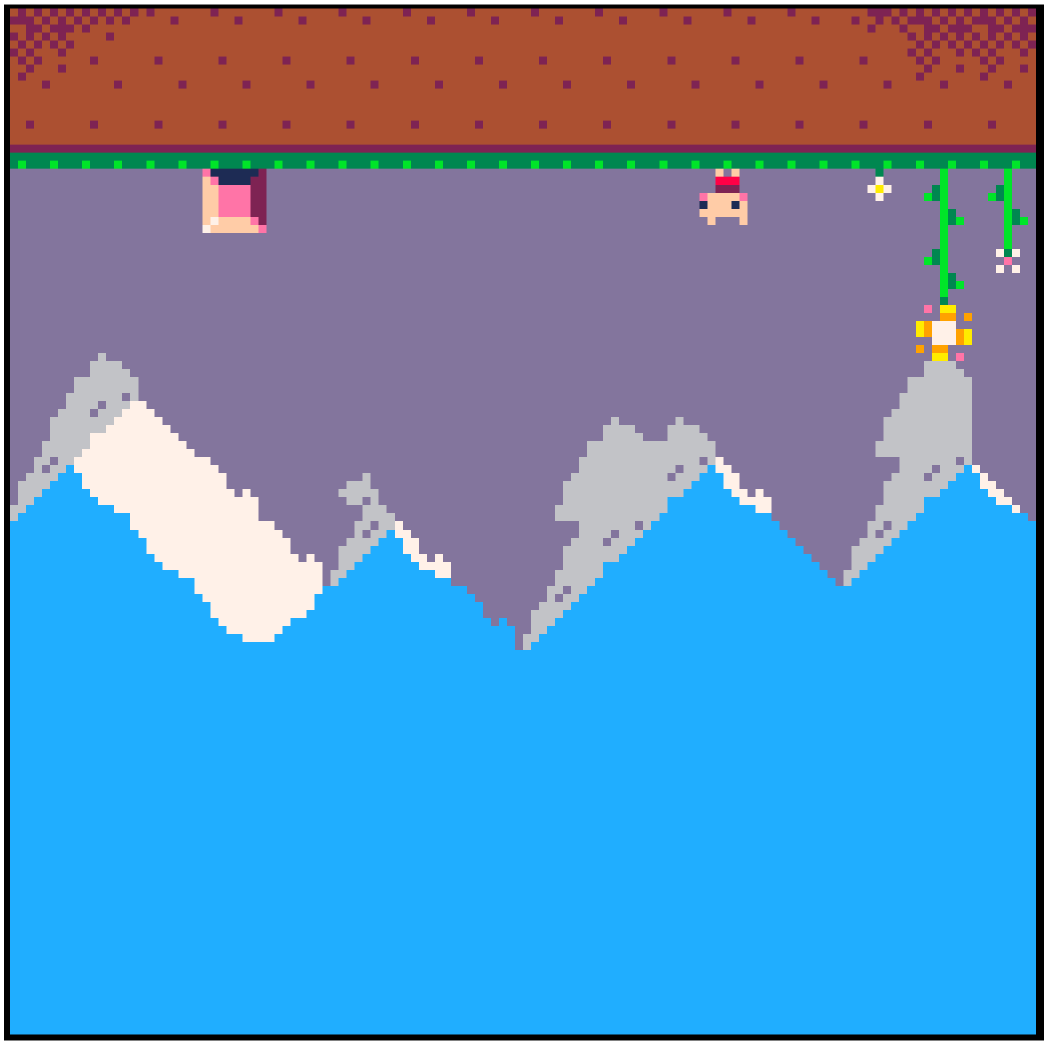
picotool, a Python library for PICO-8

dansanderson.com/articles/picotool

November 13, 2015

picotool is a suite of tools and libraries for manipulating data files for the Pico-8 fantasy game console. picotool is written in Python 3, and it includes a full Lua parser written from scratch.

