

Confessions of a Game Programmer: Robber Remake

Introduction

This article is the start of a series about game programming, which is intended for those getting into game programming for the first time or for general interest. It will attempt to explain the shown game code used and also show lateral thinking for solving problems that I may come across.

I will tend to omit code in places to avoid repetition, however, I will provide the full code at the end of the article.

This series will act not only as a record of my programming progress and achievements in both style and coding semantics but it will also serve as a good reference guide to those in need.

Disclaimer

Currently, with the exception of this game, the other games in the series won't be as badly coded. The reason for this is that at the time the GLBasic forum hosted a game programming competition and to enter, a game had to be written in GLBasic. I didn't have much experience with the IDE (Integrated Development Environment) so I elected to make a game that would be quick to do.

Since I first came across this competition via a thread in the Retro Remakes forum I decided to go down that route and I remembered an old game that I used to play on my ZX Spectrum 48K, this being the titular piece of the article you are reading now.

To code it, the first 2 levels took me about 2-3 days and I coded the rest on and off over the course of a year. The full time it took to code, in total, probably amounted to about a week.

The Foundations of Game Programming

If you are ever unsure of how to program a game then think of it this way: Do you remember the first time you saw or drew a set of “frames” in a page of a book and flicked through them to simulate animation?

Here is a wikipedia link of this for those that aren't sure what I mean:

http://en.wikipedia.org/wiki/Flip_book

Well, games are pretty much are coded in that fashion, with the pages being the frames or game cycles. Simplistically, the main function of any 2D game generally looks like this:

```
main()
{
    while gameNotFinished == true
    {
        handleKeyPressFunction()
        changeAndUpdateScreen()
    }
}
```

It doesn't look much does it? The things that fill it out are the cutscenes and other logic, such as game AI (Artificial Intelligence), but all I really wanted to do was to put you in that mindset, so that every time you start a game project you start thinking simplistically and build from there.

The Code

Before we start, I happened to mention that I was using GLBasic to write this game. This isn't a hard language to learn, in fact, it's quite a good starter language for game programming as it has functions and commands already set for you. We'll cover those as we progress.

The main function:

On line 5 is the main function that is present in all languages, in some form or another, when the program starts, this is where the code will start to execute from.

```
1] GLOBAL torchx = 289, torchy = 209, up = TRUE, down = FALSE, item[],
levelState, gameDone = FALSE, bridge1[], bridge2[], lives = 3, jelly2X = 107,
jelly2Y = 144, jelly3X = 107, jelly3Y = 170, jelly4X = 16, jelly4Y = 170, birdX
= 185, birdY = 118, bird2X = 263, bird2Y = 118, bird3X = 263, bird3Y = 157,
ballX = 211, ballY = 53

2] DIM item[3]
3] DIM bridge1[9]
4] DIM bridge2[10]

5] Main:

6]     GOSUB initGame

7]     GOSUB titleScreen

8]     GOSUB initLvl1
9]     // handle Key area

10]    WHILE gameDone = FALSE
11]        SELECT levelState
12]            CASE 1
13]                GOSUB handleKey1

14]            CASE 2
15]                GOSUB handleKey2

16]            CASE 3
17]                GOSUB handleKey3
18]        ENDSELECT
19]    WEND

20] END
```

Before the main function we can declare variables that will be available everywhere in the code and will not be “out of scope”. These are known as GLOBAL variables and lines 1 – 4 show this. DIM creates an array of a specific type.

In this context, these variables are set specifically for the game to work and will be utilised further in the code to make changes to the values or for the values to have basis for the code to work from.

The first 3 lines within the `Main` declaration contain a command called GOSUB, all this does is to tell the program to jump to that section of the code (or the SUB program) and continue executing from there. Once the program finishes executing from that SUB then it will continue to the next line in the main program, which, in this case, would be GOSUB titleScreen.

