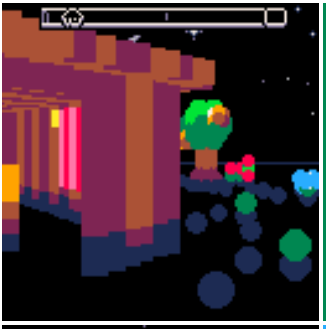
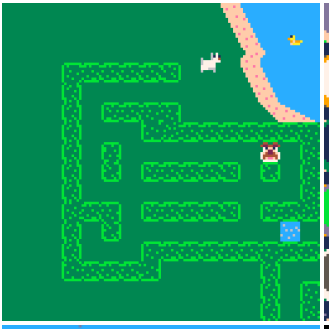
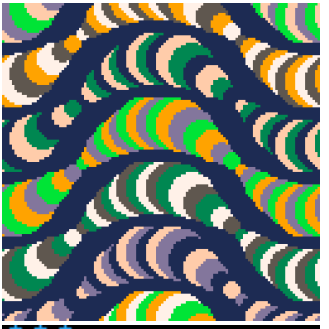
ISSUE #1

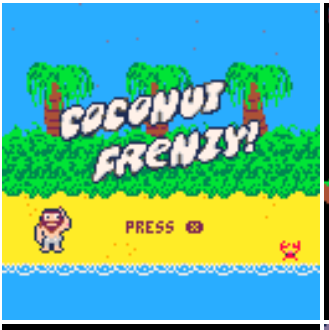
GAME DEVELOPMENT with PICO-8

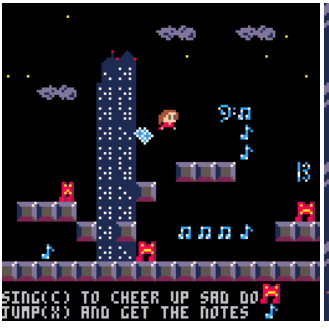
BY DYLAN BENNETT











PICO-8 COMMUNITY CREATIONS
(@:TWITTER ✚:BBS)

@FREDBEDNARSKI
@TRASEVOL_DOG
DR4IG ✚
@QUICKALAS529

@RTRNTOSPIELBURG
@PLATFORMALIST
@SCOTAIRE
@TOMSOLACROUP

@JOHANPEITZ
@GUERRAGAMES
ERIBAN ✚
@LIQUIDREAM

TABLE OF CONTENTS

INTRODUCTION

The original reason for creating this zine was simply to have printed materials for a PICO-8 workshop I was teaching with the Portland Indie Game Squad (PIGSquad). While it still has that purpose, I've decided to make it useful for anyone who wants to pick up PICO-8 and get started. I love making things in PICO-8, and I hope you will too.

The idea for this zine was absolutely inspired by Arnaud De Bock's famous PICO-8 fanzines. Just as they allowed me to easily get started with PICO-8, I can only hope this zine does the same for others.

There is so much more I would have loved to add to this issue, but I could only make it so long in the time I had. However, that means I have plenty of material for future issues, of which I hope there will be many.

I had a lot of help and support while putting this together and I appreciate all of it.

Enjoy!
Dylan (@MBoffin)

Contact: @MBoffin ~ mboffin.itch.io
Patreon: patreon.com/mboffin

A WORD ABOUT PICO-8

I was captivated by the charm of PICO-8 from the first moment I tried it. I have yet to meet anyone who doesn't take a liking to PICO-8 after seeing it in action. There's something about it that just captures people's hearts.

It's well known that creativity thrives within constraints. Nowhere is that more true than with PICO-8. Limited screen size, color palette, code length, and so on all contribute to an environment where you are actually free to be *more* creative than you might be with other game engines.

Unity is a great example of a game engine with very few constraints on what you can create. While that's good, it also means you have *many* decisions to make. PICO-8's constraints do away with many of those decisions and let you focus on just creating your game.

For example, a Unity game might have to work on dozens of screen resolutions, but in PICO-8, you get one resolution of 128x128. This frees you to put more effort into making your game work well in that one resolution.

A WORD ABOUT PICO-8

On the other side of the coin, PICO-8's constraints, such as code length, will inevitably force you to make decisions—decisions such as what is *really* important to keep in your game. Unity has no constraint on code length, so you're free to just keep adding as many features as you wish. This is not necessarily a good thing when it comes to game development.

There's another aspect to PICO-8's charm that, for me, reaches back decades. When I was a kid, I remember magazines that would include whole programs written in BASIC. You just typed in the code and ran it! Sometimes the program made a cool picture, sometimes it would be a simple game. By itself, this was interesting and fun, but what *really* captured my curiosity and hooked me for life was changing the code to make it do new things. I found something akin to this in PICO-8, and it's rooted in the culture of the PICO-8 community.

Joseph White (aka zep), the creator of PICO-8, has created a community around PICO-8 where sharing what you create is not only easy, but encouraged. When you load someone else's creation in PICO-8, you can run it, but you can *also* look at the code,

A WORD ABOUT PICO-8

change the sprites, do whatever you want. You have as much access as the person who created it! For me, this really harkens back to those BASIC programs found in magazines.

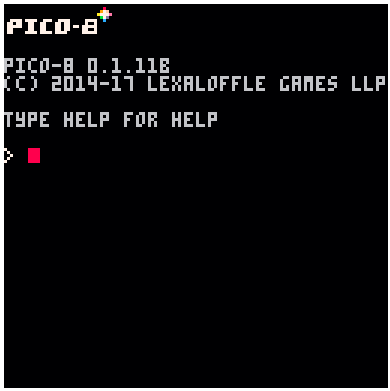
The easiest way to share your PICO-8 creations is by making a game cartridge, or cart for short. These are like digital versions of physical game cartridges. They're easy to make and easy to share because they're just images! The really cool part is that all of your game's information is stored in the image of the cart! All of it! The code, art, music, everything. If someone has that cart image, they have everything they need to run your game in PICO-8.

As you create new things in PICO-8, I encourage you to share what you create with others on the Lexaloffle web site. (See page 58 on how to do that.) Just as you can learn from what others have created, others will be able to learn from what you are creating.

I can't wait to see what you create!

USING PICO-8

When you first run PICO-8, you start in a mode where you can type in commands. From this mode you can type commands like **SAVE**, **LOAD**, and **RUN**. You can use the command **HELP** to see what other commands you can run from this mode.



Use the ESC key to switch back and forth between editor mode and command mode. When you are playing a game and hit ESC, you'll come back to command mode. Just hit ESC again to go into editor mode.

You'll find notes about each editor and shortcuts relevant to each on the following pages.

The shortcuts below are possible no matter which editor you are currently using.

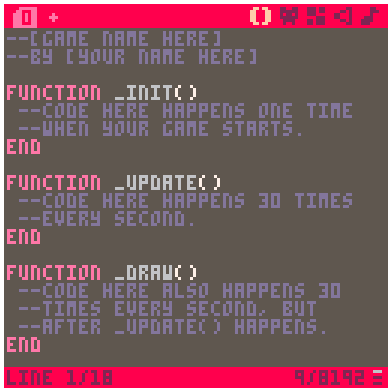
Shortcuts:

- **Alt-Right/Left** - Next/previous editor
- **Ctrl-S** - Save
- **Ctrl-R** - Run
- **Ctrl-M** - Mute/Unmute
- **Alt-Enter** - Fullscreen
- **Alt-F4, Cmd-Q** - Quit

(Use Cmd instead of Ctrl on macOS.)

CODE EDITOR

Notes:



All your code is written here. One of PICO-8's limits is how much code you can use. This limit is a bit hard to understand for people new to PICO-8. It's based on something called tokens. These are basically individual bits of code. For example, something like `X=WIDTH+1` takes up five tokens, one token for each part (`X`, `=`, `WIDTH`, `+`, and `1`). You are allowed 8192 tokens of code, so you'll be fine until you're making a fairly large game. You can see the number of tokens you've used in the bottom-right.

Shortcuts:

- **Alt-Up/Down** - Go up/down a function at a time
- **Ctrl-L** - Move to a specific line number
- **Ctrl-Up/Down** - Move to the very top/bottom
- **Ctrl-Left/Right** - Move left/right by one word
- **Ctrl-F, Ctrl-G** - Find text, or find again
- **Ctrl-D** - Duplicate the current line
- **Tab/Shift-Tab** - Indent/un-indent the currently selected line(s)
- **Ctrl-Tab/Shift-Ctrl-Tab** - Move to next/previous code tab

(Use Cmd instead of Ctrl on macOS.)

SPRITE EDITOR

Notes:

Sprites are the pieces of art that make up your game. They might be characters, map tiles, pickups, titles, backgrounds, anything.

PICO-8 allows you to have 256 8x8 sprites. These are split across 4 tabs labeled 0-3. However, the last two tabs are shared with the Map Editor. So if you have a really big map, you won't be able to use the last two tabs of sprites. But if you're using the last two tabs of sprites, you won't be able to use the lower half of the Map Editor.

Shortcuts:

- **H/V** - Flip the sprite horizontally/vertically
- **R** - Rotate the sprite clockwise
- **Q/W** or **- / =** - Move to the previous/next sprite
- **Shift-Q/Shift-W** or **_ / +** - Move one row of sprites back/forward
- **1/2** - Move to the previous/next color
- **Up/Down/Left/Right** - Loop sprite
- **Mousewheel Up/Down, < / >** - Zoom in/out
- **Space** - Pan around while space is held down
- **Right-click** - Select the color under the mouse

MAP EDITOR

Notes:

PICO-8's map tiles use the 8x8 sprites from the Sprite Editor. This means 16x16 tiles will fill an entire screen.

Even though you can have a maximum map size of 128 tiles wide and 64 tiles tall, the lower half of the map actually shares space with the last two tabs of the Sprite Editor. So you need to decide if you want a large map or if you want a lot of sprites.

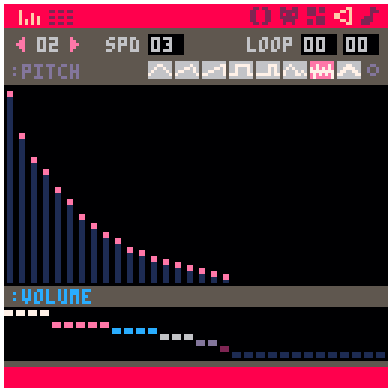
No matter what is drawn in sprite #0, that sprite is used as an "eraser" sprite. You can use it to erase map tiles.

Shortcuts:

- **Mousewheel Up/Down, < / >** - Zoom in/out
- **Space** - Pan around while space is held down
- **Q/W**, **- / =** - Move to the previous/next sprite
- **Shift-Q/Shift-W**, **_ / +** - Move one row of sprites back/forward
- **1/2** - Move to the previous/next color
- **Up/Down/Left/Right** - Loop sprite
- **Right-click** - Select the sprite under the mouse

SOUND EDITOR

Notes:



A PICO-8 cart can have up to 64 sounds. Each sound has 32 notes. You can control the frequency, instrument, volume, and effect for each note. You can also change the playback speed of the whole sound and make sections of it loop.

The Sound Editor has two modes: pitch mode and tracker mode. Pitch mode is useful for simple sound effects, whereas tracker mode is useful for music. See page 68 for a PICO-8 music reference to use for tracker mode.

Shortcuts:

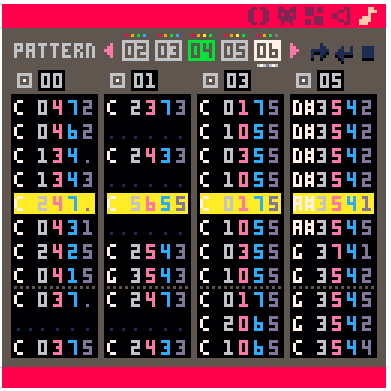
- **Space** - Play/stop
- **- / +** - Go to previous/next sound
- **< / >** - Change the speed of the current sound
- **Shift-Space** - Play the current group of 8 notes
- **Shift-Click** on an instrument, effect, or volume to change all notes in a sound at once
- **Ctrl-Up/Ctrl-Down**, **PgUp/PgDn** - Move up/down 4 notes at a time (tracker mode only)
- **Ctrl-Left/Ctrl-Right** - Switch columns (tracker mode only)

(Use Cmd instead of Ctrl on macOS.)

