A Full Explanation of the *Petscii Jetski* Code

Nick Montfort[1] *and Jesper Juul*[2]

March 2020 TROPE–20-01

Abstract

We developed a short BASIC game for the Commodore 64, *Petscii Jetski,* in which the player controls a jetski that can be moved left or right along the top edge of the screen. The player must traverse the screen, avoiding any characters that are not letters, to score points. This program was developed for the BASIC 10 Liner Contest and is a visual poem as well as a videogame. We explain the concept and the general coding challenges we faced. We go through the program statement by statement, explaining each one so that even those new to programming (if they are sufficiently patient) can understand how the program works. For those familiar with Commodore 64 BASIC and curious about the optimizations we made, we discuss those as well.

A technical report from

The Trope Tank
NYC, 126 E 12th St & MIT, 14N-336
http://nickm.com/trope_tank/

1 nickm.com
2 jesperjuul.net

Concept

Petscii Jetski is a game in which the player controls a jetski that can be moved left or right along the top edge of the screen. Points are scored by first hitting the left edge, then the right, and then continuing to alternate between the two edges of the screen. The jetski can pass through space and through letters, but the game ends if it collides with any other characters. We consider *Petscii Jetski* to be a visual poem as well as a videogame—see, for instance, *Arteroids* by Jim Andrews.

This program was developed in CBM BASIC version 2 (the version of BASIC built into the Commodore 64) for the BASIC 10 Liner Contest, and specifically for the PUR-80 category that only allows 10 lines of 80 characters. Any 8-bit system could be used in this contest; we chose the Commodore 64 because of our experience with and love of this platform. Montfort has done extensive small-scale creative work on the platform[3,4,5] and organized and co-authored a book about a Commodore 64 BASIC program.[6] Juul has been programming on the platform since the early days of the demoscene. The Commodore 64 can also produce good visual effects using only text mode and the PETSCII character set, which is unique to Commodore computers.

We sought to create a game that is playable, replayable, and indeed exhilarating.

We also wanted to require the player to "read" what is being displayed on the screen. The player of *Petscii Jetski* has to distinguish alphabetical characters from other characters, some of which are similar: the zero is not a letter 'O' and the cross symbol is not a letter 'X'. Although the game doesn't present any words, except by chance, it does require reading on this low level.

High-Level Organization of the Code

The 10-line program has lines numbered 0 through 9, and consists of:

Initialization in lines 0 and 6–7.

A *main loop* that occupies lines 1–5.

End-of-game code in line 8–9.

3 Montfort, Nick, "Lines of Commodore 64 Poetry," *Convolution* 5-7, pp. 201–212, 2019.
4 Montfort, Nick. "Three Commodore 64 Poems," *Notre Dame Review* 48, pp. 153–160, Summer/Fall 2019.
5 Montfort, Nick, "Tiny Commodore 64 Artworks," *Revista CIA* No. 6, pp. 26–27, 90–101, trans. Milton Läufer, October 17, 2017.
6 Montfort, Nick, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter. *10 PRINT CHR$(205.5+RND(1)); : GOTO 10.* MIT Press, Cambridge, Massachusetts, 2013.

Coding Challenges

There are two main challenges in writing a BASIC game in 10 lines of at most 80 characters: space and time.

A 10-liner needs to be very compact compared to typical BASIC games from the 1970s and 1980s. For instance, the version of the famous simulation game *Hammurabi* that appears in *BASIC Computers Games,* ed. David H. Ahl, 1978, is 119 lines long! Although the *Hammurabi* code could be considerably compressed, it is nevertheless an order of magnitude larger than a 10-line program.

For an action/arcade game such as *Petscii Jetski,* time is an additional challenge. *Hammurabi* is turn-based. A real-time game such as ours must contend with the speed of BASIC, an interpreted language in which even seemingly straightforward arithmetic can be painfully slow. Updating the display in visually interesting ways can also take a lot of time.

