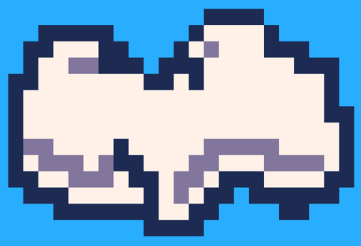
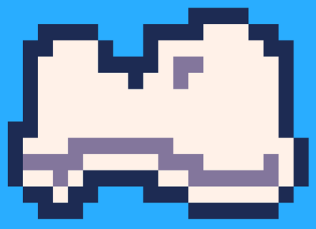
PICO-8

GETTING STARTED GUIDE

by Juan Eduardo Flores





Getting Started with PICO-8

Introduction

PICO-8 is a popular example of a **fantasy console**. A fantasy console is a fictitious video game console played on an emulator (a software that imitates the architecture of a specific hardware).

PICO-8 creates the experience of retro gaming using the Lua programming language with very specific technical limitations. It can be a fun way to learn programming and game design.

The creator of PICO-8, described the term:

> A fantasy console is like a regular console, but without the inconvenience of actual hardware. PICO-8 has everything else that makes a console a console: machine specifications and display format, development tools, design culture, distribution platform, community, and playership. It is similar to a retro game emulator, but for a machine that never existed.
>
> - Joseph White

Influences

The PICO-8 is a computer that never existed, but it is inspired by real computers from the 1980s.

Apple IIe

Commodore 64

The **Apple IIe** was released in January 1983. It was intended for home and educational use. It had a **280 x 192** pixel display with 6 colors for high resolution, and 16 colors for low resolution.

The **Commodore 64** has a **320 x 200** pixel display with 16 colors. It was released in August 1982 and became the best-selling single computer model of all time.

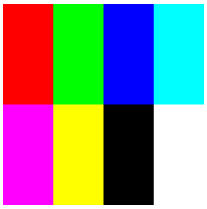
BBC Micro

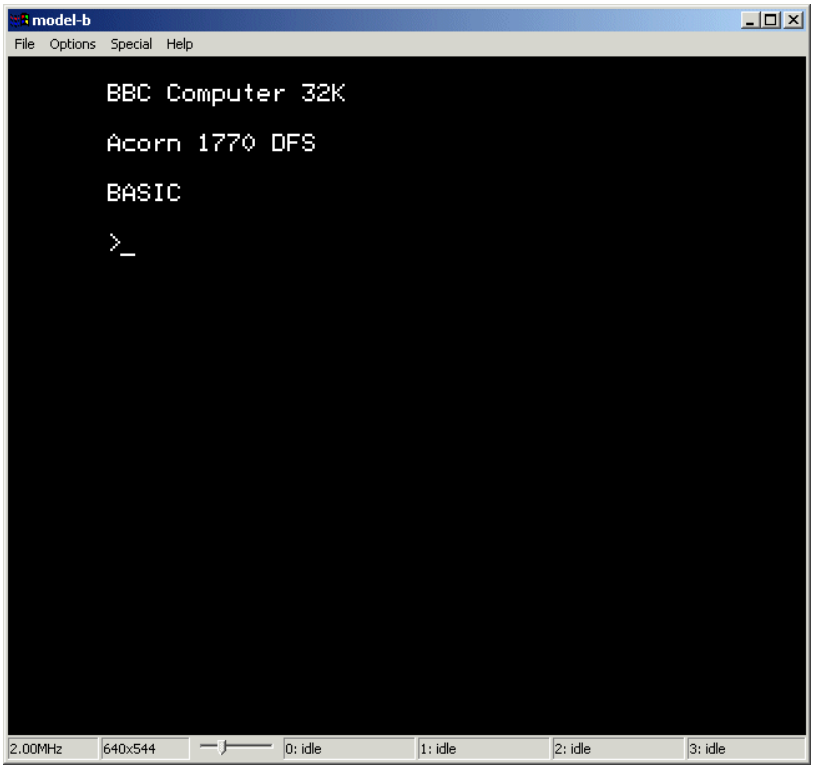
The **BBC Micro** was a family of computers designed by Acorn Computers as part of the BBC's Computer Literacy Project in the early 1980s, aimed at promoting computer literacy in the UK.

In 1982, there was a TV series called *The Computer Programme*, which focused on learning how to use this machine. It was broadcast on BBC2.

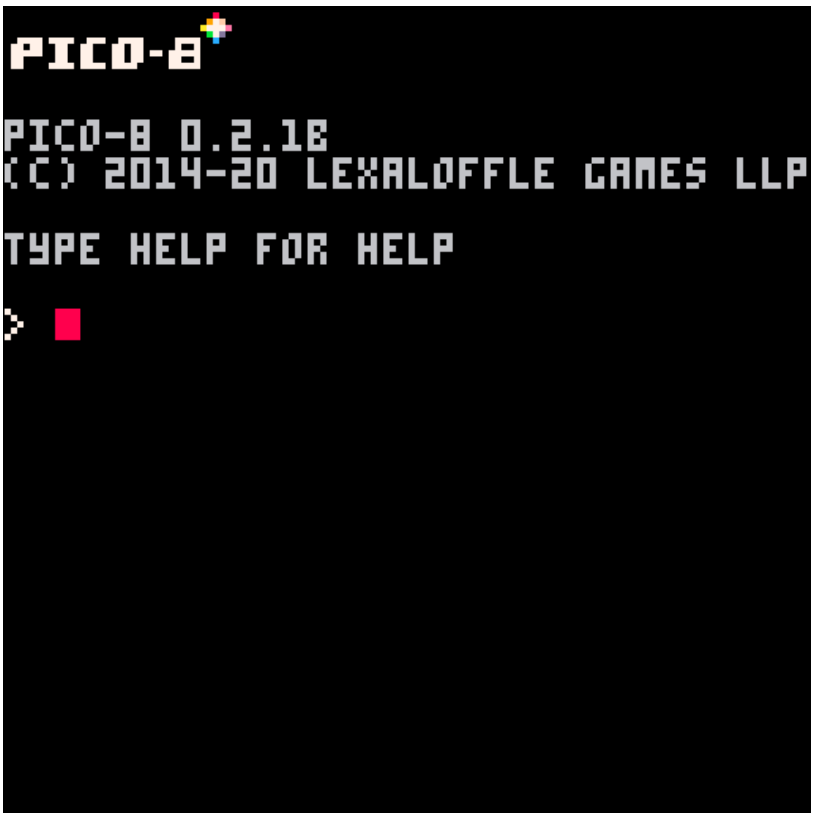
It can be switched between several display modes. Modes 0 to 6 could display a palette of sixteen colors using the eight basic colors at the vertices of the RGB color cube and eight flashing colors made by alternating the basic color and its inverse.