MUSIC EDITOR

Notes:

The Music Editor allows you to create patterns of music using the sounds from the Sound Editor. Each pattern has four channels that can each contain a sound from the Sound Editor.



Playback of patterns is controlled by the three buttons in the top-right (two arrows and a square). If the right-facing arrow is on, that marks a start point. If the left-facing arrow is on, that marks a loop point. If the square button is on, that marks a stop point.

Playback flows from the end of one pattern to the beginning of the next. However, if playback reaches the end of a pattern and finds a loop point, it will search backward until it finds a start point and play from there. If playback reaches the end of a pattern and finds a stop point, playback will stop.

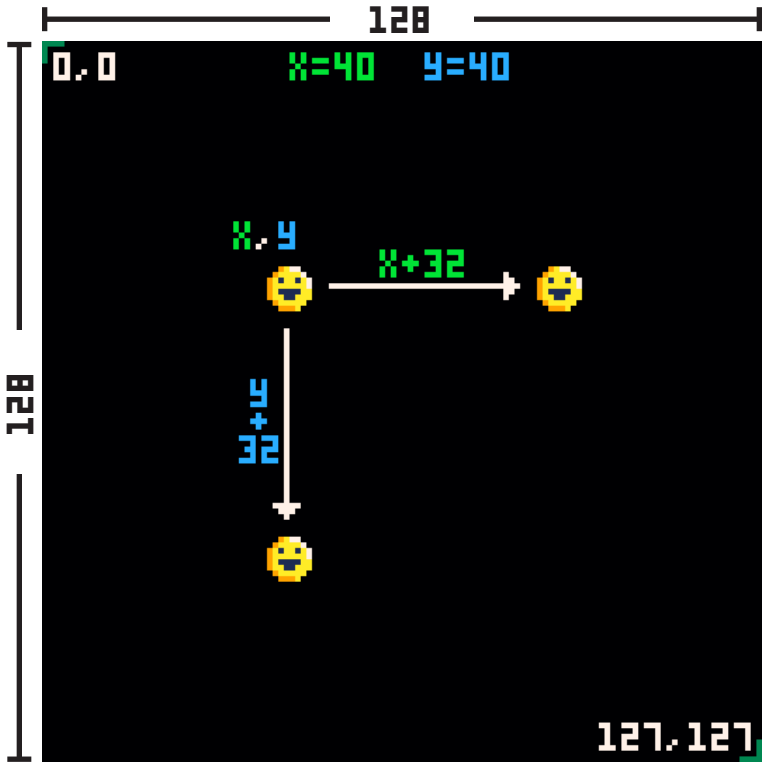
Shortcuts:

- **Space** - Play/stop
- **- / +** - Go to previous/next pattern

Note: You can edit sounds in the Music Editor, so most Sound Editor shortcuts also work here!

COORDINATES

PICO-8's screen space is 128 pixels wide and 128 pixels tall. This may not seem like a much at first, but you can do a lot in that amount of space!



Notice the coordinate `0,0` is in the top-left and coordinate `127,127` is in the bottom-right. This means positive `x` goes to the right and positive `y` goes *down*. (This may be different from what you're used to, where positive `y` is usually *up*.) Also remember that because we start counting at `0`, the position `127` is actually the 128th pixel.

PROGRAMMING BASICS

Fitting a full introduction to programming in this zine just wouldn't be worth it. I wouldn't be able to do the job justice *and* still get to all the fun stuff PICO-8 has to offer. Not to mention, there are already so many great introductions to programming on the Internet.

However, there are a few specific things I would like to ensure are covered before we get to all the fun stuff. These are particularly important. Even if you don't know much about programming, you'll be able to follow along if you just understand these few things. If know programming already, you can skip all this stuff.

Variables

Variables are ways to store information with an easy-to-remember name. As the name "variable" implies, the information stored in the variable can *vary*, or change. In PICO-8, variables can hold numbers, text, and the value `TRUE` or `FALSE`. Here are a few examples of variables:

```
X=64
NAME="DYLAN"
ALIVE=TRUE
```

Some words are reserved and you can't use them for variable names (like the word "function"). You also can't start the name of a variable with a number.

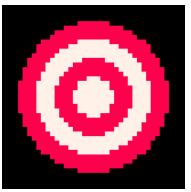
Functions

Functions are a list of instructions for the computer that are all grouped together under one name. Functions are usually created if you have a certain set of actions you want the computer to do many different times.

Functions are written with parentheses after the name of the function. This is so you can give the function extra information in case it needs that extra information to do its job. Even if no extra information is needed, you still need to write the parentheses.

Here's an example function called `DRAW_TARGET()`. It draws a target shape using filled circles. Note that it needs an X and a Y coordinate to do its job:

```
FUNCTION DRAW_TARGET(X,Y)
 CIRCFILL(X,Y,16,8)
 CIRCFILL(X,Y,12,7)
 CIRCFILL(X,Y,8,8)
 CIRCFILL(X,Y,4,7)
END
```



Maybe you noticed something: `CIRCFILL()` is a function too! It's a function built into PICO-8, so you don't have to write the steps yourself, but it's still a function. You give it an X/Y coordinate, a radius, and a color, and it draws a filled circle at X/Y, at that radius, and with that color. And `CIRCLFILL()` is just one of many built-in functions!

Usually a function just does the job you need it to do and that's that, like the `DRAW_TARGET()` function above, or `CIRCLFILL()`. But sometimes you need a function to give back, or *return*, information when it's done doing all of its steps.

Say you make a function that does a bunch of math, but you want to know the result when it's done. In other words, you want it to *return* the result back to you. Easy enough. You just use `RETURN` and then specify *what* you want it to return. Here's a real example:

```
FUNCTION AREA(WIDTH,HEIGHT)
 RETURN WIDTH * HEIGHT
END

W=8
H=5

IF (AREA(W,H) > 25) THEN
 PRINT("BIG!")
END
```

Hard to tell which letter is which? Check the font reference on page 70!

When that function gets run, the number *returned* would be `40`. Since `40` is indeed greater than `25`, the `PRINT()` function would then happen.

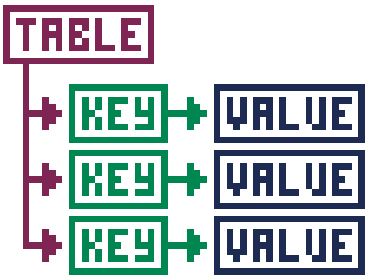
Functions are the backbone of anything you will create in PICO-8. Most games are really just many, many functions strung together, each one making changes to things in the game as the players play. Really understanding how your code moves from one function to another is the key to being able to make great games.

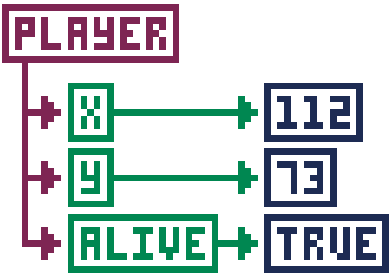
Tables

Tables are a way to store a lot of information all together under one variable name. Most PICO-8 games will use a table at some point or another, so it's good to understand how they work.

When you add a piece of information, or *value*, to a table, it gets paired with a name or a number called a *key*. The key is what you use to get the information back out of the table. You can say, "Look up the information stored in *that* table using *this* key." Keys are like the index in a book.

If you add values to a table without setting the key, the key will automatically be assigned as a number. Let's see an example of what this looks like.





Now let's see how that looks in code. Take note how we create the `PLAYER` table using empty curly braces. Then we add the values with named keys. For the `ITEMS` table, we create the table with the values inside the curly braces, but without names. The keys get automatically assigned as numbers.

```
PLAYER={}
PLAYER.X=112
PLAYER.Y=73
PLAYER.ALIVE=TRUE

ITEMS={"SWORD","CLOAK","BOOTS"}
```

That's how to get values *into* a table. But what about getting values back *out*? For keys that are names, you can just use `TABLE.KEY`, such as `PLAYER.X` or `PLAYER.ALIVE`. But for keys that are numbers, you use square brackets with the number of the key inside, such as `ITEMS[1]` or `ITEMS[3]`.

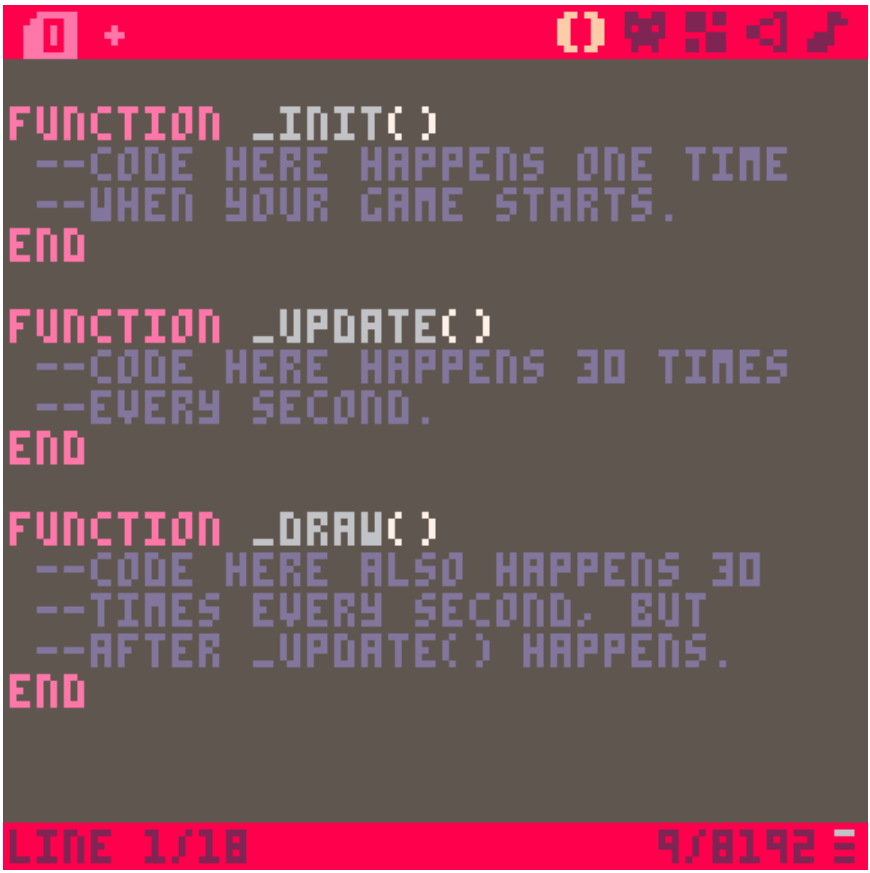
If your table uses numbers for keys, you can find out how many values are stored in a table by using the number sign (#), such as `#ITEMS`. In our example, this would give you `3`. This is useful if you have to loop through all the values in a table and do something with each value. Here's an example:

```
FOR I=1,#ITEMS DO
 PRINT(ITEMS[I])
END
```

This starts `I` at `1` and counts to `#ITEMS` (which is `3`). Each time, it will print the value at `ITEMS[I]`. Since `I` goes from `1` to `3`, every item will be printed.

THE GAME LOOP

PICO-8 uses three specially-named functions to create what's called a *game loop*. The `_INIT()` function happens one time, then `_UPDATE()` and `_DRAW()` happen in a loop until your game ends. Here's the basic structure of the PICO-8 game loop and what each functions does:



You *could* put all of your code inside these three functions, but it's generally not considered a good idea. A better solution is usually to make other

THE GAME LOOP

functions that do specific things, and then have `_INIT()`, `_UPDATE()`, or `_DRAW()` run *those* functions.