To deal with the challenge of space, we made design decisions that could be realized in compact runs of code. To deal with how much time it takes to update the display in BASIC, we developed a vertical scroller, taking advantage of a fast way to put new characters on the screen and keep them moving.

The next section explains how each section of code essentially functions, without going into optimization details. After this, we explain some of the space and time optimizations we made.

Detailed Code Explanation

For purposes of clarity, each line will be shown in this section essentially as the LIST command presents it, rather than how it was typed in. While each line of our program can be typed in using at most 80 keystrokes, the (cryptic) keyword abbreviations we use mean that lines expand to more than 80 characters when listed. A very small number of additional changes have been made below to each line to make the code even more clear: spaces are added, and '0' (zero) is used instead of '.' in cases where we used '.' to stand for zero.

Initialization

```
0 DIM L$,I,A(2) : P=RND(-TI) : S=40740 : F=54287 : C=56215 : A(1)=-1
  : A(2)=-1 : Q=40 : T=0 : W=0 : GOTO 6
6 POKE 646,5 : FOR P=0 TO 9 : POKE 53281-P/5,0 : L$=L$+"    " :
  PRINT SPC(120) : NEXT : O=1024 : R=O+Q : N=R+Q
7 POKE F,80 : POKE F+4,213 : POKE F+5,202 : POKE F+9,15 : POKE
  F+3,129 : H=0 : X=32: Y=26: TI$="000000" : GOTO 1
```

```
DIM L$,I,A(2)
```

Declares the string variable L$, numerical variables I and P, and array A, which has 3 elements, 0 through 2.

```
P=RND(-TI)
```

Seeds the (pusedo-)random number generator based on how many 1/60s of a second have elapsed since the computer was turned on, or since the game was last started, because the game resets the timer. Because RND(1) actually returns the same, deterministic sequence of numbers for any given seed value, if the random number generator is not seeded in this way, the way streams of characters appear will be exactly the same every time the computer is turned on and the game is played.

Because you can launch the game in an emulator (such as VICE) immediately after starting the emulator, it is possible to create this deterministic behavior. VICE, however, adds a random delay by default to avoid this.

Something has be done with this call to RND; it isn't valid BASIC for it to simply appear as a statement by itself. Here the value is assigned to P, so the variable P is declared (which is desirable, as described in the optimization section) and initialized. P (used for the position of the jetski) will get assigned a different value later on, at the conclusion of a FOR loop, so that the jetski always starts in the same place.

```
S=40740 : F=54287 : C=56215
```

These are used as constants; see the optimization section for an explanation of why they are declared.

```
A(1)=-1 : A(2)=1
```

Initializes the array A, used to update the position of the jetski based on input. A(1) is used to move the jetski left (displacement -1) when 1 is pressed, and A(2) is used to move the jetski right (displacement 1). This array is also effectively a constant; that is, it isn't updated during gameplay.

```
Q=40
```

Another constant.

```
T=0
```

Sets the current "target" location which the jetski is supposed to reach as the left edge, position 0.

```
W=0
```

The initial score. The score can increase until it is 10, at which point the player wins.

```
POKE 646,5
```

Changes the text color to green.

```
FOR P=0 TO 9 : POKE 53281-P/5,0 : L$=L$+"    " : PRINT SPC(120) :
  NEXT
```

A loop that happens 10 times. The POKE statement changes first the screen color, then the border color, to black. The next statement constructs a 40-character-long string, all spaces, that is stored in L$. Finally, the PRINT statement moves the cursor ahead 120 spaces (three 40-column lines) and so scrolls the display up three lines at a time—30 lines in total. Because there are only 25 lines on the screen, this removes any text that is there to begin with.

```
O=1024 : R=O+Q : N=R+Q
```

