2, which starts at line 2000.
The first `DATA` statement informs us that there are 4 walls, so the 4 subsequent data statements will provide the ends of each wall:

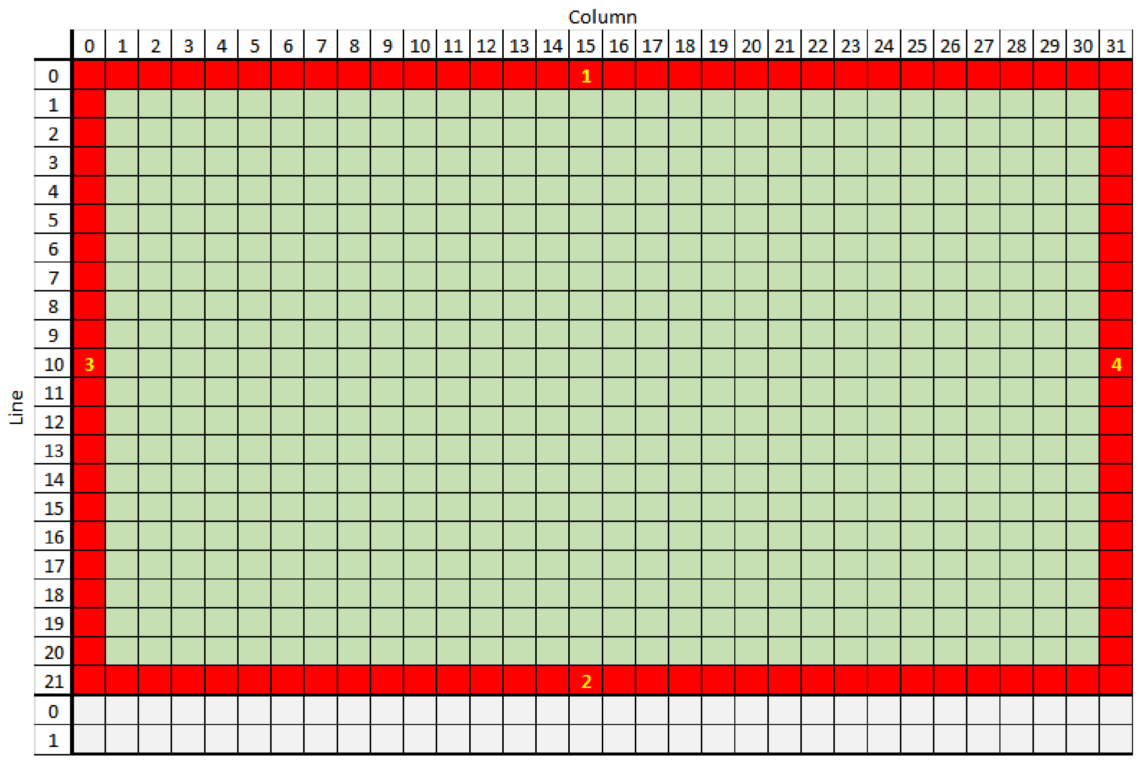
```
2000 DATA 4
```

- the first wall (line 2001) extends from line 0, column 0 to line 0, column 31;
- the second wall (line 2002) extends from line 21, column 0 to line 21, column 31;
- the third wall (line 2003) extends from line 0, column 0 to line 21, column 0;
- the fourth wall (line 2004) extends from line 0, column 31 to line 21, column 31;

```
2001 DATA 0, 0, 0, 31
2002 DATA 21, 0, 21, 31
2003 DATA 0, 0, 21, 0
2004 DATA 0, 31, 21, 31
```

So the walls in garden 2 will be laid as depicted in the following picture:

Garden 2 walls



Now, let's take a look at the code; first, for drawing the wall bricks, we set `PAPER` color to red and `INK` color to yellow:

```
1610 PAPER 2: INK 6
```

Then we read the number of walls into variable `B`:

```
1620 READ B
```

And iterate for each of the `B` walls:

```
1630 FOR K = 1  TO B
```

Fore each wall, we read the coordinates (line and column number) of each wall end:

```
1640 READ MINY: READ MINX: READ MAXY: READ MAXX
```

Then, by means of two nested loops, we iterate over each line and column within the walls ends and for each position we draw the wall bricks, by printing UDG with code 146. We also keep track of the positions occupied by the walls in the game map `M`:

```
1650 FOR I = MINY  TO MAXY
1651 FOR J = MINX  TO MAXX
1660 PRINT AT I, J;  CHR$ 146
1665 LET M(I+1, J+1) = 2
1670 NEXT J
1671 NEXT I
```

The loop on the walls number `B` ends at line 1690:

```
1690 NEXT K
```

Finally, after the walls have been drawn (or if the walls drawing has been skipped in case of Garden 1), the computer plays a sequence of notes with increasing pitch to inform the player that the game is about to begin, and then control returns to the caller:

```
1700 FOR I = -15  TO 15  STEP 2
1701 BEEP .05, I
1702 NEXT I
1900 RETURN
```