Line 5 is a comment line, which will be ignored by the program and its only purpose is for the programmer. To start a comment line you use // (two slashes) or if you want to write a block of comments the notation is /* */ (slash, star/asterix, star/asterix, slash). For example:

```
// this is a line comment
```

```
/* this is  
a block comment */
```

Comments are written to help us understand what code segments are doing, the comments can be as long or as short as you like. Although, in hindsight, this seems silly, it's damned useful when you are otherwise busy and haven't looked at your code after a few weeks and you can't quite remember why you wrote it or aren't quite clear as to what it does.

Lines 6 through to 15 is the main game loop. This is made up of a WHILE statement (a loop that keeps repeating itself until `gameDone = true`) and a SELECT statement (in other languages this is known as a SWITCH statement). The interesting thing about the SELECT statement is that it is a cleaner version of an IF..THEN..ELSE statement, although, unlike the IF..THEN..ELSE statement you can only check against one variable. Essentially, the SELECT statement is nothing more than a singular variable switchboard. Currently, it is being controlled by the `levelState` variable.

Finally, on line 16, we come to an END. This, when executed, will end the program.

InitGame:

```
// ----- //  
//   initGame   //  
// ----- //  
  
1] SUB initGame:  
2]   LOADSPRITE "wall.bmp", 0  
  
3]   LOADFONT "screen1toptextfont.bmp", 2  
  
4] ENDSUB
```

This is the first of our SUBs or sub programs and to be honest it originally wasn't implemented until I was coding the 3rd screen. It turned out that the way I'd programmed it was that the wall and text were loaded in the later SUBs and I'd used the `levelState` variable to skip ahead, subsequently missing those subs, for testing purposes.

TIP: Using the `levelState` variable to skip ahead is very useful when you need to test for bugs. Suppose you have a lot of screens and don't want to have to play through all the screens/levels just to test a new area. Changing this variable early on ensures that you can skip to that relevant area very quickly.

The `LOADSPRITE` and `LOADFONT` commands do similar things for different purposes. They both load bitmap files, for sprites and fonts respectively, into memory and use an number to refer to the file. So, for example, when I come to use the `wall.bmp` sprite I'd refer to it as 0.

TitleScreen:

```
// ----- //
// titleScreen //
// ----- //

1] SUB titleScreen:

2]   LOADSPRITE "Robber.bmp", 50
3]   STRETCHSPRITE 50, 0, 0, 320, 240

4]   PRINT "Original by", 65, 70
5]   PRINT "Keith Mitchell", 55, 85

6]   PRINT "Remake by", 50, 180
7]   PRINT "James 'namco' Dunn", 15, 195

8]   DRAWRECT 105, 210, 100, 25, RGB(0, 0, 0)
9]   PRINT "Press a key", 110, 215

10]  SHOWSCREEN

11]  KEYWAIT

12]  ENDSUB
```

The titleScreen program has a returning command being LOADSPRITE, although you'll notice that it is set at 50 because I added this last, and 5 new commands. The first one, STRETCHSPRITE, accepts a referring sprite number, an x, y position (separately) and the width and height to expand the bitmap to. The original size of the Robber.bmp bitmap was 256 x 192. The PRINT command accepts text and an x, y position; DRAWRECT accepts an x, y co-ordinate (starting from the upper left corner), the width of the rectangle, height and colour in rgb format (red, green, blue) – the values are from 0[no colour] – 255[full colour]; SHOWSCREEN draws the preceding 5 lines onto the screen (note: normally in order to show the Robber.bmp you'd have to use the DRAWSPRITE command, but the STRETCHSPRITE command will not only stretch the sprite but will also function as a DRAWSPRITE command) and KEYWAIT will pause the execution of the program and wait until a key is pressed. Interestingly this will remember what the keypress is, which will become useful later on.

Another thing to note about the SHOWSCREEN command is that it will process the previous statements **in order**. This means that swapping lines around will create odd results, for example, if we swapped:

```
DRAWRECT 105, 210, 100, 25, RGB(0, 0, 0)
PRINT "Press a key", 110, 215
```

to be:

```
PRINT "Press a key", 110, 215
DRAWRECT 105, 210, 100, 25, RGB(0, 0, 0)
```

you would find that "Press a key" has disappeared and the black rectangle looks fully filled. This is because the text is situated behind the rectangle, since that was the order the SHOWSCREEN command processed the preceding lines.