Mode 0 has a resolution of **640 x 256** pixels.





BBC Micro running on an emulator.



PICO-8's Command Line Interface.

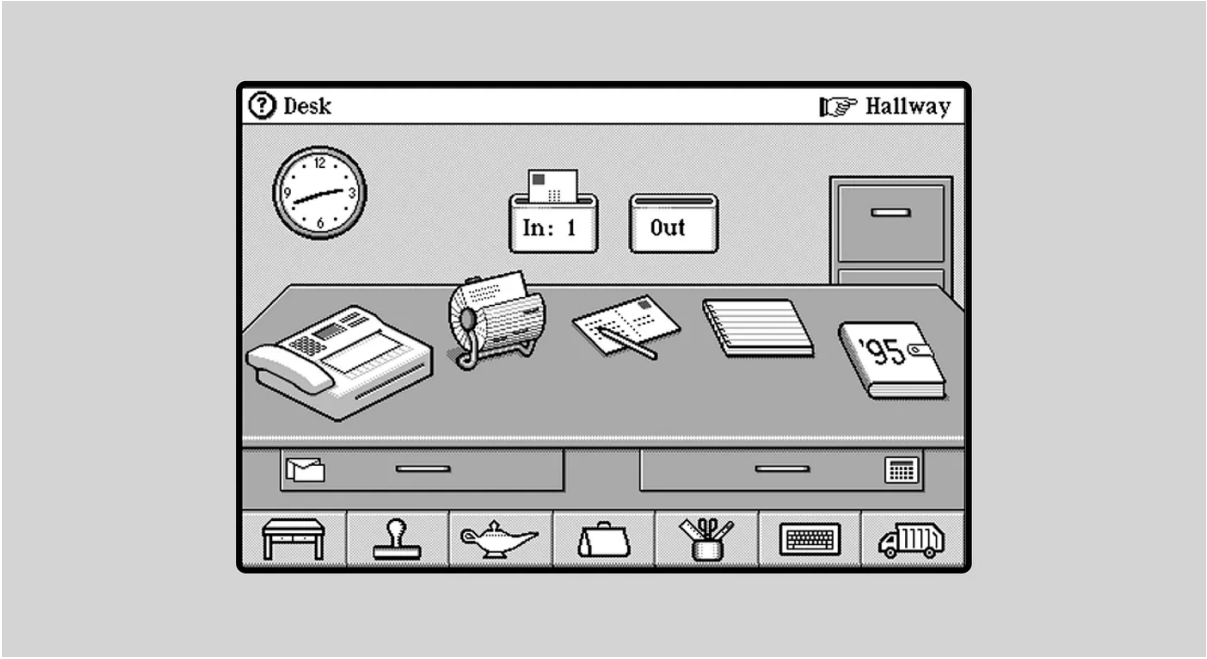
Command Line Interface (CLI) / Terminal / Console / Shell

Now that you can see the inspiration, let's get familiar with the command line interface because it is the first thing you see when you start PICO-8.

It displays the logo, some version information, and a prompt to enter commands.

The > symbol followed by the blinking red cursor indicates that the system is ready to accept user input.

Historical Background



Desktop interface of the Magic Cap operating system. It used the "desktop metaphor", which uses imagery that represent a real physical desktop.

Before Graphical User Interfaces (GUIs) that rely on the use of a mouse to click graphic icons that resemble real-world objects to open and close windows, there was the command line interface (CLI).

> According to a wikibook Operating System Design: "A command line interface or CLI is a method of interacting with a computer by giving it lines of textual commands.

CLI, **terminal**, **console**, and even **shell** are all terms that are often used interchangeably to indicate a text based system for navigating an operat-

ing system. It could get confusing, but they all refer to similar concepts. It won't hurt to take at the history of the terms to get an idea of how and why we use them.

A Terminal is a program that emulates a physical terminal, which historically was an electromechanical device that connected to a larger computer to interact with its functions.

On a terminal, you can type commands, press enter, and see the results.

It comes from a history that predates screens where a printer was used to print the results of our commands. See:

- Z3 (1941)
- Teleprinter
- Teletype Model 33 (1963)

A Console was initially the switches and indicators available literally on the console panel of a computer.

The front panel of the PDP-11/20, which ran the 1st edition of Unix in 1972.

Later the term was used for a special teletype/terminal attached to the computer. The operator could use the console to perform privileged operations.

In the world of Unix, a Shell is a command-line interpreter that provides a user interface for access to an operating system's services. There are many different Unix shells. Popular shells for interactive use include Bash, Zsh, and Fish.

In Unix based systems like Linux or MacOS, you may see this command prompt:

```
$
```

