PICO-8 ZINE #2

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

OCTOBER 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 @johanvinet
Special thanks to @nerial, @dan_sanderson and @lexaloffle

Hello everyone,

The first Pico-Zine was an incredible, unexpected success. People from all over the world downloaded the PDF and I have had to repeatedly flood my local mailing office with hundreds of paper copies. This is an amazing achievement for the Pico-8 community, both the newcomers and those who shared their knowledge and code.

In Pico-8, there is no real reason to make a distinction between the player, the consumer and the maker. If you can play, you can also make: the source and assets are always a single image you can share and change as much as you want to.

Through its well-thought-out constraints and limits, Pico-8 breaks a lot of frontiers, and that is what is amazing about it. It's a tool to make games in the largest and most inclusive way possible.

You know what? I have never been very knowledgeable about coding. I messed around with BASIC when I was younger, but since then code has lost most of its appeal to me, it became obfuscated, complicated and not-so-logical. From an outsider's point of view, code is very similar to magic. It's an arcane formulae rendering the most amazing interactions and stories. It's necessarily encrypted, protected and out of reach. It's the sacred language of an illuminated cast, the developers.

Pico-8 has broken that feeling. For the first time, I have had this amazing ability to "follow the trail" of code, directly and instantaneously. For example, I am able to look at a piece of code made by a well-known developer and change the way a character jumps.

Compared to other game engines, Pico-8 has a very specific philosophy: it's a "Fantasy console" and a walking utopia of making. This is because of two features : it's an all-in-one swiss army knife style of game-making (everything is done within the space of the engine, no plug-in, no dependencies) and it's an extremely shareable format (if you can make you can share). This means that

Pico-8 is the best tool to build an open community around a shared knowledge of making for the sake of making.

Thanks for reading !

Arnaud DE BOCK

@lucyamorris handheld mockup

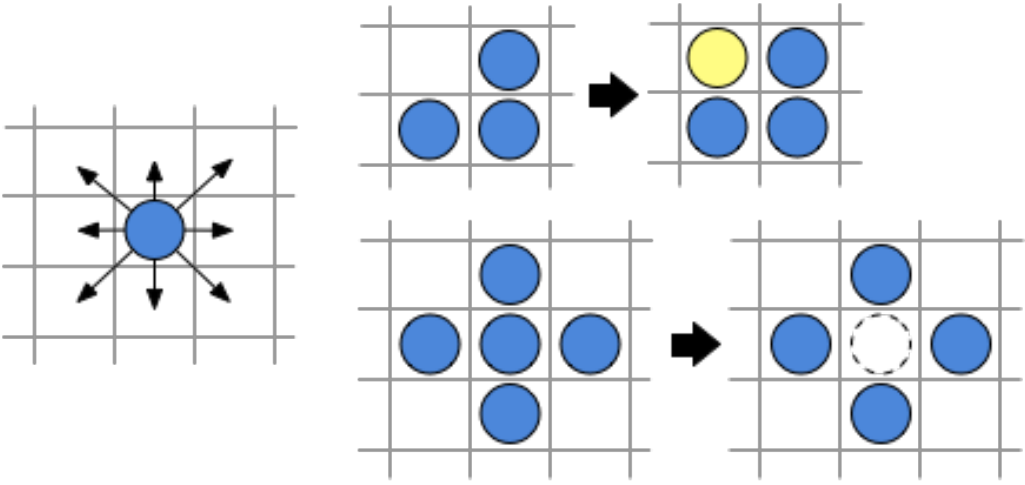
GAME OF LIFE

Mathematician John Conway published a description of his “Game of Life” in the October 1970 issue of Scientific American, spawning cellular automata as a field of research. Ever since, Game of Life has been a staple of recreational mathematics and computation. Let’s build one for PICO-8!

The Rules of the Game

The Game of Life takes place on a grid of cells, where each cell is either alive or dead. Given an arrangement of alive cells, the next generation of cells is computed from simple rules.

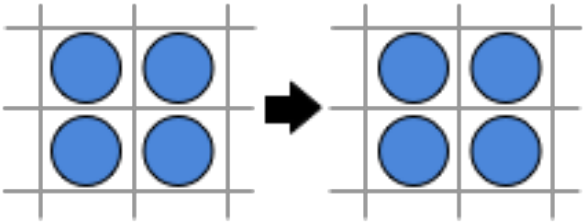
1. A cell has eight neighbors on the grid, including the diagonal directions.
2. If an alive cell has either two or three alive neighbors, then the cell survives to the next generation. Otherwise, it becomes a dead cell (due to “starvation” or “overcrowding”).
3. If a dead cell has exactly three alive neighbors, then the cell becomes alive in the next generation (“reproduction”). Otherwise, it remains a dead cell.



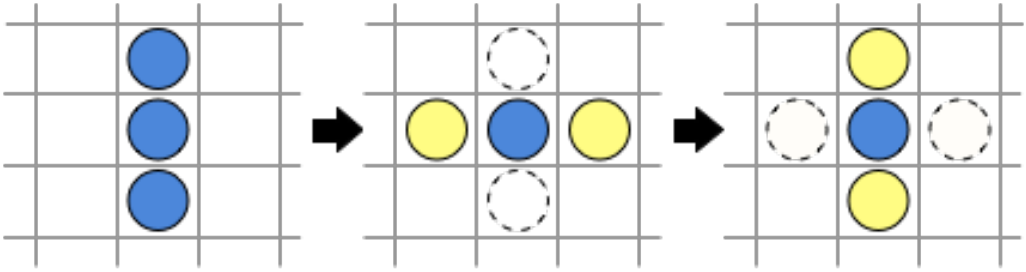
The rules of the Game of Life.

These simple rules produce a wide variety of patterns over multiple generations.

Some patterns, such as the “block,” are stable, and do not change from one generation to the next. Some patterns oscillate over a period, such as the “blinker”. Some patterns repeat themselves but in a different position, giving the appearance of a moving organism.



The “block” pattern is stable from one generation to the next.



The “blinker” pattern oscillates in place over two generations

Even simple patterns can have unpredictable behavior. The “r-pentomino” starts with five alive cells, then explodes in all directions, refusing to stabilize until 1,103 generations have elapsed.

Storing and Drawing the Board

Let’s start with an implementation that’s easy to understand. We store the current state of the board in a two-dimensional array, where each element is a cell that is either alive (1) or dead (0). We store numbers (instead of, say, Booleans) to make it easy to count a cell’s neighbors by adding up the values.

We need two boards, one to represent the current state and another to represent the next generation. We can represent this as two arrays, and switch which one is considered to be the current board at the end of each generation.

The following code sets up the boards, draws a blinker on the first board, then draws the first board repeatedly:

```
ALIVE_COLOR = 7
WIDTH = 128
HEIGHT = 128

BOARD_I = 1

BOARDS = {{},{}}

FOR Y=1,HEIGHT DO
  BOARDS[1][Y] = {}
  BOARDS[2][Y] = {}
  FOR X=1,WIDTH DO
    BOARDS[1][Y][X] = 0
    BOARDS[2][Y][X] = 0
  END
END

-- DRAW A BLINKER
BOARDS[1][60][64] = 1
BOARDS[1][61][64] = 1
BOARDS[1][62][64] = 1

CLS()

WHILE TRUE DO
  FOR Y=1,HEIGHT DO
    FOR X=1,WIDTH DO
      PSET(X-1,Y-1,BOARDS[BOARD_I][Y][X] * ALIVE_COLOR)
    END
  END
  FLIP()
END
```

Run this code. A blinker appears, but does not evolve.

The `WHILE TRUE` loop runs forever. It walks over the entire board and draws a single pixel for each cell at the corresponding location on the screen. The `FLIP()` function call tells PICO-8 to copy its screen data to the actual display. (When using the `_UPDATE()` and `_DRAW()` special functions to implement a game loop, `FLIP()` is called automatically after `_DRAW()`.)

The Next Generation

To calculate the next generation, we iterate over the cells of the current board. For each cell, we count the neighbors by reading from the board array, then set the corresponding cell on the next board according to the rules.

We have a small problem on the edges of the board. Consider what happens when we try to read the top neighbor for a cell on the top edge, like this:

```
CELL = BOARDS[BOARD_I][0][1]
```

PICO-8 arrays have indexes starting at 1, not 0. In PICO-8, when you access an array index out of range, the value is nil. Because `BOARDS[BOARD_I][0]` is `NIL` , treating that value like an array by trying to access an element (`[1]`) is a runtime error.