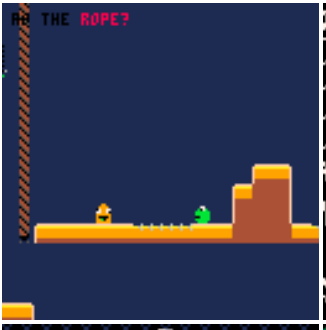
For example, instead of putting player movement code in `_UPDATE()`, write your own function called `MOVE_PLAYER()` and run *that* inside `_UPDATE()`. Here's an example of how it would all look:

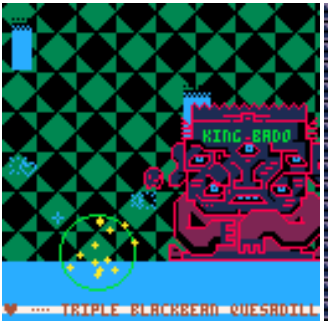
```
FUNCTION _INIT()
 MAKE_PLAYER()
END

FUNCTION _UPDATE()
 MOVE_PLAYER()
END

FUNCTION _DRAW()
 CLS() --CLEAR SCREEN
 DRAW_PLAYER()
END

FUNCTION MAKE_PLAYER()
 PX=64
 PY=64
 PSPRITE=1
END

FUNCTION MOVE_PLAYER()
 IF (BTN(0)) PX-=1 --LEFT
 IF (BTN(1)) PX+=1 --RIGHT
 IF (BTN(2)) PY-=1 --UP
 IF (BTN(3)) PY+=1 --DOWN
END

FUNCTION DRAW_PLAYER()
 SPR(PSPRITE,PX,PY)
END
```

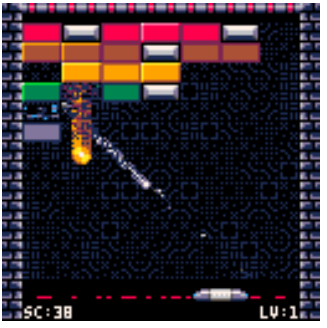
Try it out! Type this code into PICO-8 and run it!

But don't forget to make sprite #1!

See how the game loop functions are kept nice and tidy? Now you can see a good overview of how the game works just from those three functions.







PICO-8 COMMUNITY CREATIONS
(@:TWITTER ✚:BBS)

@JOHANPEITZ
@QUICKALAS529
ULTIMAN3RD ✚
@LIQUIDREAM

ELECTRICGRYPHON ✚
@MORNINGTOAST
@NEKO250
@TAECKERWYSS

ICWILK ✚
@GUERRAGAMES
@MATTHUGHSON
ULTRABRITE ✚

TUTORIALS

When doing these tutorials, go ahead and change things ! Don't want a gray cave? Make it green! Gravity is too light? Make it stronger! PICO-8 is a great environment for playing and tinkering, and these tutorials are no exception.

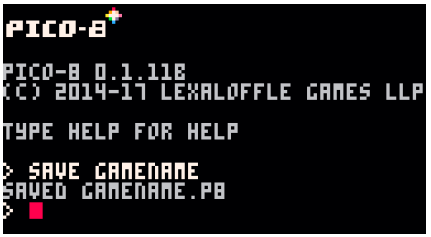
When writing the code, you'll often add on to what you've already written. Code you've already written will be gray and new code will **black**.

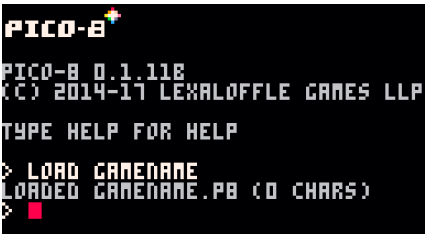
```
THIS IS CODE YOU'VE ALREADY WRITTEN!
```

```
THIS IS NEW CODE!
```

It's important to save your work as you go. But just as importantly, you need to know how to load your game later.

Use the `SAVE` command along with the name of your game to save your game. (Don't use spaces in the name, though!) PICO-8 will add `.P8` to the end of the filename so that your computer knows it's a PICO-8 game. You can use the `LOAD` command to load your game later. At any time after you've saved or loaded your game, you can hit CTRL-S (CMD-S on macOS) to save any changes.





CAVE DIVER - STEP 1

The first game we'll make is a classic, one-button, side-scrolling game. There have been hundreds of games like this, the most recent hit being *Flappy Bird*. In our variation, we're flying/bouncing through a cave trying to get as deep into the cave as we can. It's a fun, easy game!

NOTE: Start a new game by rebooting PICO-8 with the `REBOOT` command. (Hit ESC if you aren't already in command line mode.) Then be sure to save your game with the instructions on the previous page!

We only need to create three sprites in the Sprite Editor for this whole game. Sprite #1 for jumping, #2 for falling, and #3 for when you hit the walls.

CAVE DIVER - STEP 1

0 1 +

```
FUNCTION _INIT()
 GAME_OVER=FALSE
 MAKE_PLAYER()
END

FUNCTION _UPDATE()
END

FUNCTION _DRAW()
 CLS()
 DRAW_PLAYER()
END
```

Hard to tell which letter is which? Check the font reference on page 70!

0 1 +

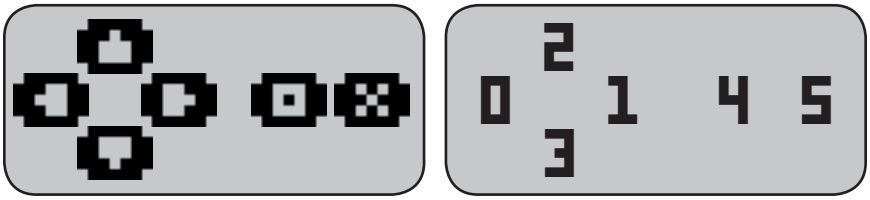
```
FUNCTION MAKE_PLAYER()
 PLAYER={}
 PLAYER.X=24     --POSITION
 PLAYER.Y=60
 PLAYER.DY=0     --FALL SPEED
 PLAYER.RISE=1   --SPRITES
 PLAYER.FALL=2
 PLAYER.DEAD=3
 PLAYER.SPEED=2 --FLY SPEED
 PLAYER.SCORE=0
END

FUNCTION DRAW_PLAYER()
 IF (GAME_OVER) THEN
  SPR(PLAYER.DEAD,PLAYER.X,PLAYER.Y)
 ELSEIF (PLAYER.DY<0) THEN
  SPR(PLAYER.RISE,PLAYER.X,PLAYER.Y)
 ELSE
  SPR(PLAYER.FALL,PLAYER.X,PLAYER.Y)
 END
END
```

SAVE & RUN IT!

Okay, so it's not mind-blowing yet. But it should at least work and show you the player on the screen.

CAVE DIVER - STEP 2

Let's make that player jump with the UP button! PICO-8 uses numbers 0 through 5 to represent each button the player can press. Here is how each number connects up to each button:



Remember, **code in gray** is code you've already written. Just add the code in **black**!

```
FUNCTION _UPDATE()
 MOVE_PLAYER()
END
```

```
FUNCTION MOVE_PLAYER()
 GRAVITY=0.2 --BIGGER MEANS MORE GRAVITY!
 PLAYER.DY+=GRAVITY --ADD GRAVITY

 --JUMP
 IF (BTNP(2)) THEN
  PLAYER.DY-=5
 END

 --MOVE TO NEW POSITION
 PLAYER.Y+=PLAYER.DY
END
```

SAVE & RUN IT!

Bouncy! The secret to this game is that the player never moves forward, just up and down!

CAVE DIVER - STEP 3

Let's modify our game loop functions to get ready to add the cave. Then we'll add the cave functions.

```
function _init()
 game_over=false
 make_cave()
 make_player()
end

function _update()
 update_cave()
 move_player()
end

function _draw()
 cls()
 draw_cave()
 draw_player()
end
```

The cave is really just a list from left to right of how low to draw the ceiling and how high to draw the floor for each column of the cave. The faster we add columns to the end and remove columns from the beginning, the faster the cave flows by!

CAVE DIVER - STEP 3

When adding new columns, we look at the last column's floor and ceiling heights. Then we go up or down randomly from there, but only just a little bit, so it looks like a natural change.

```
FUNCTION MAKE_CAVE()
 CAVE={{["TOP"]=5,["BTM"]=119}}
 TOP=45 --HOW LOW CAN THE CEILING GO?
 BTM=85 --HOW HIGH CAN THE FLOOR GET?
END

FUNCTION UPDATE_CAVE()
 --REMOVE THE BACK OF THE CAVE
 IF (#CAVE>PLAYER.SPEED) THEN
  FOR I=1,PLAYER.SPEED DO
   DEL(CAVE,CAVE[1])
  END
 END

 --ADD MORE CAVE
 WHILE (#CAVE<128) DO
  LOCAL COL={}
  LOCAL UP=FLR(RND(7)-3)
  LOCAL DWN=FLR(RND(7)-3)
  COL.TOP=MID(3,CAVE[#CAVE].TOP+UP,TOP)
  COL.BTM=MID(BTM,CAVE[#CAVE].BTM+DWN,124)
  ADD(CAVE,COL)
 END
END

FUNCTION DRAW_CAVE()
 TOP_COLOR=5 --PLAY WITH THESE!
 BTM_COLOR=5 --CHOOSE YOUR OWN COLORS!
 FOR I=1,#CAVE DO
  LINE(I-1,0,I-1,CAVE[I].TOP,TOP_COLOR)
  LINE(I-1,127,I-1,CAVE[I].BTM,BTM_COLOR)
 END
END
```

To better understand the code in this step, make sure to read the section on tables! (pg18)

SAVE & RUN IT!

CAVE DIVER - STEP 4

You probably noticed you can run into the sides of the cave and nothing happens. Let's fix that!

Also, certain things should stop happening if we hit the sides of the cave. For example, the cave and the player should stop moving. We'll use the variable `GAME_OVER` for that.

```
FUNCTION _UPDATE()
 IF (NOT GAME_OVER) THEN
  UPDATE_CAVE()
  MOVE_PLAYER()
  CHECK_HIT()
 END
END
```

```
FUNCTION CHECK_HIT()
 FOR I=PLAYER.X,PLAYER.X+7 DO
  IF (CAVE[I+1].TOP>PLAYER.Y
   OR CAVE[I+1].BTM<PLAYER.Y+7) THEN
   GAME_OVER=TRUE
  END
 END
END
```

SAVE & RUN IT!

Everything should stop if you hit the sides of the cave! You'll notice that we now get to see Sprite #3 when the game is over.

We almost have a complete game! We're so close! All we have left is to add a score (so we can see how far we've traveled), some sounds, and a way to restart the game if we crash into the cave.

CAVE DIVER - STEP 5

Our player already has a score, so let's add to it as the player moves. We'll show the score in the corner as they play. Then when the game's over, we'll tell them and show the player their score.

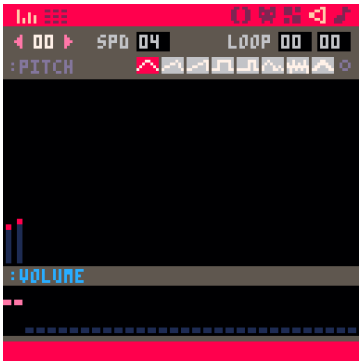
```
FUNCTION _DRAW()
 CLS()
 DRAW_CAVE()
 DRAW_PLAYER()

 IF (GAME_OVER) THEN
  PRINT("GAME OVER!",44,44,7)
  PRINT("YOUR SCORE:"..PLAYER.SCORE,34,54,7)
 ELSE
  PRINT("SCORE:"..PLAYER.SCORE,2,2,7)
 END
END
```

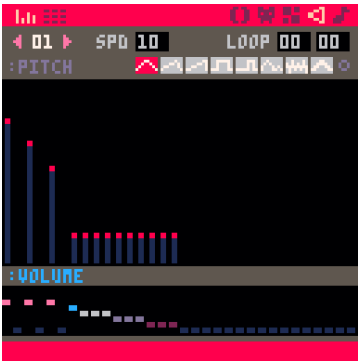
```
FUNCTION MOVE_PLAYER()
 --ADD GRAVITY
 PLAYER.DY+=0.2

 --JUMP
 IF (BTNP(2)) THEN
  PLAYER.DY-=5
 END

 --MOVE TO NEW POSITION
 PLAYER.Y+=PLAYER.DY

 --UPDATE SCORE
 PLAYER.SCORE+=PLAYER.SPEED
END
```

SAVE & RUN IT!