More constant declarations. The "origin" of the screen, memory address 1024, is indicated with the letter O (not zero). The start of the "row" that the player is on is 40 characters later, and stored in R. The start of the "next" row, used to determine if there has been a collision, is 40 characters after that, placed in N.

```
POKE F,80 : POKE F+4,213 : POKE F+5,202 : POKE F+9,15 : POKE F+3,129
```

Initializes the sound. Sets frequency, attack/decay, and sustain/release for voice 3. Then, turns the master volume up all the way. Finally, sets the noise waveform for voice 3 and turns on the gate so sound is produced. While this is just "noise," it can—and will—vary in frequency during the game. The frequency change will happen rhythmically, at the same pace at which characters are progressing up the screen. It will also be responsive to the jetski's movements.

```
H=0
```

The initial value of the frequency shift. This flips between 0 and 40 each pass though the main loop, causing the frequency of the noise to pulse up and down.

```
X=32 : Y=26
```

More constants, to check for the space (character code 32) and to check to see if a character is not a letter.

```
TI$="000000"
```

Resets the timer. This way when the game is over, it's easy to report how much time has elapsed.

The program jumps to begin the main loop for the first time after this.

Main Loop

```
1 POKE 1024+T,W+48 : POKE S+RND(1)*Q,32 : POKE S+RND(1)*Q,32 : POKE
  C+RND(1)*Q,7 : POKE C+RND(1)*Q,7
2 FOR I=0 TO 4 : POKE C+RND(1)*Q,13 : POKE C+RND(1)*Q,1 : NEXT : IF
  RND(1)<.4 THEN POKE S+RND(1)*Q,Q+RND(1)*83
3 H=(H=0)*-Q : IF P=T THEN POKE 53280,5 : T=(T=0)*-39 : W=W+1 : POKE
  53280,0 : IF W>9 GOTO 8
4 P=P+A(PEEK(203)) : P=P-(P<0)+(P=Q) : POKE R+P,160 : I=PEEK(N+P) :
  IF I<>32 THEN IF I=0 OR I>26 GOTO 8
5 FOR I=-W/2 TO 0: PRINT L$; : NEXT : POKE F,P*4+H : GOTO 1
```

```
POKE 1024+T,W+48
```

Display the digit representing the score in either the upper left or upper right corner of the screen depending upon whether the current "target" is the left or right edge. The "+T" causes the score to be in either the upper left (if T is 0) or upper right (if T is 39). The "+48" yields the correct screen code: the digits 0–9 have screen codes 48–57.

```
POKE S+RND(1)*Q,32 : POKE S+RND(1)*Q,32
```

Set two locations in the "line," the string variable L$, to be spaces, at random. Doing so

may have no effect, because it may write a space into a location that already has a space.

```
POKE C+RND(1)*Q,7 : POKE C+RND(1)*Q,7
```

Since there is a bit of room left in this line, set two positions on the bottom-to-last row to be yellow. Such color changes will only be visible if there happen to be characters (rather than spaces) in these positions, but they make the game/poem more visually interesting and also may provide a bit more challenge, since they could distract the player from playing/reading.

```
FOR I=0 TO 4 : POKE C+RND(1)*Q,13 : POKE C+RND(1)*Q,1 : NEXT
```

More color changes. This loop happens 5 times. Each time, it sets one position in the next-to-bottom row to be light green and one to be white, both at random.

```
IF RND(1)<.4 THEN POKE S+RND(1)*Q,Q+RND(1)*83
```

40% of the time this will place a character—other than a space—at a random location in the "line" L$.

```
H=(H=0)*-Q
```

Flips the frequency shift to be 0 (if it is 40) or to be 40 (if it is 0). Q is just our shorthand for 40. The expression H=0 has the value -1 if H is indeed zero; "true" on the Commodore 64 is represented as -1. Therefore if H is zero, the value of H will be updated to (-1) times -40, which is 40. If H has another value (it will be 40, in this particular case), H=0 will have the value "false" which is 0. In that case, H will be updated to 0 times 40, which is 0.

```
IF P=T THEN POKE 53280,5 : T=(T=0)*-39 : W=W+1 : POKE 53280,0
```

