

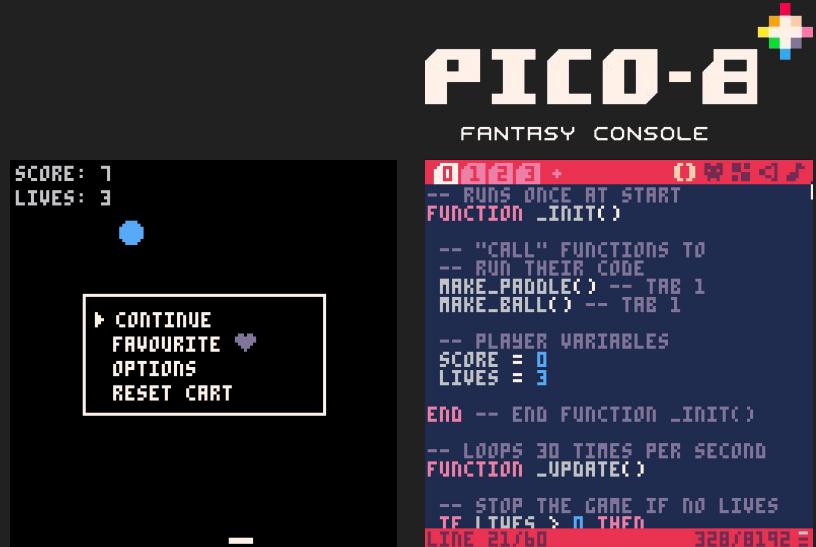
Making a Paddleball Game in PICO-8

Complete Walkthrough

by Matthew DiMatteo

Assistant Professor, Game & Interactive Media Design at Rider University
mdimatteo@rider.edu

as seen in GAM-120: Intro to Game Logic



RIDER
UNIVERSITY

Contents

- 6) About PICO-8
- 13) Getting Started in the PICO-8 Editor
 - 19) Drawing Sprites
 - 31) Saving Your Work 35) Loading a Saved File
 - 39) Microsoft VS Code as an External Code Editor
- 55) Coding in PICO-8
- 73) The PICO-8 Coordinate Plane

Contents

- 81) The PICO-8 “Game Loop” (Program Structure)
- 86) Using Variables
- 90) Understanding the Game Loop
- 102) Variables, Cont.’d
- 107) Variables and Input
- 110) Rules for Naming Variables
- 113) Detecting Input
- 119) Closing Functions and If Statements

Contents

- 134) Keeping the Paddle on Screen
- 148) Writing and “Calling” Custom Functions
- 161) Collision Detection
- 193) Making the Ball Bounce
- 199) Making the Ball Bounce Sideways
- 212) Bouncing off the Walls & Ceiling
- 217) Resetting the Ball After a Miss

Contents

- 219) Adding Sounds
- 224) Adding Score & Lives
- 232) Implementing a Game Over Screen
- 237) Restarting the Game
- 243) Paddleball with Physics
- 244) Code Examples & PICO-8 Learning Resources

About PICO-8

About PICO-8

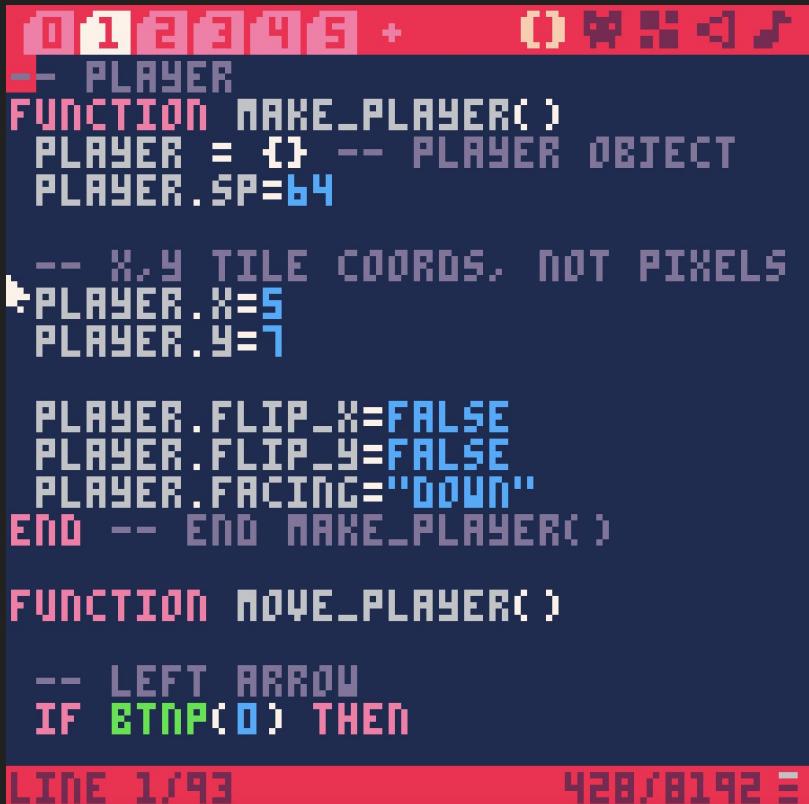
- We'll be using the free-education version, which runs in a browser

