PICO-8 ZINE #1

CODE-PLAY-SHARE-BE-THINK-DESTROY-LEARN-BREAK-LOVE-MAKE

AUGUST 2015

CONTENTS

PICO-8 is a fanzine made by and for PICO-8 users.
The title is used with permission from Lexaloffle Games LLP.
For more information: www.pico-8.com
Contact: @arnaud_debock
Cover Illustration by @dotsukiHARA

A Brief History of PICO-8

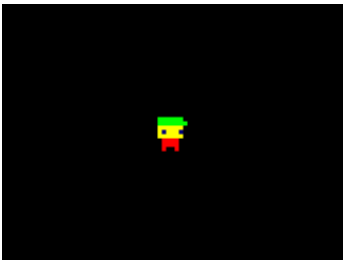
Greetings, zine readers. My name is zep, or Joseph White in real life. I'm the author of PICO-8, and was naturally very pleased to find out about this publication! As this is issue #1, I thought it might be fitting to give you a look into how PICO-8 came about in the first place.

Early Influences

It might not come as much of a surprise that I grew up with classic home computers like the Apple IIe,C64 and BBC Micro. Although my family didn't own one, I spent plenty of time crashing friends' houses and camping out in my father's psychology lab, where he used Beebs to control hardware for conducting memory experiments on pigeons. This is how I learned to program -- typing in snippets of code from the BBC manual and trying to construct anything evenly remotely resembling a playable video game.

The feeling of creating programs for those machines is a visceral childhood memory ranking right up there with the smell of the macrocarpa tree I climbed with my first girlfriend, or the metallic taste of blood after crashing my bike on a gravel path when I was 8.

There was something about plotting large colourful pixels and punching in programs on a large clunky keyboard that resonated with my 8-year-old brain. The idea of turning on the computer and seeing anything except a prompt inviting the beginning of a new program would have seemed absurd -- it was meant to be just you, the program, and the inky black canvas.

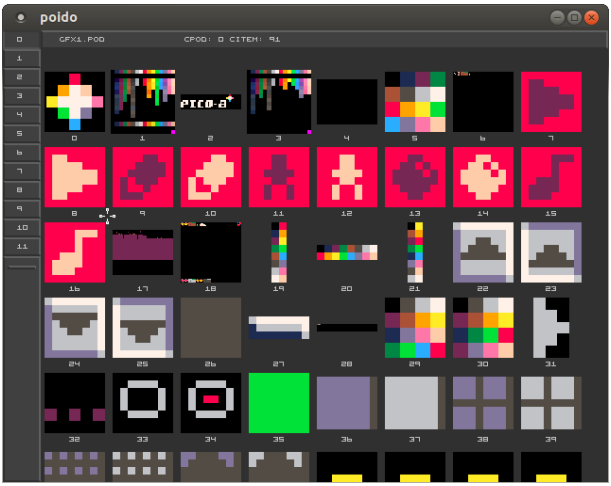


Look at this little fellow!
Where will he go? What will he do?

Tools and Editors

As I began to set about the task of creating whole games, I didn't know what kind of tools existed for making graphics, and it didn't occur to me to look. It became a kind of ritual at the start of each project to improve or rewrite a sprite editor, mapper, sound editor, or any other tools I needed to make each game.

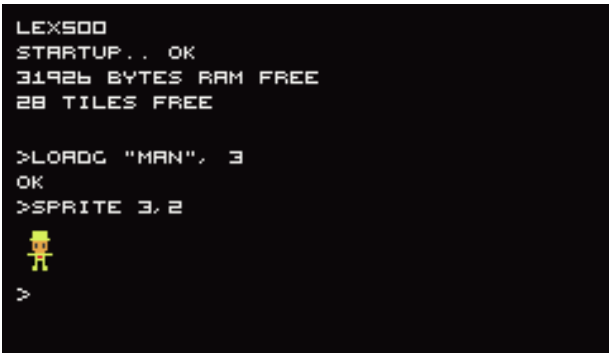
This habit ended up manifesting itself as poido ("Pointy Dough") -- a general purpose in-house editor created at the beginning of Lexaloffle. It has editing modes for sprites, palettes, models, uv maps, skeletal animation, audio, and later a modular sound synthesizer and voxel modeler.



The feature of poido I am most fond of is quite simple:
a collection of mixed resources can be edited and stored in a single file (a "pod", shown above), eliminating the need to jump between different editors or to manage sprite sheets or individual asset files.This became the model for Voxatron and PICO-8 editors.

LEX500

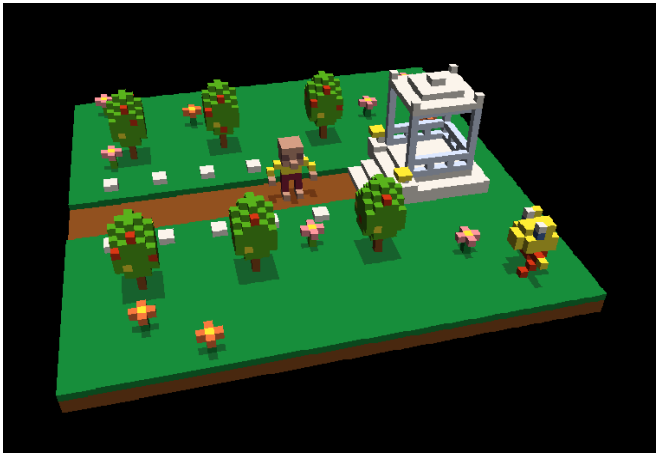
Fast forward to 2004. I'd been doing Lexaloffle for a couple of years -- released a few puzzle games that didn't sell very well, and was making small things for fun to stay sane while I did contract work to stay afloat. One of them was a project called LEX500 which was named after the Amiga 500, but was really more like a Beeb. It had a resolution of 240x240, a fixed 16 colour palette, and booted up into a BASIC interpreter environment.



LEX500 remained a bunch of design notes and mockups -- I regarded it at the time as a design exercise. There wasn't much that separated it from simply firing up a BBC Micro emulator, although it did have an integrated sprite editor. One quirky feature of LEX500 was that sprites would show up directly in the code editor when you referenced them.

Visual Formats

Apart from LEX500, I started playing with other simulated display formats that would give anything made with them their own particular visual style.
The first one was inspired by a combination of Ken Silverman's Voxlap demo, and some happy ray-tracing accidents that led to a 64x64x32 block of voxels rendered as cubes. I used it to mock up an adventure game called 'Felix and the Firebird' (below), which eventually became the basis for Voxatron and Voxatron Story.



The other display was vector based with a 2-channel colour format: one channel for hue and another for intensity. Polygons would be rendered into the channels separately and then converted into RGB at the end of each frame. I was using this for a prototype of Conflux (2008) and Swarm Racer 3000.

Working with these formats in conjunction with complementary tools gave me a taste for creating something that I vaguely understood as platforms or mediums;in the same way that games made for retro computers had a particular look and feel, the underlying platform could be treated as a separate design problem that would drive the identity of games made with them.

Voxatron

Voxatron started in 2010 as a way to do something small with my voxel display. It was initially only a Robotron-style shooter (hence the name), but after a preview trailer blew up on YouTube, I ditched the job I was doing at the time to go all in on the opportunity to expand it to include the original adventure game I had envisioned. Because of the unusual format, I also saw it as a chance to offer a general platform so that other users could try making and sharing their own voxelly work.

These varied goals created a design problem. How should all these parts be presented to the player without awkwardly slotting them into commonly understood game vocabulary? Game modes, mini-games, DLC, mods, levels, user-made levels; each one comes with a set of expectations that messes with how the content is perceived.

This was especially problematic for user-made content.
I wanted authors to have authorship; to be making their own thing on a platform rather than making a level or mod for an existing game. Some work might not even resemble levels or games at all, but toys or curiosities or visual demos.

It wasn't until half-way through the project that I stumbled into the cartridges analogy. Instead of having all of these separate types of content along with their semantic baggage and expectations, I could instead present them all as cartridges.
The notion of a cartridge could be used in a way that was general enough to capture what was needed: a standard shareable unit of expression.

PICO-8

PICO-8 began as a resurrection of LEX500 in 2012. I needed to add a general scripting component to Voxatron at some stage, but I didn't have much experience with that kind of thing. If I made LEX500, it would be a good way to get a grip on scripting, and perhaps later on it would become an accessible way to introduce Voxatron users to programming.

Another driving factor was opening the Lexaloffle office to the public several days a week as a shared workspace in the form of Pico Pico Cafe (which PICO-8 was partly named after). One of the early regulars there was Julien Quint, an all-round language theory and implementation guy, who shares my fondness for offbeat side-projects and inspired me to write a first pass at a BASIC interpretor for LEX500.