CAVE DIVER - STEP 6

Now let's add sound! Sound #0 will be the jump sound and sound #1 will be the game over sound.






```
FUNCTION MOVE_PLAYER()
 --ADD GRAVITY
 PLAYER.DY+=0.2

 --JUMP
 IF (BTNP(2)) THEN
  PLAYER.DY-=5
  SFX(0)
 END

 --MOVE TO NEW POSITION
 PLAYER.Y+=PLAYER.DY

 --UPDATE SCORE
 PLAYER.SCORE+=PLAYER.SPEED
END

FUNCTION CHECK_HIT()
 FOR I=PLAYER.X,PLAYER.X+7 DO
  IF (CAVE[I+1].TOP>PLAYER.Y
   OR CAVE[I+1].BTM<PLAYER.Y+7) THEN
   GAME_OVER=TRUE
   SFX(1)
  END
 END
END
```

SAVE & RUN IT!

CAVE DIVER - STEP 7

Our very last step is easy. When the player loses, we need to wait for a button to be pressed to restart the game. But we also need to tell the player which button that is!

(Hint: hit **shift-x** to make the ❎ character.)

```
FUNCTION _UPDATE()
 IF (NOT GAME_OVER) THEN
  UPDATE_CAVE()
  MOVE_PLAYER()
  CHECK_HIT()
 ELSE
  IF (BTNP(5)) _INIT() --RESTART
 END
END

FUNCTION _DRAW()
 CLS()
 DRAW_CAVE()
 DRAW_PLAYER()

 IF (GAME_OVER) THEN
  PRINT("GAME OVER!",44,44,7)
  PRINT("YOUR SCORE:"..PLAYER.SCORE,34,54,7)
  PRINT("PRESS ❎ TO PLAY AGAIN!",18,72,6)
 ELSE
  PRINT("SCORE:"..PLAYER.SCORE,2,2,7)
 END
END
```

SAVE & RUN IT!

You did it! You made a game! Now get your friends to play and see who can get the highest score!

CAVE DIVER - ETC

We have a complete game at this point, but there's a lot we could continue adding from here. Maybe things to pick up along the way that make you go faster or slower? Maybe extra lives? Maybe enemies to avoid? The list is endless.

However, try not to get stuck in adding every feature you want before letting others play it. Let people play it early and often! Get feedback from players on what they like and don't like. Really listen to what they have to say. What might sound fun in your head might not be as fun once it's in the game.

At this point, if you want your friends and family to play, they need to come to your computer to play. That's fine, but not always workable. On page 58 you'll find a section with easy instructions for posting your game on the web where others can play it.

Are you ready to make your next game? Let's go!

LANDER - STEP 1

The second game we'll make is a bit more complex than the first game, but still a lot of fun. In this game you're guiding a lander onto a landing pad.

Just like the first tutorial, use the **REBOOT** command to start a new game.

Let's add code to draw the lander. You can make your lander look however you want. Just make sure to put it in Sprite #1. (The second sprite spot.)

```
FUNCTION _INIT()
 MAKE_PLAYER()
END

FUNCTION _UPDATE()
END

FUNCTION _DRAW()
 CLS()
 DRAW_PLAYER()
END

FUNCTION MAKE_PLAYER()
 P={}
 P.X=60          --POSITION
 P.Y=8
 P.DX=0          --MOVEMENT
 P.DY=0
 P.SPRITE=1
 P.ALIVE=TRUE
 P.THRUST=0.075
END

FUNCTION DRAW_PLAYER()
 SPR(P.SPRITE,P.X,P.Y)
END
```

SPRITE #1

SAVE & RUN IT!

LANDER - STEP 2

As you can see, that just showed the lander. So let's add some gravity and make our lander fall!

We need a function to move the player. We move the player by adding the player's movement (`P.DX` and `P.DY`) to the player's position (`P.X` and `P.Y`).

Then to add gravity to our game, we just make sure we're always adding a gravity amount (`G`) to the player's up/down movement (`P.DY`).

Remember, **gray code** is code you already wrote!

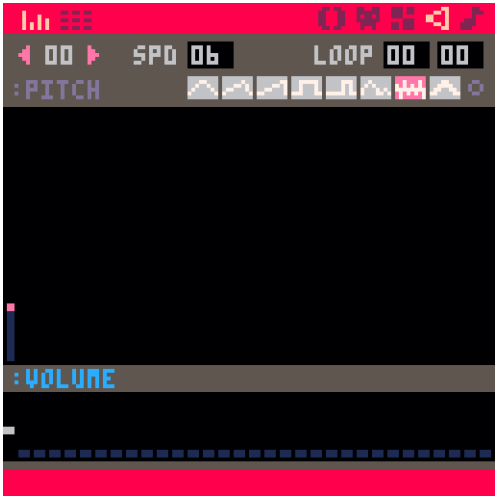
```
FUNCTION _INIT()
 G=0.025 --GRAVITY
 MAKE_PLAYER()
END

FUNCTION _UPDATE()
 MOVE_PLAYER()
END

FUNCTION MOVE_PLAYER()
 P.DY+=G --ADD GRAVITY

 P.X+=P.DX --ACTUALLY MOVE
 P.Y+=P.DY --THE PLAYER
END
```

SAVE & RUN IT!

The lander falls now! Because `MOVE_PLAYER()` happens every time the game updates (30 times a second), gravity will always be added to the player's movement. However, the player still has no control over the lander, so let's add that.

LANDER - STEP 3

When the player thrusts, we want it to play an engine sound. Use the Sound Editor to make that in Sound #0. Make sure to use the second to last instrument! (It's the most engine-thrusty sound.)



Things to note:

- Speed is at **06**
- Use this sound: 
- You only need one dot of sound! (And the lower the note used, the more engine-thrusty it will sound.)

```
FUNCTION MOVE_PLAYER()
 P.DY+=G --ADD GRAVITY

 THRUST()

 P.X+=P.DX --ACTUALLY MOVE
 P.Y+=P.DY --THE PLAYER
END

FUNCTION THRUST()
 --ADD THRUST TO MOVEMENT
 IF (BTN(0)) P.DX-=P.THRUST
 IF (BTN(1)) P.DX+=P.THRUST
 IF (BTN(2)) P.DY-=P.THRUST

 --THRUST SOUND
 IF (BTN(0) OR BTN(1) OR BTN(2)) SFX(0)
END
```

SAVE & RUN IT!

LANDER - STEP 4

We can fly our lander now, but you'll notice we can go zooming off the edge of the screen! We need a function to stay on the screen. This will check the new position to see if it's off the edge of the screen. If so, it will reset them back to the edge and cut their movement in that direction to zero.

We check if **P.X** is more than **119** because **P.X** is on the *left* side of the sprite. If **P.X** was more then **119**, it would draw past the right edge of the screen, and that's what we're trying to prevent!

```
FUNCTION MOVE_PLAYER()
 P.DY+=G --ADD GRAVITY

 THRUST()

 P.X+=P.DX --ACTUALLY MOVE
 P.Y+=P.DY --THE PLAYER

 STAY_ON_SCREEN()
END

FUNCTION STAY_ON_SCREEN()
 IF (P.X<0) THEN   --LEFT SIDE
  P.X=0
  P.DX=0
 END
 IF (P.X>119) THEN --RIGHT SIDE
  P.X=119
  P.DX=0
 END
 IF (P.Y<0) THEN   --TOP SIDE
  P.Y=0
  P.DY=0
 END
END
```

SAVE & RUN IT!

LANDER - STEP 5

We have a functioning lander! Let's make the rest of the environment. We need stars, a landing pad, and the ground, and we want them to be random. But let's just start with stars.

PICO-8's random number generator, `RND()`, will give you a random number between 0 and the number you give it. But we need a function that gives us a random number between any number we choose (not just 0) and any other number. We'll make our own "random between" function called `RNDB()` that takes a `LOW` and `HIGH` number and gives us a random number between them.

We're also using a function called srand() that ensures we get the same random numbers each time we draw the stars, so they don't jump around.

```
FUNCTION _DRAW()
 CLS()
 DRAW_STARS()
 DRAW_PLAYER()
END

FUNCTION RNDB(LOW,HIGH)
 RETURN FLR(RND(HIGH-LOW+1)+LOW)
END

FUNCTION DRAW_STARS()
 SRAND(1)
 FOR I=1,50 DO
  PSET(RNDB(0,127),RNDB(0,127),RNDB(5,7))
 END
 SRAND(TIME())
END
```

SAVE & RUN IT!

LANDER - STEP 6

Adding ground is probably the most complex part of the game, believe it or not. We're going to write our code in a few different steps, and then we'll run it.

We're going to add two new functions. The first for making the ground and the second for drawing it while we play. Making the ground happens at the beginning, so running that function happens in `_INIT()`. Drawing the ground would of course happen in `_DRAW()`. Let's just add the lines where we run those functions first.

```
FUNCTION _INIT()
 G=0.025
 MAKE_PLAYER()
 MAKE_GROUND()
END

FUNCTION _DRAW()
 CLS()
 DRAW_STARS()
 DRAW_GROUND()
 DRAW_PLAYER()
END
```

The function to create the ground requires some explanation of what's going on.

We're going to store the ground as a list of ground heights. There are 128 pixels across the screen, so we'll store 128 ground heights. When we draw our ground, we just go through that whole list, beginning to end, and draw a line from that ground height down to the bottom of the screen.

LANDER - STEP 6

First we'll figure out the position of the landing pad and make that into flat ground (all the same ground height). Then we'll add bumpy ground to the left and right of the landing pad.

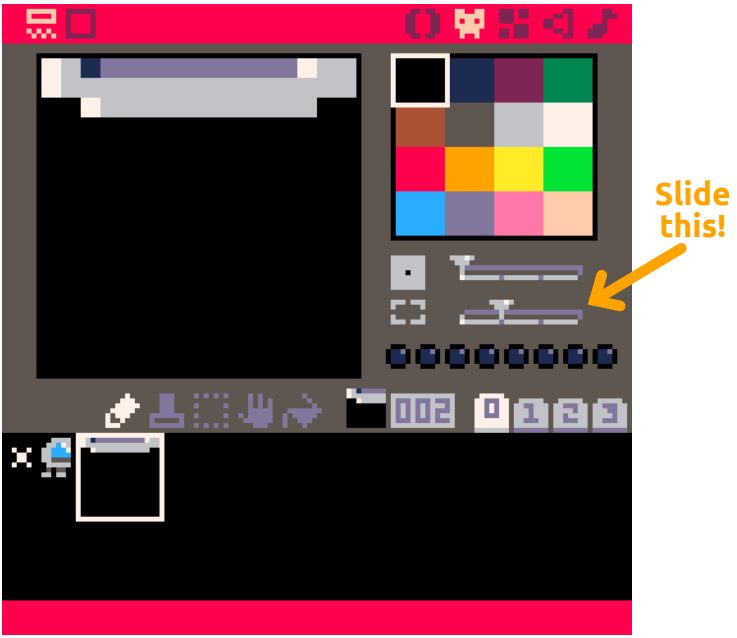
```
FUNCTION MAKE_GROUND()
 --CREATE THE GROUND
 GND={}
 LOCAL TOP=96  --HIGHEST POINT
 LOCAL BTM=120 --LOWEST POINT

 --SET UP THE LANDING PAD
 PAD={}
 PAD.WIDTH=15
 PAD.X=RNDB(0,126-PAD.WIDTH)
 PAD.Y=RNDB(TOP,BTM)
 PAD.SPRITE=2

 --CREATE GROUND AT PAD
 FOR I=PAD.X,PAD.X+PAD.WIDTH DO
  GND[I]=PAD.Y
 END

 --CREATE GROUND RIGHT OF PAD
 FOR I=PAD.X+PAD.WIDTH+1,127 DO
  LOCAL H=RNDB(GND[I-1]-3,GND[I-1]+3)
  GND[I]=MID(TOP,H,BTM)
 END

 --CREATE GROUND LEFT OF PAD
 FOR I=PAD.X-1,0,-1 DO
  LOCAL H=RNDB(GND[I+1]-3,GND[I+1]+3)
  GND[I]=MID(TOP,H,BTM)
 END
END

FUNCTION DRAW_GROUND()
 FOR I=0,127 DO
  LINE(I,GND[I],I,127,5)
 END
END
```

SAVE & RUN IT!