Go to pico-8-edu.com

- It has a code editor, sprite editor, map editor, sound editor, and music editor

About PICO-8

PICO-8 has a code editor for writing your code



```
0 1 2 3 4 5 + 0 9 8 7 6 5 4 3 2 1 0
-- PLAYER
FUNCTION MAKE_PLAYER()
PLAYER = {} -- PLAYER OBJECT
PLAYER.SP=64

-- X,Y TILE COORDS, NOT PIXELS
PLAYER.X=5
PLAYER.Y=7

PLAYER.FLIP_X=false
PLAYER.FLIP_Y=false
PLAYER.FACING="DOWN"
END -- END MAKE_PLAYER()

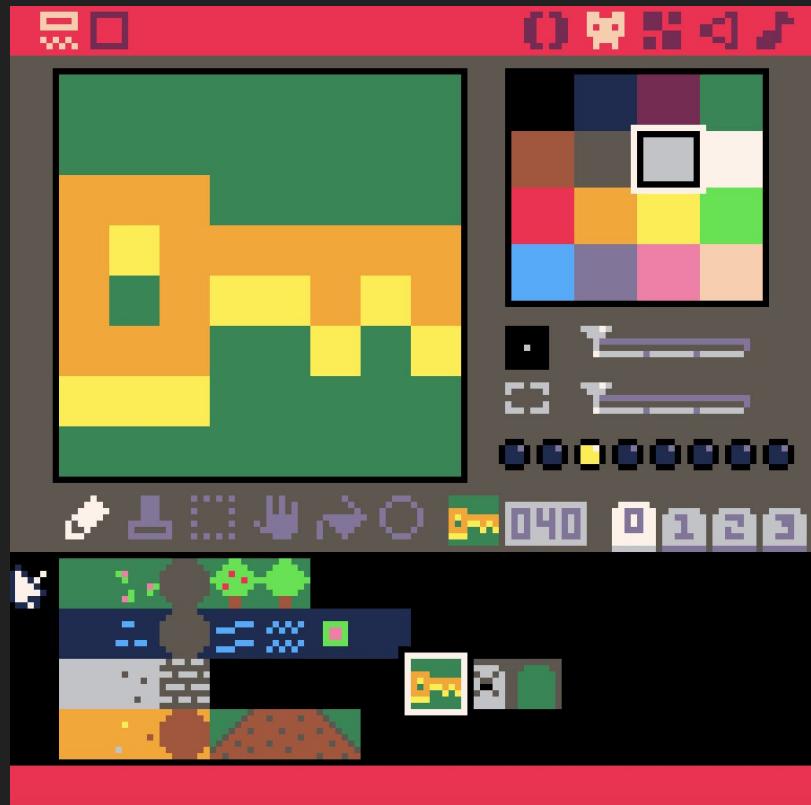
FUNCTION MOVE_PLAYER()

-- LEFT ARROW
IF BTNP(0) THEN
LINE 1/93          428/8192 E
```

About PICO-8

PICO-8 has a sprite editor for drawing graphics

You do have to draw all graphics through the editor, no uploading image files



About PICO-8

PICO-8 has a map editor
for creating your game
world

The size is 1024 x 512 px
(8 screens x 4 screens)

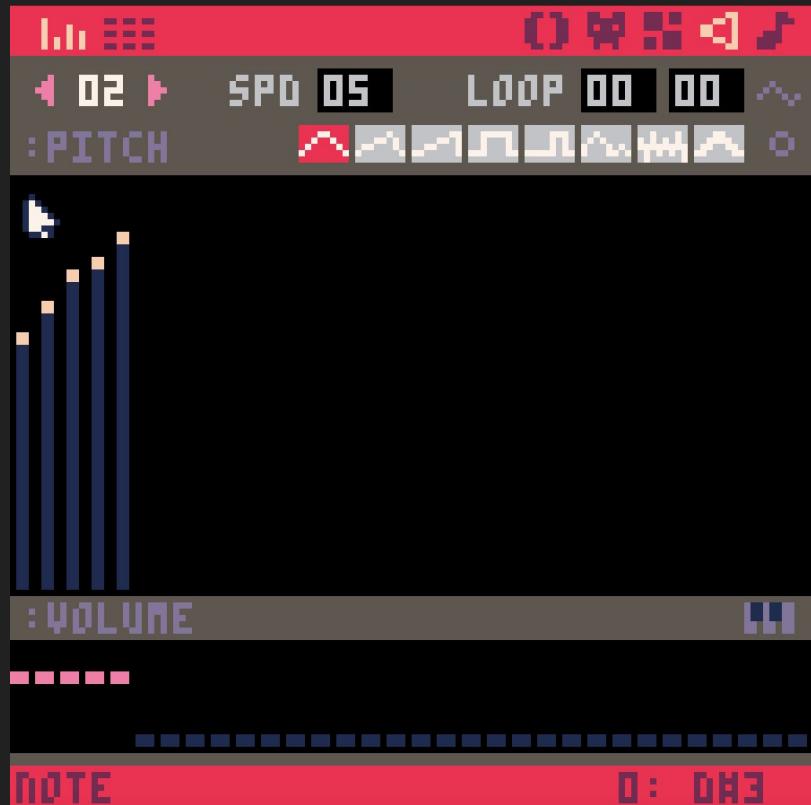
Each screen is 16 x 16 tiles or
128 x 128 px
(each tile is 8 x 8 px)