In this implementation, we'll treat the cells past the edges as dead cells. Let's use a helper function to read cells from the board that returns 0 for coordinates that are out of range:

```
FUNCTION GET(BI,X,Y)
  IF ((X < 1) OR (X > WIDTH) OR (Y < 1) OR (Y > HEIGHT)) THEN
    RETURN 0
  END
  RETURN BOARDS[BI][Y][X]
END
```

At the end of the `WHILE TRUE` loop, add the following code to calculate the next generation:

```
OTHER_I = (BOARD_I % 2) + 1
FOR Y=1,HEIGHT DO
  FOR X=1,WIDTH DO
    NEIGHBORS = (
      GET(BOARD_I,X-1,Y-1) +
      GET(BOARD_I,X,Y-1) +
      GET(BOARD_I,X+1,Y-1) +
      GET(BOARD_I,X-1,Y) +
      GET(BOARD_I,X+1,Y) +
      GET(BOARD_I,X-1,Y+1) +
      GET(BOARD_I,X,Y+1) +
      GET(BOARD_I,X+1,Y+1))
    IF ((NEIGHBORS == 3) OR
        ((BOARDS[BOARD_I][Y][X] == 1) AND NEIGHBORS == 2)) THEN
      BOARDS[OTHER_I][Y][X] = 1
    ELSE
      BOARDS[OTHER_I][Y][X] = 0
    END
  END
END
BOARD_I = OTHER_I
```

`BOARD_I` is the index of the current board in the boards array, either `1` or `2`. `OTHER_I` is the index of the other board. The expression `(BOARD_I % 2) + 1` means "take the remainder of dividing board_i by 2, then add 1," which does what we want: if `BOARD_I` is `1`, then `OTHER_I` is `2`, and vice versa.

Run this code. The blinker evolves, oscillating between its two states.

For a more interesting display, replace the code that draws the blinker with the following code, then run the program:

```
-- DRAW AN R PENTOMINO
BOARDS[1][60][64] = 1
BOARDS[1][60][65] = 1
BOARDS[1][61][63] = 1
```

```
BOARDS[1][61][64] = 1
BOARDS[1][62][64] = 1
```

The Line Buffer Method

The implementation above stores two boards and copies the active board to the screen for each iteration. The purpose of the second board is to keep a record of the neighbors around the current cell as we go down the board calculating updates. If we only had one board, changing a cell would interfere with the calculation for the cells immediately beneath and to the right of it. But we don't need to keep the entire previous board to avoid this. Instead, we could just remember the original state of the previous line and the current line as we walk down the board. This technique uses less memory than storing two boards.

We can save more memory and some time by using the screen itself as storage for the board data. The `PSET()` and `PGET()` functions can set and read pixels on the screen. Combined with the two-line buffer, this technique requires no additional storage, and does not require copying a board array to the screen because updates are written to the screen directly.

Here is a version that uses the line buffer method and writes directly to the screen:

```
ALIVE_COLOR = 7
WIDTH = 128
HEIGHT = 128

PREV_I = 1
LINE_I = 2
LINES = {{},{}}

CLS()

-- DRAW AN R PENTOMINO
PSET(64,60,ALIVE_COLOR)
```

```
PSET(65,60,ALIVE_COLOR)
PSET(63,61,ALIVE_COLOR)
PSET(64,61,ALIVE_COLOR)
PSET(64,62,ALIVE_COLOR)

FUNCTION GET(X,Y)
  IF ((X < 1) OR (X > WIDTH) OR (Y < 1) OR (Y > HEIGHT)) THEN
    RETURN 0
  END
  RETURN PGET(X,Y)
END

FUNCTION GETB(I,X)
  IF ((X < 1) OR (X > WIDTH)) THEN
    RETURN 0
  END
  RETURN LINES[I][X]
END

WHILE TRUE DO
  FLIP()

  -- CLEAR LINE BUFFER
  FOR X=1,WIDTH DO
    LINES[1][X] = 0
    LINES[2][X] = 0
  END

  FOR Y=1,HEIGHT DO
    -- SWAP LINE BUFFERS
    PREV_I = LINE_I
    LINE_I = (LINE_I % 2) + 1

    -- COPY CURRENT LINE TO BUFFER
    FOR X=1,WIDTH DO
      LINES[LINE_I][X] = PGET(X,Y)
    END
```

```
    FOR X=1,WIDTH DO
      NEIGHBORS=(
        GETB(PREV_I,X-1)+
        GETB(PREV_I,X)+
        GETB(PREV_I,X+1)+
        GETB(LINE_I,X-1,Y)+
        GETB(LINE_I,X+1,Y)+
        GET(X-1,Y+1)+
        GET(X,Y+1)+
        GET(X+1,Y+1))
    IF ((NEIGHBORS==ALIVE_COLOR*3)OR
        ((PGET(X,Y)==ALIVE_COLOR)AND
         NEIGHBORS==ALIVE_COLOR*2))
    THEN
      PSET(X,Y,ALIVE_COLOR)
    ELSE
      PSET(X,Y,0)
    END
    END
  END
END
```

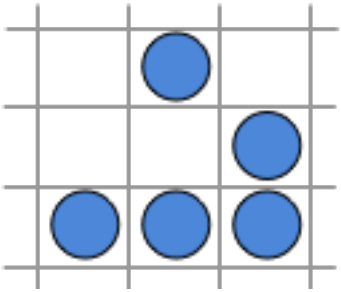
Notice that because we're reading cell data directly from the screen, our neighbor count is now expressed as a multiple of `ALIVE_COLOR`, which happens to be `7 (WHITE)`. We could tell the `GET()` and (new) `GETB()` functions to convert from the color value returned by `PGET()` to `0` or `1`, but that's CPU time we can save just by using the multiple.

In my unscientific timing tests, I noticed a savings of about 5 seconds over 50 generations with this new method compared to the two-board version.

Toroidal Game Boards?

Our implementation considers the cells beyond the edges to be dead cells. Another option is to have the board "wrap around" as if it were a torus shape, so the cells on the bottom are adjacent to the cells on the top, and the left edge is similarly adjacent to

the right edge. In the two-board version, this would be easy to implement with a small modification to the `GET()` function. Give it a try!



Try this pattern with a toroidal game board.

More Information

Wikipedia:
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

The Game of Life wiki:
http://www.conwaylife.com/wiki/Main_Page

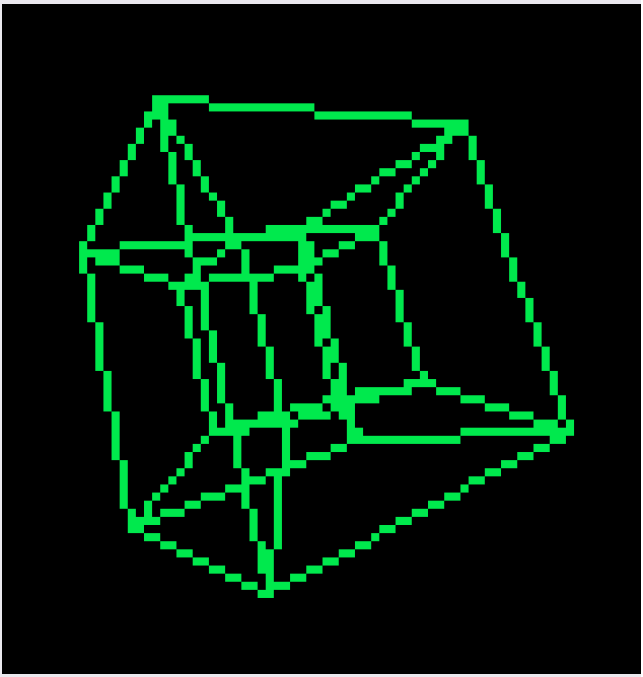
My Game of Life for PICO-8, including a built-in map editor:
http://www.lexaloffle.com/bbs/?tid=2158

by dddaaannn (@dan_sanderson)