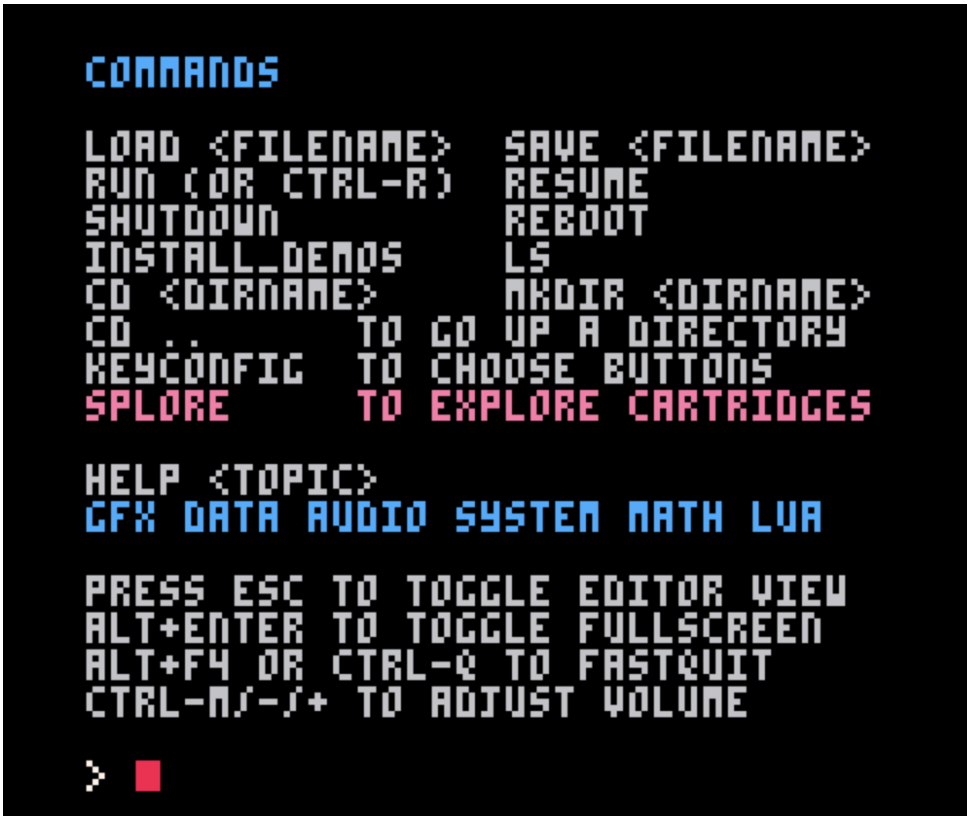
In Windows, you might see what is seen in PICO-8, the > symbol, prompting the user to enter a command. It is based on the Microsoft Disk Operating System (MS-DOS) command line interface.

```
>
```

First Commands

In the startup screen PICO-8 advises you of the **HELP** command. It says: "Type **HELP** for help". By typing help and pressing **ENTER**, you get in return a list of commands that you can use in PICO-8.

```
> HELP
```

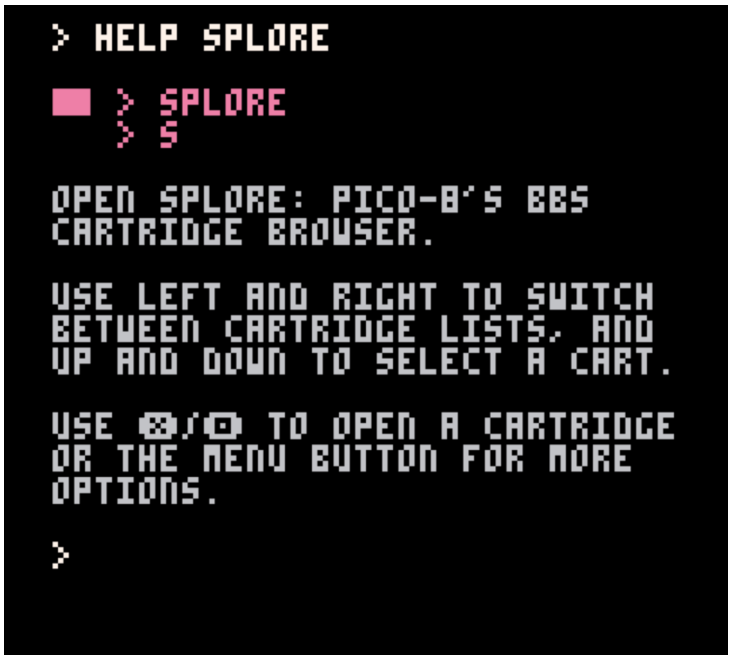


The convention of having a help command is common. You can even write **HELP** followed by another command or function for additional information about it.

```
> HELP <command>
```

The list of commands can be overwhelming. It can be difficult to know where to start and it all looks so bare and technical. The next command SPLORE is a lot more fun.

```
> HELP SPLORE
```



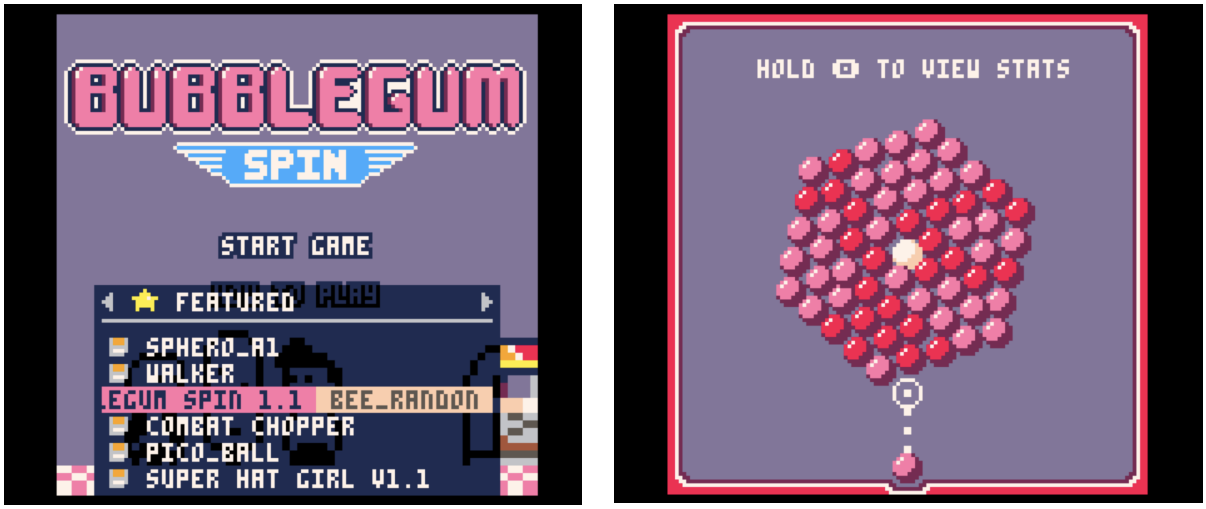
SPLORE

Let's type the command that is in pink text: SPLORE. Remember that the computer does not receive the typed command until you press the **ENTER** key.

```
> SPLORE
```

> **NOTE**
>
> In PICO-8, everything is uppercase. If you try typing uppercase letters by pressing Shift, you will find that you will get random symbols instead. Just type in lowercase letters when entering commands and when you start to write code later on.

SPLORE is a graphical interface for exploring PICO-8 cartridges that others have written and published. Here you can also publish your own games. Think of it as a way to explore and browse through a library of games that the public contributes to.