About PICO-8

PICO-8 has an audio editor for creating sound effects

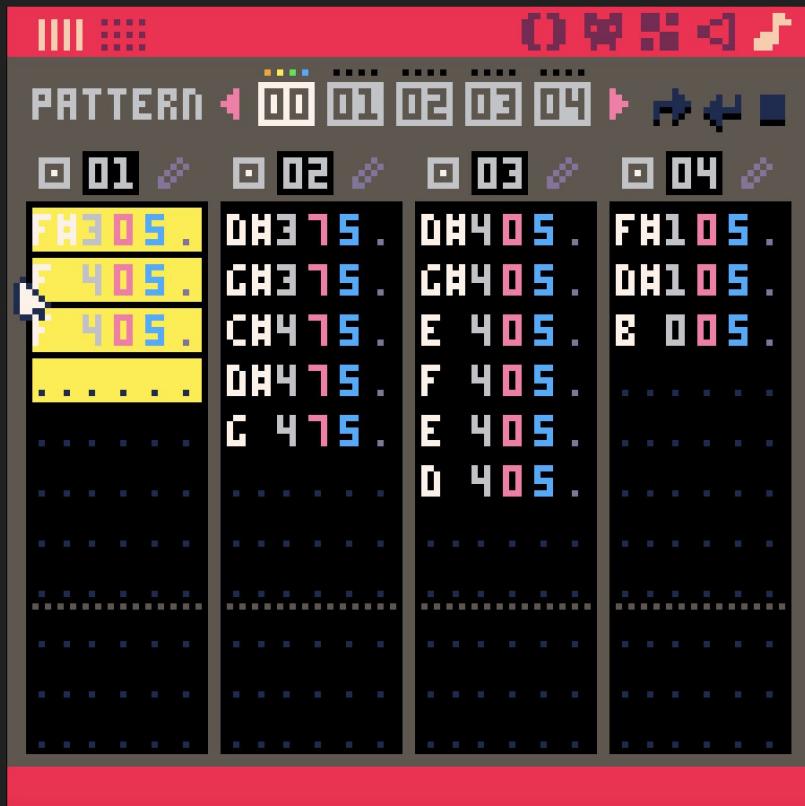
Same deal as with graphics, all must be created in the program, no file uploads



About PICO-8

PICO-8 has a music editor for creating chiptunes

Same deal as with graphics, all must be created in the program, no file uploads



Getting Started with PICO-8

Using the PICO-8 Editor



// Made by **lexaloffle & co**



Start creating using the free Web editor

- From this screen, click the play button

Using the PICO-8 Editor

- You'll see a command line interface for entering commands
- Hit Esc to swap between this screen and the editor