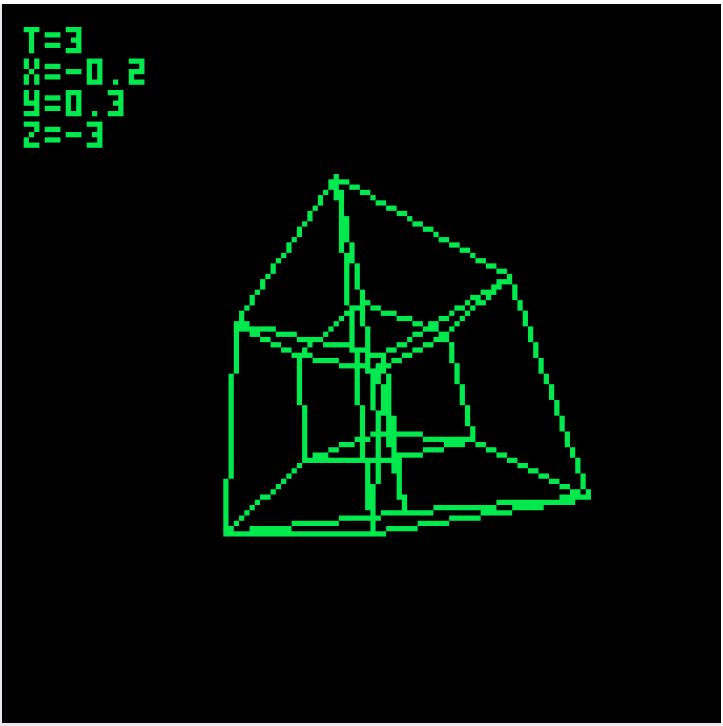
3d demo

simple 3d demo by @NoahRosamilia

```
-- HERE IS THE DATA FOR THE 3D MODEL
-- THE FIRST SECTION IS THE 3D LOCATION
-- OF EACH POINT ON THE CUBE, AND THE
-- SECOND SECTION IS THE LINES THAT
-- CONNECT THE POINTS
CUBE = {{{-1,-1,-1}, -- POINTS
   {-1,-1,1},
   {1,-1,1},
   {1,-1,-1},
   {-1,1,-1},
   {-1,1,1},
   {1,1,1},
   {1,1,-1},
   {-0.5,-0.5,-0.5}, -- INSIDE
   {-0.5,-0.5,0.5},
   {0.5,-0.5,0.5},
   {0.5,-0.5,-0.5},
   {-0.5,0.5,-0.5},
   {-0.5,0.5,0.5},
   {0.5,0.5,0.5},
   {0.5,0.5,-0.5}},
  {{1,2}, -- LINES
   {2,3},
   {3,4},
   {4,1},
   {5,6},
   {6,7},
   {7,8},
   {8,5},
   {1,5},
   {2,6},
   {3,7},
   {4,8},
   {8+1,8+2}, -- INSIDE
```



```
  {8+2,8+3},
  {8+3,8+4},
  {8+4,8+1},
  {8+5,8+6},
  {8+6,8+7},
  {8+7,8+8},
  {8+8,8+5},
  {8+1,8+5},
  {8+2,8+6},
  {8+3,8+7},
  {8+4,8+8},
  {1,9},
  {2,10},
  {3,11},
  {4,12},
  {5,13},
  {6,14},
  {7,15},
  {8,16}}}
```




```
function _init()
  cam = {0,0,-2.5} -- initialise the camera position
  mult = 64 -- view multiplier

  a = flr(rnd(3))+1 -- angle for random rotation
  t = flr(rnd(50))+25 -- time until next angle change
end

function _update()
  -- handle the inputs
  if btn(0) then cam[1] -= 0.1 end
  if btn(1) then cam[1] += 0.1 end
  if btn(2) then cam[2] += 0.1 end
  if btn(3) then cam[2] -= 0.1 end
  if btn(4) then cam[3] -= 0.1 end
  if btn(5) then cam[3] += 0.1 end
  t -= 1 -- decrease time until next angle change
  if t <= 0 then -- if t is 0 then change the random angle and restart the timer
```

```
  T=FLR(RND(50))+25 -- RESTART TIMER
  A=FLR(RND(3))+1 -- UPDATE ANGLE
  END
  CUBE=ROTATE_SHAPE(CUBE,A,0.01) -- ROTATE OUR CUBE
END

FUNCTION _DRAW()
  CLS() -- CLEAR THE SCREEN
  PRINT("T="..T,0,6*0) -- PRINT TIME UNTIL ANGLE CHANGE
  PRINT("X="..CAM[1],0,6*1) -- PRINT X, Y, AND Z LOCATION OF THE CAMERA
  PRINT("Y="..CAM[2],0,6*2)
  PRINT("Z="..CAM[3],0,6*3)
  DRAW_SHAPE(CUBE) -- DRAW THE CUBE
END

FUNCTION DRAW_SHAPE(S,C)
  FOR L IN ALL(S[2]) DO -- FOR EACH LINE IN THE SHAPE...
    DRAW_LINE(S[1][L[1]],S[1][L[2]],C) -- DRAW THE LINE
  END
END

FUNCTION DRAW_LINE(P1,P2,C)
    X0,Y0=PROJECT(P1) -- GET THE 2D LOCATION OF THE 3D POINTS...
    X1,Y1=PROJECT(P2)
  LINE(X0,Y0,X1,Y1,C OR 11) -- AND DRAW A LINE BETWEEN THEM
END

FUNCTION DRAW_POINT(P,C)
  X,Y=PROJECT(P) -- GET THE 2D LOCATION OF THE 3D POINT...
  PSET(X,Y,C OR 11) -- AND DRAW THE POINT
END

FUNCTION PROJECT(P)
  X=(P[1]-CAM[1])*MULT/(P[3]-CAM[3])+127/2
  -- CALCULATE X AND CENTER IT
  Y=-(P[2]-CAM[2])*MULT/(P[3]-CAM[3])+127/2
  -- CALCULATE Y AND CENTER IT
  RETURN X,Y -- RETURN THE TWO POINTS
END
```

```
FUNCTION TRANSLATE_SHAPE(S,T)
  NS = {{},S[2]}
  -- COPY THE SHAPE, BUT ZERO OUT THE POINTS AND KEEP THE LINES
  FOR P IN ALL(S[1]) DO -- FOR EACH POINT IN THE ORIGINAL SHAPE...
    ADD(NS[1],{P[1]+T[1],P[2]+T[2],P[3]+T[3]})
    -- ADD THE DISPLACEMENT TO THE POINT AND ADD IT TO OUR NEW SHAPE
  END
  RETURN NS -- RETURN THE NEW SHAPE
END

FUNCTION ROTATE_SHAPE(S,A,R)
  NS = {{},S[2]}
  -- COPY THE SHAPE, BUT ZERO OUT THE POINTS AND KEEP THE LINES
  FOR P IN ALL(S[1]) DO -- FOR EACH POINT IN THE ORIGINAL SHAPE...
    ADD(NS[1],ROTATE_POINT(P,A,R))
    -- ROTATE THE POINT AND ADD IT TO THE NEW SHAPE
  END
  RETURN NS -- RETURN THE NEW SHAPE
END

FUNCTION ROTATE_POINT(P,A,R)
  -- FIGURE OUT WHICH AXIS WE'RE ROTATING ON
  IF A==1 THEN
    X,Y,Z = 3,2,1
  ELSEIF A==2 THEN
    X,Y,Z = 1,3,2
  ELSEIF A==3 THEN
    X,Y,Z = 1,2,3
  END
  _X = COS(R)*(P[X]) - SIN(R)*(P[Y])
  -- CALCULATE THE NEW X LOCATION
  _Y = SIN(R)*(P[X]) + COS(R)*(P[Y])
  -- CALCULATE THE NEW Y LOCATION
  NP = {}
  -- MAKE NEW POINT AND ASSIGN THE NEW X AND Y TO THE CORRECT AXES
  NP[X] = _X
  NP[Y] = _Y
  NP[Z] = P[Z]
  RETURN NP -- RETURN NEW POINT
END
```

SHRINKING YOUR CODE

Here are some tips to help you squeeze every last byte out of pico-8. Not only will this allow you to fit more code in your game, but it will reduce the amount of scrolling you have to do in the pico-8 editor.

a word on token-counting

Starting with version 0.1.1, pico-8 uses a token-counting system rather than simply counting characters. This allows developers to fit more code on a cartridge, but it's also a little bit complicated. Here's how it works.

First, pico-8 counts tokens. Variables, functions, and operators (`=`, `+`, `-`, `[`, `]`, `(` , `)` , etc.) are one token each. Strings are also one token. Comments are ignored. Each cartridge has room for 8192 tokens.

There is also a limit on the number of characters (32768), but tokens are usually the bottleneck.