As a tech demo, I wrote a tool that uses the library to programmatically invert the sprites, map, music, and controls for a given Pico-8 game. Here is Pico-8's demo game, "Jelpi," turned upside down:

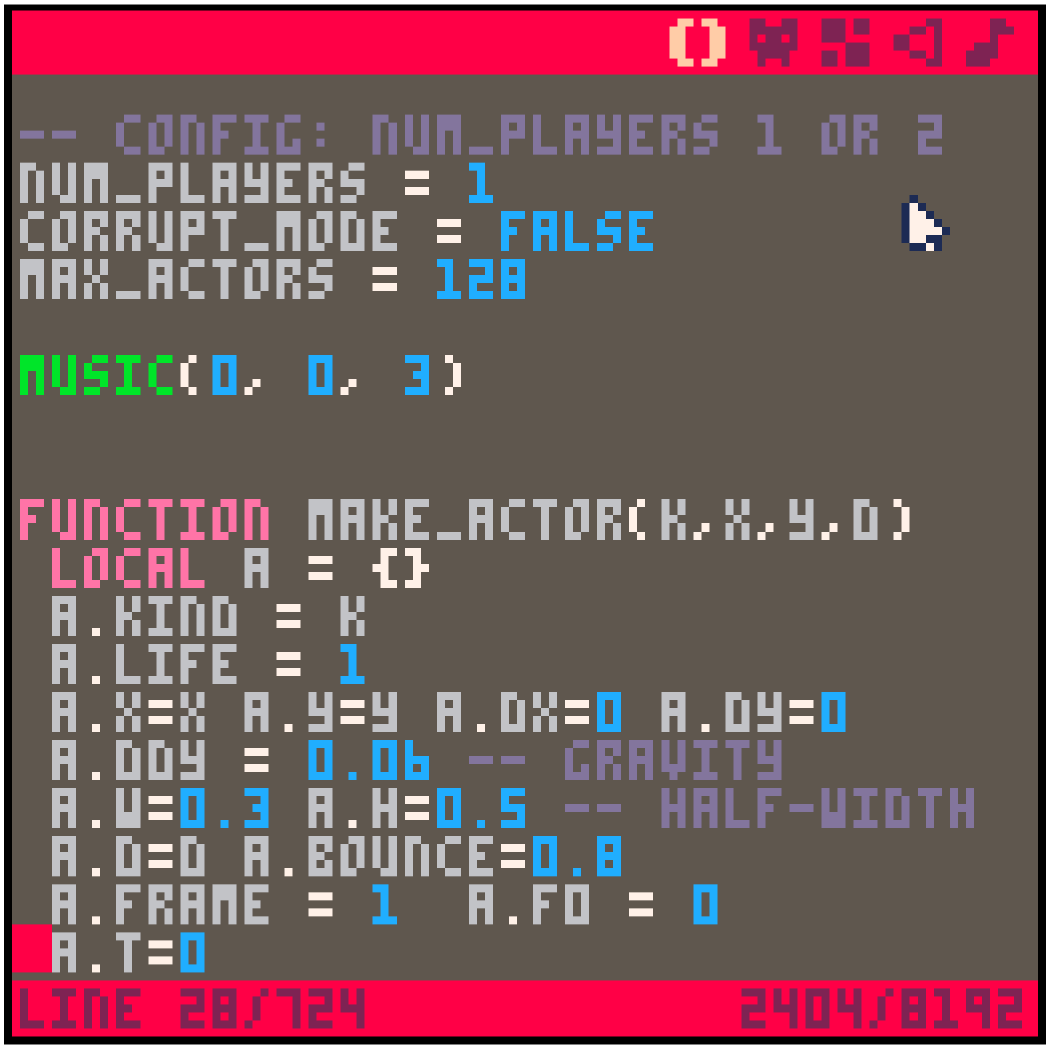


Click to play!