For completeness sake, walls data for all gardens is reported:

```
2000 DATA 4
2001 DATA 0, 0, 0, 31
2002 DATA 21, 0, 21, 31
2003 DATA 0, 0, 21, 0
2004 DATA 0, 31, 21, 31
2010 DATA 8
2011 DATA 0, 7, 9, 7
2012 DATA 5, 21, 5, 31
2013 DATA 11, 23, 21, 23
2014 DATA 16, 0, 16, 9
2015 DATA 0, 8, 9, 8
2016 DATA 6, 21, 6, 31
2017 DATA 11, 22, 21, 22
2018 DATA 15, 0, 15, 9
2020 DATA 4
2021 DATA 6, 7, 7, 24
2023 DATA 16, 7, 17, 24
2024 DATA 9, 6, 14, 7
2025 DATA 9, 24, 14, 25
2030 DATA 4
2031 DATA 0, 4, 2, 27
2032 DATA 19, 4, 21, 27
2033 DATA 0, 0, 21, 3
2034 DATA 0, 28, 21, 31
2050 DATA 8
2051 DATA 0, 0, 0, 31
2052 DATA 21, 0, 21, 31
2053 DATA 9, 3, 9, 28
2054 DATA 13, 3, 13, 28
2055 DATA 1, 0, 8, 0
2056 DATA 1, 31, 8, 31
2057 DATA 14, 0, 20, 0
2058 DATA 14, 31, 20, 31
2080 DATA 8
2081 DATA 13, 27, 17, 27
2082 DATA 17, 16, 17, 26
2083 DATA 4, 4, 8, 4
2084 DATA 4, 5, 4, 15
2085 DATA 13, 25, 15, 25
2086 DATA 15, 16, 15, 24
2087 DATA 6, 6, 8, 6
2088 DATA 6, 7, 6, 15
2090 DATA 11
```

```
2091 DATA 0, 0, 0, 14
2092 DATA 0, 17, 0, 31
2093 DATA 21, 0, 21, 14
2094 DATA 21, 17, 21, 31
2095 DATA 5, 3, 5, 28
2096 DATA 16, 3, 16, 28
2097 DATA 1, 0, 9, 0
2098 DATA 12, 0, 20, 0
2099 DATA 1, 31, 9, 31
2100 DATA 12, 31, 20, 31
2101 DATA 10, 15, 11, 16
```

Mongoose management routine

The mongoose management routine has some similarities with the apple management routine, in that we set the column number `MX` to the invalid -1 value to denote that the mongoose is currently not shown in the game area; instead, a valid value represents the actual column number when the mongoose is shown.

```
2500 REM *** MONGOOSE ROUTINE ***
```

The program only manages the mongoose at specific instants, i.e. when the frames counter value stored in `F` is one among 0, 64, 128, 129. Please note that, since `F` is set to the least significant byte of the frames counter system variable, when it reaches the 255 value, it starts again from 0:

```
2503 IF F <> 0 AND F <> 64 AND F <> 128 AND F <> 192 THEN RETURN
```

If the mongoose is currently hidden, jump to line 2600:

```
2505 IF MX = -1  THEN GO TO 2600
```

Mongoose shown

The next statements randomly make the mongoose hide with a probability of 60% and stay on screen with a probability of 40%.
First, set variable `MR` to a random value between 0 and 1:

```
2510 LET MR =  RND
```

Then if MR > 0.6, i.e. with a probability of 40%, leave the mongoose on screen and return to the game loop:

```
2515 IF MR > .6  THEN RETURN
```

Otherwise, make the mongoose invisible, by setting all its sprite (sprite number: 1) properties to 0, mark the position previously occupied by the mongoose as free in the game map `M`, set the mongoose column number `MX` to -1 and return to the game loop:

```
2520 SPRITE 1, 0, 0, 0, 0: REM HIDE MONGOOSE
2525 LET M(MY + 1, MX + 1) = 0
2530 LET MX = -1
2590 RETURN
```

Mongoose hidden

The next statements make the mongoose appear at a random position with a probability of 70%. First, set variable `MR` to a random value between 0 and 1:

```
2600 LET MR =  RND
```

Then if MR > 0.7, i.e. with a probability of 30%, leave the mongoose hidden and return to the game loop:

```
2605 IF MR > .7  THEN RETURN
```

Otherwise, randomly compute the mongoose coordinates:

```
2610 LET MX =  INT ( RND *32): LET MY =  INT ( RND *22)
```

Before drawing the mongoose, we must perform some check.