> Note: You don't have to use the default pico-8 editor. You can load your `.p8` file in any text editor you like!

use pico-8's built-in shorthands

if

```
--7 TOKENS, 31 CHARACTERS
IF VALUE THEN
  DO_THING()
END

--7 TOKENS, 20 CHARACTERS
IF (VALUE) DO_THING()

--SAVINGS: 0 TOKENS, 11 CHARACTERS
```

foreach

```
-- 13 TOKENS, 43 CHARACTERS
FOR E IN ALL(ENEMIES) DO
  TAKE_DAMAGE(E)
END

-- 6 TOKENS, 29 CHARACTERS
FOREACH(ENEMIES, TAKE_DAMAGE)

-- SAVINGS: 7 TOKENS, 14 CHARACTERS
```

operator

```
-- 6 TOKENS, 19 CHARACTERS
SIZE = COUNT(TABLE)

-- 4 TOKENS, 13 CHARACTERS
SIZE = #TABLE

-- SAVINGS: 2 TOKENS, 6 CHARACTERS
```

math

```
-- 5 TOKENS, 9 CHARACTERS
A = A + 3

-- 3 TOKENS, 6 CHARACTERS
A += 3

-- SAVINGS: 2 TOKENS, 3 CHARACTERS
```

use single-character variable and function names

You can reduce your character count by using shorter variable names, at the cost of readability. Note that variable names take up the same number of tokens regardless of length.

```
-- 3 TOKENS, 20 CHARACTERS
LONGVARIABLENAME = 3

-- 3 TOKENS, 5 CHARACTERS
A = 3

-- SAVINGS: 0 TOKENS, 15 CHARACTERS
```

If you run out of letters, you can also use an underscore (_) as a single-character name. After that, you'll have to start using two characters per name.

reduce spaces

Although it looks pretty, you don't need to include spaces between operators in Lua.

```
-- 3 TOKENS, 5 CHARACTERS
A = 3

-- 3 TOKENS, 3 CHARACTERS
A=3

-- SAVINGS: 0 TOKENS, 2 CHARACTERS
```

Similarly, you can save on characters (at the cost of readability) by removing indentations.

Since grouping symbols also count as operators in Lua, spaces before and after grouping symbols can be omitted.
For example, you don't need spaces after the parentheses in a pico-8 shorthand **IF**:

```
-- 7 TOKENS, 20 CHARACTERS
IF (VALUE) DO_THING()

-- 7 TOKENS, 19 CHARACTERS
IF (VALUE)DO_THING()
-- SAVINGS: 0 TOKENS, 1 CHARACTER
```

remove comments

Comments don't use up any tokens, but they still count as characters! Remove them to gain some space.

set constants

If there's a particular value or function you find yourself using a lot, you can assign it to a single-character variable and save characters every time you refer to it.

Initially, this will cost you an additional 3 tokens, but you will save characters over the course of your program.

```
-- COSTS 3 TOKENS, BUT SAVES 7 CHARACTERS EVERY TIME RECTFILL IS USED
R = RECTFILL

T = TRUE
F = FALSE
-- ETC.
```

use multiple returns

You can return more than 1 value from a function by separating them by commas.

```
FUNCTION MD(A,B)
  RETURN A*B,A/B
END
```

In certain situations, this can allow you to call one function instead of two. You can also choose how many of these values you want to use.

```
-- TAKE BOTH VALUES
A,B = MD(8,4)

-- TAKE ONLY ONE VALUE
A = MD(8,4)
```

If you're calling a function with multiple returns as an argument to another function (or as part of a `return` statement), it will automatically expand to use all its return values. You can prevent this by surrounding it with parentheses.

```
-- CALLS F WITH 3 ARGUMENTS: THE 2 VALUES RETURNED BY MD, AND THE CONSTANT 3
  F(MD(8,4),3)

-- CALLS F WITH 2 ARGUMENTS: THE FIRST VALUE RETURNED BY MD, AND THE CONSTANT 3
  F((MD(8,4)),3)
```

don't use parentheses for strings or tables

If you're passing a string or table to a function, the parentheses are optional.

```
  -- 4 TOKENS, 15 CHARACTERS
  PRINT("STRING")

  -- 2 TOKENS, 13 CHARACTERS
  PRINT"STRING"

  -- SAVINGS: 2 TOKENS, 2 CHARACTERS
```

Note that this only works if the string or table is the only argument to the function.

take advantage of logic (and/or)

AND returns the left argument if it is **FALSE** (or **NIL**). Otherwise, it returns the right argument.

OR returns the left argument if it is **not FALSE** (or **NIL**). Otherwise, it returns the right argument.

You can use these in conjunction to create a ternary operator.

```
-- ONLY OPEN LOCK IF MULTIPLE CONDITIONS ARE SATISFIED
LOCK_OPEN = HAS_KEY AND COLLIDE(PLAYER, CHEST)

-- USE DEFAULT VALUE FOR VARIABLE
NAME = PLAYER_NAME OR "JOE"

-- WILL PRINT "DEAD" IF DEAD, "ALIVE" OTHERWISE
PRINT(DEAD AND "DEAD" OR "ALIVE")

-- SETS SPEED
SPEED = (FALLING AND -1 OR 1) * ACCELERATION

-- YOU CAN DO IT FOR FUNCTIONS TOO!
(FROZEN AND THAW OR MOVE)()

-- DON'T FORGET ABOUT NOT!
STAND_STILL = NOT MOVING
```

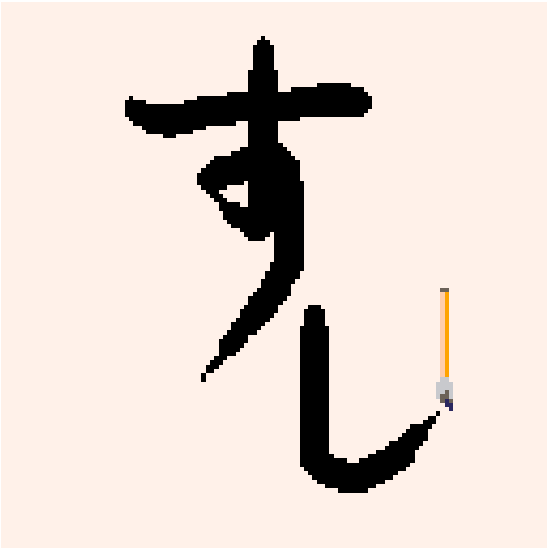
- Jonathan Stoler
@jonstoler

SHODO

@oinariman

I introduce a painting tool that I made with PICO-8, Shodo(書道). This is a demake of the 80's Macintosh software, Mac 書道(MacCalligraphy in U.S.) that simulates Japanese traditional ink-dipped brush calligraphy. Since you cannot use a mouse with PICO-8 when it runs a program, it may appear to be ridiculous to imitate brush drawings using only D-pad and AB buttons.
However, it can draw brush-like lines unexpectedly well.

In this article, I describe how to implement the brushlike line drawings, and the memory processing needed to make a painting tool.



The brush movements

To draw lines that dynamically change their thickness, I added inertia to the brush. When you press an arrow key, the brush will move in the direction. When you release the key, the brush will gradually slow its speed, and then will stop.
Pressing the Z button will cause the line to grow thick.

The line will decrease its thickness after you release the button. Operating these controls at the same time will allows you to draw brushlike lines.

The memory processing

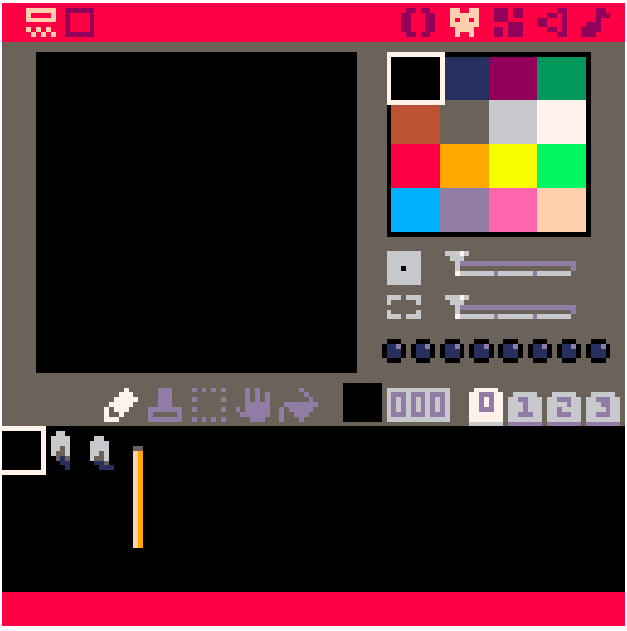
Painting tool must save its drawing data somewhere in the memory. Because the PICO-8's screen resolution is 128 x 128, we need to find space to store 128 x 128 = 16,384 pixels. Although you may save this data in a Lua array, this is not recommended idea.
I think that it will cause complications and run slowly. So, I use memset() and memcpy() . These PICO-8 API functions allow you to access continuous memories immediately.