If the jetski has reached the target, first change the border to green, then flip the target to the other side (0 if 39, 39 if 0), then increase the score by one, and change the border back to black, so that the green border simply flickers for a moment. But the line is not done...

```
IF W>9 GOTO 8
```

At this point, with the score having just increased, it may have reached 10. If that's the case, the player has won, so jump to the end-of-game code. Otherwise, continue to...

```
GET I : P=P+A(I)
```

Update the position of the jetski, taking into account the current input. GET retrieves an integer if the player has pressed a key corresponding to 0–9. (Only 1 and 2 are valid keys to press!) Then, the update of P moves the jetski left or right as appropriate.

```
P=P-(P<0)+(P=Q)
```

Update the position of the jetski if it now exceeds the far left (-1, detected with <0) or far right (40, detected with P=Q) limit. This update adds one or subtracts one as appropriate, keeping the jetski on the current line.

```
POKE R+P,160
```

Draw a solid square (the "blip" of the jetski) in the appropriate location, on the second row of the screen. 160 is just the screen code for this square, also known as a reversed space. Upward scrolling causes a trace of this square to appear above it.

After this, jump back to the top of the main loop.

```
I=PEEK(N+P) : IF I<>32 THEN IF I=0 OR I>26 GOTO 8
```

If there is a character immediately underneath the jetski, and it is not a space (32) or a letter (1 through 26), there has been a collision. Jump to the end-of-game code as the player has lost.

```
FOR I=-W/2 TO 0 : PRINT L$; : NEXT
```

This loop happens a different number of times depending upon the score: once if the score is 0 or 1, twice if it is 2 or 3, and so on, up to 5 times when the score is 8 or 9. Each time through the loop, the line is printed. Printing the line more times at this point causes the progression of characters up the screen to occur more rapdily.

```
POKE F,P*4+H
```

Changes the frequency of voice 3, which is producing noise. These changes are generally audible, although they become harder to hear at high frequencies—so high frequencies are not used. The frequency changes in two ways: Each time through the loop, it is higher or lower because of the changing value of H, which flips between 0 and 40. Also, because the frequency depends on P, it becomes lower when the jetski moves left and higher when it moves right.

End-of-Game Code

```
8 PRINT : IF W<10 THEN C=PEEK(46)*256+PEEK(45)+3 : POKE C+1,8 : FOR
  I=0 TO 22 : POKE C,RND(1)*256 : PRINT L$; : NEXT : POKE 646,1
9 POKE F+3,128 : POKE 198,0 : PRINT : PRINT "↓TIME"TI,"SCORE"W : IF
  W>9 THEN POKE 53281,13 : PRINT "↓YOU WON!"
```

```
PRINT
```

Scrolls up by one (blank) line.

```
IF W<10 THEN C=PEEK(46)*256+PEEK(45)+3
```

The rest of this line happens only if the player lost, ending the game with a score of less than 10. What this code does, essentially, is put random-like or garbage-like characters into the "line" L$ and then display that line 20 lines, creating a sort of textual "explosion." The first thing that happens is the reuse of the variable C, which becomes a pointer L$'s string data pointer.

```
POKE C+1,8
```

Set the high byte (most significant byte) of the string's data pointer. We are going to point the string into the area where the BASIC program itself is stored in memory.

```
FOR I=0 TO 22 : POKE C,RND(1)*256 : PRINT L$; : NEXT
```

This loop happens 23 times. Each time, set the low byte (least significant byte) of the string's data pointer to a random value between 0 and 255. Now the string consists of 40 characters of the BASIC program. It is displayed. The characters in the string are not tidy characters of the sort that appear in a program listing. They are the underlying representation of the BASIC program in memory, which ends up looking quite messy.

```
POKE 646,1
```

Change the character color to white. This also only happens if the player has lost.

```
POKE F+3,128
```

Closes the gate for voice 3, so that the sound is smoothly tuned off.

```
POKE 198,0
```

Clears the keyboard buffer. Otherwise whatever keypresses have not yet been processed would come through after the READY. prompt is printed, which would interfere with whatever the user wants to do next on the command line.

```
PRINT : PRINT "↓TIME"TI,"SCORE"W
```

Displays the total elapsed time (in 1/60s of a second) and the score.

```
IF W>9 THEN POKE 53281,13 : PRINT "↓YOU WON!"
```

If the player got a score of 10, change the screen color to light green in celebration and display a message. Congratulations! They won!