We started doing a monthly show-and-tell for designers called Picotachi (again with the Pico) and it turned out PICO-8 was a good thing to hack on so that I had something to show when nothing visually accessible was happening with Voxatron.

It gradually became apparent that the two projects had more in common than I had anticipated.

Because Voxatron has a complete set of design tools integrated with it, it seemed natural to do the same for PICO-8, and I could finally see why that was important.
I had been mocking up Voxatron graphics in 2D at a low resolution, and eventually went with 128x128 for PICO-8 so that it would fit into a single slice of Voxatron's volumetric display, making it a viable host platform. PICO-8 also shared Voxatron's goals of operating as a platform that could leverage the Lexaloffle BBS for distribution and collaboration.

PICO-8 had become a minimal 2D distillation of Voxatron.

An early screenshot of a PICO-8 test when I was trying to estimate how many tiles a game might need to be fun to design but not laborious to implement. It initially had a 160x120 screen with a separate 320x240 text mode layer for the code editor because I thought it wouldn't be possible to fit a readable text editor in at the native graphics resolution. Perhaps I was right about that.

The first iteration included a BASIC interpretor that was implemented by translating from BASIC to Lua internally in order to use the Lua vm.

I was gradually enamoured by Lua during this process, and ditched the BASIC facade altogether.

Although it was fun to think about what a real PICO-8 might look like, I never felt it would benefit from having an official physical form. Choosing specs was more about encouraging a certain design culture and development experience rather than being realistic or plausible. This was also true of the choice to limit controls to DPAD and 2 buttons, but a nice side effect is that users might be able to build their own PICO-8s with integrated controllers more easily one day.

Fantasy Console

So, I had all of these things going on that pointed at the concept of a "fantasy console": Cartridges, dev tools, a community platform, display formats and abstracted controls.
But I still couldn't see it! PICO-8 initially had "disks" and "programs" instead of cartridges -- for a while it was more of a fantasy home computer, which didn't sit quite right.
And Voxatron was still a game that happened to have carts and an editor included. If I tried to explain to anyone that Voxatron was actually more of a platform than anything else, they would look at me with lifeless eyes. Like a doll's eyes.

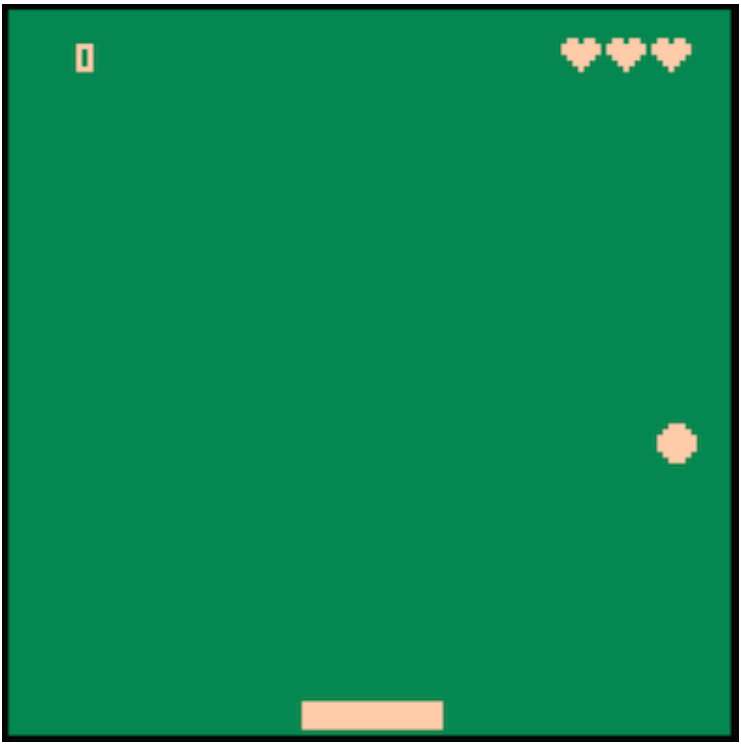
PICO-8 had grown into something that stood by itself, and looking at it next to Voxatron helped to more clearly see both of them. I don't remember how I made the final step into console territory, but by that stage there were many elements pushing in that direction. It gave Voxatron a good reason to have cartridges, offered a cute and approachable way to present these two otherwise abstract projects, and gave PICO-8 a focused identity to design around. Ideas like having a fixed 32k memory layout mapped onto the cartridge data layout would have been hard to see otherwise.

It might have been much easier if I started with the question "What would it be like to create a fictional console?" and work forwards from that. Instead I went backwards though a forest of fuzzy ideas and at the end realised: oh, these things are just consoles.

-- zep

SQUASHY

Let's make a game of squash, in the style of the classic game PONG!



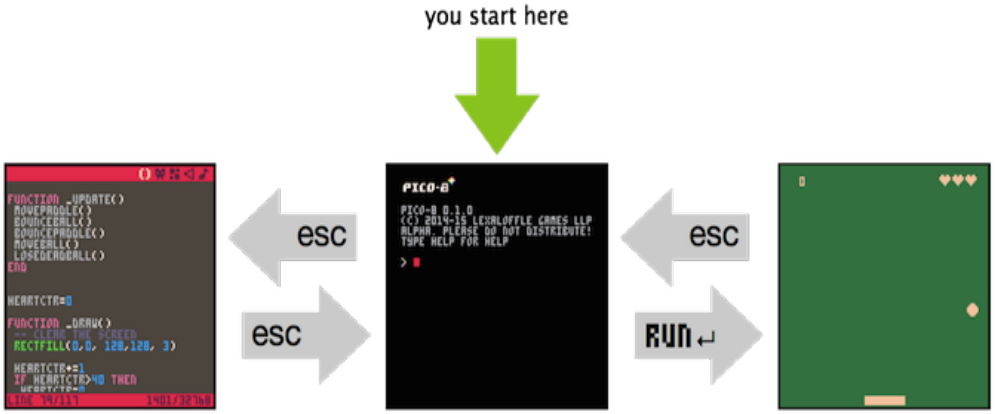
Getting around in PICO-8

When you boot PICO-8, you start in **Command Mode.**

From here, you can press the **Escape** (or ESC) key on the keyboard to go into **Editor Mode**, where you can create your games.

When you want to run your game, press **ESC** to return to **Command Mode**, and then type run and press **RETURN** (↵).

To get back to **Command Mode** again, just press ESC at any time in your game!



When you're in Command Mode, you can also save your game by typing **SAVE GAME-NAME** ↵.

To load your game up again when you come back next, type **LOAD GAME-NAME** ↵.

If you forget what you called it, type **DIR** ↵ and PICO-8 will give you a list of all the games it knows about!

For this game, you might want to use the name **SQUASHY**, so to save it you'd type **SAVE SQUASHY**

Now we're ready to get started making our first game in PICO-8!

1. A Moving Paddle

Let's make a bat move about!

Press the ESC key to go into editor mode, and then type this. It's worth pressing SPACE at the start of lines within **IF** and **FUNCTION** blocks, because it makes it much easier to read later. It's also worth putting in comments -- these are lines that are started with --, and the computer ignores them, so they're like little notes just for you!

```
-- PADDLE
PADX=52
PADY=122
PADW=24
PADH=4

FUNCTION MOVEPADDLE()
  IF BTN(0) THEN
    PADX-=3
  ELSEIF BTN(1) THEN
    PADX+=3
 END
END

FUNCTION _UPDATE()
 MOVEPADDLE()
END

FUNCTION _DRAW()
 --CLEAR THE SCREEN
 RECTFILL(0,0,128,128,3)

 --DRAW THE PADDLE
 RECTFILL(PADX,PADY,PADX+PADW,PADY+PADH,15)
END
```

Hit the ESC key and type **RUN**. When you press ← or → the bat should move!

See how we've made a function called **MOVEPADDLE()**. That'll make it easier to find what code does what later.

The commands we used

```
FUNCTION _UPDATE()--THIS GETS CALLED 30 TIMES EVERY SECOND.
                   IT'S WHERE WE WILL UPDATE EVERYTHING IN THE GAME
```

```
FUNCTION _DRAW() --THIS IS CALLED AFTER UPDATE
                 IT'S WHERE WE DRAW THE GAME

BTN(B) --CHECK TO SEE IF A BUTTON IS DOWN. THE NUMBER B MEANS THIS:

        0 IS ←,1 IS →,2 IS ↑,3 IS ↓,4 IS Z AND 5 IS X
RECTFILL(X1,Y1,X2,Y2,COL)--DRAW A RECTANGLE IN THE COLOUR COL
        X1,Y1 SHOULD BE THE COORDINATES OF THE TOP-LEFT CORNER
        X2,X2 SHOULD BE THE BOTTOM-RIGHT CORNER
```

2. Now let's add a ball

Press **ESC** twice to get back to the code editor.