InitLvl1:

```
// ----- //
//  initLvl1  //
// ----- //

1] SUB initLvl1:
2]   done1 = FALSE
3]   endlevel1 = FALSE
4]   nicked = FALSE
5]   testTorchLight = TRUE
6]   levelState = 1

7]   FOR j = 0 TO 2
8]     item[j] = FALSE
9]   NEXT

10]  GOSUB InstructionsScreen
11]  LOADSPRITE "player.bmp", 1
12]  playerx = 29
13]  playery = 209

14]  DRAWSPRITE 1, playerx, playery

15]  safekeyXPos = randXPos()
16]  safekeyYPos = randYPos()

17]  GOSUB Screen1
18]  ENDSUB
```

Here is where the proper setup for level 1 is done. By setup I mean variables for that level that get set/reset. We have: done1 – is level 1 complete?, endlevel1 – similar to done1 but it is used to escape the while loop (as we will see in handleKey1), nicked – initiates the nicked “death” scene, testTorchLight – allows collision between the torchlight and the player to occur, playerx/playery – sets up the position of the player sprite and safekeyX/YPos – returns an X/Y position for placing the safe key.

item[j], however, is an array, which is basically a table of variables of the same type and that is a table of boolean variables. These are all assigned to false with the use of the FOR loop.

InstructionsScreen:

```
// ----- //
// Instructions Screen //
// ----- //

1] SUB InstructionsScreen:
   ....
2]   PRINT "Press a key to goto next page", 30, 180
3]   PRINT "or s to start", 95, 195

4]   SHOWSCREEN

5]   KEYWAIT

6]   // 's' to start any other key for page 2 of instructions
7]   IF KEY(31)
8]     // do nothing
9]   ELSE
10]    CLEARSCREEN

11]    DRAWRECT 0, 0, 320, 240, RGB(0, 0, 200)

12]    PRINT "Screen 3:", 30, 20
13]    PRINT "Dodge jellyfish in the pool,", 30, 35
14]    PRINT "bats in linked vertical tunnels", 30, 50
15]    PRINT "and balls in the caves. Then", 30, 65
```

```

16] PRINT "go on to the secret underground", 30, 80
17] PRINT "vault!!", 30, 95
18] PRINT "Press any key to start!", 70, 150

19] SHOWSCREEN

20] KEYWAIT
21] ENDIF

22] ENDSUB

```

I've removed most of the code here as they are just a few PRINT statements, however, after the KEYWAIT command we see an IF..THEN..ELSE statement. This IF statement queries the KEY command to find out if the user pressed 's' and as I said before, the first KEYWAIT command is responsible for remembering the last key pressed. If the user presses 's' then the program will not do anything and then jump to ENDSUB, otherwise it will CLEARSCREEN (removes all the drawn items from the screen, leaving it black) and will draw a blue rectangle with the following text (lines 12 to 18).

RandXPos:

```

1] FUNCTION randXPos:
2]   num = RND(3)
3]   SELECT num
4]     CASE 0
5]       xPos = 237
6]     CASE 1
7]       xPos = 250
8]     CASE 2
9]       xPos = 263
10]    CASE 3
11]      xPos = 276
12]    DEFAULT
13]      //do nothing
14]  ENDSELECT
15]  RETURN xPos
16] ENDFUNCTION

```

This FUNCTION operates like the SUB, in that it is a sub program, with a slight difference. It can return a value. As a perfect example, the first line of this function shows num = RND(3). This RND function will generate a random number from 0 – 3 and return it to the num variable. This num variable is then used by a SELECT statement to choose an xPos variable and then RETURN that variable to whatever is assigned to this function. As we saw earlier in the initLv11 sub, that variable was safekeyXPos. The randYPos function operates in the exact same way with the differences being that it randomises from 0 – 12 and returns yPos.