Using the PICO-8 Editor

- If you type **HELP** and press Enter/Return, you'll see a list of commands
- Hit **Esc** to swap between this screen and the editor

```
> HELP
PICO-8 EDUCATION EDITION

COMMANDS:

INSTALL_DEMOS    LS
LOAD <FILENAME>  SAVE <FILENAME>
RUN (OR CTRL-R)   REBOOT
CD <DIRNAME>      MKDIR <DIRNAME>
CD ..             TO GO UP A DIRECTORY

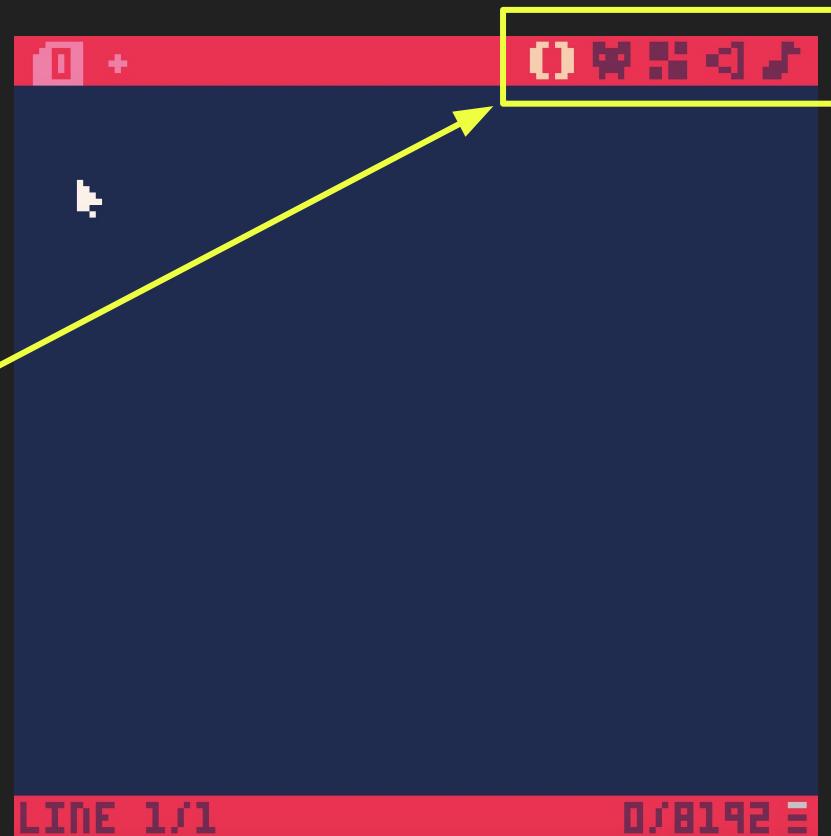
HELP <TOPIC>
GFX DATA AUDIO SYSTEM MATH LUA

CTRL-U IN CODE EDITOR FOR HELP
ON THE KEYWORD UNDER THE CURSOR

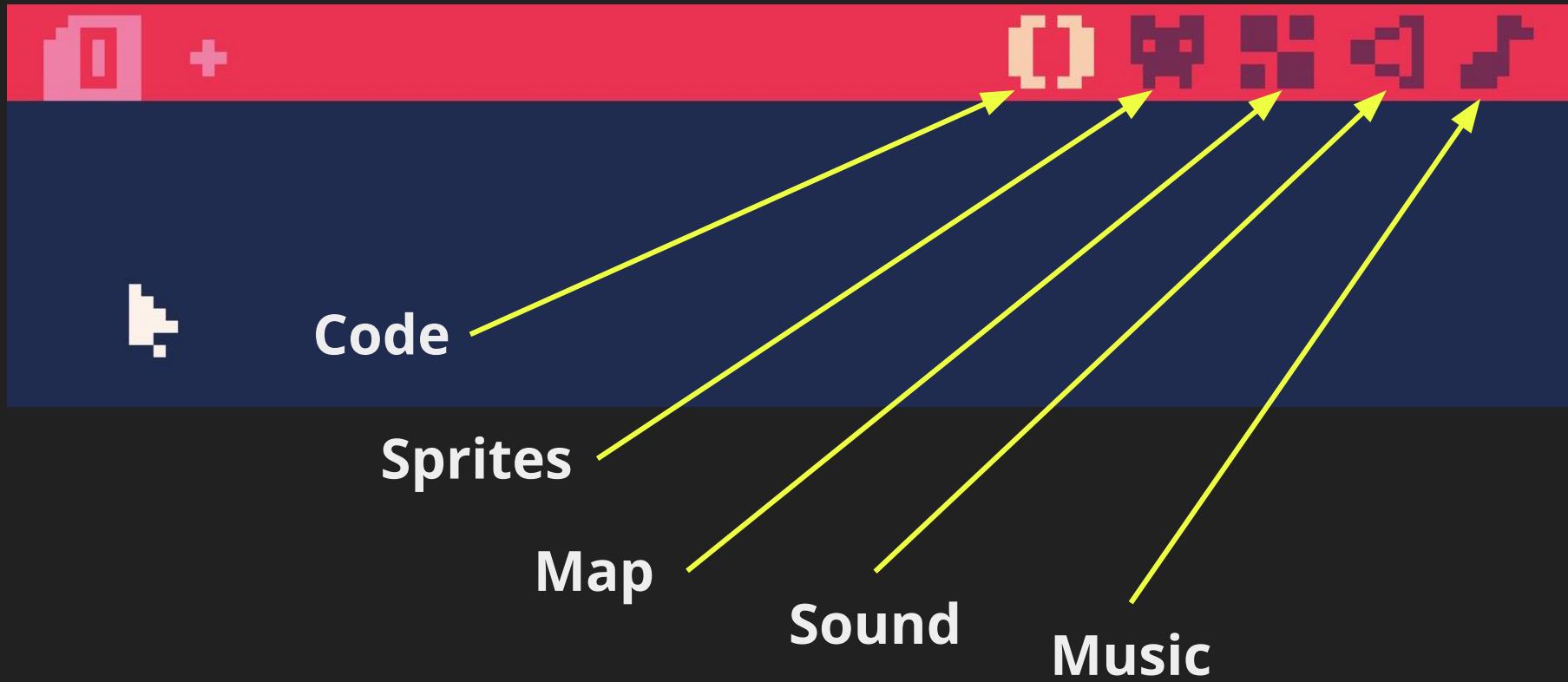
PRESS ESC TO TOGGLE EDITOR VIEW
> ■
```

