256-Byte Creative Programs

Sofian Audry, Angela Chang, Chris Kerich, Milton Läufer and Nick Montfort

June 2017 TROPE–17-02

Abstract

During the past semester, inspired by work in the demoscene, one-line and similarly short programs in recreational computing, and other concise productions, we began writing 256-byte programs to share in Trope Tank meetings. Interestingly, these turned out to be of several different particular types and helped us in different ways to investigate aspects of computing and culture. We discuss some of the types of practice that we developed and a few of the insights we reached, concluding with a note about our Salon 256, in which we invited others to join us and share short programs.

A technical report from

The Trope Tank
NYC, 545 E 14th St & MIT, 14N-233
http://nickm.com/trope_tank/

A research and creative practice project undertaken in the Trope Tank in 2017 involved coding short computer programs limited to 256 bytes. The project was initiated following some of the lab's members' participation to the Synchrony demoparty on January 27-28. This event had a compo (competition) category called "nano" that was described as:

> Maximum file size is 256 bytes for the executable. Entries are allowed in standard interpreted languages that are standard with the OS, free software, or freeware (e.g., JavaScript, Perl, Processing, BASIC, QBasic). If the demo is for an interpreted language, the source file is limited to 256 bytes.[1] [2]

This category was developed by Montfort for the first Synchrony the year before. Our core group of Trope Tank members (Sofian Audry, Angela Chang, Chris Kerich, Milton Laufer, and Nick Montfort) participated in the lab's 256-byte project, committing to create works on a regular basis. These creations were shown and experienced during lab meetings, during and after which some of these works were refactored to make them shorter. All works were made available to the public in an online repository[3].

A total of 36 works were generated over the course of the semester, to which we can add 10 refactored versions of some of these original works. The works were written in different languages (Python, JavaScript, BASIC, assembly) and for various environments (the Commodore 64, Web browsers, the Linux terminal). They also occupied a wide range of genres, from interactive fictions to classic game remakes, from educational programs to generative animations.

Coders employed different methods to keep their programs short, such as, in interpreted programs, removing unneeded whitespace, using 1-letter-long variable names, reusing variables, and employing language-specific tricks. In some cases we chose to write in assembly and to assemble an executable of 256 bytes or less.

Pedagogical Nanos

Angela Chang pursued an educational/pedagogical approach for her works by developing tools to help beginning programmers. The works are not intended to be used as standalone programs, but rather function best in a command line interpreter or iPython/Jupyter notebook. Students are meant to use the Python to edit the functions; prompting programming activity is inherent in the experience of these pieces. These

1 http://synchrony.nyc

2 Although we first expanded these constraints to include non-programming media such as text, image, and sound, only two productions turned out not to be computer programs; one was a text file, another a set of HTML files summing to less than 256 bytes.

3 http://nickm.com/trope_tank/256/

programs are a response to Nick Montfort's proposed starter programs from his *Exploratory Programming*,[4] where each work highlights some challenging programming concept that seasoned programmers take for granted. For example, *vizlist* shows that list and string indices begin at 0, while *message* and *unmessage* shows how string manipulation could be used to encode and decode a message. The *image* functions demonstrate the pixel-by-pixel addressing and sequential processing necessary for programming with images. Finally *emdas* and *pemdas* expose python's order of operations: *emdas* invites users to reason about how different expressions are evaluated, while *pembas* recursively demonstrates how parenthesis are prioritized over the other operations.

One program *stepframe* outputs this sequential flipping:

Angela writes about her work:

> What I learned, besides some advanced programming tricks for condensing code, is that small programs invite tinkering. Students find these small programs more approachable because there are few variables and operations to think about. They can more easily rearrange lines, change operators to manipulate the constructs (lists, images, and strings) directly than with longer programs. Students are also introduced to programming shorthand, coding practices that are not typically found in introductory textbooks.
> Intellectually, it is a challenge to balance code legibility with conciseness. These programs helped me think about what the essential problems of programming were to my students and distill an experience that would help them practice it.