First, we do not want the mongoose to appear in front of the snake, otherwise it would immediately byte the snake without giving the player the chance to avoid it, and the game would be over in a rather frustrating way. So, in case the mongoose position is immediately in front of the snake (same line and column numbers as the snake head), we reset its column number `MX` to -1 and return to the game loop. The mongoose will have another chance to appear in the next game loop iteration:

```
2615 IF MX = S(H, 2)  OR MY = S(H, 1)  THEN LET MX = -1: RETURN
```

Then, we check that the calculated position is free in the game map, since we do not want the mongoose to appear over a wall or over the snake. In case the mongoose position is not free, we reset its column number `MX` to -1 and return to the game loop. The mongoose will have another chance to appear in the next game loop iteration:

```
2620 IF M(MY + 1, MX + 1) <>0  THEN LET MX = -1: RETURN
```

If we've come this far, there is nothing preventing the mongoose to appear, so we can draw its sprite at the random position we calculated a little while ago, in line 2610:

```
2625 SPRITE 1, MX*8+32, MY*8+32, 3, 1: REM DRAW MONGOOSE
```

Finally, the mongoose position is marked in the game map `M`, and the routine returns to the game loop:

```
2630 LET M(MY + 1, MX + 1) = 3
2700 RETURN
```

Welcome routine

The welcome routine displays a colorful title screen, also known as loading or splash screen, and plays a short music, adapted from Pachelbel's Canon in D.

```
2800 REM *** WELCOME (LOADING SCREEN & MUSIC INTRO) ***
```

Logo screen

`LAYER OVER` 0 statement at line 2805 instructs the computer to display sprites on top of Layer 2 graphics and Layer 2 graphics on top of standard ZX Spectrum screen:

```
2805 LAYER OVER 0
```

Then, `BORDER` and `PAPER` colors are set to black:

```
2810 BORDER 0: PAPER 0
```

and the screen is cleared with the `CLS` command. `LAYER 2,1` command enables Layer 2 display, which provides a 256-color screen at the full 256x192 resolution, in which every pixel is individually colored.
Finally, the logo image stored in the `BASNAKE.NXI` file is loaded and displayed in the active layer (which is Layer 2, previously selected) by means of the `LOAD...LAYER` command.

```
2820 CLS:LAYER 2,1:LOAD "basnake.nxi" LAYER
```

> You can use Next BMP tools by Stefan Bylund to convert your images from bitmap (.bmp) format to ZX Spectrum Next Layer 2 format (.nxi).

Lines 2822 and 2825 print some messages at the top and bottom of the screen:

```
2822 PAPER 5: INK 26: PRINT AT 23,0;"  ";CHR$ 127;"2017-2019 marco's retrobits
"
2825 PRINT AT 0,0;"baSnake 3.0 for ZX Spectrum NEXT"
```

Music

In line 2827, the computer processor speed is set to 3.5 MHz in order to play the music at the right speed:

```
2827 RUN AT 0
```

The following assignments define the notes and effects, if any, that must be played on each of the 9 available sound channels:

```
2830 LET a$="UX5000T60O5W3#FED#Cbab#C"
2831 LET b$="UX5000T60O3W6Dab#fgdga"
2832 LET c$=""
2833 LET d$=""
2834 LET e$="UX2500T60O3W3Dab#fgdga"
2835 LET f$="UX2500O4W6N3D#FAG#FD#FEDbDAGBAG"
2836 LET g$=""
2837 LET h$=""
2838 LET i$="UX5000T60O5W3D#Cbag#fge"
```

Line 2840 lets the music play:

```
2840 PLAY a$,b$,c$,d$,e$,f$,g$,h$,i$
```

For all details on playing music on the ZX Spectrum Next, you can read Chapter 19 of the manual.

When the music is over, we reset computer speed to 14 MHz (`RUN AT` 2), inform the player that he must press any key to start the game and wait for the key press with the `PAUSE` 0 command.

```
2850 RUN AT 2
2860 PRINT AT 0,0; "   PRESS  ANY  KEY  TO  START   "
2870 PAUSE 0
```

After the key press, Layer 2 is disabled (`LAYER 2,0`) and the standard ZX Spectrum mode is activated (`LAYER 0`):

```
2880 LAYER 2,0: LAYER 0
```

Then, control returns to the caller:

```
2890 RETURN
```

Print turbo mode routine

The routine starting at line 3050 simply prints to screen a description, borrowed from the music terminology, of the selected game speed, which is stored in variable `TRB`:

```
3050 REM *** PRINT TURBO MODE ***
3052 INK 7
3055 PRINT AT 11, 0; "Turbo mode: ";
3056 INK 5
3060 IF TRB = 11 THEN PRINT "ANDANTE ": GO TO 3070
3061 IF TRB = 9  THEN PRINT "MODERATO": GO TO 3070
3062 IF TRB = 7  THEN PRINT "ALLEGRO ": GO TO 3070
3063 IF TRB = 5  THEN PRINT "VIVACE  ": GO TO 3070
3064 IF TRB = 3  THEN PRINT "PRESTO  ": GO TO 3070
3070 INK 7
3090 RETURN
```

Select turbo mode routine

The routine starting at line 3100 is invoked when the player cycles through available game speed in main menu, by pressing the `S` key.

```
3100 REM *** CHANGE TURBO MODE ***
```