Add some new variables at the top of the file so we know where to put the ball:

```
--BALL
BALLX=64
BALLY=64
BALLSIZE=3
BALLXDIR=5
BALLYDIR=-3
```

And then add the following to the `_DRAW()` function at the bottom of the file:

```
FUNCTION _DRAW()
 --CLEAR THE SCREEN
 RECTFILL(0,0,128,128,3)

 --DRAW THE PADDLE
 RECTFILL(PADX,PADY,PADX+PADW,PADY+PADH,15)

 --DRAW THE BALL
 CIRCFILL(BALLX,BALLY,BALLSIZE,15)
END
```

Press **ESC** to get out of the editor and type **RUN** to see the ball appear!

The new commands we used

```
CIRCFILL(X,Y,SIZE,COL)  --DRAW A CIRCLE WITH A CENTRE AT X,Y
```

3. A still ball is a boring ball

Press ESC until you're back to the code editor, then add a new function above the **_UPDATE()** function:

```
FUNCTION MOVEBALL()
 BALLX+=BALLXDIR
 BALLY+=BALLYDIR
END
```

And then make sure to call it in **_UPDATE()** like this:

```
FUNCTION _UPDATE()
 MOVEPADDLE()
 MOVEBALL()
END
```

RUN what you have, and you should have a ball that flies off to the top-right of the screen.

4. Keep it on the pitch

The ball needs to bounce off the top & sides of the screen. That's not too complicated -- we just need to check the **X** and **Y** positions of the ball.

Remember that the top-left of the screen is **0,0**
and the bottom-right of the screen is **127,127**.

To make the ball bounce off a side, we just have to flip the sign of the direction of the ball -- if the speed is greater than zero, the ball moves to the right & if the speed is less than zero, the ball moves to the left.

Make a great sound for when the ball hits the edge of the screen; something like this works!

Add a new function to do that after the end of the **MOVEBALL()** function:

```
FUNCTION BOUNCEBALL()
 --LEFT
  IF BALLX<BALLSIZE THEN
  BALLXDIR=-BALLXDIR
  SFX(0)
 END

 --RIGHT
  IF BALLX>128-BALLSIZE THEN
  BALLXDIR=-BALLXDIR
  SFX(0)
 END

 --TOP
  IF BALLY<BALLSIZE THEN
  BALLYDIR=-BALLYDIR
  SFX(0)
 END
END
```

And then call it from **_UPDATE()**:

```
function _update()
 movepaddle()
 bounceball()
 moveball()
end
```

run what you have, and you should have a ball bouncing up the screen and then down again until it falls off the bottom of the screen.

The new commands we used

```
  sfx(number) -- play a sound
```

5. HIT THAT BALL!

Figuring out whether the ball has hit the paddle is the fiddliest part of the whole game, so bear with it!

We need to check to see if the ball's x position is within the width of the paddle, and whether the ball has gone into the paddle.

We do that using the special **and** word in pico8, the same as you would in English.

Add a **bouncepaddle** function after the **bounceball** function:

```
-- bounce the ball off the paddle
function bouncepaddle()
 if ballx>=padx and
  ballx<=padx+padw and
  bally>pady then
 sfx(0)
 ballydir=-ballydir
 end
end
```

If you like, you can make a different sound for when the ball hits the paddle and play that instead!

Don't forget to call it from **_UPDATE()**

```
FUNCTION _UPDATE()
 MOVEPADDLE()
 BOUNCEBALL()
 BOUNCEPADDLE()
 MOVEBALL()
END
```

If you **RUN** that, you should be able to keep the ball in the screen by moving the paddle (though it'll still disappear when it goes off the bottom!)

6. Can we have our ball back?

When the ball flies off the bottom of the screen, we have to put it back in the middle of the screen. We should really lose a life too -- we'll get to that later though!

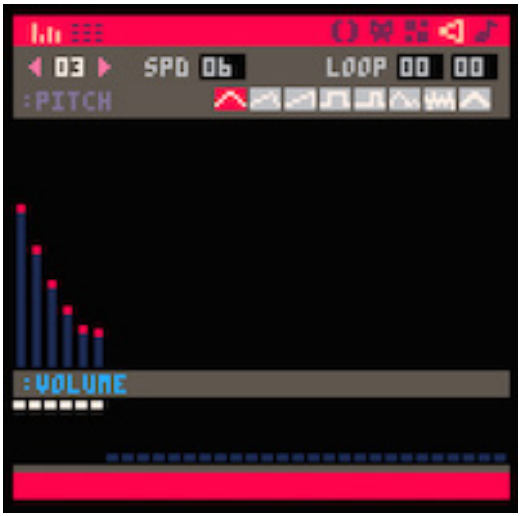
Add a new function, after **MOVEBALL():**

```
FUNCTION LOSEDEADBALL()
 IF BALLY>128 THEN
  SFX(3)
  BALLY=24
 END
END
```

Make sure to call it from **_UPDATE():**

```
FUNCTION _UPDATE()
 MOVEPADDLE()
 BOUNCEBALL()
 BOUNCEPADDLE()
 MOVEBALL()
 LOSEDEADBALL()
END
```

And make a fun sound for it falling off the screen.
Something like this works well:



When you **RUN** this, you should have the best part of a game! Now we need to move on to...

7. Scoring!

Obviously as we have a game, we want to be able to have a hi-score! We'll need a new variable at the top of the program:

```
   SCORE=0
```

Then, every time the ball bounces off the paddle, we'll increase the score. Add a line to the the **BOUNCEPADDLE** function:

```
--BOUNCE THE BALL OFF THE PADDLE

FUNCTION BOUNCEPADDLE()
 IF BALLX>=PADX AND
  BALLX<=PADX+PADW AND
  BALLY>PADY THEN
 SFX(0)
 SCORE+=10 --INCREASE THE SCORE ON A HIT!
 BALLYDIR=-BALLYDIR
 END
END
```

Then draw the score on the screen by adding a line to the `_DRAW()` function:

```
FUNCTION _DRAW()
 --CLEAR THE SCREEN
 RECTFILL(0,0,128,128,3)

 --DRAW THE SCORE
 PRINT(SCORE,12,6,15)

 --DRAW THE PADDLE
 RECTFILL(PADX,PADY,PADX+PADW,PADY+PADH,15)

 --DRAW THE BALL
 CIRCFILL(BALLX,BALLY,BALLSIZE,15)
END
```

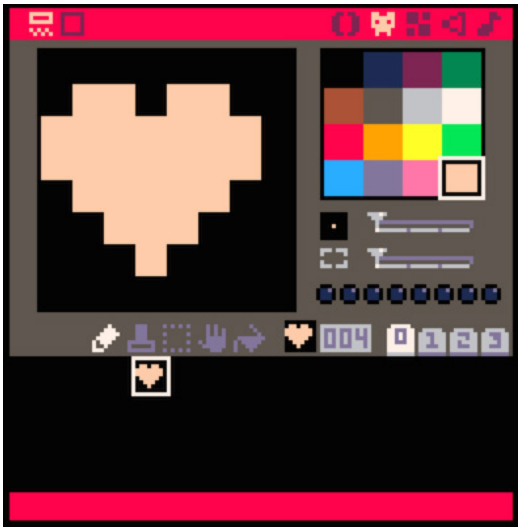
`RUN` that and Bob's your uncle!

The new commands we used

```
   PRINT(MESSAGE,X,Y,COL) --WRITE A MESSAGE ON THE SCREEN.
                            X,Y IS THE BOTTOM-LEFT OF THE FIRST LETTER
```

8. HEARTS

The next piece of the puzzle is limiting the number of lives the player has. We'll need to make a *sprite* (a small picture) to show a heart, so open up the Sprite Editor in PICO-8 and make a sprite like this one:



Remember the sprite number **004** in this case!
Now you can add a new variable at the top of the file: **LIVES=3**
And the code to draw it in **_DRAW()**:

```
FUNCTION _DRAW()
 --CLEAR THE SCREEN
 RECTFILL(0,0,128,128,3)

 --DRAW THE LIVES
 FOR I=1,LIVES DO
  SPR(004,90+I*8,4)
 END

 --DRAW THE SCORE
 PRINT(SCORE,12,6,15)

 --DRAW THE PADDLE
 RECTFILL(PADX,PADY,
  PADX+PADW,PADY+PADH,15)

 --DRAW THE BALL
 CIRCFILL(BALLX,BALLY,BALLSIZE,15)
END
```

(Make sure the number after spr matches the number of the sprite you made!)