Screen1:

```
// ----- //
//   Screen 1   //
// ----- //
1] SUB Screen1:
2]   LOCAL x, y
3]   CLEARSCREEN
4]   DRAWRECT 0, 0, 320, 240, RGB(0, 0, 200)
5]   // LOADSPRITE "wall.bmp", 0
6]   LOADSPRITE "stethoscope.bmp", 2
7]   LOADSPRITE "doorkey.bmp", 3
8]   LOADSPRITE "safekey.bmp", 4

9]   LOADFONT "screen1toptextfont.bmp", 2
10]  SETFONT 2

11]  // draw top text
12]  PRINT "Stethoscope", 0, 0
13]  PRINT "Door Key", 120, 0
14]  PRINT "Safe Key", 200, 0
15]  // end draw top text

16]  // draw wall
17]  wy = 27
18]  wx = 3
19]  GETSPRITESIZE 0, sx, sy
20]  DRAWSPRITE 0, wx, wy
21]  FOR y = 0 TO 14
22]    DRAWSPRITE 0, wx, wy + sy
23]    DRAWSPRITE 0, 302, wy + sy
24]    wy = wy + sy
25]  NEXT

26]  FOR x = 0 TO 22
27]    DRAWSPRITE 0, wx + sx, 27
28]    DRAWSPRITE 0, wx + sx, 222
29]    wx = wx + sx
30]  NEXT
31]  // end draw wall

32]  // draw exit
33]  LOADFONT "screen1exitfont.bmp", 1
34]  SETFONT 1
35]  DRAWRECT 15, 40, 50, 20, RGB(0, 0, 0)
36]  PRINT "EXIT", 15, 40

37]  wx = 3
38]  FOR x = 0 TO 2
39]    DRAWSPRITE 0, wx + sx, 66
40]    wx = wx + sx
41]  NEXT

...

42] ENDSUB
```

I'm not going to post the full listing for this since most of it is the same, but here is a selection of that code. The two new commands used are LOCAL and GETSPRITESIZE. LOCAL defines two variables (x and y) that only “exist” within the Screen1 sub program. Once the program exits Screen1 the LOCAL variables of x and y will cease to exist! GETSPRITESIZE accepts a referring sprite number and will return the width and height of the sprite number into sx and sy. In order to draw the screens I used a FOR loop to draw lines of the wall sprite (13 x 13 pixels) in certain positions. There is a better way of implementing this and that is to use a tile map (its a lot easier with a lot less code). Luckily, I'm doing that for Brains!!! so once that is completed I will show you that method then.