It calculates the next `TRB` by subtracting 2 from current value (in fact, lower values of the `TRB` variable correspond to higher game speeds):

```
3105 LET TRB = TRB - 2
```

If the lower bound is exceeded, `TRB` is set to the maximum value:

```
3110 IF TRB = 1  THEN LET TRB = 11
```

The selected speed is then shown on screen, by calling the turbo mode print routine:

```
3115 GO SUB 3050
```

After a small pause, control returns to the caller:

```
3120 PAUSE 300
3130 RETURN
```

Input routine

The input routine starting at line 4000 loops, checking the keyboard status and if a key corresponding to the snake direction has been pressed, the snake direction is updated. The loops terminates and the routine ends when the time elapsed since the beginning of the game loop matches the requested game speed, and so the next game loop iteration can start. The change of direction in the snake movement is allowed only if the new direction is perpendicular to current direction, i.e. if the snake is currently moving horizontally, the allowed new directions are from top to bottom and from bottom to top; instead, if the snake is moving vertically, the allowed directions are from left to right and from right to left.

```
4000 REM *** INPUT ***
```

The loop starts by saving current snake direction `D` into temporary variable `DT`:

```
4010 LET DT = D
```

Direction can assume one of the following values:

Value	Direction
1	up
2	down
3	left
4	right

Then the code of currently pressed key, if any, is saved in `K` variable (`CODE` function returns the code of the character provided as input and `INKEY$` returns a string containing currently pressed character, if any):

```
4100 LET K = CODE(INKEY$)
```

The keys and character codes that allow the player to change the snake direction are:

Key/Character	Code	Direction
↑	11	1: up
Q	81	1: up
q	113	1: up
↓	10	2: down
A	65	2: down
a	97	2: down
←	8	3: left
O	79	3: left
o	111	3: left
→	9	4: right
P	80	4: right
p	112	4: right

The following lines store in DT the new snake direction, based on current direction and the code of the key pressed by the player:

```
4210 IF (K = 11 OR K = 81 OR K = 113)  AND (D = 3 OR D = 4)  THEN LET DT = 1
4220 IF (K = 10 OR K = 65 OR K = 97)  AND (D = 3 OR D = 4)  THEN LET DT = 2
4230 IF (K = 8 OR K = 79 OR K = 111)  AND (D = 1 OR D = 2)  THEN LET DT = 3
4240 IF (K = 9 OR K = 80 OR K = 112)  AND (D = 1 OR D = 2)  THEN LET DT = 4
```

If no key is pressed, the snake continues moving with current direction.

The input loop is executed until the game loop iteration elapsed time, computed as the difference between the current value of the frames counter and the value of the frames counter, saved at the beginning of the game loop in variable `F`, exceeds the value stored in `TRB`.

```
4280 IF ABS(PEEK 23672 - F) < TRB THEN GO TO 4100
```

So, smaller values of `TRB` will determine a faster game speed, because the number of frames that must elapse before the next game loop iteration is mall; instead, high values of `TRB` will determine a slower game speed. `TRB` value is assigned in the main menu, when the player selects the game speed by pressing the `s` key.

When the necessary time to wait for the next game tick elapses, the chosen direction is saved into `D` variable:

```
4290 LET D = DT
```

and then control returns to the game loop:

```
4300 RETURN
```

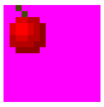
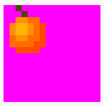
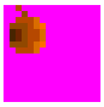
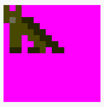
Sprites loading and initialization routine

This routine's responsibility is mainly loading sprite image data from the `BASNAKE.SPR` file and initializing the sprite system. Sprites are a new feature introduced with the ZX Spectrum Next and were not available on the classic ZX Spectrum; you can read more about sprites in NextBASIC new commands and features or on the ZX Spectrum Next Sprites Wiki.

```
5000 REM *** SPRITES INITIALIZATION ***
```

> You can use *UDGeedNext* tool by David Saphier for painting your own sprites or converting an existing image to ZX Spectrum Next sprites and saving them to a sprite data file. The tool is included in the NextBuild IDE.

baSnake uses 8 sprite images, identified by numbers from 0 to 6; each sprite is 16x16 pixels wide; magenta color defines transparency:

Sprite image number	Image description	Sprite image (3x zoom)
0	Red apple	
1	Golden apple	
2	Rotting apple	
3	Mongoose	
4, 5, 6,7	*baSnake* writing, displayed in menu screen and made up of 4 sprite images	

Line 5001 loads the content of the sprites data file into bank 12 of ZX Spectrum Next RAM:

```
5001 LOAD "basnake.spr" BANK 12
```

Line 5010 defines all sprites patterns using data previously loaded into bank 12.

```
5010 SPRITE BANK 12
```

Line 5020 resets the sprites attributes and settings to defaults:

```
5020 SPRITE CLEAR
```