The last bit we need to add loses a life when the ball goes off the bottom, and ends the game when the player runs out of lives. We need to make the **LOSEDEADBALL** function a bit more complicated - change it to this:

```
FUNCTION LOSEDEADBALL()
 IF BALLY>128-BALLSIZE THEN
  IF LIVES>0 THEN
   --NEXT LIFE
   SFX(3)
   BALLY=24
   LIVES-=1
  ELSE
   --GAME OVER
   BALLYDIR=0
   BALLXDIR=0
   BALLY=64
  END
 END
END
```

You can make a fun sound for **game over** too! Play it with the **SFX()** function in the section marked **game over.**

The new commands we used

```
SPR(NUMBER,X,Y) --DRAW A SPRITE ONTO THE SCREEN WITH THE TOP-LEFT AT X,Y
```

----ALEX MOLE
----@TheRealMolen

Let's Make Some Music!

When working with the PICO-8 tracker, there are two tools you should get yourself comfortable with.

1. The Sfx Editor

With this, you'll create the individual components of your songs as well as sound effects.
Let's break it down! From top left to bottom right, we got:

1. The currently selected sequence.
2. The speed the sequence will be played at.
3. The loop start and end point.
4. The octave a new note will be set at.
5. The Instrument a new note will be played with.
6. The Volume a new note will be played at.
7. With this, you'll create the individual components of your songs as well as sound effects.

The bottom half shows your notes, with four columns holding eight notes each.

Each note holds the following information:

•a letter, indicating the note's frequency.
•a dot or hash, indicating if it's a half or full tone.
•a grey number indicating the octave.
•a red number, indicating the instrument.
•a blue number, indicating the volume.
•a dark grey number, indicating an effect.

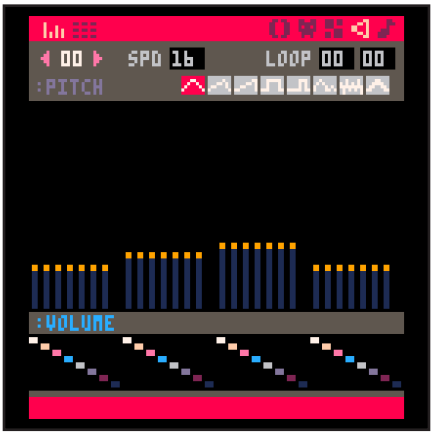
Notes are entered via a standard musical keyboard layout.
The pictured sequence shows all available notes in two octaves.

The corresponding keys are:
•Column 1: 2, 3, 5, 6, 7
•Column 2: q, w, e, r, t, y, u, i
•Column 3: s, d, g, h, j
•Column 4: z, x, c, v, b, n, m

Available effects are:
0 none **1** slide **2** vibrato **3** drop **4** fade_in
5 fade_out **6** arp fast **7** arp slow

1.1 The Sfx Editor-Graph Mode

By clicking on the little icon in the top left, you can switch to the sequencer mode of the sfx editor. The main difference of this mode is that each note is represented graphically instead of numerically, as it would be in a sequencer. Furthermore, you can use your mouse to create notes, adjust note frequencies, octaves, volume and so on.



Personally, I don't use this mode due to it's lack of precision over the hard numbers of the tracker, so I can't tell you much about it.

2. The Pattern Editor

In the pattern editor, you arrange the sequences you made in the sfx editor into songs.
All functionalities from the sfx editor are retained in the pattern editor.

Additionally, there is:

1. A list of patterns with four colored dots each representing the selected sequences there in.
2. The behaviour of the current pattern. Loop start, loop back & stop.
3. The individual sequences of the currently selected pattern. Each pattern can hold up to four.

In your code, you select a pattern number to be played with **MUSIC(NUMBER)**. The appropriate pattern will then play once. Depending on the pattern behaviour set in the top right of each pattern, it will then either play the next pattern in sequence, loop the current pattern indefinitely or stop.

You can edit the notes of your sequence in the same way as in the sfx editor.The only functionalities not present in the pattern editor are the speed and loop point settings.

3. Making Music

Now that you know the functionalities of the PICO-8 tracker, you can start making some sick chiptune!

Here is a list of hotkeys that might be useful:

- Play / Stop: Space
- Enter note: q2w3er5t6y7ui zsxdcvgbhnjm
- Delete note: Backspace(Alternatively, set volume to 0)
- Increase / decrease pattern, speed etc. by 4:
  Shift + left click / right click
- Sfx editor - set all notes to instrument / effect:
  Shift + click instrument / effect
- Release looping sequence : A

Here's a couple tips to get you going:

- The drop effect, 3, is good for bassdrums
- The noise instrument, 6, is good for snares and hihats
- A full song should consist of at least an intro, a main loop a and a final, surprising, loop back pattern.
- A continuous loop should consist of at least 4 different patterns to not sound samey. The more, the better!
- When making your music, keep in mind that you only have four channels to play your music+ sound effects on.
- A bass with octave 0-1, a middle to high lead with octave 1-3

and drums are a good basis for a full sound.
• Play around with many combinations of effects, instrument, octaves, volume and frequencies!
• Change the default speed of your songs. You can make sick drum loops by having a high speed and far apart notes.

4. Playing Music

```
PLAYING=0
MUSIC(0)

FUNCTION _UPDATE()
END

FUNCTION _DRAW()
CLS()
PRINT("TRACK"..PLAYING)
END
```

In order to play the music you've made inside of your game, all you need to do is type `MUSIC(n)`, where n represents the number of the pattern, shown as element **(1)** in the Pattern Editor screenshot in section 2 of this tutorial. You can play any pattern you want, just note that empty patterns won't play anything back, of course. The rest of the code shown is added because PICO-8 would otherwise assume this an empty program and quit it instantly.

To avoid this, we have added a little code to clear the screen and show us which track is playing.

Line by line:

`PLAYING=0` is set as a variable that is 0.
`MUSIC(0)` is called once to start playing our music.
`FUNCTION _UPDATE()` and `FUNCTION _DRAW()` cause otherwise our game had nothing to do and wouldn't run.

You can actually just add both of these functions, leave them empty, and PICO-8 will play your music just fine. The rest is just extra.

CLS() clears the screen every frame, so the code in the next line will only be visible once.

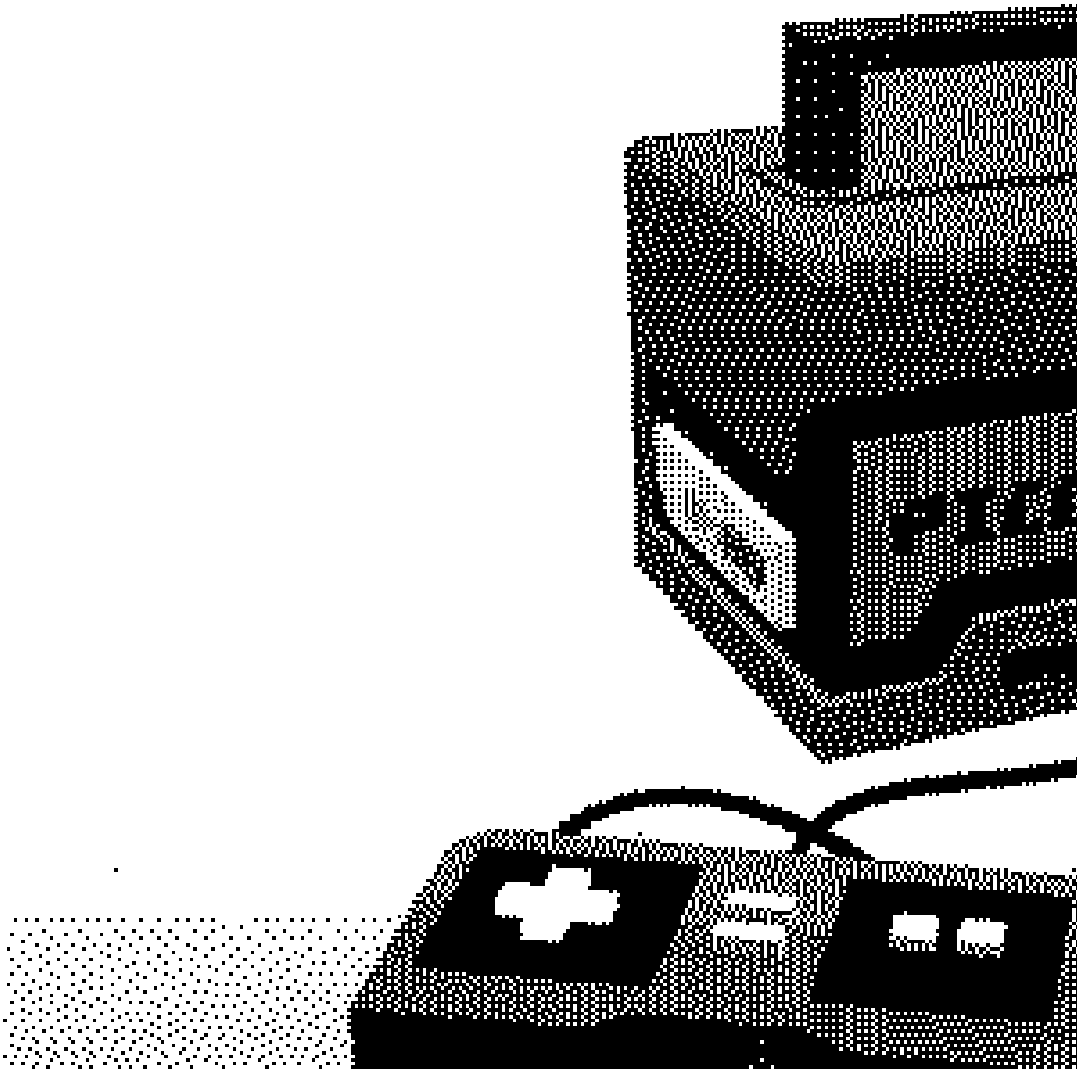
PRINT("TRACK"..PLAYING) writes TRACK to the screen along with the variable **PLAYING**, which holds our Track number.