Text-based Games

Nick Montfort came up with a work of interactive fiction called *Wastes* written in Python 3. The piece, launched from the command line, presents the following:

```
WASTES
an adventure

Moor 53

>
```

4 Montfort, Nick. *Exploratory Programming for the Arts and Humanities.* MIT Press, Cambridge, Massachusetts, 2016.

The prompt invites the user to type instructions. For example, she can use cardinal directions (north, south, east, west) to navigate within the world, soon to discover a wide, mostly empty wasteland that presents itself as a 10x10 grid. This space can, if the player wishes, be mapped out on paper. The work remains true to the interactive fiction form, having a sort of minimal world model and parser. Wastes allows the player to "look" and to "quit" the game and even provides an opportunity to win the game if the player locates a treasure in one of the Moor locations – always the same one; there is no procedural generation or random placement in canonical IF. If they player does win, the game addresses her as "you" ("You won!") in a manner typical of text adventures, multisequential novels, and the like.

The code was written in Python 3, which allows assigning a function to a variable, thus saving space by rewriting function print as p; note also that the game's main loop is a while loop:

```
p=print;l=53;i='l';p('\nWASTES\nan adventure\n')
while'q'!=i:
      l+=(i=='s')*(l<90)*10+(i=='e')*(l%10<9)-(i=='n')*(l>9)*10-
(i=='w')*(l%10>0);x=l==7;p(('?','Moor '+str(l)+"\nLo, a gem."*x)[i
in'nsewl']);i=input('\n>')[:1]
      if(i in'gt')&x:p('You won!\n');i='q'
```

Shortly after writing this send-up of interactive fiction, which presented a rather pointless treasure hunt and a tabula rosa player character, Nick wrote a 256-byte classic hypertext fiction, *Vita,* using several HTML files. This one allows the player to choose, for instance, "...was it because of the war, or your mother?"

The Flu Shot Stand, created by Sofian, is also an investigation of a classic sort of program – an attempt to reinterpret the 1973 business simulation game *Lemonade Stand* by Bob Jamison. The original piece put the user in charge of a lemonade stand, asking her to make decisions such as how many glasses of lemonade to make, how much advertising to buy, as well as the price to charge for each glass. Based on these decisions and on random events such as weather conditions and street closure, the user would know how many glasses were sold, and would see the profit or loss for the round.

In Sofian's version, written in BASIC for the Commodore 64, the lemonade maker is turned into a pharmacist who buys and sells flu shots. In each round, the user is presented with a forecast of next day's temperature (in Celsius) and needs to input the number of shots to buy and how much to sell them. Then, the actual temperature is presented (which may differ from the prediction) and the demand for shots is computed. This demand is calculated using a simple linear model, simulating the number of infected people due to low temperature while into account the adversarial effect of medication cost, as follows:

$$demand = (149 - temperature) - (5 \times price)$$

Based on this demand, the number of flu shots produced, and the price of each shot, the program computes the profits made (if any) and moves to the next round. In order to

save on space, there is no check on available resources, and the user's available assets can become negative (we suppose it is possible to borrow money).

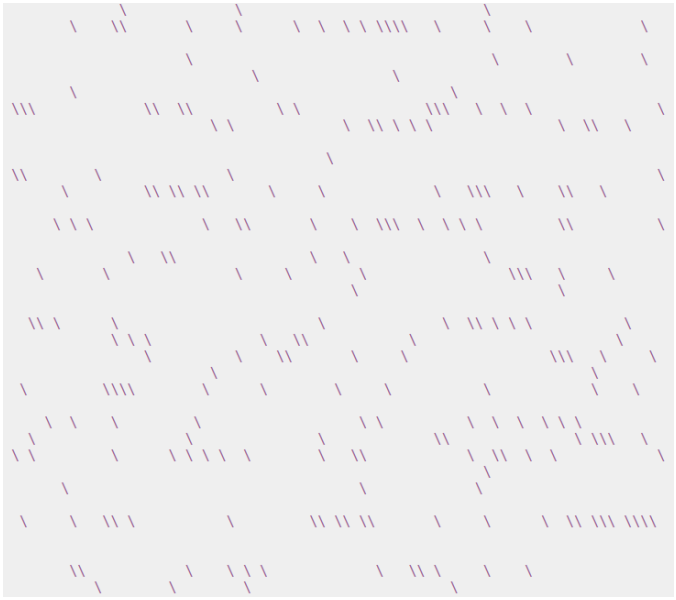
The code of 256 characters reads as follows, with the computation of shots sold in bold:

```
1 b=-50:m=99:c=50
2 w=rnd(0):r=int(w*m+b):e=rnd(0)/3-.3:f=int((w+e)*m+b):print "$";c:print
"outlook:";f;"c"
3 input "buy n. shots";n:input "$/shot";p:print "today:";r;"c":s=int(m-
m*w-5*p):if s>n then s=n
5 if s<0 then s=0
6 print "sold";s:c=c-n+s*p:goto 2
```