Optimizing the Code

We made several sorts of optimizations, explained below. In this section, we discuss the code as it was originally typed in:

```
0dIl$,i,a(2):p=rN(-ti):s=40740:f=54287:c=56215:a(1)=-1:a(2)=1:q=40:t=.:w=.:gO6
1pOo+t,w+48:pOs+rN(1)*q,32:pOs+rN(1)*q,32:pOc+rN(1)*q,7:pOc+rN(1)*q,7
2fOi=.to4:pOc+rN(1)*q,13:pOc+rN(1)*q,1:nE:ifrN(1)<.4tHpOs+rN(1)*q,q+rN(1)*83
3h=(h=.)*-q:ifp=ttHpO53280,5:t=(t=.)*-39:w=w+1:pO53280,0:ifw>9gO8
4gEi:p=p+a(i):p=p-(p<0)+(p=q):pOr+p,160:i=pE(n+p):ifi<>32tHifi=.ori>26gO8
5fOi=-w/2to.:?l$;:nE:pOf,p*4+h:gO1
6pO646,5:fOp=.to9:pO53281-p/5,.:l$=l$+"    ":?sP120):nE:o=1024:r=o+q:n=r+q
7pOf,80:pOf+4,213:pOf+5,202:pOf+9,15:pOf+3,129:h=.:x=32:y=26:ti$="000000":gO1
8?:ifw<10tHc=pE(46)*256+pE(45)+3:pOc+1,8:fOi=.to22:pOc,rN(1)*256:?l$;:nE:pO646,1
9pOf+3,128:pO198,0:?:?"↓time"ti,"score"w:ifw>9tHpO53281,13:?"↓you won!"
```

In this text, a lowercase letter means this letter was typed in without SHIFT held down. An uppercase letter was typed in with SHIFT held down.

We discuss categories of optimizations with examples, rather than going through the code statement-by-statement as in the previous section.

Keyword Abbreviations

The Commodore 64 allows BASIC programmers to use keyword abbreviations. For instance instead of typing in “dim” for the dimension statement, one can enter “dI” (unshifted D, shifted I). BASIC tokens are stored using a single byte no matter what, but keyword abbreviations allow programmers to pack more code into a line. We used all available keyword abbreviations.

No Spaces

Spaces are optional in BASIC unless they occur in string literals. Adding such unnecessary spaces takes up more room and can't help speed, so we did not use any.

Line Numbers 0–9

To maximize the amount of space available for code, we used line numbers 0–9 rather than, for instance, 10, 20, 30 ... 100. Line numbers occupy space at the beginning of each line and also need to be referred to in GOTO statements.

Declaration of Variables

The first statement is a DIM statement declaring L$, I, and P and dimensioning the array A to have 3 elements. It is not needed, but it allows the most frequently-used variables (L$ and I) to be placed at the beginning of variable memory where BASIC can find them more rapidly. The array A could simply have the default dimension of 11, but since we wish to use only 3 elements we have declared that explicitly here. After this DIM statement, variables are declared and assigned values based on how frequently they are used in the main part of the main loop.

By the way, the array a gets moved up in memory every time another variable is declared/assigned later in the initialization process. This takes time. That's fine, because we are not trying to optimize the speed of initialization. We care about the speed of the main loop, and by the time we enter the main loop, we are finished declaring variables and the array a has moved to its permanent location.