Using the PICO-8 Editor

- Inside the editor, you'll begin on the code editing screen
- Use the **top-right navigation** to switch to the sprite, map, and sound editors



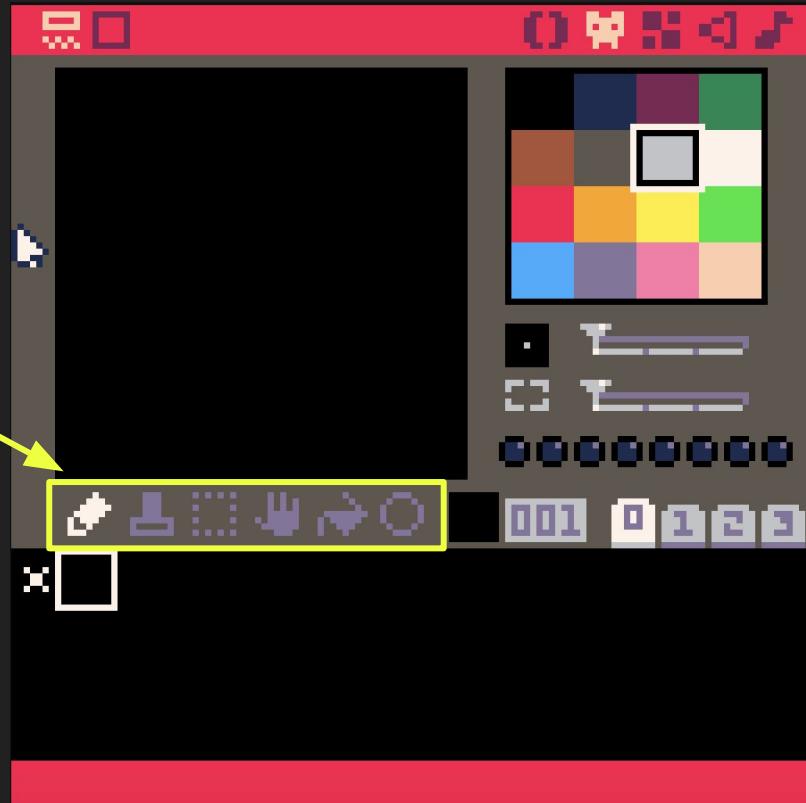
Using the PICO-8 Editor



Drawing Sprites

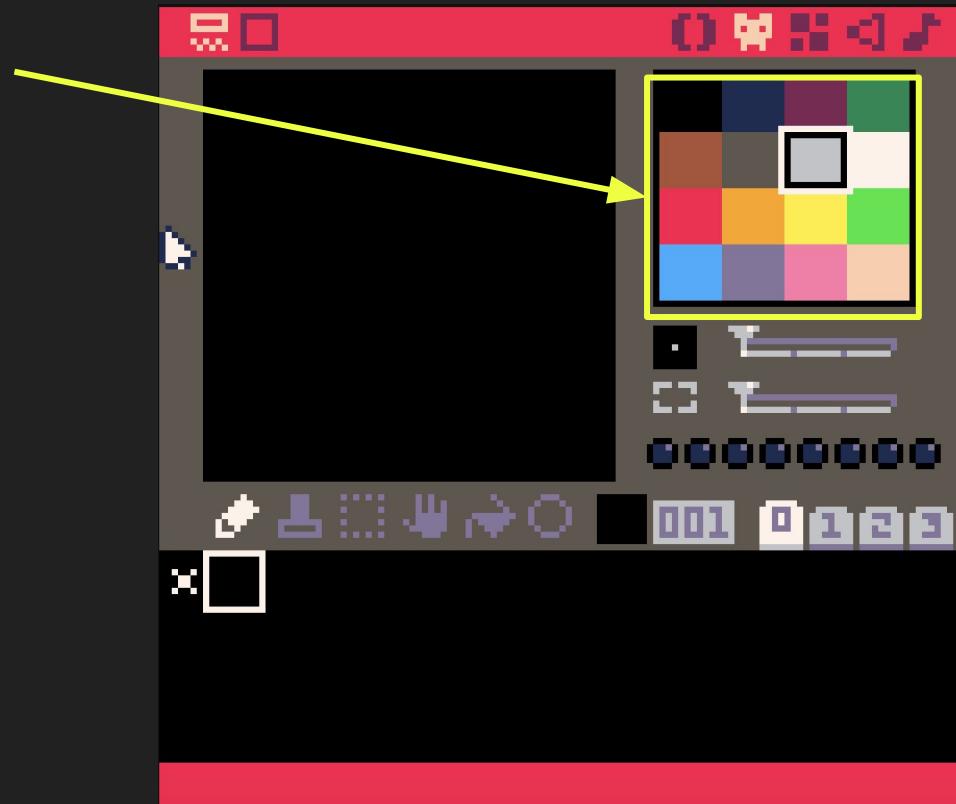
Drawing Sprites

- In the sprite editor, you have a few simple tools for drawing pixel art
- By default, each sprite is 8 x 8 pixels