Line 5030 enables sprites:

```
5030 SPRITE PRINT 1
```

Line 5040 prevents sprites to be displayed on the screen border:

```
5040 SPRITE BORDER 0
```

Finally, line 5050 returns to the initializations code, whence this subroutine was called:

```
5050 RETURN
```

Once the sprite images have been loaded and the sprite system initialized, the rest of the program will be able to draw sprites to the screen, by means of the `SPRITE` command which, for each sprite, will define the sprite image to be displayed and the sprite properties, such as its position and visibility flag.

baSnake program will use 6 sprites:

Sprite number	Sprite description	Sprite image numbers
0	Apple sprite; will be associated to one if the 3 sprite images number, depending on the value of the apple lifecycle counter `FD`	0, 1, 2
1	Mongoose sprite	3
2, 3, 4, 5	*baSnake* writing: 4 sprites are required to display simultaneously the 4 sprite images making up the writing	4, 5, 6, 7

Full source code

For completeness sake, the full *baSnake* source code is reported.

> The *baSnake* program has not been typed directly on a ZX Spectrum Next, but on a text editor running on a PC. Then, the text file containing the source code has been converted to the `BASNAKE.BAS` file using the **.txt2bas** dot command created by Garry Lancaster. At the moment of writing this document, .txt2bas is still in testing phase, but should be publicly released soon.
> The `#autostart` text after line 5 is not a basic command, but a directive instructing .txt2bas to make the converted BASIC program automatically start execution at the following line.

```
   1 REM ***********************
   2 REM *    baSnake 3.0.0    *
   3 REM *  ZX Spectrum Next   *
   4 REM * Marco Varesio 2019  *
   5 REM ***********************
#autostart
   7 LET TRB = 7 : RUN AT 2
   8 GO SUB 5000: GO SUB 2800
   9 GO SUB 900
  10 REM *** MAIN MENU ***
  11 LET HI = 0
  12 GO SUB 1500
  15 RANDOMIZE
  20 DIM M(22, 32): DIM S(704, 2)
  22 BRIGHT 0: BORDER 5: PAPER 4: CLS
  25 GO SUB 1600
  30 LET SC = 0
  35 LET T = 1: LET H = 1
  40 LET S(1, 1) = 11: LET S(1, 2) = 8
  60 LET Y = S(1, 1): LET X = S(1, 2)
  70 LET D = 4: LET G = 4
  80 LET FX = -1: LET FY = -1: LET FD = 0
  81 LET MX = -1: LET MY = -1: LET MR = 0
  90 PRINT #1;  AT 1, 0; "Score: "; SC, "High: "; HI
 100 REM *** MAIN GAME LOOP ***
 110 LET F = PEEK 23672: IF G = 0  THEN GO SUB 1000
 250 IF D = 1  THEN LET Y = Y - 1 : GO TO 270
 255 IF D = 2  THEN LET Y = Y + 1 : GO TO 270
 260 IF D = 3  THEN LET X = X - 1 : GO TO 270
 265 IF D = 4  THEN LET X = X + 1 : GO TO 270
 270 IF Y < 0  THEN LET Y = 21 : GO TO 290
 275 IF Y > 21  THEN LET Y = 0 : GO TO 290
 280 IF X < 0  THEN LET X = 31 : GO TO 290
 285 IF X > 31  THEN LET X = 0 : GO TO 290
 290 GO SUB 2500
 300 IF X <>FX  OR Y <>FY  THEN GO TO 400
 310 LET G = G + 3: SPRITE 0, 0, 0, 0, 0: LET FX = -1: REM HIDE APPLE
 320 IF FD > 16 THEN LET SC = SC + 5: ELSE IF FD > 8 THEN LET SC = SC + 10: ELSE
LET SC = SC + 1
 330 PRINT #1;  AT 1, 0; "Score: "; SC, "High: "; HI
 400 INK 0
 405 REM *** Draw new snake head ***
 410 PRINT AT S(H, 1), S(H, 2);  CHR$ 144
 420 LET H = H + 1: IF H = 705  THEN LET H = 1
 430 LET S(H, 1) = Y: LET S(H, 2) = X