The space to save your picture

According to PICO-8.txt, memset() and memcpy() may only access the 32k memory area called "base ram." The list below from PICO-8.txt describes the layout of the base ram.

```
0x0 gfx
0x1000 gfx2/map2 (shared)
0x2000 map
0x3000 gfx_props
0x3100 song
0x3200 sfx
0x4300 userdefined
0x5f00 draw state (/cart data) (192 bytes incl. unused)
0x5fc0 (reserved for persistent data in development)
0x6000 screen (8k)
```

The area that begins with 0x4300 is the “user-defined” area. That is what programmers can use freely. The area occupies 896 bytes (0x4300 to 0x5eff). Because the PICO-8’s color format is 2 pixels per byte, we need 16,384 / 2 = 8,192 bytes (4 kilobytes) of memory area to save all of the pixels on the screen.
The userdefined area is not sufficient at all. So, I use the area from 0x1000 to 0x2fff. This area is basically for sprites and maps copied from a cart. I don’t need it because Shodo uses just five sprites, and the first gfx area (from 0x0 to 0x0fff) has enough memory to store them.



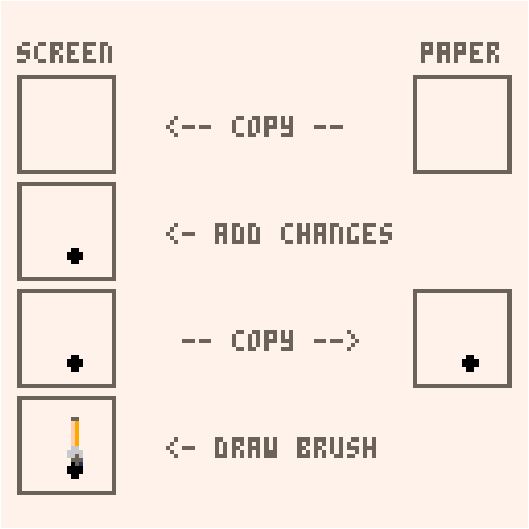
The sprites used in Shodo

The screen updating

I named the 4k area from 0x1000 “paper area.” PICO-8 displays the pixel data in the “screen area” (beginning at 0x6000) to the screen. So, in order to display the userdrawn picture, we should just copy the data that is stored in the paper area to the screen area.

The list below is the procedure to update the screen in the _draw() function which is called every 1/30 seconds.

1. Copy whole data in the paper area to the screen area (using memcpy())
2. Add the changes made by the user to the screen
3. Copy all of the data in the screen area to the paper area (using memcpy())
4. Draw an image of a brush on the screen



In order to keep the only user-drawn content in the paper area, it is important to follow these instructions in the order in which they are listed.

The entire program appears below. It’s not very long or complicated. I didn’t implement any undo/redo functionality or eraser tool to respect the Japanese shodo tradition.

However, it might be fun to add these things or a color palette.

Ryosuke Mihara

```
-- SHODO 1.0.1
-- BY RYOSUKE MIHARA

-- BRUSH ATTRIBUTES
--
-- X    : X-COORDINATE
-- Y    : Y-COORDINATE
-- VX   : X COMPONENT OF VELOCITY
-- VY   : Y COMPONENT OF VELOCITY
-- DOWN : WHEN TRUE, THE BRUSH IS PUT DOWN TO THE PAPER
-- R    : BRUSH THICKNESS
BRUSH = {}
BRUSH.X = 64
BRUSH.Y = 64
BRUSH.VX = 0
BRUSH.VY = 0
BRUSH.DOWN = FALSE
BRUSH.R = 0

-- BRUSH CONSTANTS
--
-- BRUSH_ACC : ACCELERATION
-- BRUSH_BRAKE : BRAKE VALUE
-- BRUSH_MUL : SPEED MAGNIFICATION WHEN THE BRUSH IS DOWN
-- BRUSH_RMAX : MAXIMUM THICKNESS
-- BRUSH_RACC : ACCERELATION OF THICKNESS CHANGES
BRUSH_ACC = 0.175
BRUSH_BRAKE = -0.1
BRUSH_MUL = 0.65
BRUSH_RMAX = 3
BRUSH_RACC = 0.2

-- PAPER ATTRIBUTES AND CONSTANTS
PAPER = {}
PAPER.Y = 0
PAPER.VY = 0
PAPER_INIT_VY = 5
PAPER_ACC = 0.3
----
```

```
FUNCTION REPLACE_PAPER ()
  IF PAPER.Y > 0 THEN
    RECTFILL ( 0 , 0 , 127 , 127 , 7 )
    LOCAL Y = FLR (PAPER.Y)
    MEMCPY ( 0 X6000 , 0 X1000 + Y * 64 , 64 * ( 128 - Y ))
    LINE ( 0 , 127 - Y - 1 , 127 , 127 - Y - 1 , 6 )

    PAPER.Y += PAPER.VY
    PAPER.VY += PAPER_ACC
    IF PAPER.Y >= 127 THEN
      MEMSET ( 0 X1000 , 0 X0077 , 128 * 64 )
      PAPER.Y = 0
    END
  END
END

FUNCTION DRAW_BRUSH ()
  IF BRUSH.DOWN THEN
    SPR ( 3 , BRUSH.X , BRUSH.Y - 23 , 1 , 3 )
    SPR ( 2 , BRUSH.X , BRUSH.Y )
  ELSE
    SPR ( 3 , BRUSH.X , BRUSH.Y - 24 , 1 , 3 )
    SPR ( 1 , BRUSH.X , BRUSH.Y )
  END
END

FUNCTION DRAW_LINE ()
  IF BRUSH.R > 0 THEN
    CIRCFILL (BRUSH.X + 4 , BRUSH.Y + 6 , BRUSH.R , 0 )
  END
END

----
FUNCTION MOVE_BRUSH ()
  -- WHEN THE BRUSH IS PUT DOWN TO THE PAPER , SLOW ITS SPEED
  LOCAL MUL = 1
  IF BRUSH.DOWN THEN
    MUL = BRUSH_MUL
  END
```

```
  BRUSH.X += BRUSH.VX * MUL
  BRUSH.Y += BRUSH.VY * MUL

  -- BRAKE THE BRUSH
  -- STOP THE BRUSH WHEN ITS X/Y COMPONENT OF VELOCITY IS INVERTED
  LOCAL PREV
  IF BRUSH.VX ~= 0 THEN
    PREV = BRUSH.VX
    BRUSH.VX += BRUSH.VX * BRUSH_BRAKE
    IF PREV * BRUSH.VX < 0 THEN BRUSH.VX = 0 END
  END
  IF BRUSH.VY ~= 0 THEN
    PREV = BRUSH.VY
    BRUSH.VY += BRUSH.VY * BRUSH_BRAKE
    IF PREV * BRUSH.VY < 0 THEN BRUSH.VY = 0 END
  END

  -- STOP THE BRUSH WHEN IT REACHES THE EDGE OF THE SCREEN
  IF BRUSH.X <- 4 THEN BRUSH.X =- 4 END
  IF BRUSH.X > 123 THEN BRUSH.X = 123 END
  IF BRUSH.Y <- 6 THEN BRUSH.Y =- 6 END
  IF BRUSH.Y > 123 THEN BRUSH.Y = 123 END
END

FUNCTION UPDATE_LINE_WIDTH ()
  IF BRUSH.DOWN THEN
    BRUSH.R += BRUSH_RACC
  ELSE
    BRUSH.R -= BRUSH_RACC
  END
  IF BRUSH.R < 0 THEN BRUSH.R = 0 END
  IF BRUSH.R > BRUSH_RMAX THEN BRUSH.R = BRUSH_RMAX END
END

----
FUNCTION INPUT ()
  BRUSH.DOWN = BTN ( 4 )
  IF BTN ( 0 ) THEN BRUSH.VX -= BRUSH_ACC END
  IF BTN ( 1 ) THEN BRUSH.VX += BRUSH_ACC END
```

```
  IF BTN(2) THEN BRUSH.VY -= BRUSH_ACC END
  IF BTN(3) THEN BRUSH.VY += BRUSH_ACC END

  IF PAPER.Y == 0 AND BTNP(5) THEN
    SFX(0)
    PAPER.Y = 1
    PAPER.VY = PAPER_INIT_VY
  END
END

----
FUNCTION _INIT()
  MEMSET(0X1000, 0X0077, 128*64)
END

FUNCTION _UPDATE()
  INPUT()
  MOVE_BRUSH()
  UPDATE_LINE_WIDTH()
END

FUNCTION _DRAW()
  IF PAPER.Y > 0 THEN
    REPLACE_PAPER()
  ELSE
    -- COPY WHOLE PIXELS IN THE PAPER TO THE SCREEN
    MEMCPY(0X6000, 0X1000, 128*64)
    -- ADD CHANGES MADE BY THE USER TO THE SCREEN
    DRAW_LINE()
    -- COPY WHOLE PIXELS IN THE SCREEN TO THE PAPER
    MEMCPY(0X1000, 0X6000, 128*64)
  END
  DRAW_BRUSH()
END
```

Water Waves

Add some visual flair to your game with a wave distortion.