Links:

Pico-8 and limitations

Pico-8 is a "fantasy game console" that vivifies the memory of developing video games on the 8-bit home computers of the 1980's. Which is to say, it isn't a complete virtual machine with simulated hardware, but rather a game platform with a set of constraints. The platform imposes tight limits on screen resolution, sound capabilities, memory, and code size to encourage experimentation, creativity, and community building. Much of Pico-8's charm comes from a built-in development environment for editing code, graphics, sound, and music. These tools run within the same constraints as the games themselves, including the 128x128 16-color screen.



```
-- CONFIG: NUM_PLAYERS 1 OR 2
NUM_PLAYERS = 1
CORRUPT_MODE = FALSE
MAX_ACTORS = 128

MUSIC(0, 0, 3)


FUNCTION MAKE_ACTOR(K,X,Y,D)
 LOCAL A = {}
 A.KIND = K
 A.LIFE = 1
 A.X=X A.Y=Y A.DX=0 A.DY=0
 A.DDY = 0.06 -- GRAVITY
 A.W=0.3 A.H=0.5 -- HALF-WIDTH
 A.D=D A.BOUNCE=0.8
 A.FRAME = 1  A.F0 = 0
 A.T=0
```

LINE 28/724 2404/8192

My own 8-bit experience hit a milestone when I noticed that professional games appeared to exceed the capabilities of my Commodore 64's built-in BASIC programming environment. I was vaguely aware that these games used a more powerful "machine language," but it wasn't until years later that I understood that many of these games were not developed on Commodore 64s themselves, as my own games necessarily were. Rather, game engineers would use tools on more powerful computers like Unix workstations to *cross-compile* code and data to use the C64 hardware as efficiently as possible. I was jealous that these tools were out of my reach at the time.

In a way, picotool is my attempt at capturing that part of early game development that I missed. The library can parse, modify, or generate any aspect of a Pico-8 game, including the Lua code region. I can use it to construct tools to extract the most value out of Pico-8's artificial limitations.

Why bother squeezing artificial limitations when I can just use a more powerful platform? A huge part of Pico-8 is the community that has grown around it, people producing games and demos with a wide variety of levels of experience. Beginners use it to learn programming and game development, and pros use it as a sketchpad and prototyping platform. Key to the community is the commonality of the platform: we're all using the same tool and working within the same constraints.

To that end, using external tools to make Pico-8 games, as opposed to using the built-in dev environment exclusively, is a cheat. But it's no more a cheat than the professional developers of the '80s using similar tools. In the end, the games still run on the target platform.

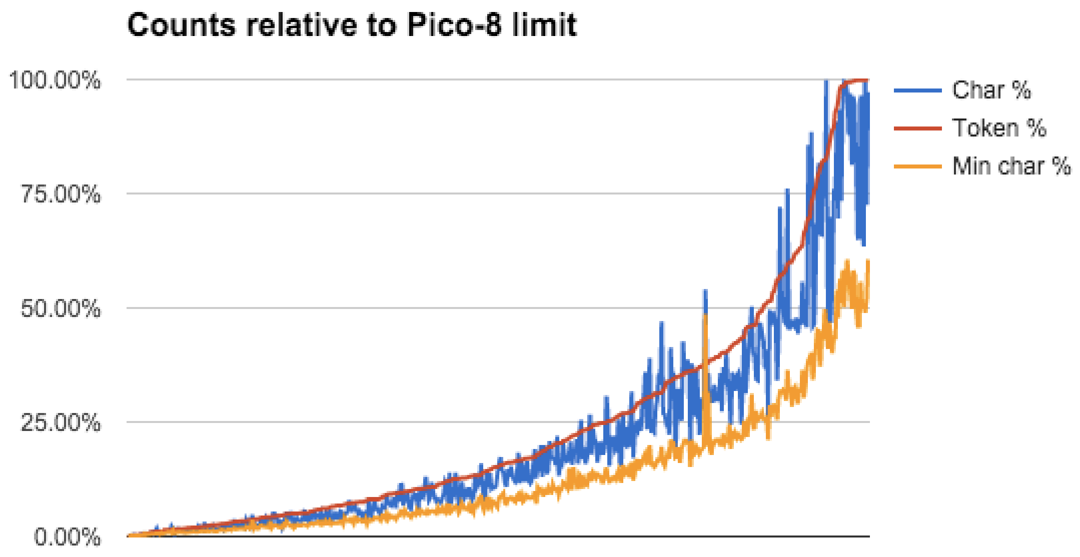
Code formatters

Originally I started picotool as a personal challenge to write a code formatter. Most Pico-8 games are written haphazardly and in the tiny window of the built-in editor, so coding style is not usually a priority for most developers. By design, it is easy to crack open any published game and see how it works, and I thought it'd be useful to have a script that can clean up indentation and such to make the code easier to read.