 440 PRINT AT S(H, 1), S(H, 2);  CHR$ 145
 450 IF G > 0  THEN GO TO 490
 452 REM *** Delete snake tail ***
 455 LET TY = S(T, 1): LET TX = S(T, 2)
 460 PRINT AT TY, TX; " "
 465 LET M(TY + 1, TX + 1) = 0
 470 LET T = T + 1: IF T = 705  THEN LET T = 1
 475 REM BEEP .008, -20
 480 GO TO 500
 490 LET G = G - 1
 495 IF G = 2  THEN BEEP .04, -10
 496 IF G = 1  THEN BEEP .04, -20
 497 IF G = 0  THEN BEEP .04, -5
 500 LET MV = M(Y+1, X+1)
 501 IF MV = 1  THEN LET M$ = " You bit yourself ": GO TO 700
 502 IF MV = 2  THEN LET M$ = " You hit the wall ": GO TO 700
 503 IF MV = 3  THEN LET M$ = " The mongoose bit you ": GO TO 700
 510 LET M(Y+1, X+1) = 1
 590 GO SUB 4000
 600 GO TO 100
 700 REM *** GAME OVER ***
 701 FOR I=0 TO 1: SPRITE I, 0, 0, 0, 0: NEXT I: REM HIDE APPLE AND MONGOOSE
 705 INK 3
 710 LET I=H
 715 REPEAT
 717  LET C=144: IF I = H THEN LET C=147
 720  IF I = H OR S(I, 1) <> S(H, 1) OR S(I, 2) <> S(H, 2) THEN PRINT AT S(I,
1), S(I, 2); CHR$ C
```

```
 725  WHILE I <> T
 730  LET I=I-1
 735  IF I = 0 THEN LET I = 704
 737  PAUSE 1
 740 REPEAT UNTIL 0
 810 PAPER 1
 811 FOR I = 6  TO 15
 812 PRINT AT I, 0; "                                "
 813 NEXT I
 814 BORDER 6: PAPER 2: INK 7
 815 PRINT AT 9, (32 -  LEN (M$)) / 2; M$
 820 INK 6: FLASH 1
 821 PRINT AT 7, 10; " GAME OVER! "
 825 FLASH 0: PAPER 5: INK 1
 830 IF SC <=HI  THEN GO TO 850
 840 LET HI = SC
 841 LET M$ = " New high score : "+  STR$ (HI) + " "
 845 PRINT AT 11,  INT ((32- LEN (M$))/2); M$
 850 PAPER 1: INK 7
 870 FOR I = 15  TO -30  STEP -2
 871 BEEP .05, I
 872 NEXT I
 875 PRINT AT 13, 1; "PRESS "; : INVERSE 1 : PRINT "M"; : INVERSE 0 : PRINT " TO
RETURN TO MAIN MENU"
 876 PRINT AT 14, 1; "OR ANY OTHER KEY TO PLAY AGAIN"
 880 LET K$ =  INKEY$
 885 IF K$ = "" THEN GO TO 880
 890 IF K$ = "M" OR K$ = "m" THEN GO TO 10
 895 GO TO 15
 900 REM *** UDGs ***
 901 RESTORE 903
 902 FOR I = 0  TO 31: READ L: POKE 65368+I, L: NEXT I
 903 DATA 60, 66, 129, 129, 129, 129, 66, 60: REM SNAKE BODY 144
 904 DATA 60, 66, 165, 129, 165, 153, 66, 60: REM SNAKE HEAD 145
 905 DATA 4, 4, 4, 255, 64, 64, 64, 255: REM WALL 146
 906 DATA 60, 66, 165, 129, 129, 153, 66, 60: REM SNAKE SAD 147
 910 RETURN
1000 REM *** APPLE ROUTINE ***
1001 REM APPLE NOT YET FALLEN
1005 IF FX <>-1  THEN GO TO 1100
1010 LET FX =  INT ( RND *32): LET FY =  INT ( RND *22): LET FD = 40+ INT ( RND
*30)
1015 IF M(FY + 1, FX + 1) <>0  THEN LET FX = -1 : LET FD = 0: RETURN
1030 SPRITE 0, FX*8+32, FY*8+32, 0, 1: REM RED APPLE
1040 RETURN
1100 REM APPLE ALREADY FALLEN
1105 LET FD = FD - 1
1110 IF FD = 8  THEN SPRITE 0, FX*8+32, FY*8+32, 2, 1 : RETURN : REM ROTTEN
APPLE
1111 IF FD = 16  THEN SPRITE 0, FX*8+32, FY*8+32, 1, 1 : RETURN : REM YELLOW
APPLE
1115 IF FD = 0  THEN SPRITE 0, 0, 0, 0, 0: LET FX = -1: REM HIDE APPLE
1130 RETURN
1500 BORDER 0: PAPER 0: CLS
1505 FOR I=0 TO 3: SPRITE I+2,120+16*I,32,I+4,1: NEXT I
1506 INK 7: PRINT AT 1,26; "v. 3.0": INK 5
1507 PRINT "Guide the snake "; : INK 4: PRINT CHR$ 145;  CHR$ 144;  CHR$ 144;
 CHR$ 144; : INK 5: PRINT " through the"
```

```
1508 PRINT "garden, eating the apples   that": SPRITE 0,240,56,0,1
1509 PRINT "fall from the tree,  before they"