LANDER - STEP 7

Ignoring the fact that you can still fly through the ground, you'll notice our landing area is kind of boring. Let's make it awesome!

The landing pad will take up two sprites side by side. Use the second slider in the middle to make your drawing space 2x2 sprites. Make sure to draw the pad flat across the *top* of the drawing space. Don't forget to make it awesome!




```
FUNCTION DRAW_GROUND()
 FOR I=0,127 DO
  LINE(I,GND[I],I,127,5)
 END
 SPR(PAD.SPRITE,PAD.X,PAD.Y,2,1)
END
```

SAVE & RUN IT!

LANDER - STEP 8

It's important to keep track of whether or not our game is over. For example, if the game is over, we shouldn't be able to move the player.

But just keeping track of whether the game is over isn't enough. When the game is over, we also need to know whether the player won or lost.

We just need to track **TRUE** or **FALSE** for these two pieces of information. And when the game starts, the game is not over and the player hasn't won, so these two things start out as **FALSE**.

```
FUNCTION _INIT()
 GAME_OVER=FALSE
 WIN=FALSE
 G=0.025
 MAKE_PLAYER()
 MAKE_GROUND()
END

FUNCTION _UPDATE()
 IF (NOT GAME_OVER) THEN
  MOVE_PLAYER()
 END
END
```

Now we need to check to see if the player has landed on the ground or not. This is a bit complicated because there a few ways the player can land. They can:

- land fully on the pad, but *not* going too fast
- land fully on the pad, but going too fast
- land partially or fully on the ground, not the pad

Only the first one means the player wins. We need

LANDER - STEP 8

a function that will check each of these, one by one. Whether they win or lose, we're going put the changing of `GAME_OVER` and `WON` in its own function. It's always a good idea to put things you need to do again and again into their own function.

```
FUNCTION _UPDATE()
 IF (NOT GAME_OVER) THEN
  MOVE_PLAYER()
  CHECK_LAND()
 END
END

FUNCTION CHECK_LAND()
 L_X=FLR(P.X)   --LEFT SIDE OF SHIP
 R_X=FLR(P.X+7) --RIGHT SIDE OF SHIP
 B_Y=FLR(P.Y+7) --BOTTOM OF SHIP

 OVER_PAD=L_X>=PAD.X AND R_X<=PAD.X+PAD.WIDTH
 ON_PAD=B_Y>=PAD.Y-1
 SLOW=P.DY<1

 IF (OVER_PAD AND ON_PAD AND SLOW) THEN
  END_GAME(TRUE)
 ELSEIF (OVER_PAD AND ON_PAD) THEN
  END_GAME(FALSE)
 ELSE
  FOR I=L_X,R_X DO
   IF (GND[I]<=B_Y) END_GAME(FALSE)
  END
 END
END

FUNCTION END_GAME(WON)
 GAME_OVER=TRUE
 WIN=WON
END
```

SAVE & RUN IT!

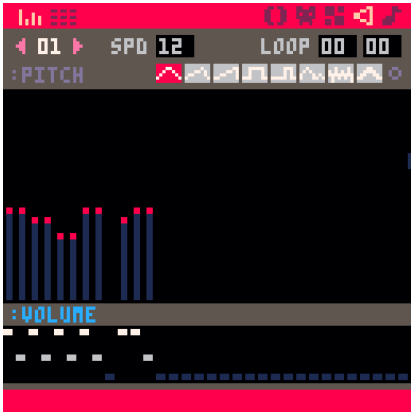
You'll notice when we land, the game just... stops. Let's make the end a bit more dramatic than that.

LANDER - STEP 9

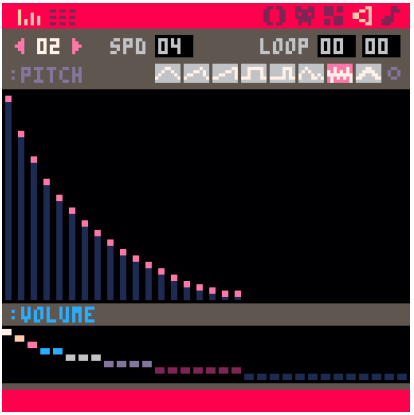
We'll make the end of our game dramatic in two steps. First we'll add sound, then we'll add visuals.

We need two sounds: one for winning and one for losing. We'll use our end_game() function to play these sounds.

In addition to the notes used, pay close attention to the speed setting, the instrument used, and volume levels.



SOUND #1



SOUND #2

```
FUNCTION END_GAME(WON)
 GAME_OVER=TRUE
 WIN=WON

 IF (WIN) THEN
  SFX(1)
 ELSE
  SFX(2)
 END
END
```

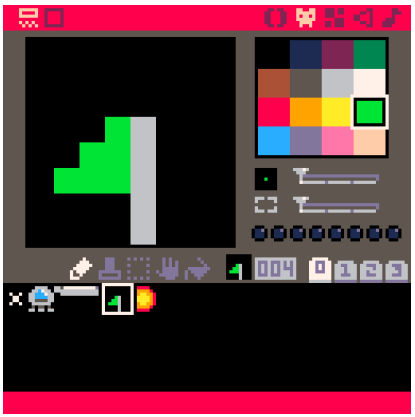
SAVE & RUN IT!

LANDER - STEP 10

Now let's add the visuals. If the player lands successfully, we'll raise a flag of victory. But if the player lands on the treacherous terrain or hits the pad too hard, we'll show a fiery explosion.

We need to add sprites #4 and #5 for this.

We'll use the `DRAW_PLAYER()` function. We'll only draw the sprites if `GAME_OVER` is `TRUE`, but we'll show different sprites whether `WIN` is `TRUE` or not.



SPRITE #4

SPRITE #5

```
FUNCTION DRAW_PLAYER()
 SPR(P.SPRITE,P.X,P.Y)
 IF (GAME_OVER AND WIN) THEN
  SPR(4,P.X,P.Y-8) --FLAG
 ELSEIF (GAME_OVER) THEN
  SPR(5,P.X,P.Y) --EXPLOSION
 END
END
```

SAVE & RUN IT!

LANDER - STEP 11

We're almost done! One last step! We just need to let the player know when the game is over and give them a way to restart the game.

In `_UPDATE()`, we'll check to see if the game is over, and if it is, we'll listen for a button press. If the button is pressed, we'll run `_INIT()` which will restart everything.

In `_DRAW()`, if the game is over, we'll tell the player whether they won or not and how to play again. (Use shift-X for the ❎ symbol.)

```
FUNCTION _UPDATE()
 IF (NOT GAME_OVER) THEN
  MOVE_PLAYER()
  CHECK_LAND()
 ELSE
  IF (BTNP(5)) _INIT()
 END
END

FUNCTION _DRAW()
 CLS()
 DRAW_STARS()
 DRAW_GROUND()
 DRAW_PLAYER()

 IF (GAME_OVER) THEN
  IF (WIN) THEN
   PRINT("YOU WIN!",48,48,11)
  ELSE
   PRINT("TOO BAD!",48,48,8)
  END
  PRINT("PRESS ❎ TO PLAY AGAIN",20,70,5)
 END
END
```

SAVE & RUN IT!

LANDER - ETC

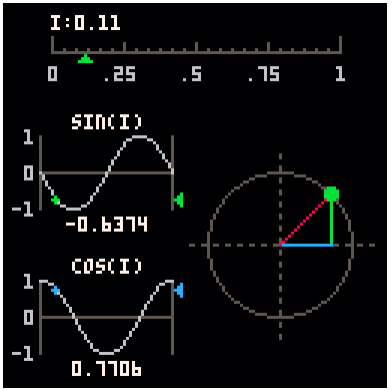
We're done! Even though we have a working game, there's so much more we could add. For instance, we could add wind, or obstacles, or a smaller pad, or fuel that runs out. But, we'll stop here.

As with the first game, we can publish it at this point. The section starting on page 58 has easy instructions on publishing your game.

As noted at the beginning of this tutorial, this game was a bit more complex than the first one. However, the basic approach to creating the game is the same. We just build it up one piece at a time, testing each time we add something new.

PICO-8 FOR GAMEDEVS

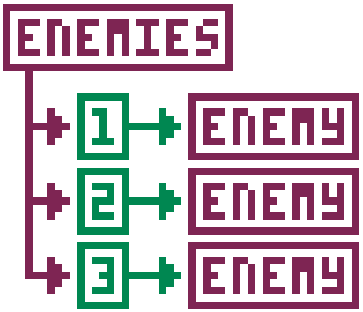
While PICO-8 is certainly a fun environment for new game developers who are first learning to make games, it's just as fun for experienced game developers who already know how to make games.



Because of PICO-8's constraints, you are free to quickly try stuff out and just have fun, but without the heavier time investment required of other game-making environments. However, experienced game developers might not know how to get PICO-8 to do some of the more advanced things they're used to doing in other environments (like particle systems). To help with that, the following few pages cover some more advanced topics that are likely familiar to experienced game developers.

Even if you are not already an experienced game developer, I urge you to work your way through these next pages. You may need to read through it a few times and play around with the examples until you really understand them, but it will be worth it.

MORE ON TABLES



One reason tables are so useful in games is the ability for a table to store *other tables* as values. Most games use this quite a lot. For example, each enemy in a game might be stored as a table, but a master enemies table would store each of those individual enemy tables as values.

Tables of tables don't have to be as exciting as a list of enemies. They can also be as simple as, say, the terrain we created in the Lander tutorial, or the cave walls in the Cave Diver tutorial. If you have to keep track of a collection of things in your game, a table of tables is probably your best bet.

There are many ways to deal with a table of tables. We'll examine a common (and easy) method here.

Let's take the example of keeping track of enemies in a game. As mentioned above, each enemy on its own will be a table, but each enemy table will be stored as a value in a master table called `ENEMIES`.

There are really two parts to dealing with a table of tables. First we need a few generic functions to deal with each individual table value in the master table. Then we'll need to loop through the master table in `_UPDATE()` and `_DRAW()` and run those generic functions on each individual table value.