1. Assumptions

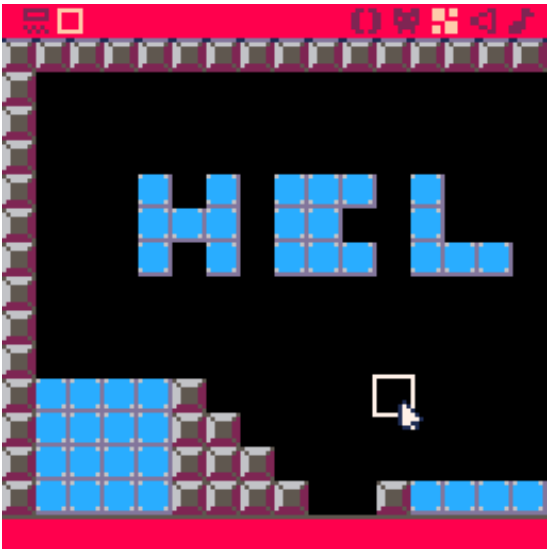
This tutorial makes use of a sprite flag to know where to warp the screen pixels. It assumes the whole available area of the tile-map is draw to the screen at the 0,0 coordinate. If you draw your tilemaps in a different manner, you will need to adjust the code accordingly.

Three variables are required to determine the warp effect, t is for time, which is updated by 1 each _update(), and cam_x & cam_y which are used to offset the camera position in the _draw() function.

2. Tiles

Draw some sprites to use in your game, I drew a light blue square, a solid block and a ball. The blue square will be the tile that waves apply to; the lines and dots in the blue square will help accentuate the wave's ripple. Set the 5th (light blue) sprite flag to be true for each sprite that you want to ripple (sprite flags are set with the little circles below the color palette).

Draw a tilemap which includes areas of the ripple tiles, as well as non-wavy tiles.



3. The Warp Function

This is the meat of our exercise, I'll show the whole function, then explain line by line what is going on. You should be able to add this to existing games, as you can add this to your _draw() function.

```
function warp_w()
  local wave_speed=0.1667
  local displace_dist=0.5

  for y=max(0,flr(cam_y/8)),min(flr(cam_y/8)+16,16*2) do
    for x=max(0,flr(cam_x/8)),min(flr(cam_x/8)+16,16*8) do
      local val=mget(x,y)
      if (fget(val,4)) then
        for y1=0,7 do
        local line={0,0,0,0,0,0,0,0}
        for x1=0,8 do
          line[x1]=pget(x*8-flr(cam_x)+x1,y*8-flr(cam_y)+y1)
        end
        local newx=sin((t*wave_speed+y1)/8)*displace_dist
        for x1=0,8 do
          pset(x*8+x1+newx,y*8+y1,line[x1])
        end
       end
```

```
  end
 end
end
end
```

What does all this code do then? Let's start at the top.

```
local wave_speed=0.1667
local displace_dist=0.5
```

Here we define two variables which affect the displacement effect. wave_speed affects how fast the ripples go, displace_dist changes how far the pixels are moved in either direction. The value of 0.5 allows a pixel to be moved from -0.5 to 0.5, a max distance of 1 total pixel. Change these numbers to see the different effects they can have, but I would suggest keeping the numbers quite small.

```
for y=max(0,flr(cam_y/8)),min(flr(cam_y/8)+16,16*2) do
  for x=max(0,flr(cam_x/8)),min(flr(cam_x/8)+16,16*8) do
```

The next two lines begin loops, starting at the current camera position (or zero if negative), and end a screen later. Since we are using tilemaps (remember the sprite flag?) to mark the effect, we only need to check a maximum of 16 tiles in either direction (one tile is an 8 pixel image, screen res of 128, 128/8=16)

```
local val=mget(x,y)
```

This line gets the number of the sprite used at the position of x,y which we will use in the next line

```
if (fget(val,4)) then
```

where we check if the sprite has the light blue flag selected. If it does, then we will continue

```
for y1=0,7 do
  local line={0,0,0,0,0,0,0,0}
```

The first line starts a loop where we are going to iterate through each vertical slice of our 8x8 tile. The second line creates a collection of values for the pixels we are going to move.

```
FOR x1=0,8 DO
  LINE[x1]=PGET(x*8-FLR(CAM_X)+x1,y*8-FLR(CAM_Y)+y1)
END
```

These lines loop through the horizontal pixels, recording the not-yet-manipulated values to the line collection. Notice that we subtract the value of cam_x (or y) from our current x (or y) values. This is because we are grabbing the current pixels from the frame buffer and need the values to be within 0-127. Our code will affect anything already drawn to the screen, and it will not affect anything we draw after applying this effect.

```
LOCAL newx=SIN((t*WAVE_SPEED+y1)/8)*DISPLACE_DIST
```

Now we need a new x offset for our line of pixels. We use a wave formula to smoothly adjust our offset back and forth as time (t) increases. The wave formula is: value = sin(angle) * range. Our formula has a slight adjustment in the angle, as we add y1 and then divide by 8. The reason for that is to make a smooth transition between each of our 8 vertical pixels, each one offset by 1/8th the angle of the previous pixel. This also makes it so multiple wavy tiles above and below each other will all appear to seamlessly match motion.

```
FOR x1=0,8 DO
  PSET(x*8+x1+newx,y*8+y1,LINE[x1])
END
```

This last loop then sets the screen pixels to the values at the newx position.

4. Other code

As I mentioned earlier, we need to have some variables in our game for this to work. At the top of your code, outside of any functions, add these lines:

```
T=0
CAM_X=0
CAM_Y=0
```

These will initialize our variables, and start each at default values of zero.

At the very end of our _update() function, we need to be sure that time is increasing. If you started a project from an example like Jelpi, this may already be in there.

```
FUNCTION _UPDATE()
 T+=1
 -- ALL YOUR UPDATE CODE GOES HERE

 -- EXAMPLE CODE, MOVES CAMERA
 IF (BTN(0,0)) CAM_X-=0.4
 IF (BTN(1,0)) CAM_X+=0.4
 IF (BTN(2,0)) CAM_Y-=0.4
 IF (BTN(3,0)) CAM_Y+=0.4
```

The last bit of code is that our _draw() function needs to call warp_w() at the appropriate time. Most likely, you want to call it after all tilemaps and sprites have been drawn, but before any HUD elements, such as scores or lives have been drawn. Below is an example _draw() function, yours will most likely have more.

```
FUNCTION _DRAW()
 -- CLEAR SCREEN
 RECTFILL(0,0,127,127,0)

 -- MOVE CAMERA
 CAMERA(CAM_X,CAM_Y)
 -- DRAW WHOLE TILEMAP
```

```
  map(0,0,0,0,16*8,16*2,0)

  -- draw ball, wave position up/down
  local y_wave=(cam_y+8*8)+sin(t*0.0025)*32
  spr(3,cam_x+8*8,y_wave,1,1)

  -- warp
  warp_w()

  -- reset camera
  camera(0,0)

  -- draw hud above warp, it will not distort
  print("score:000",8,8,7)
end
```

I added some code to this which will help show how the affect works.

```
  -- draw objects
  local y_wave=(cam_y+8*8)+sin(t*0.0025)*32
  spr(3,cam_x+8*8,y_wave,1,1)
```

These lines make use of the wave function again, this time as a way of moving our object up and down. spr() is the drawing call for drawing a sprite (#3 in this case) to the screen. Since it is drawn before calling warp_w(), the ripple effect will apply to it when over one of the marked tiles.

```
  -- draw hud above warp, it will not distort
  print("score:000",8,8,7)
```

This line adds some text to the screen, a score counter that does nothing at the moment. It will not be affected when placed over a marked tile, since its drawn after the ripple effect occurs.