1510 PRINT "rot. Avoid the walls "; : PAPER 2: INK 6: PRINT CHR$ 146;  CHR$ 146;
 CHR$ 146; : PAPER 0: INK 5: PRINT " and the"
1511 PRINT "mongoose   .": SPRITE 1, 104, 80, 3, 1
1512 PRINT
1513 INK 7:PRINT "There are 8 different gardens:"
1514 INK 6: PRINT "press key from "; : INVERSE 1 : PRINT "1";
1515 INVERSE 0: PRINT " to "; : INVERSE 1: PRINT "8";: INVERSE 0
1516 PRINT " to choose."
1517 PRINT
1518 GO SUB 3050
1519 INK 6: PRINT "Press "; : INVERSE 1: PRINT "S"; : INVERSE 0 : PRINT " to
change speed."
1520 INK 7: PRINT : PRINT "Snake controls:": INK 6: INVERSE 1
1521 PRINT "Q"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "UP"
1522 PRINT "A"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "DOWN"
1523 PRINT "O"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "LEFT"
1524 PRINT "P"; : INVERSE 0: PRINT " OR "; : INVERSE 1: PRINT "RIGHT"
1525 INVERSE 0: PRINT
1527 INK 5: PRINT "Check out my other retro stuff @": INK 7
1528 PRINT "https://retrobits.altervista.org"
1529 PRINT #1;  AT 0, 0; "   https://retrobits.itch.io    "
1530 PRINT #1; "Enjoy!     Marco 'Pulce' Varesio"
1550 LET L$ =  INKEY$
1555 IF L$ >= "1" AND L$ <= "8" THEN GO TO 1570
1557 IF L$ = "S" OR L$ = "s" THEN GO SUB 3100
1560 GO TO 1550
1570 FOR I=0 TO 63: SPRITE I,0,0,0,0: NEXT I
1580 RETURN
1600 REM *** DRAW SELECTED GARDEN ***
1601 IF L$ = "1" THEN GO TO 1700
1602 IF L$ = "2" THEN RESTORE 2000
1603 IF L$ = "3" THEN RESTORE 2050
1604 IF L$ = "4" THEN RESTORE 2010
1605 IF L$ = "5" THEN RESTORE 2020
1606 IF L$ = "6" THEN RESTORE 2030
1607 IF L$ = "7" THEN RESTORE 2080
1608 IF L$ = "8" THEN RESTORE 2090
1610 PAPER 2: INK 6
1620 READ B
1630 FOR K = 1  TO B
1640 READ MINY: READ MINX: READ MAXY: READ MAXX
1650 FOR I = MINY  TO MAXY
1651 FOR J = MINX  TO MAXX
1660 PRINT AT I, J;  CHR$ 146
1665 LET M(I+1, J+1) = 2
1670 NEXT J
1671 NEXT I
1690 NEXT K
1700 FOR I = -15  TO 15  STEP 2
1701 BEEP .05, I
1702 NEXT I
1900 RETURN
2000 DATA 4
2001 DATA 0, 0, 0, 31
2002 DATA 21, 0, 21, 31
2003 DATA 0, 0, 21, 0
```

```
2004 DATA 0, 31, 21, 31
2010 DATA 8
2011 DATA 0, 7, 9, 7
2012 DATA 5, 21, 5, 31
2013 DATA 11, 23, 21, 23
2014 DATA 16, 0, 16, 9
2015 DATA 0, 8, 9, 8
2016 DATA 6, 21, 6, 31
2017 DATA 11, 22, 21, 22
2018 DATA 15, 0, 15, 9
2020 DATA 4
2021 DATA 6, 7, 7, 24
2023 DATA 16, 7, 17, 24
2024 DATA 9, 6, 14, 7
2025 DATA 9, 24, 14, 25
2030 DATA 4
2031 DATA 0, 4, 2, 27
2032 DATA 19, 4, 21, 27
2033 DATA 0, 0, 21, 3
2034 DATA 0, 28, 21, 31
2050 DATA 8
2051 DATA 0, 0, 0, 31
2052 DATA 21, 0, 21, 31
2053 DATA 9, 3, 9, 28
2054 DATA 13, 3, 13, 28
2055 DATA 1, 0, 8, 0
2056 DATA 1, 31, 8, 31
2057 DATA 14, 0, 20, 0
2058 DATA 14, 31, 20, 31
2080 DATA 8
2081 DATA 13, 27, 17, 27
2082 DATA 17, 16, 17, 26
2083 DATA 4, 4, 8, 4
2084 DATA 4, 5, 4, 15
2085 DATA 13, 25, 15, 25
2086 DATA 15, 16, 15, 24
2087 DATA 6, 6, 8, 6
2088 DATA 6, 7, 6, 15
2090 DATA 11
2091 DATA 0, 0, 0, 14
2092 DATA 0, 17, 0, 31
2093 DATA 21, 0, 21, 14
2094 DATA 21, 17, 21, 31
2095 DATA 5, 3, 5, 28
2096 DATA 16, 3, 16, 28
2097 DATA 1, 0, 9, 0
2098 DATA 12, 0, 20, 0
2099 DATA 1, 31, 9, 31
2100 DATA 12, 31, 20, 31
2101 DATA 10, 15, 11, 16
2500 REM *** MONGOOSE ROUTINE ***
2503 IF F <> 0 AND F <> 64 AND F <> 128 AND F <> 192 THEN RETURN
2505 IF MX = -1  THEN GO TO 2600