MORE ON TABLES

In our example, the master **ENEMIES** table will be created in **_INIT()**. Then we will need to create **MAKE_ENEMY()**, **MOVE_ENEMY()**, and **DRAW_ENEMY()**. Lastly, we'll loop through the **ENEMIES** table in both **_UPDATE()** and **_DRAW()**. Let's get started.

```
FUNCTION _INIT()
 ENEMIES={}
END
```

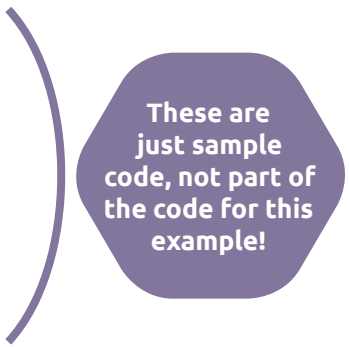
The function **MAKE_ENEMY()** needs to accept an **X/Y** coordinate so we can specify *where* to create the enemy. The enemy table **E** is created as a local (temporary) variable because we don't need each enemy as its own variable *after* we add it to the **ENEMIES** table. For **MOVE_ENEMY()**, we'll move them randomly. If they move off-screen, we'll have them "die" by removing them from the **ENEMIES** table.

```
FUNCTION MAKE_ENEMY(X,Y)
 LOCAL E={}
 E.X=X
 E.Y=Y
 E.SPRITE=1
 ADD(ENEMIES,E)
END

FUNCTION UPDATE_ENEMY(E)
 E.X+=RND(2)-1
 E.Y+=RND(2)-1
 IF (E.X<0 OR E.X>119
  OR E.Y<0 OR E.Y>119) THEN
  DEL(ENEMIES,E)
 END
END

FUNCTION DRAW_ENEMY(E)
 SPR(E.SPRITE,E.X,E.Y)
END
```

MORE ON TABLES

With those functions written, we need to add code to loop through the **ENEMIES** table. There are a few methods. As an example, here are three ways of looping through **ENEMIES** to draw each enemy:

```
--METHOD 1
FOREACH(ENEMIES,DRAW_ENEMY)

--METHOD 2
FOR I=1,#ENEMIES DO
 DRAW_ENEMY(ENEMIES[I])
END

--METHOD 3
FOR E IN ALL(ENEMIES) DO
 DRAW_ENEMY(E)
END
```



Let's use method 1 in both **_UPDATE()** and **_DRAW()**.

```
FUNCTION _UPDATE()
 FOREACH(ENEMIES,UPDATE_ENEMY)
END

FUNCTION _DRAW()
 CLS()
 FOREACH(ENEMIES,DRAW_ENEMY)
END
```

Lastly, if we want to create, say, 20 enemies in random locations at the start of the game, we'll just add that to the **_INIT()** function.

```
FUNCTION _INIT()
 ENEMIES={}
 FOR I=1,20 DO
  MAKE_ENEMY(RND(128),RND(128))
 END
END
```

And that's it! This can be used for keeping track of all sorts of things in your game. Try it out!

PARTICLE SYSTEMS

Particle systems are extremely useful in games. They can be used for so many things: sparks, rain, smoke, trails, debris, fireworks, you name it.

Particle systems are not hard to create. At their core, they're just another table of tables. Also, each particle usually has a lifetime and dies on its own when its lifetime is over. The key to a good particle system is having enough variables to give you good control over each particle.

Let's create a particle fountain with enough variables to give us good control over it. We'll use many concepts from the section above on tables. Feel free to examine each part to determine what it does and tweak variables to see what they do. You can also press ❎ to create a burst of particles.

```
FUNCTION _INIT()
 PS={}          --EMPTY PARTICLE TABLE
 G=0.1          --PARTICLE GRAVITY
 MAX_VEL=2      --MAX INITIAL PARTICLE VELOCITY
 MIN_TIME=2     --MIN/MAX TIME BETWEEN PARTICLES
 MAX_TIME=5
 MIN_LIFE=90 --PARTICLE LIFETIME
 MAX_LIFE=120
 T=0            --TICKER
 COLS={1,1,1,13,13,12,12,7} --COLORS
 BURST=50

 NEXT_P=RNDB(MIN_TIME,MAX_TIME)
END

FUNCTION RNDB(LOW,HIGH)
 RETURN FLR(RND(HIGH-LOW+1)+LOW)
END
```

PARTICLE SYSTEMS

```
FUNCTION _UPDATE()
 T+=1
 IF (T==NEXT_P) THEN
  ADD_P(64,64)
  NEXT_P=RNDB(MIN_TIME,MAX_TIME)
  T=0
 END
 --BURST
 IF (BTNP(🅾️)) THEN
  FOR I=1,BURST DO ADD_P(64,64) END
 END
 FOREACH(PS,UPDATE_P)
END

FUNCTION _DRAW()
 CLS()
 FOREACH(PS,DRAW_P)
END

FUNCTION ADD_P(X,Y)
 LOCAL P={}
 P.X,P.Y=X,Y
 P.DX=RND(MAX_VEL)-MAX_VEL/2
 P.DY=RND(MAX_VEL)*-1
 P.LIFE_START=RNDB(MIN_LIFE,MAX_LIFE)
 P.LIFE=P.LIFE_START
 ADD(PS,P)
END

FUNCTION UPDATE_P(P)
 IF (P.LIFE<=0) THEN
  DEL(PS,P) --KILL OLD PARTICLES
 ELSE
  P.DY+=G   --ADD GRAVITY
  IF ((P.Y+P.DY)>127) P.DY*=-0.8
  P.X+=P.DX --UPDATE POSITION
  P.Y+=P.DY
  P.LIFE-=1 --DIE A LITTLE
 END
END

FUNCTION DRAW_P(P)
 LOCAL PCOL=FLR(P.LIFE/P.LIFE_START*#COLS+1)
 PSET(P.X,P.Y,COLS[PCOL])
END
```

GAME STATES

Games often have multiple modes, or *states*. For example, a single game might have a menu state, a gameplay state (where you actually play the game), and a game over state. These states often use very different code from each other. With PICO-8's game loop, it's easy to separate the code for your game's various states.

The root of how you do this lies in the ability for function names to act like variables. Basically, the *value* being stored in a function's name is the function's *code*. Just like the *value* stored in a variable `x` might be a *number*, the *value* stored in, say, `pset` is the *code* to draw pixels on the screen.

Since PICO-8 will always run the game loop functions `_update()` and `_draw()`, you can just assign the code from *other* functions to the game loop function names `_update` and `_draw`.

Lastly, it helps to put each game state in its own code tab. This makes it easier to mentally separate the functions for each state. For example, you could put menu code in tab `0`, gameplay code in tab `1`, game over code in tab `2`, and various utility functions in tab `3`.

Let's look at an example of how this would work. Notice that each state is on a different tab. Also notice the way we move from one state to another if the player hits ❎. (Use shift-O for the ❎ symbol.)

GAME STATES - EXAMPLE

```
FUNCTION _INIT() MENU_INIT() END

FUNCTION MENU_INIT()
 _UPDATE=MENU_UPDATE
 _DRAW=MENU_DRAW
END

FUNCTION MENU_UPDATE()
 IF (BTNP(🅾️)) GAME_INIT() --PLAY THE GAME
END

FUNCTION MENU_DRAW()
 PRINT("MENU!") --MENU DRAW CODE
END
```

```
FUNCTION GAME_INIT()
 _UPDATE=GAME_UPDATE
 _DRAW=GAME_DRAW
END

FUNCTION GAME_UPDATE()
 IF (BTNP(🅾️)) GAMEOVER_INIT() --GAME OVER
END

FUNCTION GAME_DRAW()
 PRINT ("GAME!") --GAME DRAW CODE
END
```

```
FUNCTION GAMEOVER_INIT()
 _UPDATE=GAMEOVER_UPDATE
 _DRAW=GAMEOVER_DRAW
END

FUNCTION GAMEOVER_UPDATE()
 IF (BTNP(🅾️)) MENU_INIT() --BACK TO MENU
END

FUNCTION GAMEOVER_DRAW()
 PRINT("GAME OVER!") --GAME OVER CODE
END
```

COROUTINES

Most functions in your games need to happen all at once inside the time it takes to draw one frame (like moving the player). But sometimes you need a single function to take longer than a single frame. Or you might want other things to be able to happen *while* the function is running its course. This is where coroutines help.

Coroutines are special functions that can give back, or *yield*, control to what's calling them even if the coroutine isn't complete. The coroutine can then be resumed at a later point.

This is very useful for, say, scripted animation or showing dialog one letter at a time, all the while still listening for key presses from the player. In both examples, you would want the function to play out over time, not happen all at once.

PICO-8 has four functions to work with coroutines:

`cocreate(function_name)` - Creates and returns a coroutine, but does not start the coroutine.

`coresume(coroutine)` - Passes control to the coroutine. (If it hasn't started yet, this will start it.)

`costatus(coroutine)` - Returns the status of the coroutine as `"running"`, `"suspended"`, or `"dead"`.

`yield()` - Gives control back to whatever called the coroutine.

COROUTINES - EXAMPLE

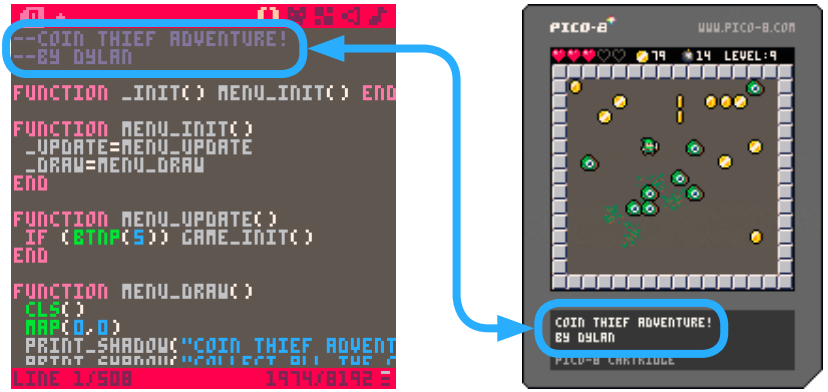
This will make a circle move in a pattern around the screen. Any button will reset the animation.

```
FUNCTION _INIT()
 C_MOVE=COCREATE(MOVE)
END

FUNCTION _UPDATE()
 IF C_MOVE AND COSTATUS(C_MOVE)!="DEAD" THEN
  CORESUME(C_MOVE)
 ELSE
  C_MOVE=NIL
 END
 IF (BTNP()>0) C_MOVE=COCREATE(MOVE)
END

FUNCTION _DRAW()
 CLS(1)
 CIRC(X,Y,R,12)
 PRINT(CURRENT,4,4,7)
END

FUNCTION MOVE()
 X,Y,R=32,32,8

 CURRENT="LEFT TO RIGHT"
 FOR I=32,96 DO
  X=I
  YIELD()
 END

 CURRENT="TOP TO BOTTOM"
 FOR I=32,96 DO --TOP TO BOTTOM
  Y=I
  YIELD()
 END

 CURRENT="BACK TO START"
 FOR I=96,32,-1 DO --BACK TO START
  X,Y=I,I
  YIELD()
 END
END
```

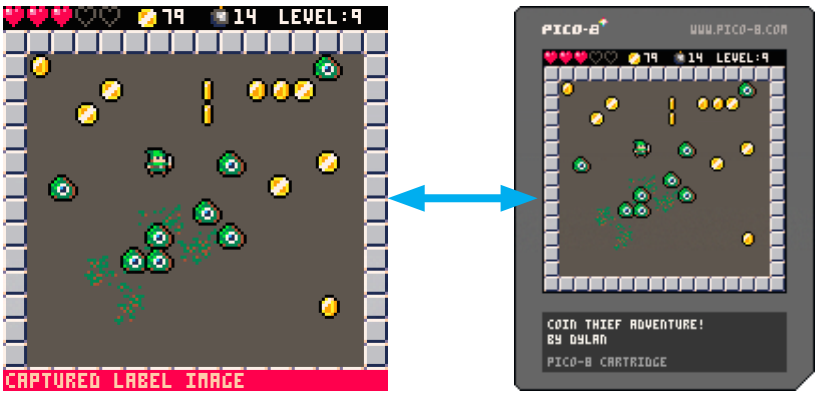
Try it out! Type this code into PICO-8 and run it!

PUBLISHING YOUR GAMES

Publishing your game is easy, but there are a few steps you will need to do to get your game ready. The first step is a title for your game. Just add two comments to the top of your code. These will be added to the cart image as the game's title.



The next step is creating the cart's label image. Play your game, and when the screen looks how you want it to look, hit F7. Then make sure to save!