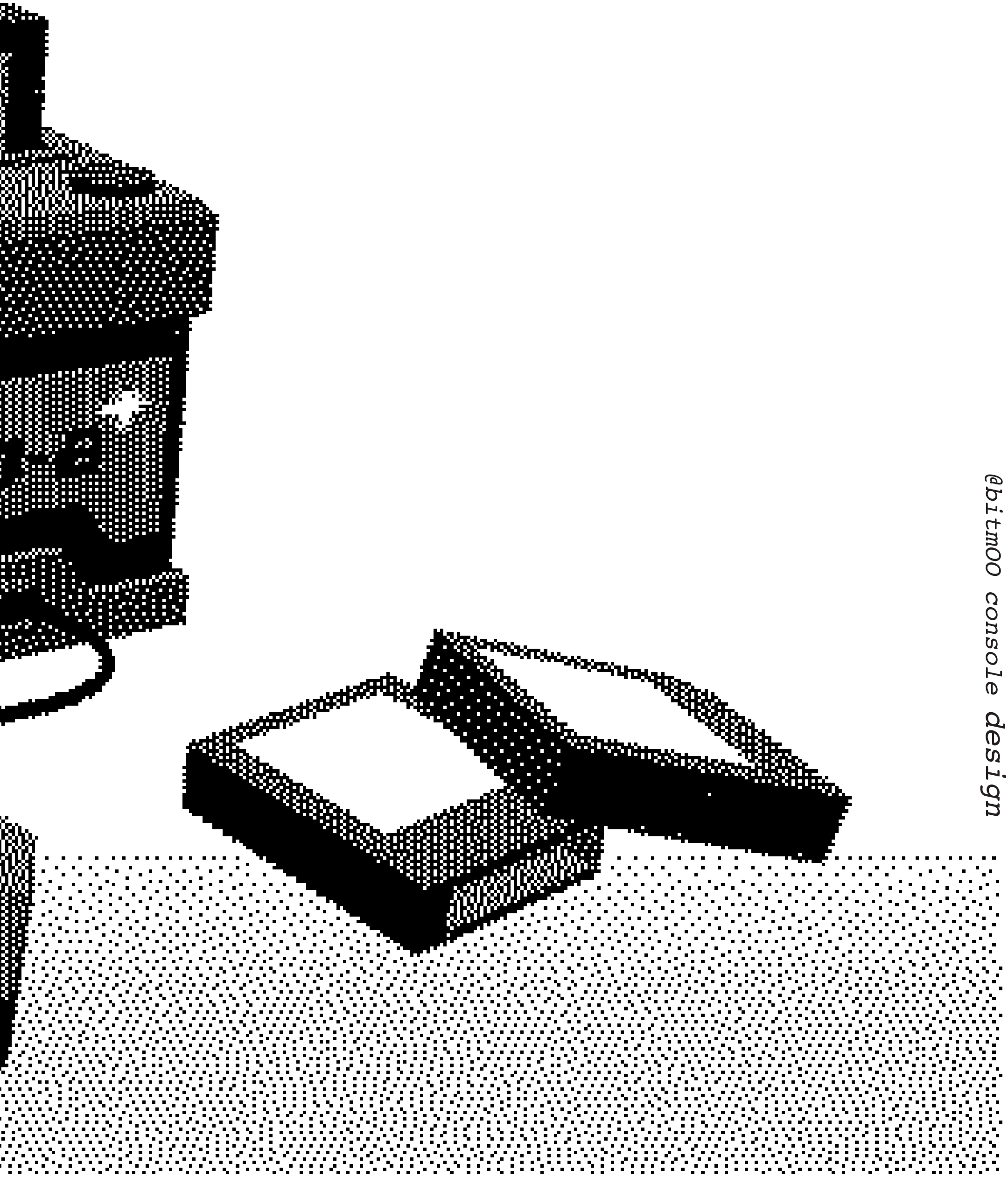
PATTERN BEHAVIOUR:

Depending on the pattern behaviour set in the Pattern Editor, shown as element **(2)** in the before mentioned screenshot, the **MUSIC** function will either play the next pattern in sequence once your selected pattern was played in full once if you've selected neither loop start, loop back or end. If you've selected the third symbol, end, the music will stop after the pattern was played. If you've selected loop back, the second symbol, the last pattern before the currently played pattern that had loop start activated will be played next. That way you can loop bigger sets of patterns, by giving the last in sequence a loop back, and the first a loop start. If you set both loop back and loop start on any pattern, it will repeat until it is stopped. That's it! Now combine this knowledge with other tutorials and make some great PICO-8 programs!

Thanks for reading, I hope I could help you understand trackers a bit better.
If you have any more questions, tweet at me @pizzamakesgames.

--FELIKS BALZER



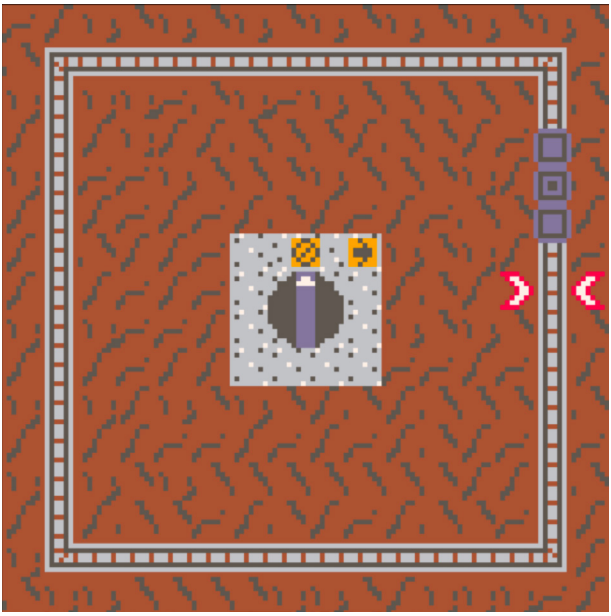


@bitm00 console design

TOY TRAIN

I always loved watching my grandpa's toy trains zip around their tracks, and I'm working on a complicated train game right now, but I decided to try and contribute something simpler to this zine, in the same vein.

This is not so much a game as a little toy - a train that zips around its little track, perfectly negotiating right-angle turns. Very impressive engineering on the part of the toymakers!