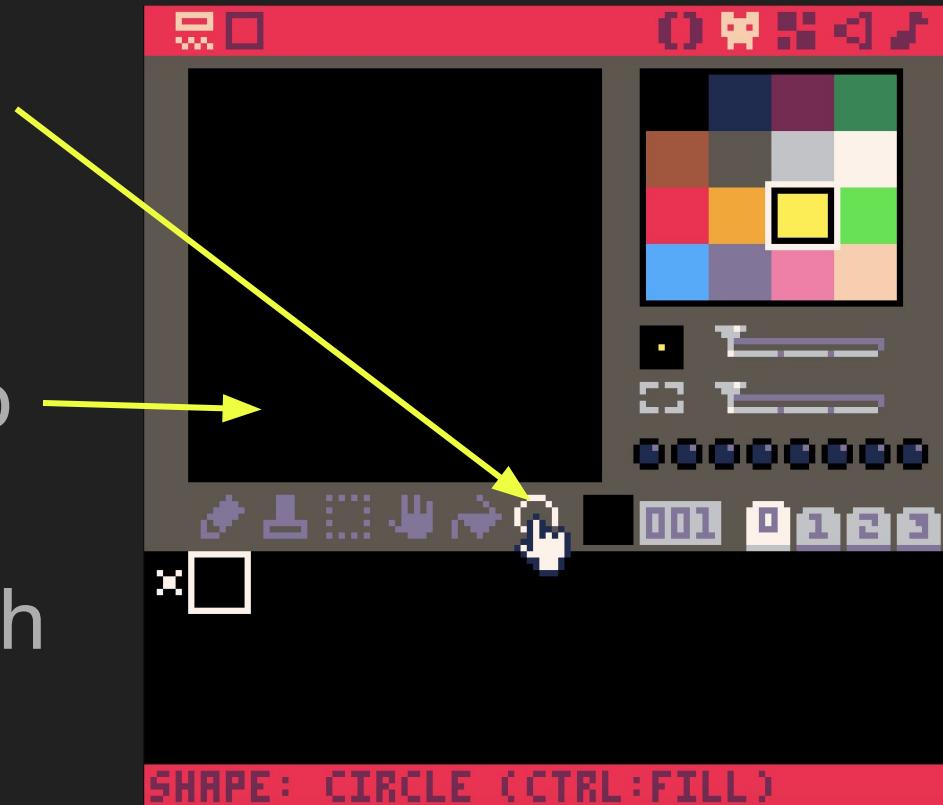
Drawing Sprites

- Select a color here



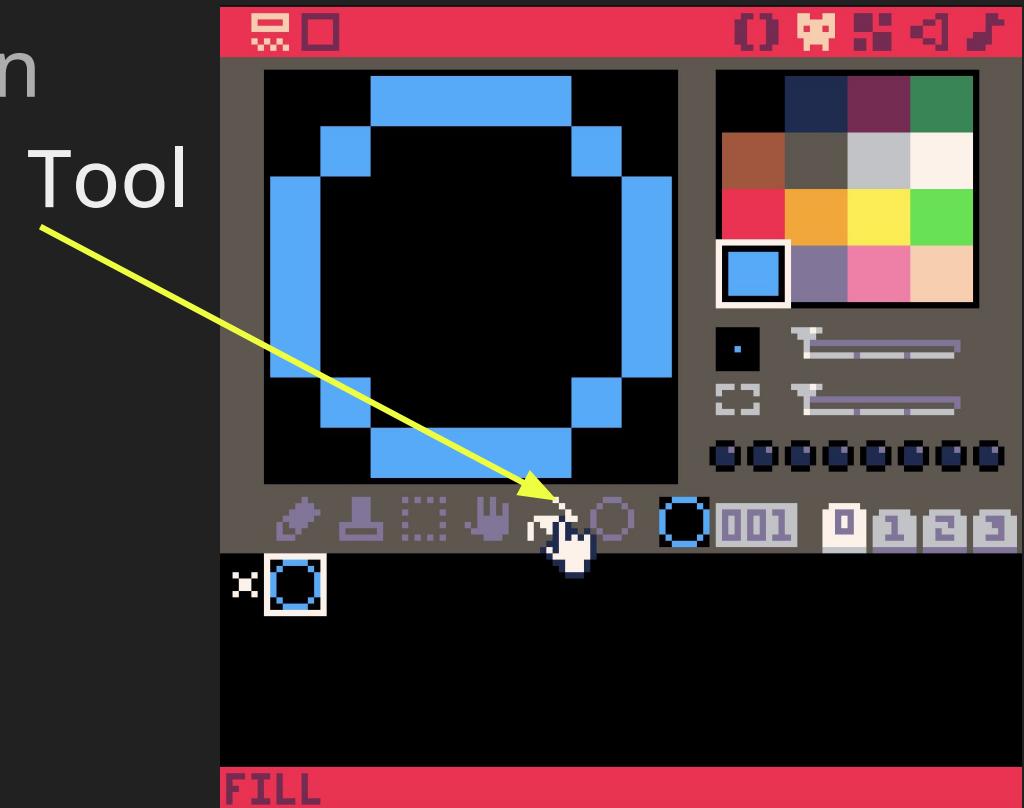
Drawing Sprites

- Click the Circle Tool to select it
- Then click and drag inside the canvas to draw a circle
- Hold CTRL to fill with the selected color



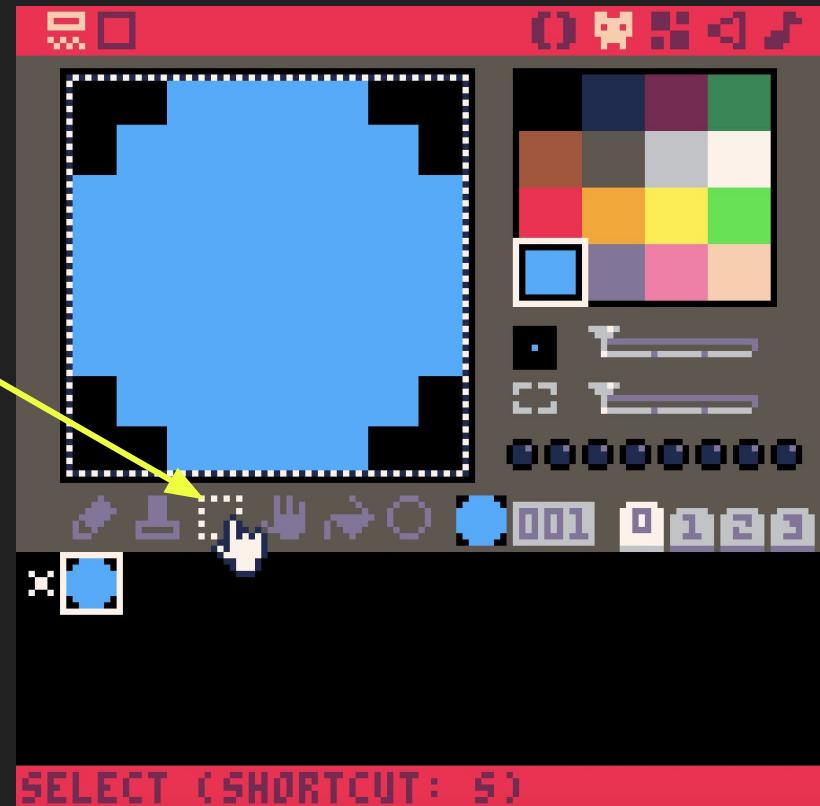