HandleKey1:

```
// ----- //
//   handleKey1   //
// ----- //
1] SUB handleKey1:
2]   WHILE endlevel1 = FALSE
3]     GOSUB Screen1

4]       IF delay = 10
5]         delay = 0
6]         // up key
7]         IF KEY(200)
8]           // if collide with bottom exit wall
9]           IF BOXCOLL(playerx, playery, 13, 13, 20, 79, 30, 13)
10]          // if collide with right exit wall
11]          ELSEIF SPRCOLL(1, playerx, playery, 0, 68, 66)
12]          // if collide with main top wall

13]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 81, 30, 220,
13)

14]          // if collide with top extra wall
15]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 133, 53, 78,
13)
16]          // if collide with top upper L-shaped room walls
17]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 146, 92, 39,
13)
18]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 176, 105, 26,
13)
19]          // if collide with top lower L-shaped room walls
20]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 146, 121, 65,
13)
21]          // if collide with bottom upper L-shaped room walls
22]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 146, 157, 65,
13)
23]          // if collide with bottom lower L-shaped room walls
24]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 146, 196, 39,
13)
25]          ELSEIF BOXCOLL(playerx, playery, 13, 13, 183, 183, 26,
13)
26]          // if collide with extra walls (right side of L-shaped
rooms)
27]          ELSEIF SPRCOLL(1, playerx, playery, 0, 224, 196)
28]          ELSEIF SPRCOLL(1, playerx, playery, 0, 224, 157)
29]          ELSEIF SPRCOLL(1, playerx, playery, 0, 224, 132)
30]          ELSEIF SPRCOLL(1, playerx, playery, 0, 224, 106)
31]          ELSEIF SPRCOLL(1, playerx, playery, 0, 224, 41)
32]          // do nothing
33]          // if collide with stethoscope item
34]          ELSEIF SPRCOLL(1, playerx, playery, 2, 159, 105)
35]          playery = playery - 13
36]          item[0] = TRUE
37]          // if collide with safe key item
38]          ELSEIF SPRCOLL(1, playerx, playery, 4, safekeyXPos,
safekeyYPos + 13)
39]          playery = playery - 13
40]          item[2] = TRUE
41]          // Open exit door when all items are collected
42]          ELSEIF item[0] = FALSE AND item[1] = FALSE AND item[2] =
FALSE AND BOXCOLL(playerx, playery, 13, 13, 55, 66, 15, 2)
43]          // do nothing
44]          // if all items gotten
45]          ELSEIF item[0] = TRUE AND item[1] = TRUE AND item[2] =
TRUE AND BOXCOLL(playerx, playery, 13, 13, 55, 66, 15, 2)
46]          playery = playery - 13
47]          done1 = TRUE
48]          // if caught by torchlight
49]          ELSEIF testTorchLight = TRUE
50]          playery = playery - 13
51]          beam = torchLight(torchy)
52]          IF BOXCOLL(playerx, playery, 13, 13, torchx - 13,
```

```

torchy, -beam, 13)
53]                                     nicked = TRUE
54]                                     ENDIF
55]                                     ELSE
56]                                     playery = playery - 13
57]                                     ENDIF
                                     ENDIF

.....

58]                                     ELSE
59]                                     delay = delay + 1
60]                                     ENDIF

```

This handleKey1 SUB is actually cut in half, the first section deals with the handling of key presses in the game and the other half is for updating the screen. Again, for purposes of keeping the repetition out of the code, most of it has been cut out. In hindsight, I should have separated the key handling process from the screen drawing one but again, this was done quickly so that I could assess how GLBasic was used.

The entire SUB is encased in a WHILE loop (line 2) that checks whether endlevel1 is still false. The variable endlevel1 will not be true until certain conditions within the code are met. The WHILE loop is further encased in an IF..THEN..ELSE statement that delays the execution of the key presses within the program. For example, if we were to take away the IF...THEN...ELSE statement then pressing up would not gradually move the player up x number of spaces but would jump the player from one area of the screen to another.

Within the delayed IF..THEN..ELSE statement is a set of IF..THEN statements that checks for key presses with the KEY() function. There is only one argument this takes and that is the id of the key, it returns a 1 or 0 depending whether a key has been pressed (1) or not (0). You can find this id by using the Keycodes tool in the Tools menu (or by using the Alt+K shortcut).

Lines 9 – 31 deal with the sprite collisions between the player sprite and the walls, as you can see they check for collisions and if true no code is executed. This creates a boundary and prevents the player sprite from leaving the predetermined area or walking through walls. It is achieved by the use of two functions BOXCOLL and SPRCOLL. BOXCOLL accepts 8 parameters: X and Y position of the top left corner of the first box followed by the width and height of the same first box and X and Y position of the top left corner of the second box followed by its width and height. BOXCOLL uses these parameters to check if the first box collides with the second box, if so then the function returns true. SPRCOLL accepts 6 parameters: the id of sprite A, the X and Y position of sprite A, the id of sprite B and X and Y positions of sprite B. This function tests whether sprite A overlaps with sprite B and returns true if this is the case.

Lines 34 and 38 deal with sprite collisions between the player and the pickupable items (the stethoscope and the safe key). If the collisions return true, then we set item[x] to be true and move the character in the appropriate direction (`playery = playery - 13`). We won't deal with the visual implications of this (the picking up of the items) until the next part of this section.

Lines 42 and 45 set the conditions the player must achieve in order to exit the level. If no items have been picked up, i.e. item[x] is all false, then the player cannot exit the area, however, if item[x] is all true then the player can move into the exit area and `done1` can be set to true.