The map

We're going to start with an `_INIT()` method, which is called whenever you use the **RUN** command (or **CTRL-R**, or **CMD-R**). Think of it as the setup for the game.

```
FUNCTION _INIT()
CLS()
SWITCH_STATE=0
TRAIN={{64,8},{72,8},{80,8}}
END
```

The really handy thing about `_INIT()` is that you can run it again at any time and it will seem like the game was reset. So try to put all your important setup stuff in there. In this case I just have two variables, `SWITCH_STATE` and `TRAIN`. `SWITCH_STATE` just tells me whether the switch for the train is on `(1)` or off `(0)`. We always want to start with the train off so the player gets the satisfaction of turning it on. And then `TRAIN` is a table, which is basically a list of lists. Each pair of values is a two-element list, representing a segment of the train. The first value is the `X` position, and the second value is the `Y` position. You can see at a glance there will be three segments in this train, but you can add as many as you like. Just be careful where you're positioning them. We'll see why in a moment.

```
FUNCTION MOVE_SEGMENT(S,DIR)
SPD=DIR*2
IF(S[2]==8)--TOP SIDE
THEN
     IF(S[1]==112)--TOP RIGHT
     THEN
          S[2]+=SPD
     ELSE
          S[1]+=SPD
     END
ELSE
IF(S[1]==112)--RIGHT SIDE
THEN
     IF(S[2]==112)--BOTTOM RIGHT
     THEN
          S[1]-=SPD
     ELSE
     S[2]+=SPD
     END
ELSE
IF(S[2]==112)--BOTTOM SIDE
THEN
     IF(S[1]==8)--BOTTOM SIDE THEN
```

```
            S[2]-=SPD
      ELSE
            S[1]-=SPD
      END
ELSE
IF(S[1]==8)--LEFTSIDE
THEN
      IF(S[2]==8)--TOPLEFT
      THEN
            S[1]+=SPD
      ELSE
            S[2]-=SPD
      END
END
END
END
END
END
```

The **MOVE_SEGMENT** function will let us advance one segment of the train (which we call **S**) along the track, taking into account the turns it will need to make along the way.

It does this by checking the co-ordinates of **S** and deciding whether it needs to move in the X direction or the Y direction at this time. At the beginning, **S** has a Y/vertical position of **8**, and an X/horizontal position NOT matching 112.

Therefore when the **SWITCH_STATE** is **1**, we will be increasing the X position of **S** by 2 every execution until X reaches 112 (indicating the top-right corner).

When **S** reaches the top-right corner, its Y/vertical position is 8, and its X/horizontal position is 112. Now we are executing slightly different parts of the code. Instead of increasing the X position (moving right), we must move down. So the train's Y position is increased, sending it down the screen until it reaches the next corner. We repeat this method for the other three corners, and the result is that the train segment is moved along in one dimension until it reaches a limit, at which point it begins moving in the other dimension, and so on.

```
FUNCTION ADV_SWITCH()
IF(SWITCH_STATE<1)
THEN
     SWITCH_STATE+=1
ELSE
     SWITCH_STATE=0
END
END
```

This function moves the switch to the next position (ADV-ances it). You can use this to manage as many switch states as you like, just by increasing the number in the IF statement. Having **1** in that **IF** statement allows us to have two switch states, **2** would allow three states, etc. Different states could do very different things!

```
FUNCTION MOVE_TRAIN()

FOR T IN ALL(TRAIN)
DO
     MOVE_SEGMENT(T,SWITCH_STATE)
END

END
```

In the **MOVE_TRAIN** function, we do something very simple: loop through all the train segments, and move each one. **FOR T IN ALL(SOME_LIST)** will let you do some operations (inside the **DO...END**) on each element, temporarily referred to as **T**. In this case, we've already done the hard work, so we're just going to call that other function, **MOVE_SEGMENT**, for each. We pass in **SWITCH_STATE** as well, because if **SWITCH_STATE** is **0**, we don't want the train moving. Alternatively we could just check **SWITCH_STATE** and only do the **FOR** loop if its value is **1**, but this way we could potentially add more stuff to the **MOVE_SEGMENT** function later (see below).

```
FUNCTION _UPDATE()
IF(BTNP(4))
THEN
    ADV_SWITCH()
END
MOVE_TRAIN()
END
```

The **_UPDATE** function, as you have already seen in a previous lesson, is called every time the game updates itself (30 times a second). The **BTNP** function checks if a given button has *just* been pressed, this very frame, and will wait a few frames before activating again if the button is held down. So you can use it very neatly for switches. Pressing Player 1's button 4 (usually Z) will call the **ADV_SWITCH** method shown above, and turn the switch. **MOVE_TRAIN** is something we want to call whether a button is being pressed or not, so we put that outside the **IF...END** block.

```
FUNCTION DRAW_TRAIN()
LOCAL LEN=COUNT(TRAIN)
FOR T=1,LEN
DO
    IF(T==1)
    THEN
        SPRITE=11
    ELSE
    IF(T==LEN)
    THEN
        SPRITE=13
    ELSE
        SPRITE=12
    END
    END
    SPR(SPRITE,TRAIN[T][1],TRAIN[T][2])
END
END
```

Now we have to actually draw the train. This is usually the easy part, it's just a little less neat because I wanted to have

special starting and ending segments. This time we use a different sort of FOR loop, which starts out with T set to 1, and continues, adding one each time, until T is equal to the length of the TRAIN list. This allows us to not only do things with each segment, but also know where in the list we are easily. In this case, the first segment/element of the list is the last segment of the train, so we want to put a slightly more boring sprite there. The last element of the list is the first segment of the train, so we want to make that one more interesting. And all the ones in between will have the same sprite for each of them.

```
FUNCTION DRAW_SWITCH()
IF(SWITCH_STATE==1)
THEN
SSPR(0,16,16,16,56,56,16,16)
ELSE
SSPR(64,16,16,16,56,56,16,16)
END
END
```

This function checks the SWITCH_STATE and picks the correct switch image to draw based on that. The switch images are two sprites wide by two sprites high (16x16px). We use the SSPR ('stretch sprite') function to make this work easily, although we could also just draw each sprite one by one. The first two arguments tell the program where to start drawing (x=0, y=16 if the switch is on). The next two arguments tell the program how big the area we want to pull from the sprite sheet is. As we saw above, that's going to be 16 across and 16 down from the original point. Then we have to give an X and Y coordinate to start drawing the sprite, and again the size of the area we want to fill. If you doubled the last two arguments, to (32, 32), you would draw the switch twice as large. SSPR is fun to play around with, but it's also useful even if you don't stretch sprites at all.

```
FUNCTION _DRAW()
MAP(0,0,0,0,16,16)
DRAW_SWITCH()
DRAW_TRAIN()
END
```

Sprites

This function is called every time **_UPDATE** is called (unless PICO-8 is running slowly). It uses the **MAP** function to pull a big section of the map data and display it on screen. The first two arguments are where to start drawing on the PICO-8 screen (where 0,0 is the top left corner), the second pair of arguments tell it where in the map data to start, and the third pair of arguments tell it how many sprites in each direction to draw. One full screen is 16x16 sprites, or 128x128 pixels. So it pulls out 16x16 sprites, based on the map data, and draws them to the screen before doing anything else. Then it calls the **DRAW_SWITCH** and **DRAW_TRAIN** functions which we looked at above. (These things are drawn in ordered layers, so if, for example, you wanted to draw a little bridge over the train, you would draw it *after* the train.)

We saw above that **MAP** draws sprites based on map data, but it might not be clear what is actually going on. The 'map' which you can see in the first screenshot is only a long list of numbers, telling the program which sprite to put where. So the first row of sprites in the map data would look something like "02 03 19 18 03 02" and so on. It just goes to the sprite sheet (second screenshot) and picks out the sprite at that position. This is particularly cool because you can at any time change your sprite 02 and you will instantly see the change in all your maps.

Mapping and sprite stuff is a whole thing in itself, but I enjoy playing with it very much. It's a fun way to be kind of working on your game without getting too technical.

Homework:

- There is at LEAST one way to neaten up the **MOVE_SEGMENT** function. Try to make it smaller.
- Now you've neatened something up, make things messier by trying to make the train go the other way.
- You could probably add a third switch position to make the train go, stop, and reverse.
- For serious cool points, try to make the train slow down after it's turned off, rather than stopping dead. You could even make it 'chuff' up to speed when turned on.
- Change any (or all) of the sprites to change the feel of the game, or just to make it look better!
  What other things could use the same mechanics but with different visuals?