Use of Variables for Constant Values

A statement such as `s=40740` is also not necessary, but can have benefits both in terms of space and time. When a values based on the same large integer (such as f) occur many times, it is more concise to define and use a constant. Also, when s is used instead of 40740, BASIC can retrieve the value from the variable area and does not have to parse the digits 40740 into an underlying numerical representation, so there is a speed advantage.

To be clear, these sort of assignments aren't constants in the strictest sense. C64 BASIC does not have user-definable constants or variables that can only be assigned once. And the "constant" c is actually reused in end-of-game code when it is no longer needed.

Use of One-Letter Variable Names

Variables can have one or two-letter names. (You can type in longer names, but at most the first two letters end up being used.) For space reasons, of course, it's better than they be one letter long. This also improves speed, since only a single letter needs to be parsed.

Array a to Update Position

The player types 1 or 2 to move the jetski. Having an array a with the appropriate displacements defined means that a(i) can simply be added to the current position.

There is no need for an IF statement or other more elaborate logic.

. Instead of 0 (Zero)

Curiously enough, it is fine to represent zero in C64 BASIC as 0, 0.0, 0., .0, or simply . without anything next to it. It happens that . is not only as concise as 0, it is also the representation that takes the least time to process.

Main Loop on Lines 1–5

The program has to begin with the first line (0) so some initialization code must go there. But instead of continuing with initialization on line 1, the programs jumps to line 6, and the main loop is placed on lines 1–5. This is because BASIC searches from the beginning of the program down through each line when it needs to GOTO some particular line. So the GOTO statement on line 5, which returns to the top of the main loop, is sped up slightly by having the main loop occur earlier.

Speed Optimization in Main Parts of Main Loop

The speed optimizations need to be made for the sake of gameplay, not everywhere in the program. Initialization can be slow, and indeed the player should be able to get into the game gradually. After the game ends, it is not important that the code runs as rapidly as possible. And even within the main loop, the code that deals with the jetski reaching the edge of the screen and the player earning a point can be slow, because we would like the green flash of the border to be visible.

So, while we made some speed enhancements outside the main parts of the main loop, our emphasis was on the code that must run every turn.

Modifying a String Using POKE

Instead of using the typical functions to modify strings, such as LEFT$, MID$, and RIGHT$ combined with the + operator, the program uses POKE to directly change the 40 bytes where the string data is stored. Because the string is constructed in the same way each time the progam is run, the string data always ends up at 40740–40779. If we had simply assigned L$ with forty characters of spaces, we could still modify the string. But in this case, the string pointer would point into the code itself, and our modifications would alter the program's code. Even if this was all right with us, it would have violated the rules of the BASIC 10 Liner contest.

Unrolling Loops

These four statements could have been written more concisely using a FOR loop that does each statement twice:

```
pOs+rN(1)*q,32:pOs+rN(1)*q,32:pOc+rN(1)*q,7:pOc+rN(1)*q,7
```

But there is room in the program to write each statement twice. We avoid the overhead of the loop by simply laying those statements out.

For Loop Terminating on Zero, not an Expression

The FOR loop beginning `fOi=-w/2to.` seems to be written in reverse; the normal way to

do it, since the value of i is not actually used within the loop, would be to count up from zero (represented as .) to w/2. However, it is faster to test whether i is zero than to do the division each time and test whether i is w/2.

PRINT, not POKE, to Output Characters on the Screen

BASIC's PRINT statement uses the built-in CHROUT routine, in the Kernal, to rapidly put characters on the screen and to scroll the display up when needed. This is much faster than using POKE to update a character at a time. Also, of course, we need to scroll the screen, and PRINT does that for us.

A BASIC Case

We hope our explanation of this specific code is of interest to expert Commodore 64 BASIC programmers and also helps those new to programming understand something about how computer programs work in general. Whatever your level of expertise, we also hope you will develop a game for a future BASIC 10 Liner contest.