Drawing Sprites

- You can also fill an area using the Fill Tool



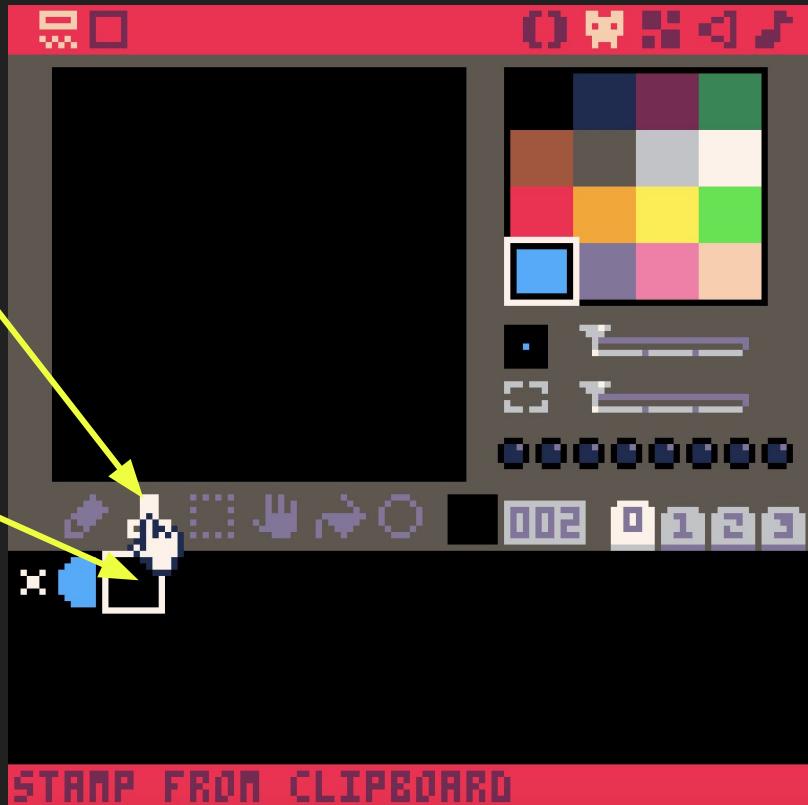
Drawing Sprites

- You can copy a sprite (or a portion of one) to your clipboard using the Select Tool and pressing CTRL C
- *Even on a Mac, the shortcut is CTRL, not CMD*