For the last step, hit ESC to go to Command Mode and type `SAVE YOURGAME.PNG` to save as a shareable image. Now it's ready to be shared!

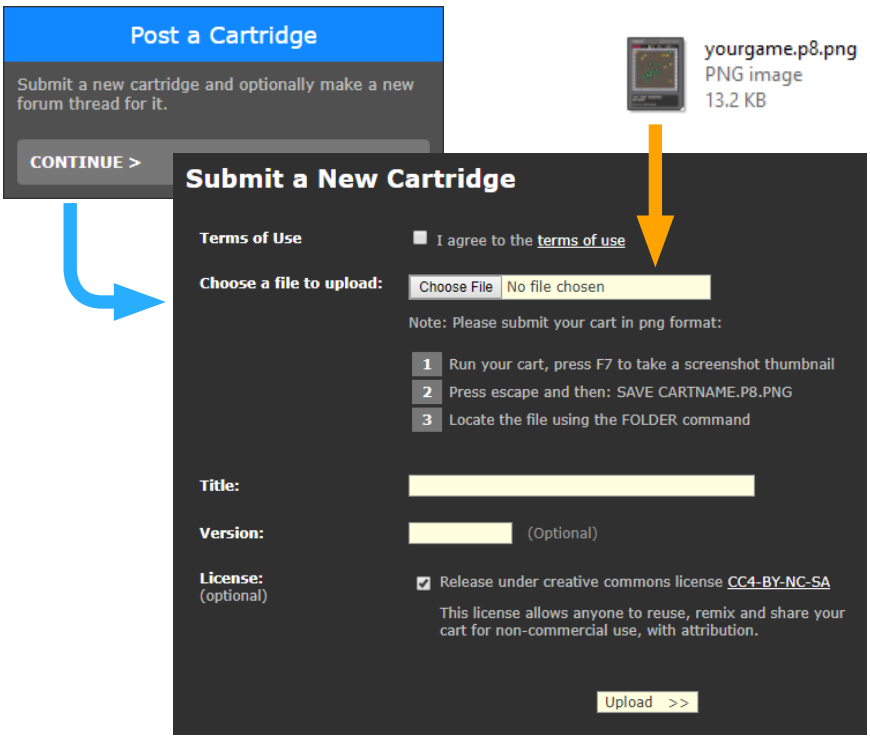
PUBLISHING TO THE BBS

The Lexaloffle forum is a great place to publish your PICO-8 games. It's referred to as "the BBS" by the PICO-8 community, after the bulletin board systems of the '80s and '90s. There you'll find a wonderful, welcoming community of creators.

To submit your cart to the BBS, go to this address:

http://lexaloffle.com/pico-8.php?page=submit

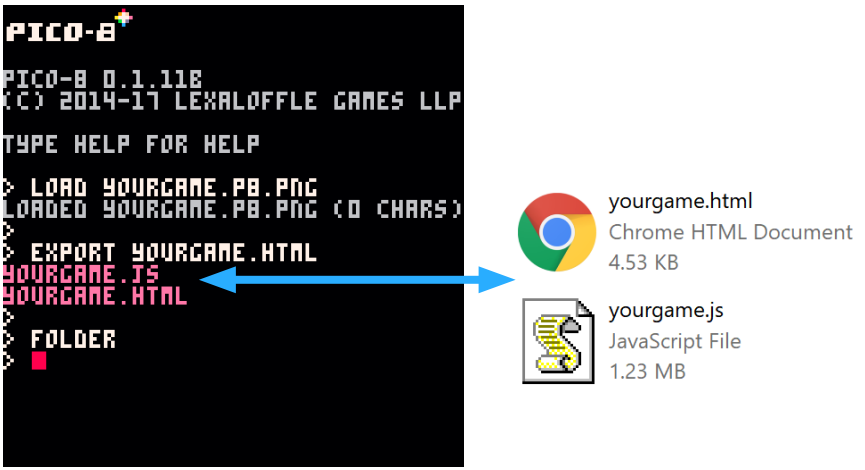
After choosing Post a Cartridge, you'll come to a page where you can submit your cartridge. When asked for which file to upload, choose the cart image you made (the `YOURGAME.P8.PNG` file).



EXPORTING FOR THE WEB

Once you have your cart image ready, you can publish your game as an HTML5 game. This lets you load and play the game in a web page, on its own, without the need to have PICO-8 installed.

Once your game is loaded in PICO-8, hit ESC to go to Command Mode, if you're not already there. Type the command **EXPORT YOURGAME.HTML** to export your game. After it creates **YOURGAME.JS** and **YOURGAME.HTML**, follow the instructions and type **FOLDER** to see the files that were created.



Go ahead and open the HTML file in your browser to test it out. Everything should work just fine.

The HTML file is just a template to nicely embed the Javascript file, so feel free to edit the HTML file to make it look how you want. However, if you upload the game to the web, make sure you don't forget to upload the Javascript file as well.

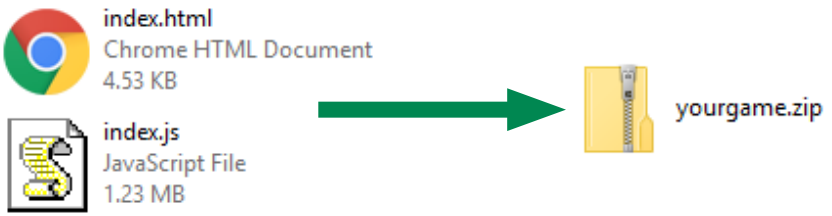
PUBLISHING ON ITCH.IO

The web site itch.io is a publishing platform for independent game developers. It's probably one of the most developer-friendly publishing platforms around. Literally within minutes of exporting your PICO-8 game, you can have your game published on itch.io for all to see and play.

The first step is to follow the instructions on page 58 for prepping your game. The next steps are *almost* the same as exporting to the web, but the differences are important.

Follow the instructions for exporting to the web, but when you type the `EXPORT` command, type `EXPORT INDEX.HTML` instead of your game's name. It's very important you use `INDEX.HTML` in this step!

The next step is to zip the two files, `INDEX.HTML` and `INDEX.JS`, into one zip file. (You can name the zip file whatever you want.) On Windows, select the two files, right click on them, and then choose "Send to compressed (zipped) folder". On macOS, select the two files, right click on them, and then choose "Compress 2 Items".

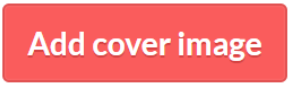


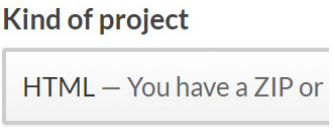
PUBLISHING ON ITCH.IO

All the remaining steps take place on the itch.io web site. Obviously the first step is to create an account!

Once your account is created, click the little drop-down arrow next to your profile icon in the top-right and choose "Upload new project" from the list. You'll be taken to the new project creation page. Most of the steps are self-explanatory, but we'll walk through the parts that need particular attention.

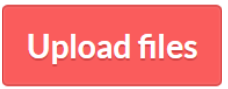
While you are playing your game, you can take a screenshot with the F6 key. These make good cover images!

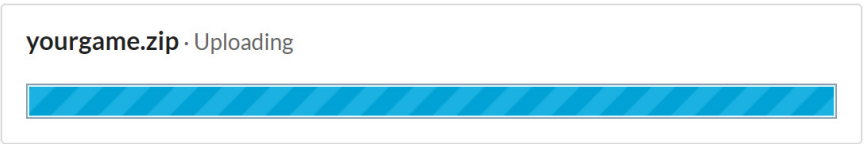




Make sure to set the "Kind of project" to HTML. This means the game will be playable in the browser, without having to download anything.

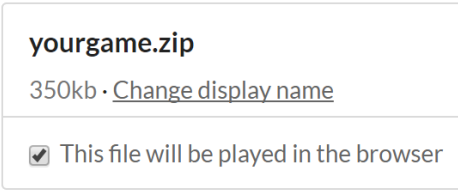
The file you need to upload is the zip file you made on the previous page. Because PICO-8 games are so small, it should only take a short amount of time to upload your zip file. Nevertheless, make sure the file finishes uploading fully! This is important.

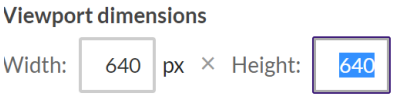




PUBLISHING ON ITCH.IO

Once your file is done uploading, that section will change to give you new options. In that section, check the box that says "This file will be played in the browser."

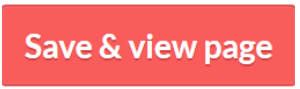




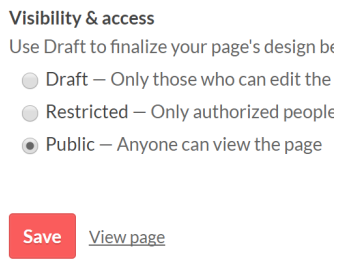
Since PICO-8 games use a square screen, set *both* viewport dimension settings to 640 pixels.

As mentioned before, the rest of the settings are fairly self-explanatory, so take some time to go through them and change them as you see fit. For instance, you may want to add the "pico-8" tag in the Tags section to make it easier for other PICO-8 users to find your game.

When you're done, click the "Save & view page" button at the bottom of the page. This will take you to a preview of what others will see when they view your game.



If all is well, click the  button at the top of the preview page. This will take you back to the bottom of the project page. Change the option from Draft to Public and click Save.



Done! You're now a published game developer! Huzzah!

CODE REFERENCE

On the following pages you'll find a short summary of the more common functions built into PICO-8. They are loosely organized by category. For a list and descriptions of *all* functions, check the manual file called `PICO-8.TXT` in PICO-8's install folder.

Green brackets `[LIKE THIS]` mean the information is optional and the function will still run without it.

Graphics

`CLS([C])` - Clear the screen to black, or to color `C`

`PSET(X,Y,[C])` - Set the pixel at `X,Y` to color `C`

`PGET(X,Y)` - Returns the color at `X,Y`

`LINE(X1,Y1,X2,Y2,[C])` - Draw a line from `X1,Y1` to `X2,Y2` with color `C`

`CIRC(X,Y,R,[C])` - Draw a circle at `X,Y` of radius `R` with color `C`

`CIRCFILL(X,Y,R,[C])` - Draw a filled circle at `X,Y` of radius `R` with color `C`

`RECT(X1,Y1,X2,Y2,[C])` - Draw a rectangle from `X1,Y1` to `X2,Y2` with color `C`

`RECTFILL(X1,Y1,X2,Y2,[C])` - Draw a rectangle from `X1,Y1` to `X2,Y2` with color `C`

`SPR(S,X,Y,[W,H],[FLIP_X,FLIP_Y])` - Draw sprite `S` at `X,Y`, optionally `W,H` sprites wide and tall, and optionally flipped horizontally or vertically if `FLIP_X` or `FLIP_Y` are `TRUE`

`COLOR([C])` - Set the default color to `C` for functions that use a color

`CURSOR(X,Y)` - Set the `PRINT()` function's cursor position to `X,Y`

`PRINT(T,[X,Y],[C])` - Print value `T` at `X,Y` using color `C`

Tables

`ADD(T,V)` - Add value `V` to table `T` and return `V`

`DEL(T,V)` - Delete value `V` from table `T`

`ALL(T)` - Used in `FOR` loops to go through every item in table `T`, as long as `T` is using number-based keys. Example:

```
FOR I IN ALL(T) DO
 PRINT(I)
END
```

`FOREACH(T,F)` - Go through every value in table `T` and run function `F` with each value as a single parameter for the function `F`

`PAIRS(T)` - Used in `FOR` loops to go through every item in table `T` and provide the key and value of each item. Example:

```
FOR K,V IN PAIRS(T) DO
 PRINT("KEY:"..K)
 PRINT("VALUE:"..V)
END
```

Input

`BTN(B,[P])` - Return the state of button `B` (`0-5`), for player `P` (`0-7`), as `TRUE` or `FALSE`

`BTNP(B,[P])` - Return `TRUE` or `FALSE` depending on whether button `B` (`0-5`), for player `P` (`0-7`), was newly pressed or not

CODE REFERENCE

Audio

`SFX(n,[c],[o],[l])`
Play sound number `n` on channel `c` starting at note `o` and continuing for `l` notes

`MUSIC(n,[fade],[chan])`
Play music starting at track `n`, fading in over `fade` milliseconds, reserving the channels defined by the `chan` bitmask for music

Map

`MGET(x,y)` - Return the number of the sprite at map location `x,y`

`MSET(x,y,[s])` - Set the sprite at map location `x,y` to use sprite number `s`

`MAP(mx,my,sx,sy,w,h,[l])` - Draw map tiles, starting with map tile `mx,my`, to screen coordinate `sx,sy`, and draw `w` tiles wide and `h` tiles tall, and if `l` is specified, draw only the cells that have sprites with matching bits on

Math

`MAX(x,y)` - Return the max of the values `x` and `y`

`MIN(x,y)` - Return the min of the values `x` and `y`

`MID(x,y,z)` - Return the middle value of `x`, `y`, and `z`, no matter the order. For example, `MID(6,2,9)` returns `6`

`FLR(x)` - Return the closest integer below `x`. So `FLR(4.6)` returns `4` and `FLR(-4.6)` returns `-5`.