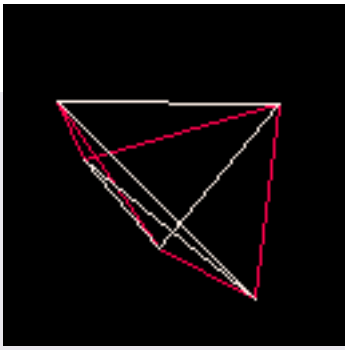
5. Taking it further

You can use this effect to show different needs, such as water, lava, heat, gravity or whatever you would like. The effect looks really great when you add vertical elements to your drawings such as seaweed or pipes. Try changing the code so the ripple effect is vertical instead of horizontal, or maybe change the speed based on objects inside the tile area.

I'd love to see what cool things you make with this code. Send me some gif's on twitter to @mattfox12. Happy coding!

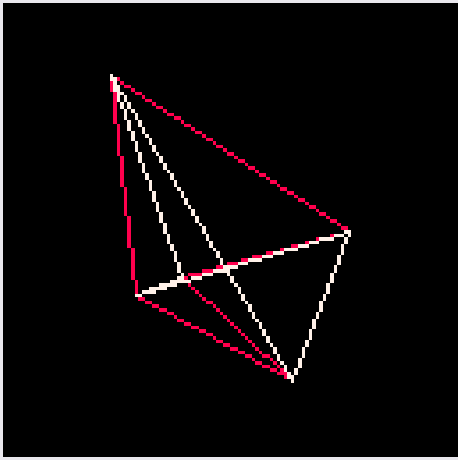
Matthew Klundt

Screensaver



```
POINT1 = {}
POINT1.X = 64
POINT1.Y = 64
POINT1.A = 90
POINT1.S = 2

POINT2 = {}
POINT2.X = 64
POINT2.Y = 64
POINT2.A = 270
POINT2.S = 2

POINT3 = {}
POINT3.X = 64
POINT3.Y = 64
POINT3.A = 180
POINT3.S = 2

POINT4 = {}
POINT4.X = 64
POINT4.Y = 64
POINT4.A = 0
POINT4.S = 2

POINT5 = {}
POINT5.X = 64
POINT5.Y = 64
POINT5.A = 0
POINT5.S = 2

FUNCTION _UPDATE()

  POINT1 = ANIMATE(POINT1)
  POINT2 = ANIMATE(POINT2)
  POINT3 = ANIMATE(POINT3)
  POINT4 = ANIMATE(POINT4)
  POINT5 = ANIMATE(POINT5)
END
```

```
function animate(point)

  angleoffset = ((point.a)%360)/360

  point.x = point.x + point.s*cos(angleoffset)
  point.y = point.y + point.s*sin(angleoffset)
  --collision detection
  if point.x > 128 then
    point.a = 135 + flr(rnd(90))
  end
  if point.x < 0 then
    point.a = 45 - flr(rnd(90))
  end
  if point.y > 128 then
  point.a = 45 - flr(rnd(90))
  end
  if point.y < 0 then
    point.a = 225 + flr(rnd(90))
  end

  return point

end

function _draw()

  rectfill(0,0,127,127,0)

  --main prism
  line(point1.x,point1.y,point3.x,point3.y,8)
  line(point2.x,point2.y,point4.x,point4.y,8)
  line(point3.x,point3.y,point5.x,point5.y,8)
  line(point4.x,point4.y,point1.x,point1.y,8)
  line(point5.x,point5.y,point2.x,point2.y,8)

  --red lines

  line(point1.x,point1.y,point2.x,point2.y,7)
  line(point2.x,point2.y,point3.x,point3.y,7)
  line(point3.x,point3.y,point4.x,point4.y,7)
  line(point4.x,point4.y,point5.x,point5.y,7)
  line(point5.x,point5.y,point1.x,point1.y,7)
end
```



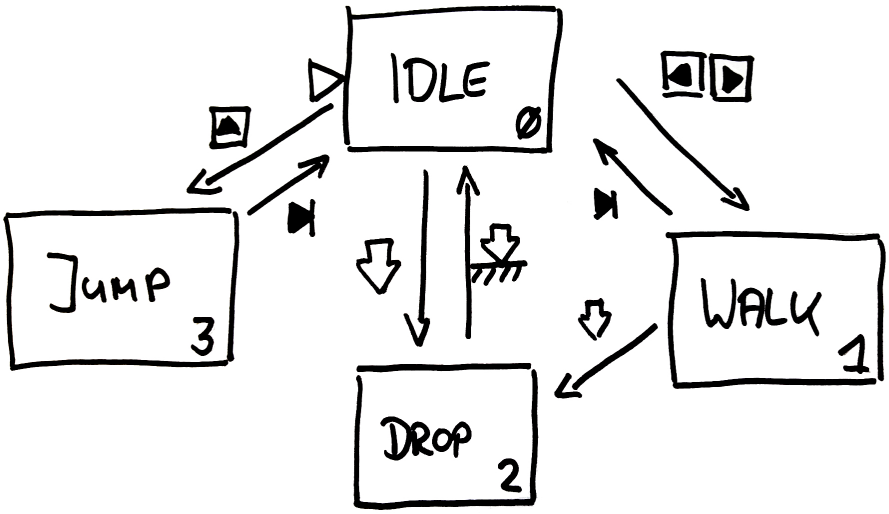
Devine Lu Linvega
@aliceffekt

PicoJump

Platformers have appeal. They are quite literally the pixel-manifestation of the Hero's Journey and have shaped the gaming landscape over for decades. I grew up with an Amiga 500 and games like Turrican, Prince of Persia and The Shadow of the Beast! Well, and let's not forget about Sonic and Super Mario, right?

This tutorial assumes you know the basics of PICO-8 programming, how to draw sprites, the `_init()`, `_update()` and `_draw()` functions and how to write simple programs with them.

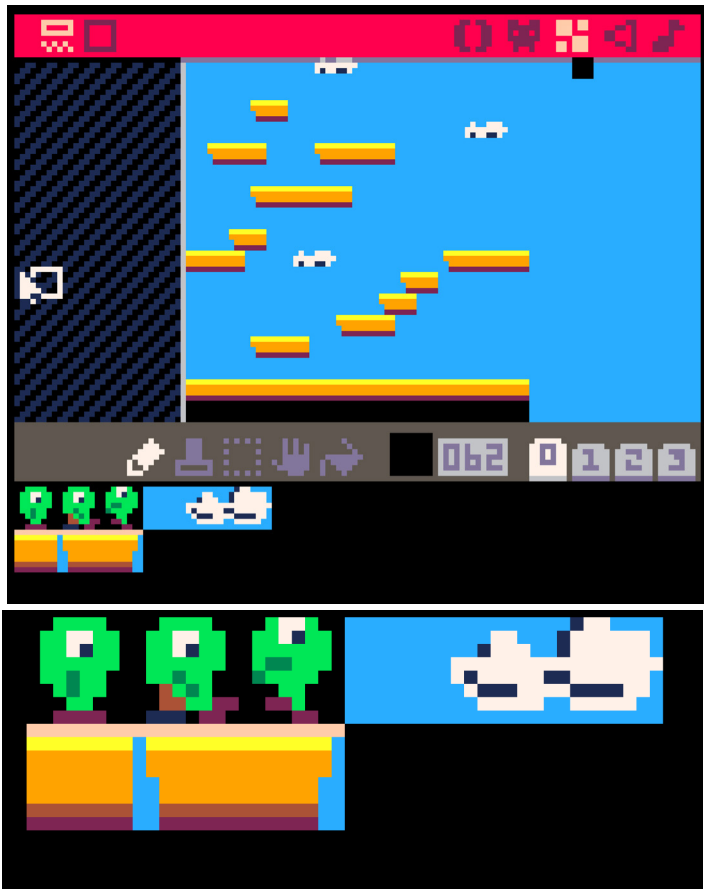
At first it is always good to think about what we want the character (I think I will name him Tutorial-Bob) in our platformer game will be able to do. For the time being, let's start with walking, jumping, falling and being idle. To model this behaviour, we will make use of a mechanism generally used in computing and engineering. We will build a finite state machine or in short FSM. And in our case it looks like this: the boxes represent the states out player can be in. The arrows between them are the transitions, which - when a certain condition is met - will point to the next state Bob will be in.



For example, if Bob walks along and suddenly there is no more ground, he will start falling. Once he hits ground again, he will stop falling and be idle. Or if the user presses the left or right arrow, Bob will start walking. When we have made up our state machine, it is very straightforward to start writing code.

Before we begin, here some sprites and a little world in preparation for the game. Any map sprite that we want to use as a ground Bob can stand on, is tagged with the first sprite flag being set to True. This way we can easily determine with what to collide using `FGET()` on the map tile.