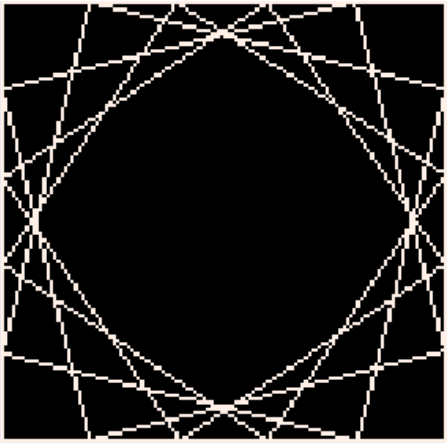
Thanks for reading! I hope this has been somewhat educational for you. Let me know if you enjoyed this and especially if you made anything sweet using my game as a base!
Cheers,

James (PROGRAM_IX)

You can find the original version here :
http://www.lexaloffle.com/bbs/?tid=2253

GEODEZIK

@aliceffekt
http://xxiivv.com



```
FRAME=0

FUNCTION _UPDATE()
FRAME+=1

 --LOOP AT F127

 IF FRAME>127 THEN
 FRAME=0
 END
END

FUNCTION _DRAW()

 RECTFILL(0,0,127,127,0)
 I=0
 WHILE(I<20)DO
 E=(I*0.5)
 LINE(0,(FRAME*E),127-(FRAME*E),0,7)
 LINE((FRAME*E),127,0,(FRAME*E),7)
 LINE(127,127-(FRAME*E),(FRAME*E),127,7)
 LINE(127-(FRAME*E),0,127,127-(FRAME*E),7)
 I+=1

 END

END
```

SMOKE PARTICLE

By **Mozz** http://mozz.itch.io/

This tutorial assumes that you know the basics of a PICO-8 program, including the functions `_INIT()`, `_UPDATE()` and `_DRAW()`. Particle effects are the most popular method to create dust and smoke and sparkles in a game: they allow for game worlds to feel more alive.Basically, a particle effects system is one which creates a series of individual particles that have some sort of physics applied to them. In this tutorial, the particles will be a series of circles, that will grow based on the life of the particle, emulating the dissipation of smoke and clouds.

For this effect we will need to write three functions: one to make the smoke; one to move the smoke; and one to draw the smoke.

1.Making the Particle

When you make the smoke you want to think about what variables you might need to have in the future and what you want to have control over when you create the smoke. For this example I decided I wanted to set the initial x-y values, color of the smoke and the starting size of the smoke particle.
These variables go in the function parameters. The rest of the variables in the function will hold the current x and y value of the particle, as well as variables that will control the physics later in the program. the letter `D` in front of a variable is mathematical shorthand for delta, or "a change in". So `DX` would be "the change in `X`" and will set the velocity of the particle in that direction.
This will help us remember what the variable is for later.
Variable `T` and `MAX_T` set how long a particle lasts in seconds, `WIDTH` and `WIDTH_FINAL` will make the particle start out a certain size and grow over time and ddy is the change in `DY`, and will simulate acceleration on the `Y` axis.
We then add the list "`S`" to a list "`SMOKE`" that will contain every particle we create in the game. So in the end we will have a list named "`SMOKE`" that contains a number of lists that holds the data for each instance of smoke particle.

```
FUNCTION MAKE_SMOKE(X,Y,INIT_SIZE,COL)
    LOCAL S={}
    S.X=X
    S.Y=Y
    S.COL=COL
    S.WIDTH=INIT_SIZE
    S.WIDTH_FINAL=INIT_SIZE+RND(3)+1
    S.T=0
    S.MAX_T=30+RND(10)
    S.DX=(RND(.8).4)
    S.DY=RND(.05)
    S.DDY=.02
    ADD(SMOKE,S)
    RETURN S
END
FUNCTION _INIT()
    SMOKE={}
    CURSORX=50
    CURSORY=50
    COLOR=7
END
```

2.Moving the particle

Now that we have a function that will add a smoke particle to the game, we need to define how it moves. In **MOVE_SMOKE**, we will change the **X** and **Y** values of the particle based on variables we set when we created it. With each step of **MOVE_SMOKE**, we will first check to see if the particle has reached its max life, and if it has we remove it from "**SMOKE**". Then we grow the width of the particle if we are within 15 steps of the end of its life (not to exceed the **WIDTH_FINAL** variable). Then we apply the **DX** and **DY** values as well as **DDY** (which stands in for gravity) to future calls of **DY**.
In **_UPDATE** I decided just to create a smoke particle each step by calling **MAKE_SMOKE**, just for testing. The program also check to see if you have pressed one of the arrow keys and changes the "**CURSORX**" and "**CURSORY**" values based on the player's input. We use those values to determine where we create new particles. Color is set randomly when you press "**BUTTON1**".

```
function move_smoke(sp)
if (sp.t>sp.max_t)then
    del(smoke,sp)
end
if (sp.t>sp.max_t15)then
    sp.width+=1
    sp.width=min(sp.width,sp.width_final)
end
sp.x=sp.x+sp.dx
sp.y=sp.y+sp.dy
sp.dy=sp.dy+sp.ddy
sp.t=sp.t+1
end
function _update()
foreach(smoke,move_smoke)
if btn(0,0)then cursorx=1 end
if btn(1,0)then cursorx+=1 end
if btn(2,0)then cursory-=1 end
if btn(3,0)then cursory+=1 end
if btn(4,0)then color=flr(rand(16))end
make_smoke(cursorx,cursory,rnd(4),color)
end
```

3. Drawing the particle

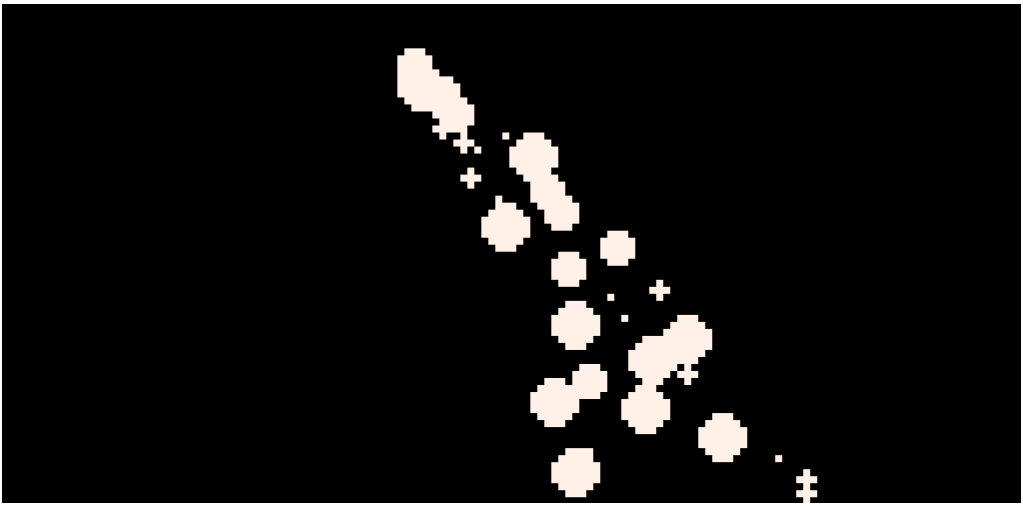
Ok, we've created a particle and it can move but we will need to get it to draw to the screen before we can see it. The method for this is fairly simple: Each time the **_draw** function is called, **foreach()** will call the function **draw_smoke** for each entry in the list "**smoke**". **draw_smoke** will then use the values of the current particle to draw a filled circle at its current x and y values, with its current width and color.

```
function draw_smoke(s)
circfill(s.x,s.y,s.width,s.col)
end
function _draw()
cls()
foreach(smoke,draw_smoke)
end
```

4.Further Steps

You can now play with different variables to change the effect by increasing the velocity and gravity of spawned particles to create a different feeling. You can also create an emitter that creates different kinds of particles for complicated layered effects. And if you would like the particle to be something more interesting than just a circle, replace `circfill()` with `spr()` and use your own sprites.

For more examples of what you can do with particles, check out the Advanced Particle System Library, posted by Viza, which you can find through this QR code
http://www.lexaloffle.com/bbs/?tid=1920

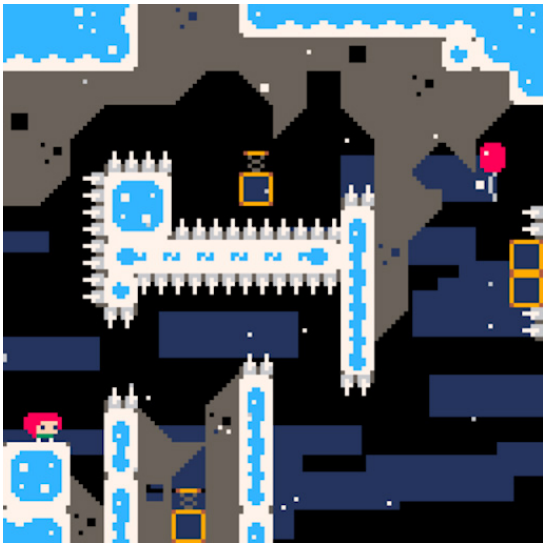


Welcome to PICO 8!

@terrycavanagh

New to PICO-8? Here are a few carts that are a great place to start:

CELESTE



Author's comment:
"We used pretty much all our resources for this. 8186/8192 code, the entire spritemap, the entire map, and 63/64 sounds. Let us know what you think!"