2510 LET MR =  RND
2515 IF MR > .6  THEN RETURN
2520 SPRITE 1, 0, 0, 0, 0: REM HIDE MONGOOSE
2525 LET M(MY + 1, MX + 1) = 0
2530 LET MX = -1
```

```
2590 RETURN
2600 LET MR =  RND
2605 IF MR > .7  THEN RETURN
2610 LET MX =  INT ( RND *32): LET MY =  INT ( RND *22)
2615 IF MX = S(H, 2)  OR MY = S(H, 1)  THEN LET MX = -1: RETURN
2620 IF M(MY + 1, MX + 1) <>0  THEN LET MX = -1: RETURN
2625 SPRITE 1, MX*8+32, MY*8+32, 3, 1: REM DRAW MONGOOSE
2630 LET M(MY + 1, MX + 1) = 3
2700 RETURN
2800 REM *** WELCOME (LOADING SCREEN & MUSIC INTRO) ***
2805 LAYER OVER 0
2810 BORDER 0: PAPER 0: REM POKE 23739,244
2820 CLS:LAYER 2,1:LOAD "basnake.nxi" LAYER
2822 PAPER 5: INK 26: PRINT AT 23,0;"  ";CHR$ 127;"2017-2019 marco's retrobits
"
2825 PRINT AT 0,0;"baSnake 3.0 for ZX Spectrum NEXT"
2827 RUN AT 0: REM OUT 9275, 7: OUT 9531, 0
2830 LET a$="UX5000T60O5W3#FED#Cbab#C"
2831 LET b$="UX5000T60O3W6Dab#fgdga"
2832 LET c$=""
2833 LET d$=""
2834 LET e$="UX2500T60O3W3Dab#fgdga"
2835 LET f$="UX2500O4W6N3D#FAG#FD#FEDbDAGBAG"
2836 LET g$=""
2837 LET h$=""
2838 LET i$="UX5000T60O5W3D#Cbag#fge"
2840 PLAY a$,b$,c$,d$,e$,f$,g$,h$,i$
2850 RUN AT 2: REM OUT 9275, 7: OUT 9531, 2
2860 PRINT AT 0,0; "   PRESS  ANY  KEY  TO  START   "
2870 PAUSE 0
2880 LAYER 2,0: LAYER 0
2890 RETURN
3050 REM *** PRINT TURBO MODE ***
3052 INK 7
3055 PRINT AT 11, 0; "Turbo mode: ";
3056 INK 5
3060 IF TRB = 11 THEN PRINT "ANDANTE ": GO TO 3070
3061 IF TRB = 9  THEN PRINT "MODERATO": GO TO 3070
3062 IF TRB = 7  THEN PRINT "ALLEGRO ": GO TO 3070
3063 IF TRB = 5  THEN PRINT "VIVACE  ": GO TO 3070
3064 IF TRB = 3  THEN PRINT "PRESTO  ": GO TO 3070
3070 INK 7
3090 RETURN
3100 REM *** CHANGE TURBO MODE ***
3105 LET TRB = TRB - 2
3110 IF TRB = 1  THEN LET TRB = 11
3115 GO SUB 3050
3120 PAUSE 300
3130 RETURN
4000 REM *** INPUT ***
4010 LET DT = D
4100 LET K = CODE(INKEY$)
4210 IF (K = 11 OR K = 81 OR K = 113)  AND (D = 3 OR D = 4)  THEN LET DT = 1
4220 IF (K = 10 OR K = 65 OR K = 97)  AND (D = 3 OR D = 4)  THEN LET DT = 2
4230 IF (K = 8 OR K = 79 OR K = 111)  AND (D = 1 OR D = 2)  THEN LET DT = 3
4240 IF (K = 9 OR K = 80 OR K = 112)  AND (D = 1 OR D = 2)  THEN LET DT = 4
4280 IF ABS(PEEK 23672 - F) < TRB THEN GO TO 4100
4290 LET D = DT
```

```
4300 RETURN
5000 REM *** SPRITES LOAD & INITIALIZATION ***
5001 LOAD "basnake.spr" BANK 12
5010 SPRITE BANK 12
5020 SPRITE CLEAR
5030 SPRITE PRINT 1
5040 SPRITE BORDER 0
5050 RETURN
```

Recommended readings

At the time of writing this document, the ZX Spectrum Next manual has not yet been released; however, it will be the main reference for BASIC programming on the Spectrum Next.

The classic ZX Spectrum BASIC Programming manual and the comp.sys.sinclair FAQ are always good reads; after all, the ZX Spectrum Next is an evolution of the classic ZX Spectrum.

NextZXOS and NextBASIC documentation by Garry Lancaster, which is provided in the official distribution, is essential for understanding the new features introduced with the Spectrum Next.

The SpecNext Developers H.Q. website is the official and the most important web resource, with developer blogs, tutorials and an ever growing wiki.

Acknowledgments

Thanks to the ZX Spectrum Next team and community!