Character-based animations

Chris's works, all of them programmed in Python, present a progression mainly focused in the use of four (4) ASCII characters: /, -, \, and | (though not every work uses all of those characters). In his first contribution, titled *Hex*, lines of text are printed directly on the terminal window. As the text starts scrolling, it creates an animation of growing hexagons, the only regular figure that appears constantly in nature.

This idea developed in different variations, each one a representation of different phenomena or landscapes. For instance, *Wave* and *Rain* are similar programs with very distinctive outputs:



As in *Hex*, these two works make use of automatic vertical scrolling. Unlike *Hex, Wave* and *Rain* apply randomness, and are not constrained to one particular geometrical figure. Then, the effect of both of them is accomplished by switching, in a coherent manner, from one of **/**, ****, or **|** to another. The original version of *Wave* is 252 bytes:

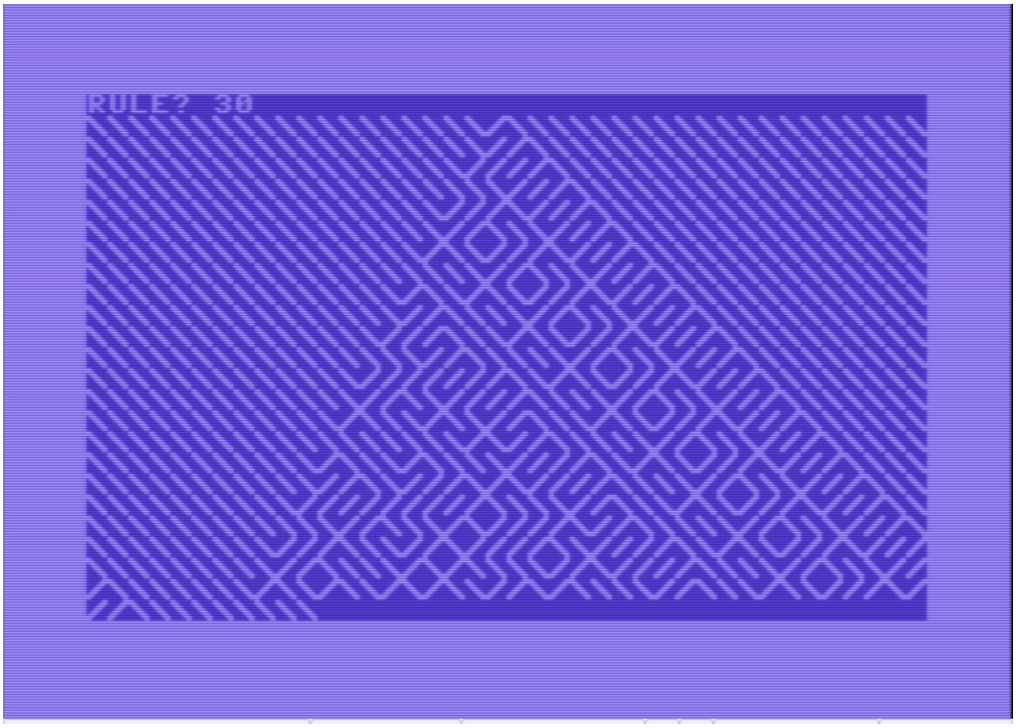
```
import random as r, time as t, math
s=1
o="|"
p={'\\':1,'/':-1,'|':0}
while 1:
	c=["\\","/","|"]
	s+=p[o]
	print s*" "+o*s
	t.sleep(.07)
	if o=="\\":c.remove("/")
	if o=="/":c.remove("\\")
	o=r.choice(c)
	if (o=="\\"and s>=99)or(o=="/"and s<=1):o="|"
```

The group shortened the code to 156 bytes using a series of optimizations, some of which are of general interest. By finding a better representation for the three characters (in a string with three characters rather than a list of quoted characters), and indexing efficiently into that representation, several reductions in size were made possible. The result is even arguably clearer overall:

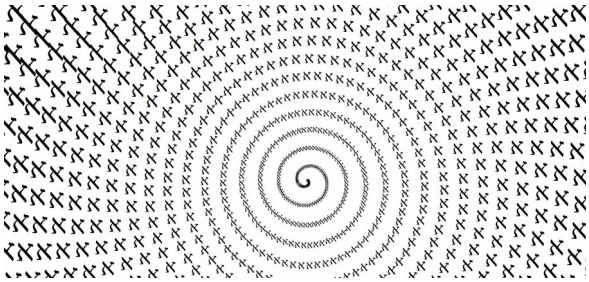
```
import random as r,time as t
s=d=1
while 1:
	s+=d-1;print s*' '+'/|\\'[d]*s;t.sleep(.07);d=r.choice([0,1,2]
[d>1:3-(d<1)])
	if(s<2)&(d<1):d=1
	if s>39:d=d%2
```

Later in the semester, Chris produced *City*, a scrolling ASCII work that represents fictional generative buildings and uses only two characters (| and _), as the result is meant to be square. The scrolling is horizontal and creates an image suggestive of a city skyline, one that could be used as a platform or background for a scrolling game.

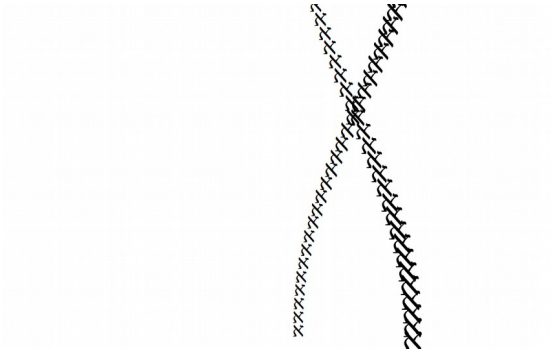
This type of visual development through the use of particular characters was also used by Sofian in *CA*, a Commodore 64 program that uses PETSCII characters:



Milton Läufer's work, like Chris's work in Python, was all done in the same language, HTML5/Javascript. This selection gave him the ability to easily share his work on the Web. His first two pieces have a similar code, though the result is very different:



```
<body
onload=d=document;z=0;setInterval("a=d
.createElement('div');t=a.style;t.font
Size=10+z/16;r=z*.1;s=(1+7*r);t.left=7
00+s*Math.cos(r);t.top=400+s*Math.sin(
r);t.position='absolute';a.innerHTML='
א';d.body.appendChild(a);z++",9)>
```



```
<body
onload=d=document;z=0;p=d.body;setInte
rval("for(x=-
1;x<2;x+=2)a=d.createElement('div'),t=
a.style,t.position='absolute',a.innerH
TML='א',t.top=z*25,t.fontSize=(i=7+x*M
ath.cos(z*.1))*i,t.left=i/.01,p.append
Child(a);p.scrollTop=p.scrollHeight;z+
+",99)>
```

The use of <body> takes advantage of the fact that HTML5 does not require declaration of either <html> or <head>; it is also not required that this tag be closed. Using a <script> tag instead, would be shorter, but that approach has the disadvantage of implying the creation, by JavaScript, of the body element. The choice of the Hebrew aleph character was mainly a aesthetic one, according to the author, but has a price associated with being a three-byte character, as this character doesn't belong to the standard ASCII table.