I had never implemented a code formatter before. I knew that a formatter usually needs a full parser for the language being formatted, and Lua's syntax is relatively simple. How hard could it be? I'll just say that it's non-trivial. picotool's Lua parser isn't the best, and many formatting tasks can be accomplished using just the lexer (the part of the parser that recognizes individual terms). But naturally once I started writing it, I figured it'd be cool for a general purpose library to have a full Lua parser.

Pico-8 stores a game's Lua code directly in the game data and executes it with a Lua interpreter. To represent the code size limitation implied by an 8-bit computer storing instructions in a limited amount of memory, Pico-8 imposes an arbitrary limit on the size of the Lua source code. This limit was originally specified as a number of characters, but this led to developers purposefully making their code difficult to read with short function and variable names. A subsequent version of Pico-8 added a new limit that counts language tokens (words) instead of characters (letters), and this is intended to be the limit that matters, though the character limit is still in place.

With mixed feelings and mostly because it was easy to do, picotool includes a Lua minifier that takes easy-to-read code and shortens all of the names and deletes whitespace and comments so that the code takes as little space as possible. I'm not fond of making published code impossible to read because I consider code inspection to be an important part of the community. In the picotool announcement, I included a brief analysis of the character and token limits that I generated with picotool, concluding that yes in fact the token limit is the important one for typical games, and minifying doesn't help. (Shortening words doesn't reduce the number of words.)