The last ELSE..IF statement handles the collision between the player and the torch beam. First, it moves the player to the applied direction and then gets the width of the beam via the torch Y position through the use of the function and parameter `torchLight(torchy)`. Finally, it checks if the player collides with the beam with BOXCOLL and sets `nicked` to true. You will notice in the

parameters of BOXCOLL that I've used `-beam`, this is because the box for that beam is drawn backwards (from right to left) since the torch is pointing in the opposing direction from what is drawn on the screen.

Lastly, if none of the other conditions are met the player is free to move in the desired direction.

```

61]         IF screenDelay = 10
62]             screenDelay = 0
63]             DRAWSPRITE 1, playerx, playery

64]         IF nicked = TRUE
65]             LOADSPRITE "robber2.bmp", 7
66]             DRAWSPRITE 5, torchx, torchy
67]             DRAWRECT torchx, torchy, -beam, 13, RGB(255, 255, 255)
68]             DRAWSPRITE 7, playerx, playery
69]             DRAWRECT 100, 105, 120, 30, RGB(0, 0, 0)
70]             SETFONT 2
71]             PRINT "You're nicked!!", 100, 105
72]             PRINT "Press a key", 115, 120
73]             SHOWSCREEN
74]             endlvl1 = TRUE
75]             KEYWAIT
76]             GOSUB initLvl1
77]         ELSEIF done1 = TRUE
78]             DRAWSPRITE 5, torchx, torchy
79]             DRAWRECT torchx, torchy, -beam, 13, RGB(255, 255, 255)
80]             DRAWSPRITE 7, playerx, playery
81]             DRAWRECT 100, 105, 90, 15, RGB(0, 0, 0)
82]             SETFONT 2
83]             PRINT "Well Done!!", 100, 105
84]             SHOWSCREEN
85]             endlvl1 = TRUE
86]             levelState = 2
87]             KEYWAIT
88]             GOSUB initLvl2
89]         ELSE
90]             GOSUB TorchAI
91]             IF item[0] = TRUE
92]                 DRAWSPRITE 2, 40, 11
93]             ELSE
94]                 DRAWSPRITE 2, 159, 92
95]             ENDIF

96]             IF item[1] = TRUE
97]                 DRAWSPRITE 3, 150, 11
98]             ELSE
99]                 DRAWSPRITE 3, 159, 170
100]            ENDIF

101]            IF item[2] = TRUE
102]                LOADSPRITE "safekey1.bmp", 6
103]                DRAWSPRITE 6, 225, 11
104]            ELSE
105]                DRAWSPRITE 4, safekeyXPos, safekeyYPos
106]            ENDIF

107]            SHOWSCREEN
108]        ENDIF
109]    ELSE
110]        screenDelay = screenDelay + 1
111]    ENDIF

112] WEND
113] ENDSUB

```

The screen update section is encased in another `IF..THEN..ELSE` statement that delays the program from updating the screen and is the same as the `handleKey1` delay condition. Within that the player

sprite is drawn on the screen with the updated `playerx` and `playery` variables and is followed by more IF..THEN..ELSE statements.

The first condition (lines 64 – 76) displays what happens when `nicked` becomes true, in this a new sprite is loaded (the “death” sprite) and is drawn onto the screen along with the torch and the torch beam. On top of this is a black rectangle with text on and the `endlevel1` variable is set to true. KEYWAIT is used to pause until the user has pressed a key and the game resets.

The second condition (lines 77 – 88) is the same as the first condition with the exception that it doesn't load the “death” player sprite and it sets the `levelState` variable to the next level, before going to the initialisation SUB for that level. The last condition updates the artificial intelligence of the torch and draws the positions of the sprites for `item[x]`, depending on whether the item has been picked up or not. If the item is picked up then the sprite's position changes from it's set position on the level to the collected items area at the top of the screen.