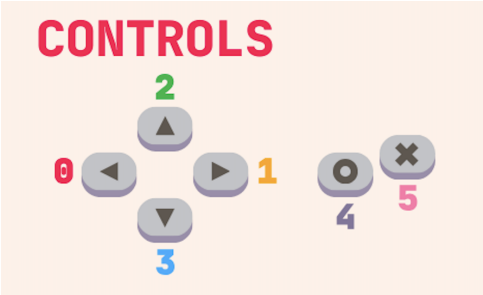


BUBBLEGUM SPIN by BEE_RANDON. Being played after navigating SPLORE

Navigate around using the arrow keys on your keyboard, and use the **Z** key to select an option and to run a cart. The **ESC** key is also very useful for many purposes. It is typically used to exit or to pause the game.

Limits for a Design Culture

This is an opportunity to introduce one quirk of PICO-8 games: they only use up to six different buttons for input, comprising the four arrow keys and two additional buttons.



By default, the **Z** key is the [O] and the **X** key is the [X] button. You could also use C/V or N/M respectively. These buttons are configurable, allowing you to change them if desired.

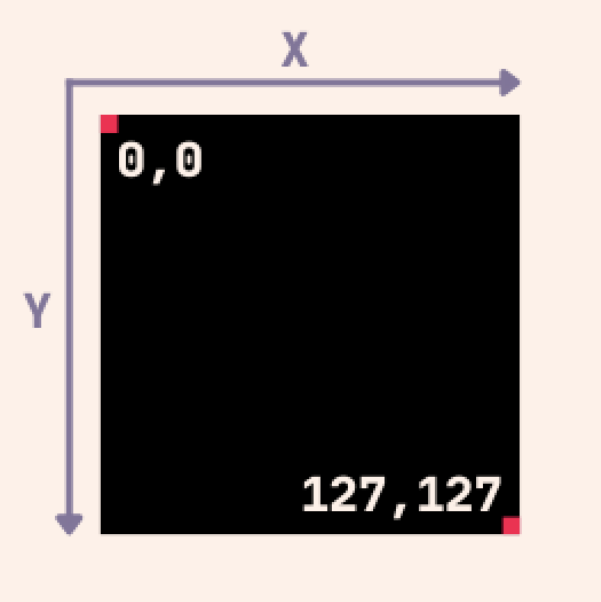
> KEYCONFIG

This command is used to change the key assignments for the six buttons, for each of the two players. Be cautious when using this command, as it will modify the key bindings for all PICO-8 games until you reset them.

If that happens, you can reset to default by entering keyconfig and then pressing the **DELETE** key on your keyboard for each input.

While getting to have fun in SPLORE, you might notice that the games are limited to a 16 color palette and a small resolution of **128 x 128 pixels.** This is part of the design culture of PICO-8, which embraces limitations to inspire creativity.

The PICO-8 is limited to 16 colors (with some trickery you can get 16 more!)



The display resolution of 128 x 128 pixels. Note how 0 is at the top left corner, and how it increases the X by going right, and increased Y by going down.

Other limitations worth mentioning:

- 4-channel sound
- 8192 tokens (code size, the number of characters you are allowed to use)

File Navigation

In the command line, we don't have a GUI to click on folders or files, so we need to learn how to get around our file system using text commands.

What we know as folders in GUIs are called **directories** in command line interfaces.

To view the files and directories in the current directory, we can use the **LS** command.

```
> LS
```

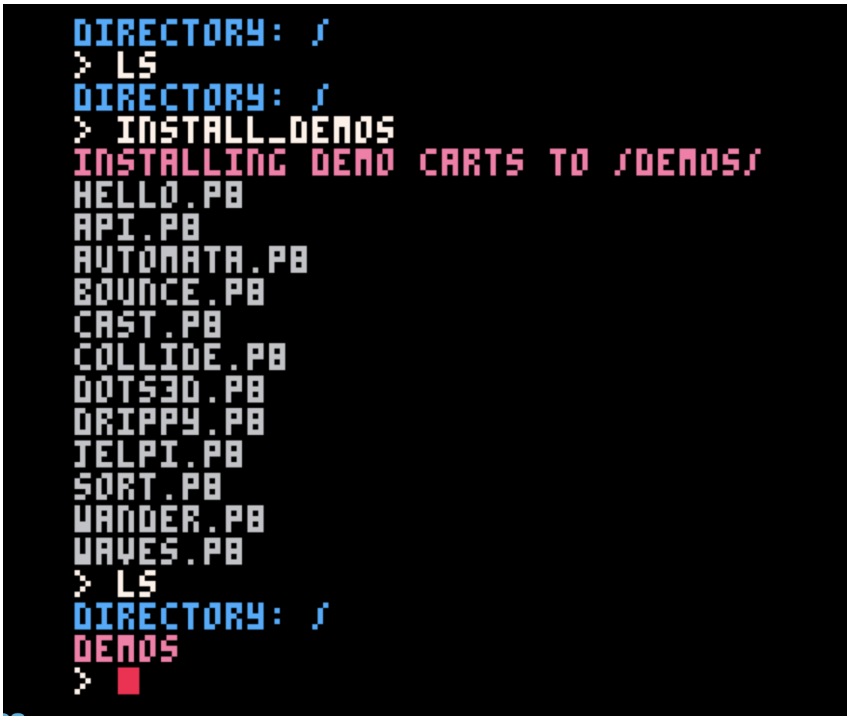
LS stands for "list". It lists all files and directories in the current directory. "List all contents"

Currently, we don't have any directories, so nothing is listed under what is our home directory. This directory is represented by the lone forward slash /.

Let's try this command next:

```
> INSTALL_DEMOS
```

This will install some demo cartridges into our file system. Try typing **LS** again to see the newly created directory.



At the bottom of the screen, we can see:

```
DIRECTORY: /
DEMOS
```

A DEMOS directory has appeared under our home directory (/). Directories are in a pink color, while files are in a lightgray color. To enter the DEMOS directory, we can use the **CD** command.

```
> CD <directory_name>
```

CD stands for "change directory." It allows us to navigate into a different directory.