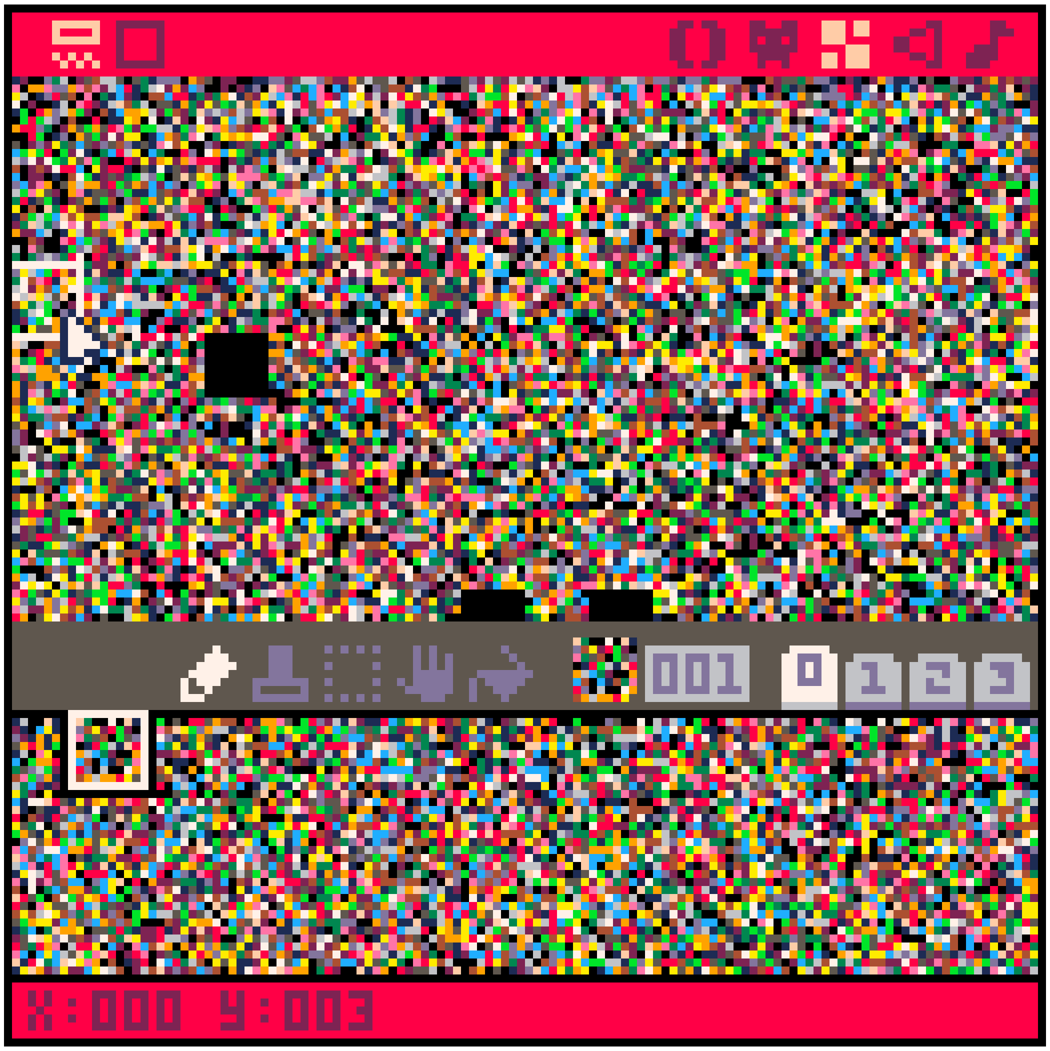
A Tale of Two Cities

One of the most important techniques of cross-platform development for small target platforms is compression, the use of algorithms to cram as much data into the cartridge space as possible. This data is decompressed as needed while the game is playing.

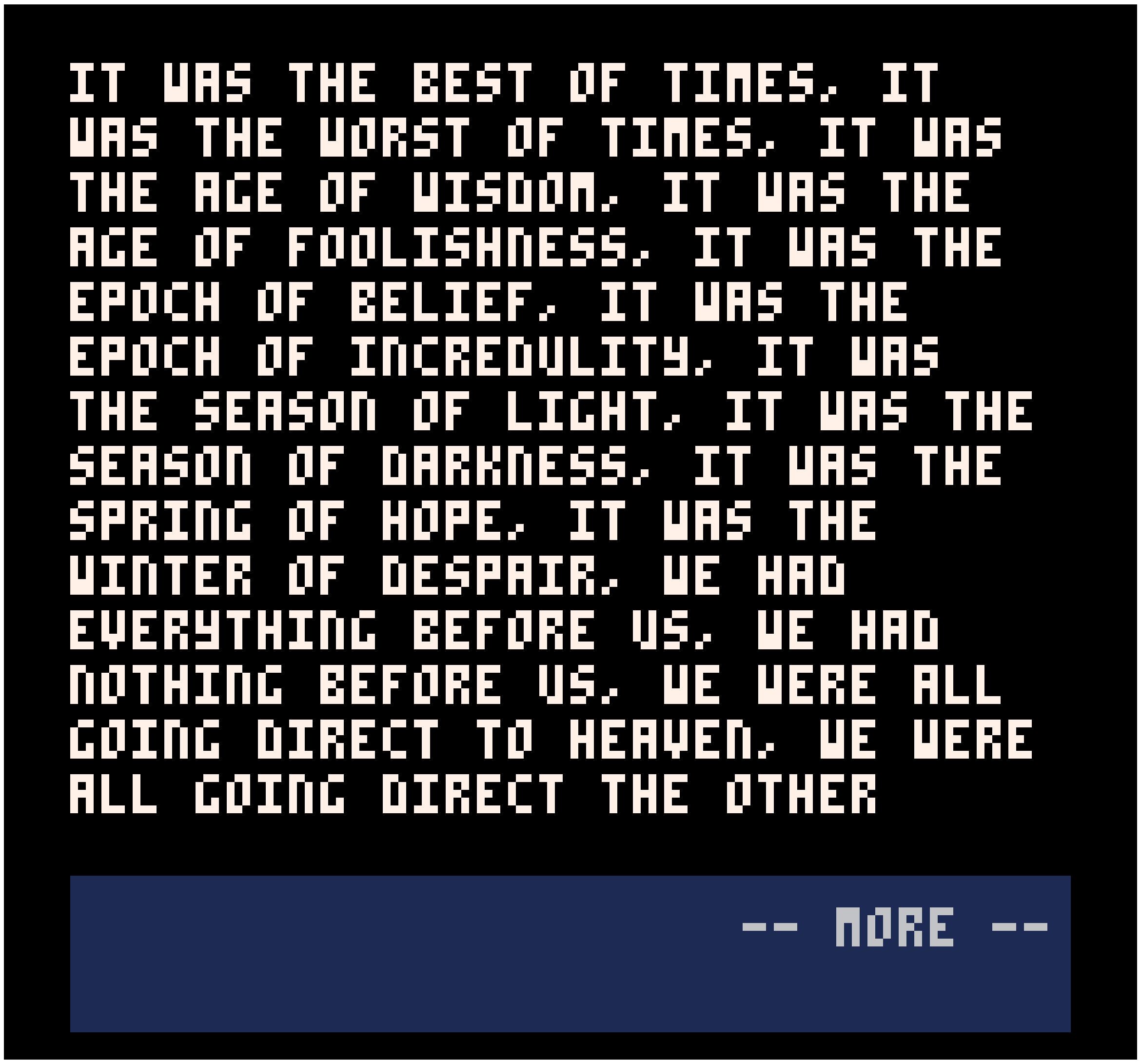
With ambitions to build a graphic adventure game engine based on compressed text, vector graphics, and animated sprites, I started a tool called p8advent. I only got as far as text compression, but it's a good example of how to use picotool to build a development toolchain for compressed data.