`COS(x)` - Return the cosine of `x`, where the start of

a circle is `0.0` and a full circle is `1.0`

`SIN(X)` - Return the inverse sine of `X` (because positive `Y` is down in PICO-8's screen coordinate system), and like cosine, the start of a circle is `0.0` and a full circle is `1.0`

`ATAN2(DX,DY)` - Convert `DX,DY` into an angle from `0.0` to `1.0` that represents the direction pointing from `0,0` to `DX,DY`

`SQRT(X)` - Return the square root of `X`

`ABS(X)` - Return the absolute value of `X`

`RND([X])` - Return a random number between `0.0` and `X`, or between `0.0` and `1.0` if `X` is not given

`SRAND(X)` - Initialize the random number generator using `X` to get predictable random numbers

`TIME()` - Return the seconds since PICO-8 started

`TONUM(S)` - Return the string `S` as a number

Strings

`#S` - Return the number of characters in string `S`

`S1..S2` - Join string `S1` to string `S2`

`SUB(S,B,[E])` - Get a sub-section of string `S`, starting at character `B`, until the end of the string, or for `E` number of characters

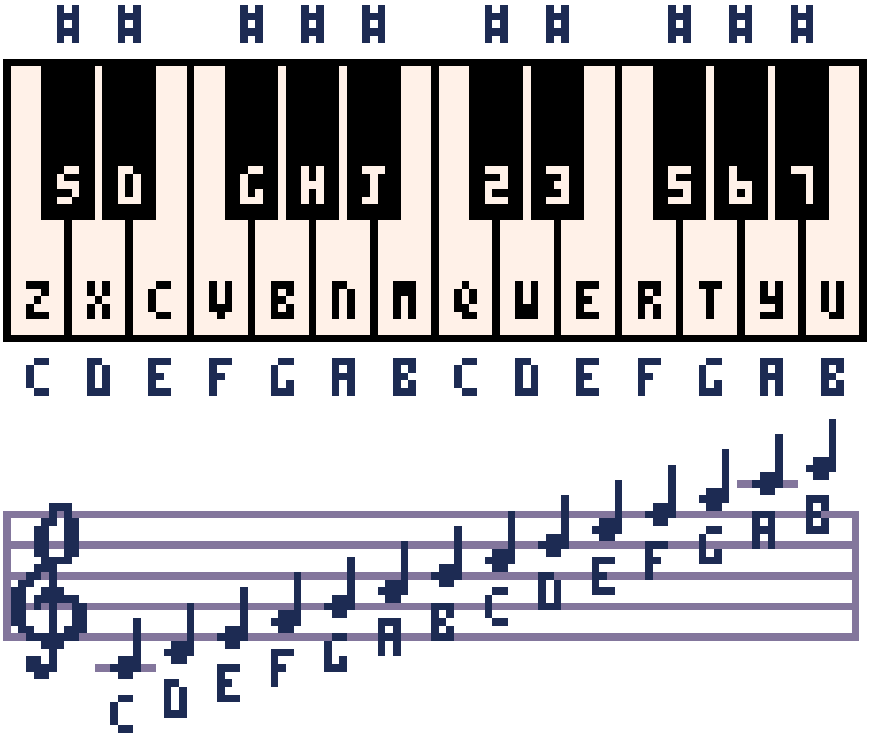
`TOSTR(N)` - Return the number `N` as a string

Colors

MUSIC REFERENCE

In the top-left of PICO-8's Sound Editor, you can switch to tracker mode. In this mode, you can type in specific notes using your computer keyboard like a piano.

On the piano below, you can see which keyboard keys (in black and white) correspond to which musical notes (in blue).



MORE PICO-8 RESOURCES

There are many resources on the Web for you to learn more about PICO-8. I'm going to list just a few of them here.

Offical Lexaloffle PICO-8 Site
www.pico-8.com

The PICO-8 BBS
www.lexaloffle.com/bbs/?cat=7

Unofficial PICO-8 Wiki
pico-8.wikia.com/wiki/Pico-8_Wikia

PICO-8 Fanzines by Arnaud De Bock
sectordub.itch.io/pico-8-fanzine-1
sectordub.itch.io/pico-8-fanzine-2
sectordub.itch.io/pico-8-fanzine-3
sectordub.itch.io/pico-8-fanzine-4

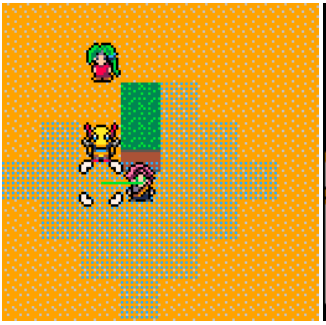
PICO-8 Cheatsheet by Carlos Aguilar
neko250.github.io/pico8-api

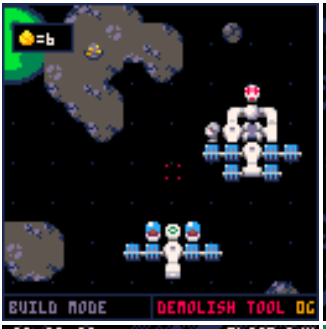
Awesome PICO-8 - Curated List by Felipe Bueno
github.com/felipebueno/awesome-PICO-8

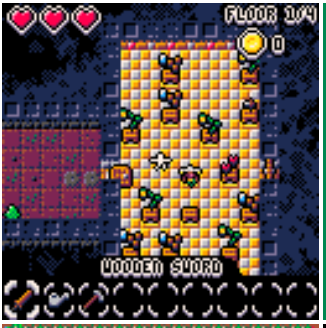
PICO-8 Resources by Marco Secchi
pico-8-resources.zeef.com/marco.secchi

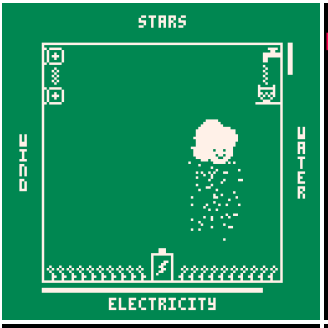
PICO-8 FONT REFERENCE

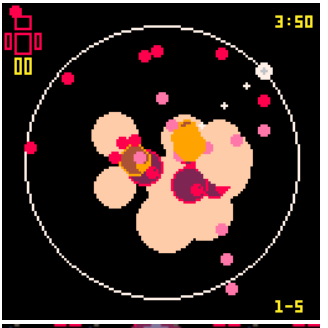
Until you're used to the font in PICO-8, sometimes it can be hard to tell which character is which. This handy chart should help you out.

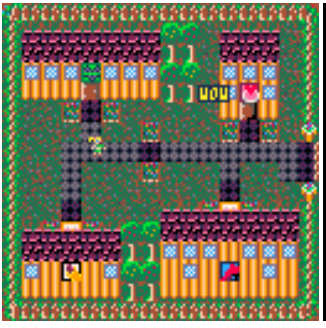






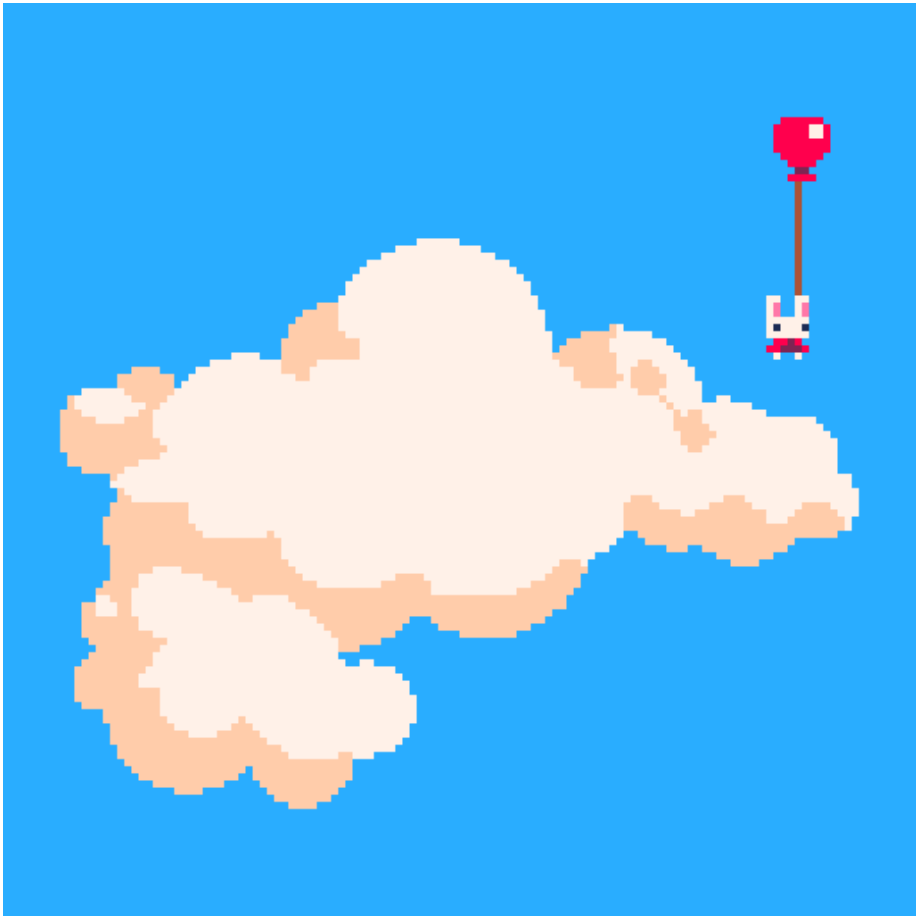






PICO-8 COMMUNITY CREATIONS
(@:TWITTER ✚:BBS)

@ENARGY
@PIXEL_COD
@TRASEVOL_DOG
JCWILK ✚
@CASTPIXEL
@MUSURCA
@PLATFORMALIST
ELECTRICGRYPHON ✚
@JOHANPEITZ
@GUERRAGAMES
@MORNINGTOAST
ULTRABRITE ✚



@LEXALOFFLE

PIGSQUAD

PORTLAND INDIE GAME SQUAD

The Portland Indie Game Squad is a non-profit dedicated to supporting the health and continued expansion of game developers in Portland, the Pacific Northwest, and online. They provide events, resources, and networking activities for art and technology creatives.

A huge thanks goes to PIGSquad for the creation of this zine. It would not have been possible to create without their help and support. If you'd like to find out more, visit **pigsquad.com** and see what they are all about. Consider donating at **patreon.com/pigsquad** to support this wonderful group.