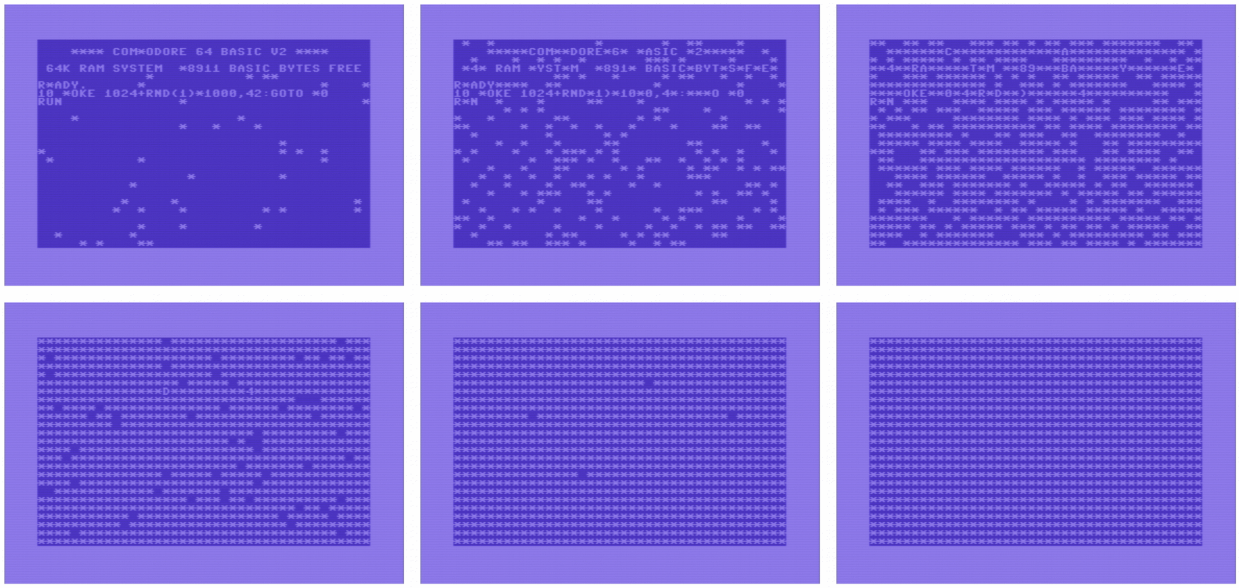
Milton's other two pieces emulated, by the alternation of the * and ‘ characters and the erratic movement of both, the motion of a fly, which would bounce off the top and bottom areas of the screen. The change implemented in the second piece is that this one represents two flies. Internally, both of them are similar to the aleph-based ones, using the JavaScript trigonometrical functions and a time loop.

Nano-nano

The shortest work in the lot is a 36 characters long BASIC work for the C64 called *Storm*. The piece was written by Sofian on March 15th 2017, and was inspired by a two-days long snowstorm that hit the East Coast on March 14-15. The one-line code randomly fills the screen with the star (*) symbol (PETSCI code 42) until it covers the whole area:

```
10 poke 1024+rnd(1)*1000,42:goto 10
```

Here are images of the work as it evolves through time:



About this work, Sofian writes:

> Working under such harsh size constraints as those imposed by this project is not a habit for me. I am used to working with complex, high-level software architectures. In most 256 bytes works I created, I started with a big idea, realized mid-way that I would not be able to make it as I wanted, and struggled my way through to keep as much of the core features as possible. I made *Storm* with a totally different approach. I was sitting in the kitchen in the morning after the blizzard, looking at my backyard covered in snow, I had this idea and I just wrote it down, without thinking much about it. *Storm* is the result of a process of discovery and acceptance, that simple programs have the potential to generate rich, evocative experiences.

After this program was presented, we looked into *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*[5] and found, on page 153, a program called "Icicle Storm" which is identical except for using 78 (the screen code for a diagonal line from upper right to lower left) instead of 42. Sofian's re-creation of this program is apt, as many hobbyist programs (such as the famous 10 PRINT) were reworked, reimplemented, and persisted almost as folklore does in culture. Another feature of very short programs is that aspects of them, and sometimes the whole program, become memorable enough for programmers to redo, or, in some cases, are suitable and pleasing enough that they are often rediscovered.

5 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.

Salon 256

On May 1st 2017, we organized Salon 256, an open event that took place in Building 14 at MIT. People were invited to sign up before the event or when they arrived to to show a nano piece of their making, discuss it, and answer questions. Audry, Chang, Kerich, and Montfort presented, but there was also interest and participation from across campus and from the local community, with presentations that included a LISP interpreter written in Lisp (thanks to Henry Lieberman of CSAIL), an assembly-language demo representing and written on a train (Dr. Claw, the organizer of the local demoparty, @party), a JavaScript bookmarklet to jumble the words in Web pages (MIT undergraduate Willy Wu), and a system that uses a word list to skip through slightly different English words (Doug Orleans, a local computer scientist). The work here showed a new sort of engagement with existing data sets and systems, including LISP, word lists, and pages on the Web.