> CD DEMOS

```
> CD DEMOS
/DEMOS/
>
```

Now, if we type **LS** again, we can can list the contents in the **DEMOS** directory, which should be the newly installed demo cartridges.

```
> CD DEMOS
/DEMOS/
> LS
DIRECTORY: /DEMOS
API.P8
COLLIDE.P8
BOUNCE.P8
WANDER.P8
WAVES.P8
DRIPPY.P8
DOTS3D.P8
HELLO.P8
CAST.P8
JELPI.P8
AUTOMATA.P8
SORT.P8
>
```

> **NOTE**
>
> PICO-8 games are saved with the **.P8** extension, and are called *cartridges*, just like old video game cartridges.

Imagined cartridge for PICO-8.

Cartridge for the Nintendo Entertainment System (NES)

To run a cartridge, we can use the **LOAD** command followed by the name of the cartridge.

```
> LOAD <cartridge_name>
```

Loads the specified cartridge.

TIP
A tip when typing file or directory names: you can use the TAB key to auto-complete names. Just type the first few letters and press TAB, and it will fill in the rest for you if there is a unique match. Try typing: **LOAD AP** and press enter to see it autocomplete.

```
> LOAD API.P8
```

Lastly, use the **RUN** command to run the cartridge.

```
> RUN
```

Runs the currently loaded cartridge.

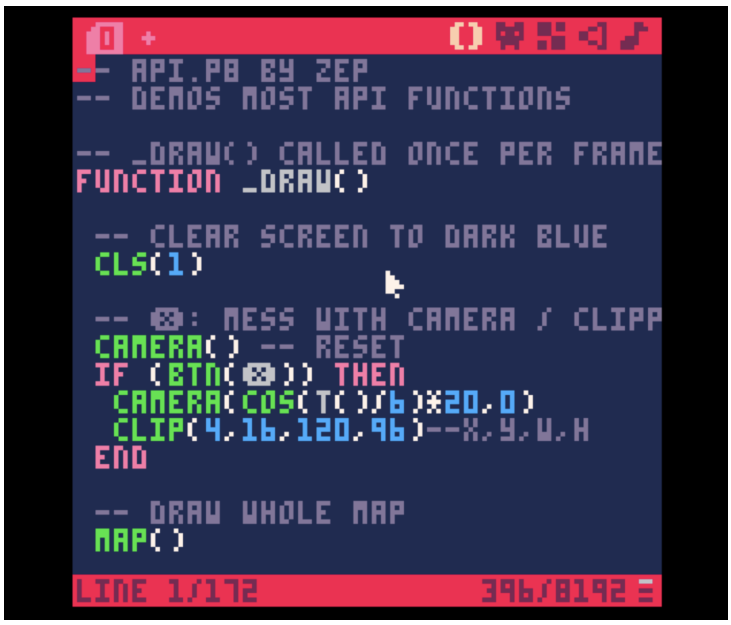
API.P8 is now running! Pressing **ESC** will bring you back to the command line interface.

> **TIP**
> You can press **Ctrl + L** (or **Cmd + L** on MacOS) to clear the terminal screen!

> **TIP**
> Instead of typing **RUN** each time, you could just press **Ctrl + R** (**Cmd + R**) to run the currently loaded cartridge or restart it.

The Code Editor

To access the code editor, press the **ESC** key while in the command line interface.



You may notice how we have a mouse! Yes, this is a graphical user interface (GUI) for writing code. On the top right are icons to navigate through different types of editors, but for now, press **ESC** again to return to the command line interface.

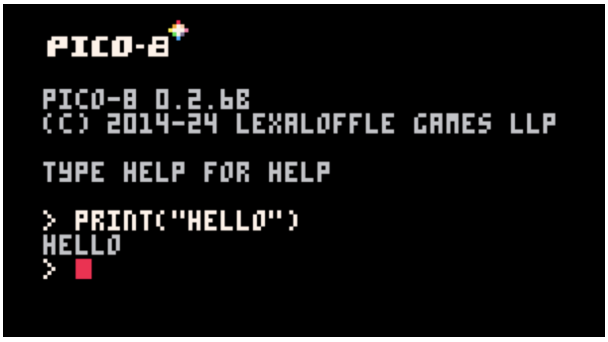
Simple Functions

Before writing a game, you can get acquainted with some simple functions in the console.

PRINT

Print (draw) some text.

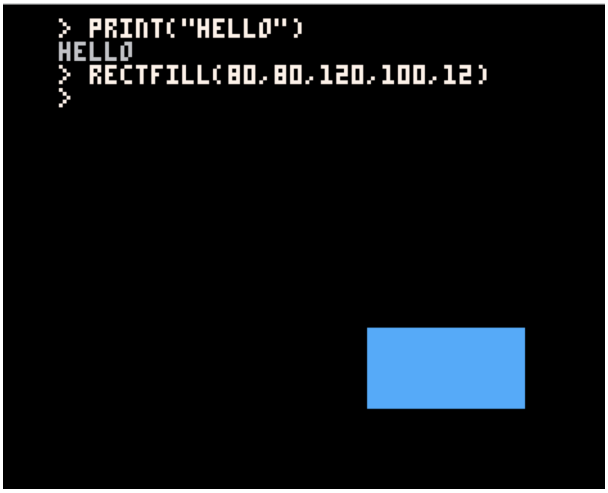
```
-- Print "HELLO" to the console
PRINT("HELLO")
```



RECTFILL

Draw a rectangle with a filled color.

```
-- Draw a rectangle at:
-- x = (80, 80)
-- y = (120, 100)
-- with a fill color of 12
RECTFILL(80,80,120,100,12)
```



