

Analysis of a five liner game written in Basic v2.0 for the Commodore Vic-20

Footnote: Apologies in advance if you know all this stuff already and you are an old hand at Basic 10-liners.

UFO rush is game written in five lines of Basic for the unexpanded Commodore Vic-20. The game has been written by Jeffrey Daniels (Jeff-20 on the Commodore Vic Denial forum) and is featured in his forthcoming book 'The Vic-20 Quick Reference Handbook' on page 121. What I think makes the game significant is its high compactness and the functionality it delivers for that compactness. The core game loop is only two lines of Basic code.

Although none of the techniques used in the program are particularly new, what is new (to me at least) is the way all of these techniques have been combined to make a very efficient program.

So first let's run the game. I have made some superficial changes to the code to allow it to run in Vic emulators without having to fiddle with it.

1. I changed special characters to chr\$ codes for easy copying and pasting.
2. I removed color special characters to make the code readable. (I'm sorry it's less colorful now)
3. I changed the steering keys from the commodore cursor keys to N and M (left and right respectively). Once again it makes it easier for someone using a Vic-20 emulator.
4. I changed the basic keywords to their abbreviations.

(My apologies for mangling your program a bit Jeffrey)

So now go and run the code. Copy and paste it into your favorite Vic emulator and then type run and press the return key. What no screenshot? Yes no screenshot, go run the game.

```
-----  
1 v=36876:pOv+3,8:pOv+1,0:?cH(147):?"s:"a,"ufo rush":a=a+1:c=8196  
2 fOt=0to17:?""spc(rN(1)*28):nE:pOv+2,9  
3 i=pE(197):b=c-22+(i=28)-(i=36):d=pE(b):pOb,65:pOc,32:ifc<7746then1  
4 c=b:fOt=0to40-a*2:nE:ifd=32then3  
5 pOv+1,220:fOi=1to15step.1:pOv+2,15-i:pOv+3,8+(iaN1):nE:wA197,64,64:rU  
-----
```

So why does the program above not look like familiar Basic syntax? Well as mentioned before its using abbreviations. Below is the un-abbreviated code so that one can get a better understanding of what's happening:

```
-----  
1 v=36876:poke v+3,8:poke v+1,0:print chr$(147):print "s:"a,"ufo rush":a=a+1:c=8196  
2 for t=0 to 17:print "***"spc(rnd(1)*28):next:poke v+2,9  
3 i=peek(197):b=c-22+(i=28)-(i=36):d=peek(b):poke b,65:poke c,32:if c<7746 then 1  
4 c=b:for t=0 to 40-a*2:next:if d=32 then 3  
5 pokev+1,220:fori=1to15step.1:pokev+2,15-i:pokev+3,8+(iand1):next:wait197,64,64:run  
-----
```

Ok now that's much better. I have analyzed each line of code below and for this analysis I referenced two books.

1. The VIC-20 Programmers reference guide (Published by: Commodore Business Machines)
2. Mapping the VIC (Published by: Compute! Books)

Both are available online in PDF format and are obtainable from your local internet VIC historical book archive.

LINE: 1 v=36876:poke v+3,8:poke v+1,0:print chr\$(147):print"s: "a,"ufo rush":a=a+1:c=8196

v=36876 - The Video Interface Chip (VIC) addresses starts at 36864, so why is v defined as 36876? It's efficient to pick an address close to the registers that one is going to use.

poke v+3,8 - Turn the screen and border color to black. See the Programmers reference guide page 265 for the color combination codes.

poke v+1,0 - The fourth sound oscillator which is the white noise oscillator is silenced (Any value below 128 is off)

print chr\$(147) - Clear the screen. One can also use print "{shift home}"

print"s: "a,"ufo rush" - Print the score and game name. Variable 'a' is printed and initializes to 0 at the start (a default for numeric variables unless they're specifically assigned a value) so that's just dandy.

a=a+1 - add one to the score variable, but the score has already been displayed, so that's for the next round.

`c=8196` - Define c at 10 bytes past the end of screen memory at 8186. 'c' is used for the old location of the players spaceship.

LINE: 2 `for t=0 to 17: print "***"spc(rnd(1)*28):next:poke v+2,9`

`for t=0 to 17: print "***"spc(rnd(1)*28):next` - Populate a starfield by iterating 18 times with a random amount of up to 27 spaces and then print a double star. Because the addition of successive random number amounts of spaces within bounds will form a normal distribution, this means that about half of the screen at the bottom should remain empty. This is neat because it allows the player to get their bearings before they encounter the starfield.

`poke v+2,9` - Set the global sound volume to 9 of 15

LINE: 3 `i=peek(197):b=c-22+(i=28)-(i=36):d=pE(b):pOb,65:pOc,32:ifc<7746then1`

`i=peek(197)` - 'Mapping the Vic' on page 58 says that location 197 in Vic memory is - 'The matrix coordinate of the last key pressed. A value of 64 is returned if a key is not pressed.' Assign the value of a keypress to variable 'i'.