Note: Re-examining the code you can actually get the same delay result if you deleted/commented out lines 4, 5, 58, 59, 60, 61, 62, 109, 110 & 111 and used LIMITFPS 10 inbetween lines 3 and 4.

torchLight():

```
1] FUNCTION torchLight: y
2]     SELECT y
3]         CASE 53
4]             beamWidth = 208
5]         CASE 66
6]             beamWidth = 234
7]         CASE 105
8]             beamWidth = 130
9]         CASE 157
10]            beamWidth = 130
11]        CASE 131
12]            beamWidth = 273
13]        CASE 196
14]            beamWidth = 273
15]        DEFAULT
16]            beamWidth = 52
17]    ENDSELECT
18]    RETURN beamWidth
19] ENDFUNCTION
```

The torchlight function takes in a parameter (`y`) and uses it to set a number for `beamWidth` and returns it. This is to give the torch a set beam width so that it cannot pass through walls. Typically, the `y` parameter would be the global variable `torchy`.

TorchAI:

```
// ----- //
// TorchAI //
// ----- //

1] SUB TorchAI:
2]   LOADSPRITE "torch.bmp", 5

3]   IF torchy <= 40 AND up
4]     GOSUB drawTorchDown
5]     up = FALSE
6]     down = TRUE
7]   ELSEIF torchy >= 209 AND down
8]     GOSUB drawTorchUp
9]     up = TRUE
10]    down = FALSE
11]  ELSEIF torchy <= 209 AND up
12]    GOSUB drawTorchUp
13]    up = TRUE
14]    down = FALSE
15]  ELSEIF torchy >= 40 AND down
16]    GOSUB drawTorchDown
17]    up = FALSE
18]    down = TRUE
19]  ENDIF

20]  IF AIUpDelay = 5
21]    up = RND(1)
22]    AIUpDelay = 0
23]  ELSE
24]    AIUpDelay = AIUpDelay + 1
25]    beam = torchLight(torchy)
26]    DRAWRECT 289, torchy, -beam, 13, RGB(255, 255, 255)
27]    DRAWSprite 5, torchx, torchy
28]  ENDIF

29]  IF AIDownDelay = 5
30]    down = RND(1)
31]    AIDownDelay = 0
32]  ELSE
33]    AIDownDelay = AIDownDelay + 1
34]    beam = torchLight(torchy)
35]    DRAWRECT 289, torchy, -beam, 13, RGB(255, 255, 255)
36]    DRAWSprite 5, torchx, torchy
37]  ENDIF

38] ENDSUB
```

Here, for most of you, is the most interesting part of the program – the artificial intelligence (AI). Coding AI can be an absolute nightmare depending on how complex you want your enemy sprites or (Non Player Characters) NPCs to behave, but then again, it can be quite simple. In this particular instance all I want the torch to do is to move up and down in a random fashion, enough so that the AI is trying to simulate searching for a player.

The torch sprite is loaded into memory (line 2), then it is tested to see if the torch has come to the edge of the level and which direction it is going (lines 3 – 19). If so, the torch can be reversed by drawing it going the opposite direction and swapping the direction boolean variables. The “searching” part of the AI can now be looked at by delaying the time or number of moves it takes to change direction by counting to a certain number and randomly changing the direction boolean variable followed by a redraw.

Note: The way I implemented the AI makes for an interesting “bug” in that there are times that the torch appears to stop for a moment. This occurs when both `up` and `down` variables contain 1 or 0, since there is no condition to account for this, the torch is drawn on the screen anyway but isn't moved. I could have rectified this but I decided to leave it as it makes the torch seem more realistic and gives an uncertain challenging edge for the player.

`drawTorchUp:`

```
// ----- //
// drawTorchUp //
// ----- //
1] SUB drawTorchUp:
2]   torchy = torchy - 13
3]   beam = torchLight(torchy)
4]   DRAWRECT 289, torchy, -beam, 13, RGB(255, 255, 255)
5]   DRAWSPRITE 5, torchx, torchy
6] ENDSUB
```

The `drawTorchUp` and `drawTorchDown` SUB just exist to set the direction of the torch movement and draw it onto the screen. The only difference between the two is that line 2 changes the minus to a plus.