p8advent takes a Lua source file and a Pico-8 data file as input, and it parses the source file for specially marked strings. The strings are added to a compressed string dictionary, then replaced in code with a function call and a dictionary index. The corpus is written not to the code region but to the graphics region. (You can specify an offset if you want to make room for graphics as well as text.) p8advent uses an LZW compressor is written in Python, and the decompressor is written in Lua and is added to the game code by the tool.

The compressed text data looks pretty in the graphics editor:



As a demo, I made an ebook reader containing the first three chapters of *A Tale of Two Cities* by Charles Dickens.


IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES, IT WAS THE AGE OF WISDOM, IT WAS THE AGE OF FOOLISHNESS, IT WAS THE EPOCH OF BELIEF, IT WAS THE EPOCH OF INCREDULITY, IT WAS THE SEASON OF LIGHT, IT WAS THE SEASON OF DARKNESS, IT WAS THE SPRING OF HOPE, IT WAS THE WINTER OF DESPAIR. WE HAD EVERYTHING BEFORE US, WE HAD NOTHING BEFORE US, WE WERE ALL GOING DIRECT TO HEAVEN, WE WERE ALL GOING DIRECT THE OTHER

-- MORE --

Click to play!

Here's the forum thread about text compression.

Future plans

picotool and Pico-8 have minor disagreements about what counts as a token, including cases that might be bugs in one or the other. Since picotool was released, Pico-8 has made revisions to reduce the effective token count with regards to the limit that would disagree with a typical lexer, such as not counting `end` as a token. I'd love to hunt down the last few discrepancies so that picotool and Pico-8 match, especially so that it can be known in advance whether generated code doesn't fit within Pico-8's limit. With the recent Pico-8 changes, it is likely that picotool overcounts in the typical case, so if picotool allows it, so will Pico-8.

A need often discussed by Pico-8 developers is the ability to write reusable libraries of code. Pico-8 does not have built-in support for Lua's `require()` mechanism. Several people have written simple tools that concatenate multiple source files and install it in the game data, and this would be easy to do with picotool as well. But with picotool we can do even better: with full access to the parsed code tree, we can do *dead code elimination* on the result and generate a cart with only the routines actually used by the game. This is necessary if we want to use large libraries and inter-library dependencies. A naive version of this would not be difficult, and I'd love to take this on some day.

picotool can read both of Pico-8's file formats, a text-based `.p8` format and the steganographic `.p8.png` format that hides the game data inside an image of a game cartridge. (Yes the PNG format is an extensible container and it's not strictly necessary to use steganography to attach game data to an image, but it's fun.) Currently, picotool can only write the text-based format. I'd like to add the ability to write the `.p8.png` format as well. I'll have to come up with a default image of a game cartridge to use when creating a data file from scratch, though I suspect it'll be more common to overwrite data in an existing file.

picotool has been a fun on-going project with multiple dimensions. I don't know if anyone other than myself will ever use it for something, but I'd like to keep it in good enough condition to make that possible.