Here's a little program so that you can obtain the matrix coordinate value of a key press:

`1 wait 197,64,64`

`2 a=peek(197):print a:goto 1`

`b=c-22+(i=28)-(i=36)`

- This the meat and potatoes of the game.
- 'b' is the new location of the players spaceship
- 'c' is the old (start) location of the player spaceship - c starts at one line below the end of screen memory
- 'b' firstly gets a -22 (one screen row) - so the spaceship is moving upward
- Here's comes a very cool bit (Two logical 'if' statements are embedded in the formula)
- (i=28) - if 'i' (which is the value of the key press) is 28 ('n') then add one position to the spaceship position
- (i=36) - if 'i' (which is the value of the key press) is 36 ('m') then subtract one position from the spaceship position
- wait a minute, shouldn't that be the other way around. n subtracted and m added?
- Well that has to do with the logic result of the 'if' statement. A 'FALSE' is 0, but a 'TRUE' is -1
- An example: n is pressed thus $b=c-22+(-1)-(0)$

`d=pE(b)` - d gets loaded with whatever is in the ships new position. This is used for collision detection.

`pOb,65:pOc,32` - erase the old character (32 is a space) and draw the spaceship in the new position (65 is a spade)

ifc<7746then1 - if the spaceship has reached the top of the screen then loop back to 1. Remember 'a' which had a one added to it? Well now the score update will reflect that and the whole update process starts again with the drawing of a new starfield, the reset of the spaceship position and so on. If the spaceship is not at the top of the screen then proceed to line 4.

LINE: 4 c=b:for t=0 to 40-a*2:next:if d=32 then 3

c=b - The variable 'c' (the old spaceship position) is now loaded with the latest (new) spaceship position

for t=0 to 40-a*2:next - Generate a delay but reduce the delay by two steps for every point that has been scored. In other words the ship is moving faster after every new screen.

if d=32 then 3 - if the new position ('d') is an empty space (32) then loop back to beginning of the core code (core code is: line 3&4) otherwise the spaceship has crashed

LINE: 5 poke v+1,220: for i=1 to 15 step.1: poke v+2,15-i: poke v+3,8+(iand1): next: wait 197,64,64: run

poke v+1,220 - load the noise oscillator with a pitch of 220

for i=1 to 15 step.1: poke v+2,15-i: poke v+3,8+(iand1): next

- Crash sound and border flicker

- v+2 is the global volume parameter. Its decreasing from 15 to 1 at a constant noise pitch of 220

- The step value of i is only 0.1, so that means that the volume will only decrease by one every 10th iteration. Basic in its wisdom only passes the integer value to the v+2 location. Basic V2 is nifty.

- **poke v+3,8+(iand1)** - v+3 is the screen/border color combination, 8 is black screen with border but if one adds the AND of i with 1, it will alternate between 8+0 and 8+1.

To illustrate:

09 (1001 in binary) AND 1 = 1

10 (1010 in binary) AND 1 = 0

11 (1011 in binary) AND 1 = 1

wait 197,64,64: run - Monitor memory location 197 for a change. It checks the keyboard matrix for 64 (no key press) through the first parameter which ANDs (filters) 64. The second parameter is an XOR which means the moment a key is pressed the 64 bit is flipped, the status changes and the wait is over (See P34 of the programmers reference guide). If a key is pressed then restart the program through a RUN. A run resets all variables back to empty and will even RESTORE dimensional array pointers. It's much better than using a GOTO seeing as one gains full re-initialization with the run statement.

And so that's it.

Except for a nifty compact piece of code, what's the use case?

For myself I used the game as a basis for my own Basic 10-liner games.

In 2011 a contest was started called the Basic 10-Liner contest. Link here:

<https://gkanold.wixsite.com/homeputerium>

It's been running every year and the idea is to produce a game for an 8-bit platform which has a Basic interpreter and to write it in 10 lines or less. There are a couple of categories but the one I'm most interested in is the classic PUR-80 category.

The rules for Category "PUR-80":

- Program a game in 10 lines (max. 80 characters per logical line, ABBREVIATIONS ARE ALLOWED)
- Only BASIC variants that were originally installed in the computer system are permitted in this category.
- No reloading of data or program parts
- The 10 lines must not contain any machine programs
- The program may be compiled
- All code must be visible in the listing: self-modifying code or hidden initializations are not allowed
- POKEn in memory locations is allowed
- The program must be submitted on a diskette image or tape image
- The program must be listable
- Along with the program, a text file with the program description and instructions should be submitted, including a brief description of how to start using the emulator, a screenshot in jpg or png format or an animated screenshot in gif format should also be attached, and you should also enclose proof of this showing that the program has no more characters than allowed in the category
- There can be up to 0.5 bonus points in the rating for program descriptions and code explanations

Personal quick notes and tips:

- DATA statements can be shoved in anywhere (and I mean anywhere). The problem is a READ statement which if it gets to execute again without a RESTORE OR RUN will throw an OUT OF DATA error. So put your READ statements high up (preferably line 1) and your DATA statement in any hole you can find.

- 'if' statements are a headache. They have to be the last statement on a line because nothing is processed after the 'if' except when the 'if' is TRUE. One can have it do multiple things if a condition is found to be TRUE. Example:

```
if a=1 then b=2:c=3:d=4:goto2
```

What the above line does is it checks if a=1

If a=1 it processes the then statement b=2 followed by the rest c=3,d=4 and then GOTOs to line 2

If 'a' does not equal '1' it goes to the next line and ignores all the statements after the if!

So that is why line 3 in the game is so cool to me. You're gaining logical if conditional logic without the headache.

Anyway if time and opportunity will allow I'll have a submission ready for 2023. Look forward to your entry.

Greetings

Huffelduff

(user handle on the Vic-20 Denial forum)