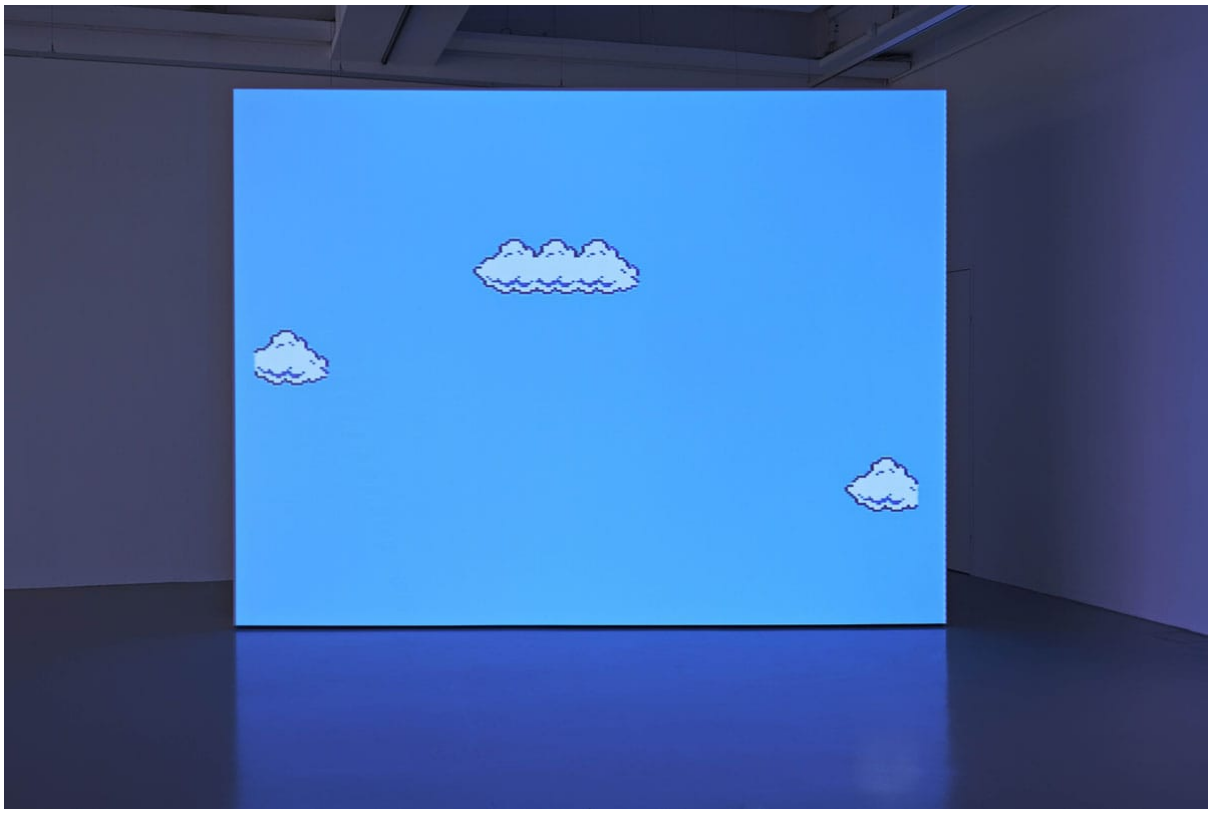
Core Functions

There are three core functions that PICO-8 uses to run a program.

```
function _init()
    -- This function runs once when the program starts
end

function _update()
    -- This function runs 30 times per second
end

function _draw()
    -- This function also runs 30 times per second
end
```

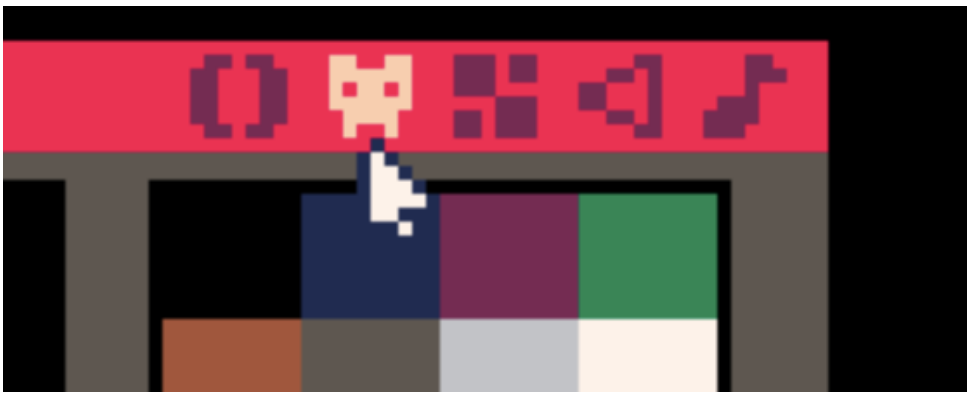
Making a Simple Animation

As a first exercise let's create a floating cloud animation inspired by Cory Arcangel's Super Mario Clouds artwork, made by modifying a Super Mario Bros. cartridge.

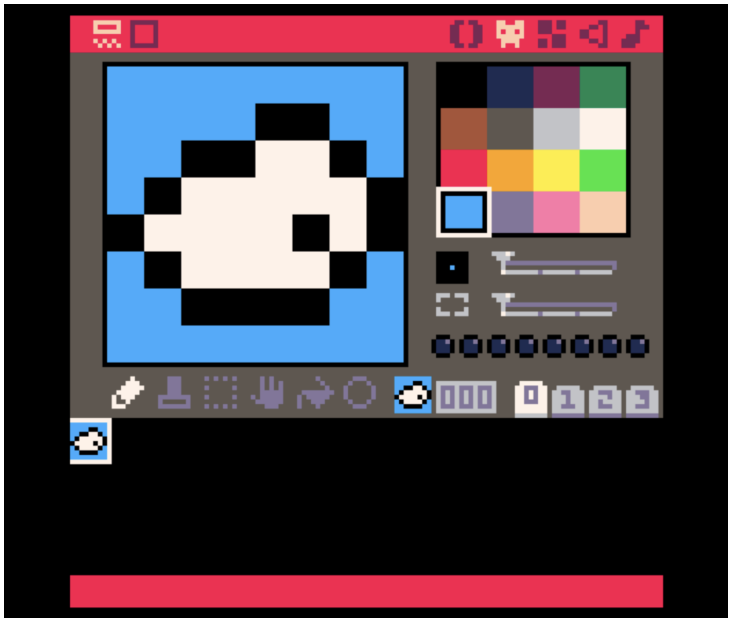


Sprite Editor

To navigate to the sprite editor, go to the editor and click on the icon that looks like a little character with two eyes.



Here, you can draw a sprite within this 8x8 pixel canvas.



Note how I draw it in the top left-most tile, which is assigned the number "000".

Return to the terminal screen by pressing **ESC** again.

The SPR() function

You can draw a sprite onto the screen by using the **SPR()** function.