Lets start writing the program. Here the player initialization and drawing routines for the game.



```
function _init()
  px=20  -- x-position
  py=64  -- y-position
  pstate=0 -- current player state
  pspr=0  -- current sprite
  pdir=0  -- current direction
  pat=0 -- player state timer
end

function _draw()
  -- draw the world
  map(0,0,0,0,16,16)
  -- draw the player, we use dir to mirror sprites
  spr(pspr,px,py,1,1,pdir==-1)
end
```

In the `_init()` method we are setting a whole lot of variables to keep track of Bob.
We have `px` and `py` for the world position in pixel, `pdir` is for the direction we are looking at, we have `pspr` to store the current sprite and a `pstate` for the active state itself.
Look at the state machine graph again. See the numbers in the corner of the boxes? This is the identifier for each state which will be assigned to `pstate`.
We also hold a variable called `pat`, which is the state counter or sometimes called state clock. Every time we run `_update()` that counter will increase by one. On every change of state, the counter will be reset to zero. This way we always know how long bob has been in his current state and use that information for animation and movement calculations.

Before we write the state machine behaviour, we need one more helper function to tell us, if Bob is hovering in the air (`true`) or standing on the ground (`false`).
Be aware that `mget()` wants map coordinates, so we have to divide the player position by 8.

```
function canfall()
  -- get the map tile under the player
  v=mget(flr((px+4)/8),flr((py+8)/8))
  -- see if it's flagged as wall
  return not fget(v,0)
end
```

Now we get to the state machine part.
Each time we move from one state to another, we have to set the **pstate** variable to the new state id and also reset the **pat** counter. Let's do this in a dedicated function. Later on, when things get more complex, we could also implement state OnEnter behaviour in here. For now, this is beyond the scope of this tutorial and our function just looks like this.

```
function change_state(s)
  pstate=s
  pat=0
end
```

Our **_update()** function will deal with all the actual state behaviour and starts like this

```
function _update()
  b0=btn(0) -- button0 state
  b1=btn(1) -- button1 state
  b2=btn(2) -- button2 state

  px=(px+128)%128 -- no bounds left and right
  pat+=1 -- increment state clock
```

We capture what buttons are pressed by the user and also make sure that when Bob leaves the screen on one side, he will come back in on the other. Also notice, that we increment the **pat** on every call of **_update()**

Next, we gonna implement the four states, one by one.
For each state we will write what is happening to Bob while he is in the state, as well as define the conditions under which we are transitioning into another behaviour.

```
-- IDLE STATE
IF PSTATE==0 THEN
  PSPR=0
  IF (B0 OR B1) CHANGE_STATE(1)
  IF (B2) CHANGE_STATE(3)
  IF (CANFALL()) CHANGE_STATE(2)
END
```

If Bob is in the Idle state, we set the current sprite to idle. This is pretty much all that happens here.
The three if-statements check for user input and send Bob into the walking, jumping or falling state according to our diagram above.

```
-- WALK STATE
IF PSTATE==1 THEN
  IF (B0) PDIR=-1
  IF (B1) PDIR=1
  PX+=PDIR*MIN(PAT,2)
  PSPR=FLR(PAT/2)%2

  IF (NOT (B0 OR B1)) CHANGE_STATE(0)
  IF (B2) CHANGE_STATE(3)
  IF (CANFALL()) CHANGE_STATE(2)
END
```

The walk state is a little more elaborate. Based on what button the user pressed, we set the sprite direction. We also increment or decrement Bobs x position **PX**. Note that we are using **PAT** to make him move just one pixel in the first tick of the state and then two in any following, to create a sense of acceleration. We also set the current sprite alternating between 0 and 1 based on the **PAT** again. See that I also have divided the **PAT** by 2 two to slow down the sprite change to not get too flickery. This - again - is followed by the transitions into falling, jump and idle.

The state implementation for falling looks a bit more complicated, as it has to deal with collisions and intersections.

```
-- FALL STATE
IF PSTATE==2 THEN
  PSPR=2
  IF (CANFALL()) THEN
    IF (B0) PX-=1 -- STEER LEFT
    IF (B1) PX+=1 -- STEER RIGHT
    PY+=MIN(4,PAT) -- MOVE THE PLAYER
    IF (NOT CANFALL()) PY=FLR(PY/8)*8 -- CHECK GROUND CONTACT
  ELSE
    PY=FLR(PY/8)*8 -- FIX POSITION WHEN WE HIT GROUND
    CHANGE_STATE(0)
  END
END
```

Inside the fall state, we check for ground contact - which is the only transition out of here into the idle state.
If we are falling, we allow the player to move left and right to steer the fall.
Like in the real world, with every tick falling, we accelerate to fall a little bit faster. This is done by adding the **PAT** to the y-position of Bob, that speed is capped at a terminal velocity of 4.
In case Bob is hitting a ground tile, we need to make sure to fix his position back on top of a tile.
The mechanism used here is kept rather simple, but works. We just round the y-position back to the tile under Bob. That way we cannot get stuck halfway inside the ground

The last state is the JUMP state. It works similar to the FALL state.

```
-- JUMP STATE
IF PSTATE==3 THEN
  PSPR=2
  PY-=6-PAT
  IF (B0) PX-=2
  IF (B1) PX+=2
  IF (NOT B2 OR PAT>7) CHANGE_STATE(0)
END
```

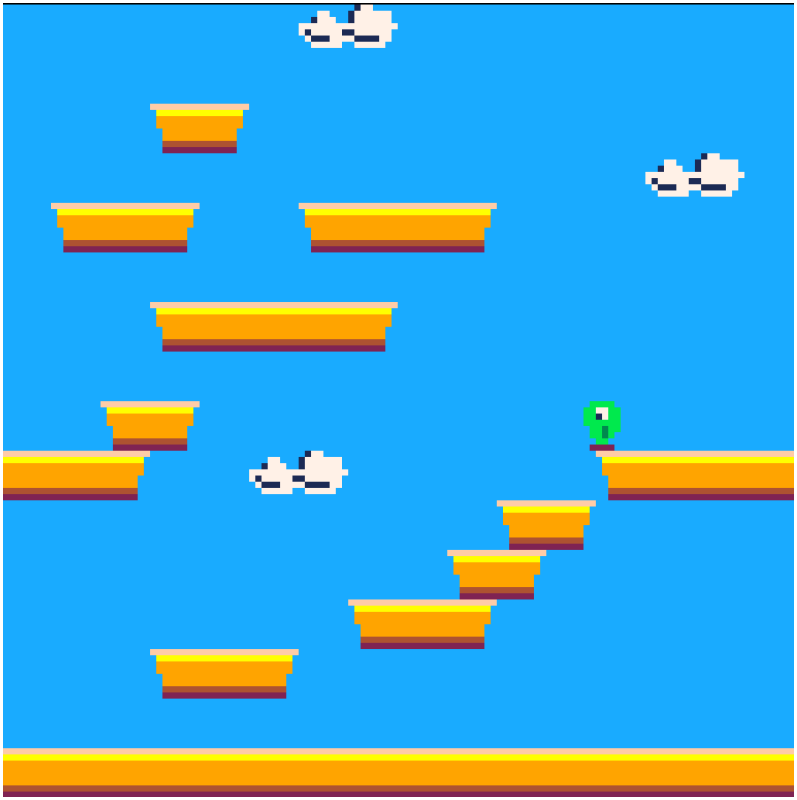
And let's not forget about this one

```
END -- END OF THE UPDATE FUNCTION
```

Now you have a character driven by a simple state machine. The movement is still a bit jerky and we can't run, shoot, duck or slide.
But the biggest benefit of an FSM is, it is easy to add other states and transitions and also more complicated math to smoothen out the animation of the character later on.

Happy Coding!

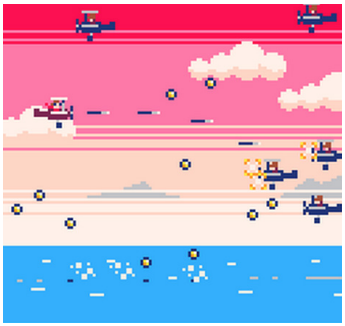
Johannes Richter
http://www.lexaloffle.com/bbs/?tid=2520



PICO-8 Colour Palettes

PICO-8's limited colour palette creates an interesting challenge when creating art for your games/cartridges. Here are some of my favourite examples of colour palettes used in PICO-8.

Gabby DaRienzo



"Baron Figs"
by zep

Unnamed project by- Christina Antoinette

"Across the River"
by Benjamin Soule

"Naughty Painters"
by oinariman

"Mortuary Simulator"
by gabdar

"Celeste"
by Noel Berry and Matt Thorson