PICO 8's killer app. If you only play one PICO-8 game, make it this one.

http://www.lexaloffle.com/bbs/?tid=2145

Stories at the Dawn

A minimal story platformer with four endings. A great example of what can be done really well in PICO 8's constraints.

http://www.lexaloffle.com/bbs/?tid=1919

PAT Shooter

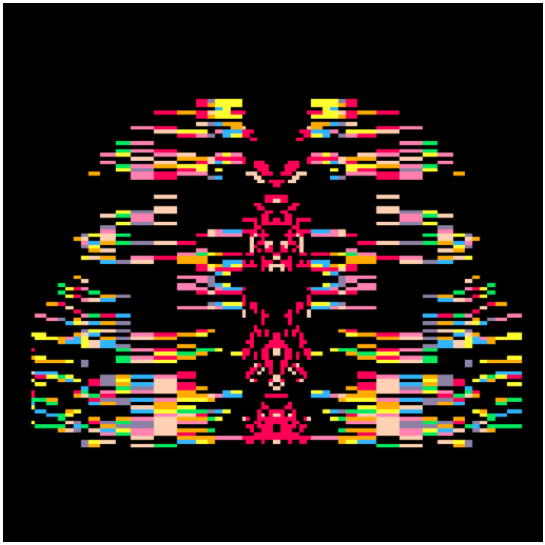
Author's comment:
when asked "What does P.A.T. stand for?": "Nothing. Actually I was hoping nobody will ask."

http://www.lexaloffle.com/bbs/?tid=1867

Transdimensional Butterfly

There are loads of carts like this on the BBS, a bit like 90s graphics demos with music.
This one, by PICO-8 creator Lexaloffle, is gorgeous, and has a great soundtrack too.

http://www.lexaloffle.com/bbs/?tid=2109



The Tower of Archeos

Author's comment:
"Reach the 8th floor to fight Archeon."

A fantastically crafted puzzle game, by the prolific Benjamin Soule (who also wrote PAT Shooter along with several other PICO-8 games).

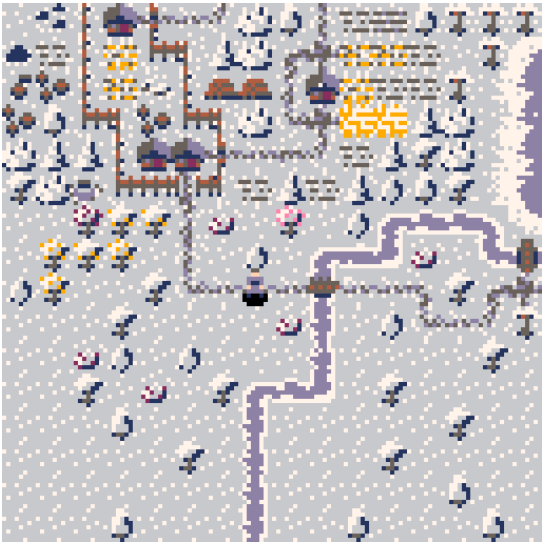
http://www.lexaloffle.com/bbs/?tid=1907

Tempest

Author's comment:
"Not much of a game yet, but working on a little adventure/sim/survival game. Having a lot of fun!"

Build a shelter and find food to survive. Still a work in progress, but very promising!

http://www.lexaloffle.com/bbs/?tid=2186



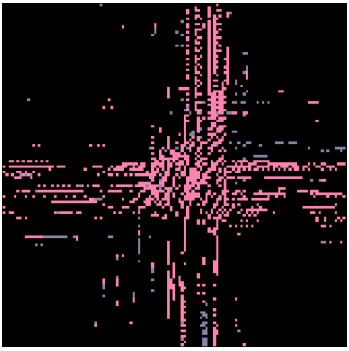
WORMWORMWORMWORM

Author's comment:
"INSPIRED BY DIARY OF UNSPOKEN TRUTHS, ARTIST, AND I, ROBOT BY NIALL, MICHAEL, AND PERSON"

PICO-8 has been a pretty great source for glitch art. This one is particularly good!

http://www.lexaloffle.com/bbs/?tid=2006

mtrx



Author's comment:
"an endless running painting with noise"

Enjoyed WORM, but found it had too much interaction and vowels? Try jph's mtrx!

http://www.lexaloffle.com/bbs/?tid=1936

Delia Mute in Grave Grotto

Author's comment:
"small grid roguelike. each inventory slot can only be used in the direction it is picked up in."

http://www.lexaloffle.com/bbs/?tid=2166

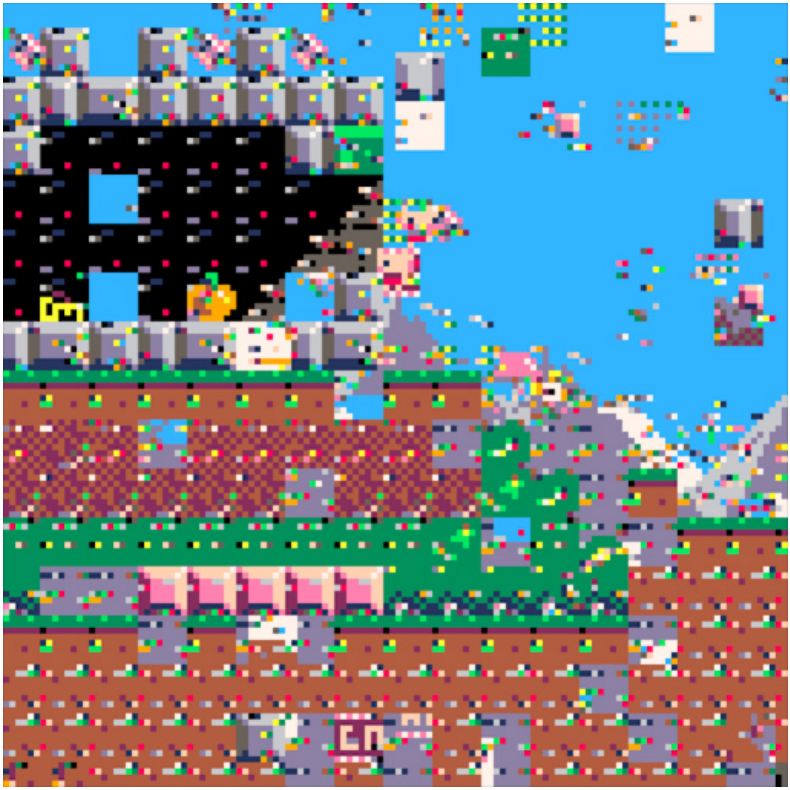
The Adventures of Jelpi (with Corrupt mode):

[included demo game]

Author's comment:

"I thought it might be nice to have a glitch monster who pokes random values into core memory -- you have to get out before the level is no longer completable. Although -- there's always the chance it will poke an exit index into the map right in front of you. Or: a superhero game set in a city riddled with corruption. (wakawaka)"

Everyone with PICO 8 should try this - load up Lexaloffle's included demo game, The Adventures of Jelpi. Right at the start of code, look for the variable corrupt_mode, and set it to true. How many times you can get across the stage?



Some other cool stuff to check out:

=-=-=-

Stray Shot
http://www.lexaloffle.com/bbs/?tid=1923

Endless Train
http://www.lexaloffle.com/bbs/?tid=2122

Random Sound Generator
http://www.lexaloffle.com/bbs/?tid=1965

Video Poker
http://www.lexaloffle.com/bbs/?tid=2020

Piano Simulator
http://www.lexaloffle.com/bbs/?tid=2208

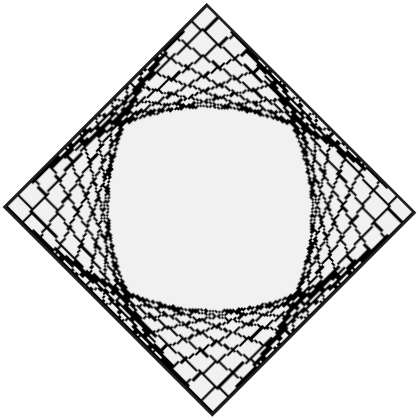
Duangle 2015 intro
http://www.lexaloffle.com/bbs/?tid=1984

Thopter Escape
http://www.lexaloffle.com/bbs/?tid=2196

Bounce
http://www.lexaloffle.com/bbs/?tid=1947

Puzzle Cave:
Raiders of the Lost Potato:
http://www.lexaloffle.com/bbs/?tid=2039

Sumo Pico
http://www.lexaloffle.com/bbs/?tid=2191



PICO-8