The first parameter is *n*, which is the assigned **number of the sprite.** In our case, we drew our sprite in the top left-most box, which is assigned the value of 0.

```
-- spr(sprite_number)

-- draw sprite 0,
spr(0)
```

The second and third parameters are X and Y coordinates for where it will be drawn, referring to the sprite's top-left pixel.

```
-- spr(sprite_number, x, y)

-- draw sprite 0, at (x=20, y=20)
spr(0, 20, 20)
```

There are even more parameters! They are optional. You could specify how many tiles wide or tall to draw from the sprite sheet.

For example, if you had more drawn sprites in the sprite editor:

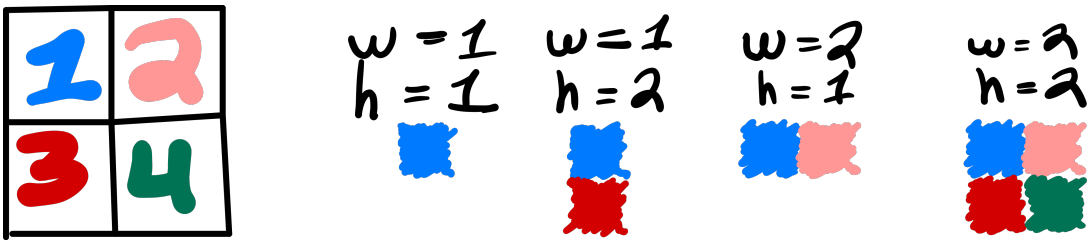
You could draw more than one using the sprite command if that is useful to you.

```
-- spr(sprite_number, x, y, w, h)

-- draw from sprite 0 at (20, 20)
-- with tile width of 1 and height of 2
spr(0, 20, 20, 1, 2)
```


```
-- draw from sprite 0 at (20, 20)
-- with tile width of 2 and height of 1
spr(0, 20, 20, 2, 1)
```

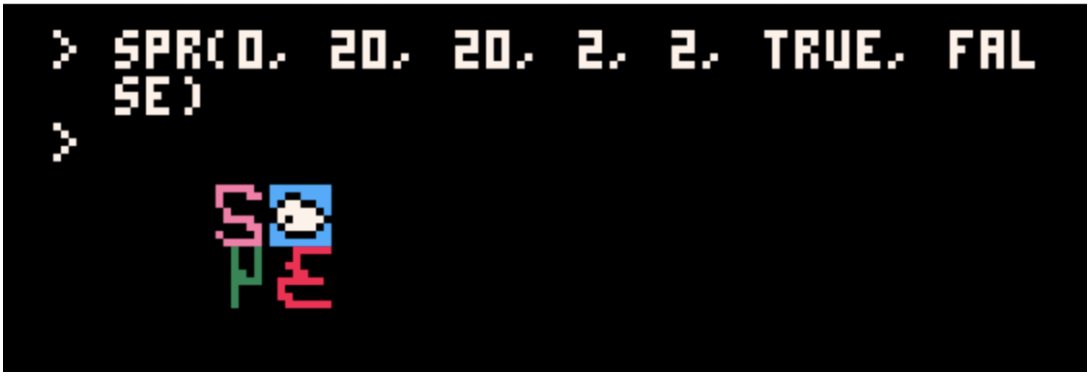
```
-- draw from sprite 0 at (20, 20)
-- with tile width of 2 and height of 2
spr(0, 20, 20, 2, 2)
```

Here is a drawing with the different `w` and `h` values and what they would draw:

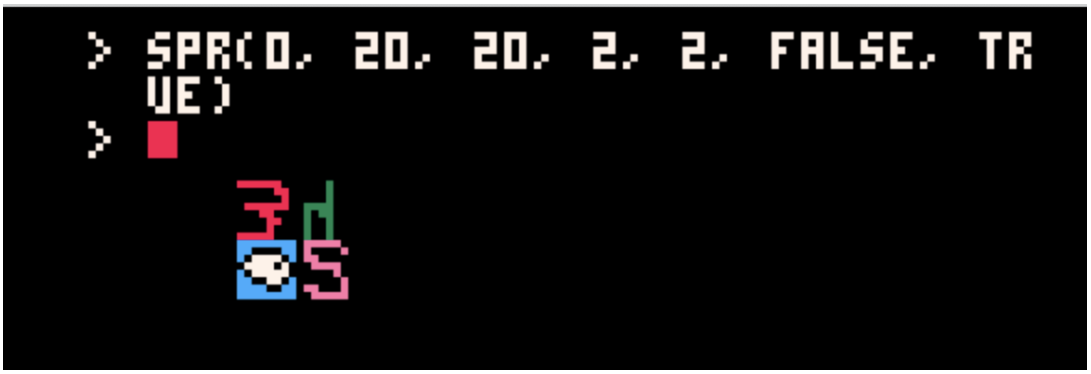


The last two optional parameters are booleans (true or false). They are `flip_x` and `flip_y`, used to flip the sprite vertically or horizontally, as desired.

```
-- draw 2x2 tiles at (20, 20), flip it vertically (flip x).
spr(0, 20, 20, 2, 2, true, false)
```



```
-- draw 2x2 tiles at (20, 20), flip it horizontally (flip y).
spr(0, 20, 20, 2, 2, false, true)
```




```
-- draw 2x2 tiles at (20, 20), flip it vertically.
spr(0, 20, 20, 2, 2, true, true)
```

Making it Move

Let's draw our cloud sprite and make it move across the screen!

First let's choose an initial X and Y position for our cloud, plug it into the `spr()` function, and write that inside the `_draw()` function.

```
function _draw()
  -- draw cloud sprite at (20, 20)
  spr(0, 20, 20)
end
```

Variables:

We can use variables to store the X and Y position of our cloud. In order to make the cloud move horizontally, we will need to change its X position over time, this is where variables come in handy.

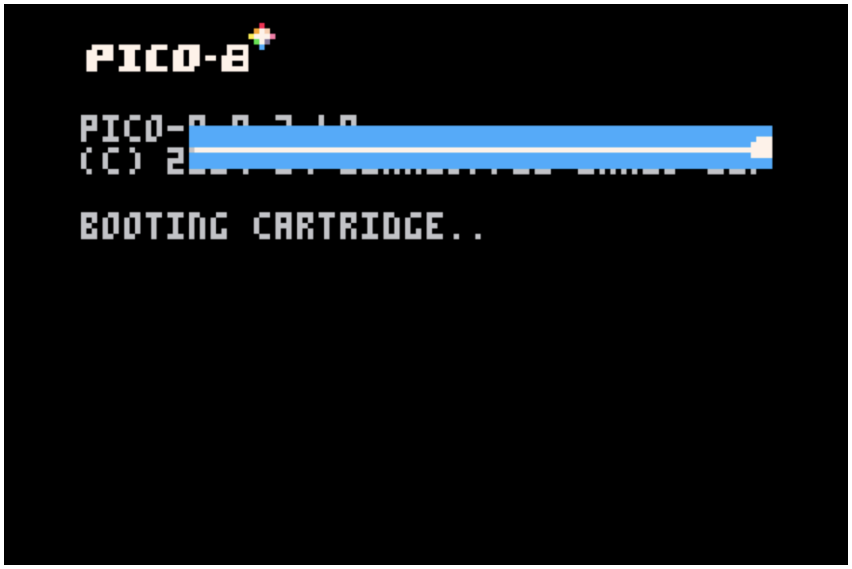
Let's create a variable for the cloud's x position: `cloud_x`.

```
cloud_x = 20

function _draw()
  -- draw cloud sprite at (20, 20)
  spr(0, cloud_x, 20)
end
```

Incrementation:

To make the cloud move, we can increment the cloud_x variable within the _update() function. This function runs 30 times per second, so by increasing `cloud_x` by a small amount each time, the cloud will appear to move across the screen from left to right.

```
cloud_x = 20

function _draw()
  -- draw cloud sprite at (20, 20)
  spr(0, cloud_x, 20)
  -- increment by 0.1 each time draw loop happens
  cloud_x = cloud_x + 0.1
end
```



The sprite appears to move, but it leaves a trail behind it. This is because the screen is not being cleared before each new frame is drawn. By cleared, we mean filling the screen with a solid color before drawing the new frame.

Clearing the Screen; CLS() Function:

To clear the screen, we can use the **CLS()** function at the beginning of the _draw( ) function. This function fills the screen with a solid color (default is black) before drawing anything else.

```
cls(0)
```

```
function _draw()
    -- clear the screen by drawing sa olid black color
    cls(0)
    -- draw cloud sprite at (20, 20)
    spr(0, cloud_x, 20)
    -- increment by 0.1 each time draw loop happens
    cloud_x = cloud_x + 0.1
end
```

NOTE

`cls(0)` is the equivalent of using:

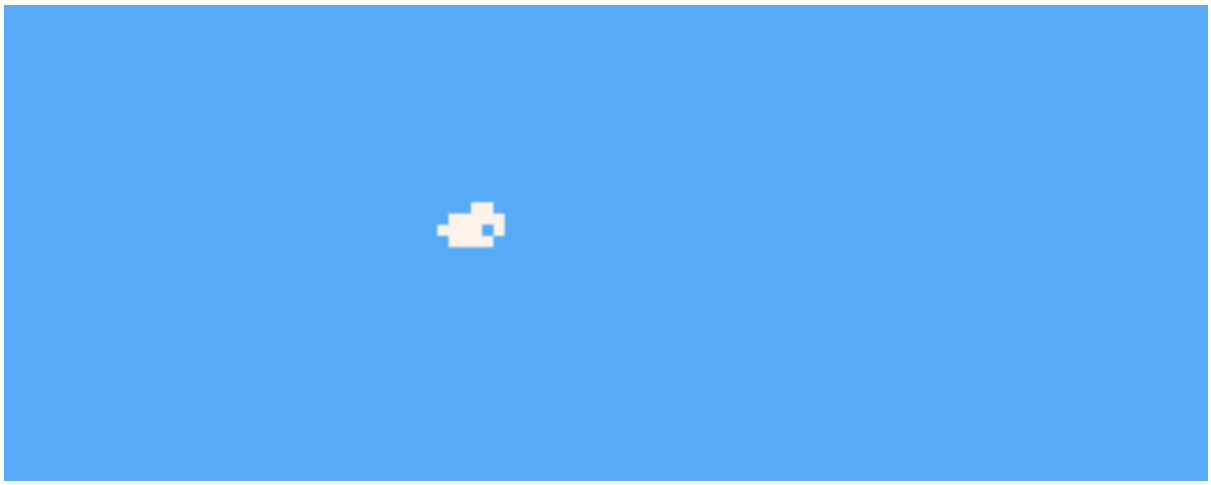
```
rectfill(0,0,127,127,0)
```

This looks like a moving cloud! I drew the cloud sprite with a blue background, though..

Instead of using the default clearing the screen with a black solid color, let's use the same blue:

Looking at my color palette with their assigned numbers, I know that the blue I used is assigned the value of 12.

```
function _draw()
    -- clear the screen by drawing a solid black color
    cls(12)
    -- draw cloud sprite at (20, 20)
    spr(0, cloud_x, 20)
    -- increment by 0.1 each time draw loop happens
    cloud_x = cloud_x + 0.1
end
```



That looks like a sky! But something happened with my black outline. This is because black is used as the default transparency color. If you really want black to be opaque, you can add this line in `_init()`:

```
function _init()
  palt(0, false)
end
```

Or you could use a different color. I'll just use the dark blue color instead.

Wrapping Around the Screen

When the cloud completely exits the frame, we are left with just a blank blue screen. We should reset it to the left side so that it becomes visible again.

To do this, we can use an **if statement**.

> "**if** the cloud is out of frame on the right, then reset it to a position on the left side"

We can write the code in the _update() function.

```
function _update()
  if cloud_x > 127 then
    cloud_x = -8
  end
end
```

If cloud_x becomes higher than 127, it means it has completely left the frame; therefore, that is when the reset should happen.

The reset position should be a negative value that is the width of the sprite size, so -8.

Full Minimal Program

```
cloud_x = 20

function _update()
    -- reset cloud to the left side when out of frame
    if cloud_x > 127 then
        cloud_x = -8
    end
end

function _draw()
    -- clear the screen by drawing a solid black color
    cls(12)
    -- draw cloud sprite at (20, 20)
    spr(0, cloud_x, 20)
    -- increment by 0.1 each time the draw loop happens
    cloud_x = cloud_x + 0.1
end
```

Adding More Clouds (Making Objects)

> **TIP**
> To take a screenshot of your screen, press **Ctrl + 6.**