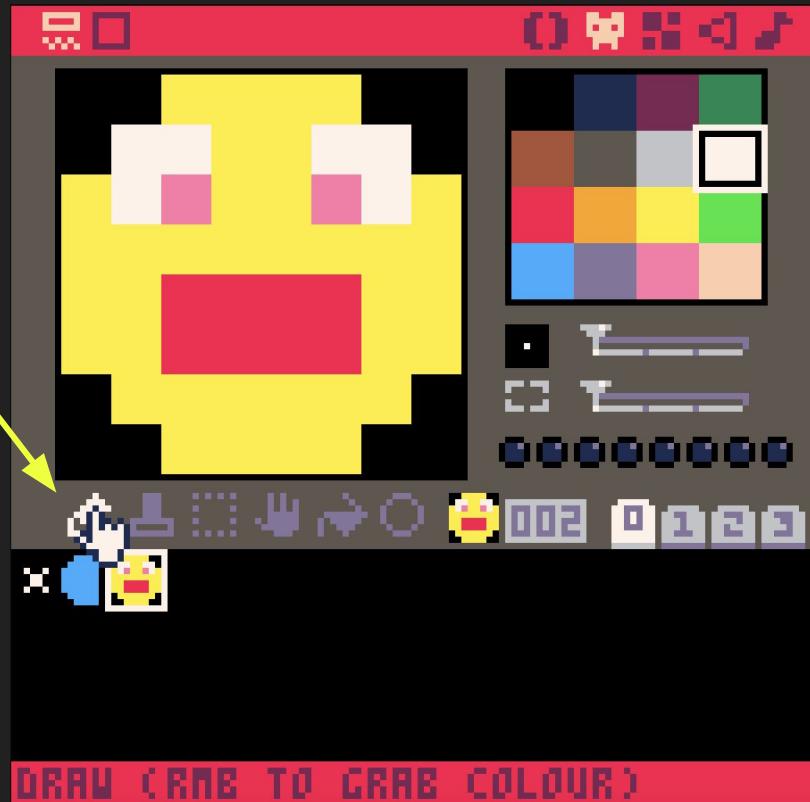
Drawing Sprites

- Use the Stamp Tool to paste what's on your clipboard (or use CTRL-V)
- Switch to a different sprite slot below the canvas to paste into a new sprite



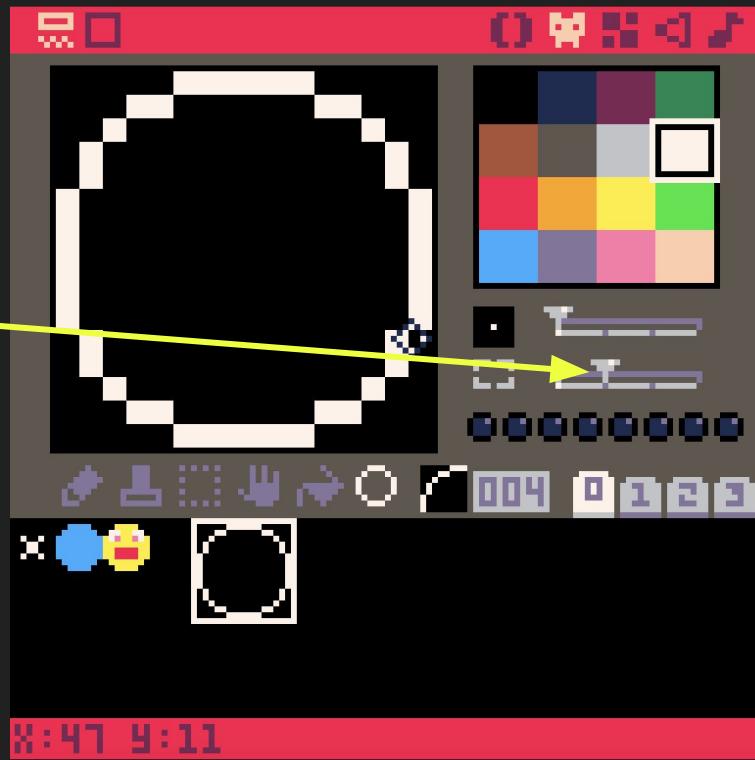
Drawing Sprites

- Use the Draw Tool to draw individual pixels with the selected color



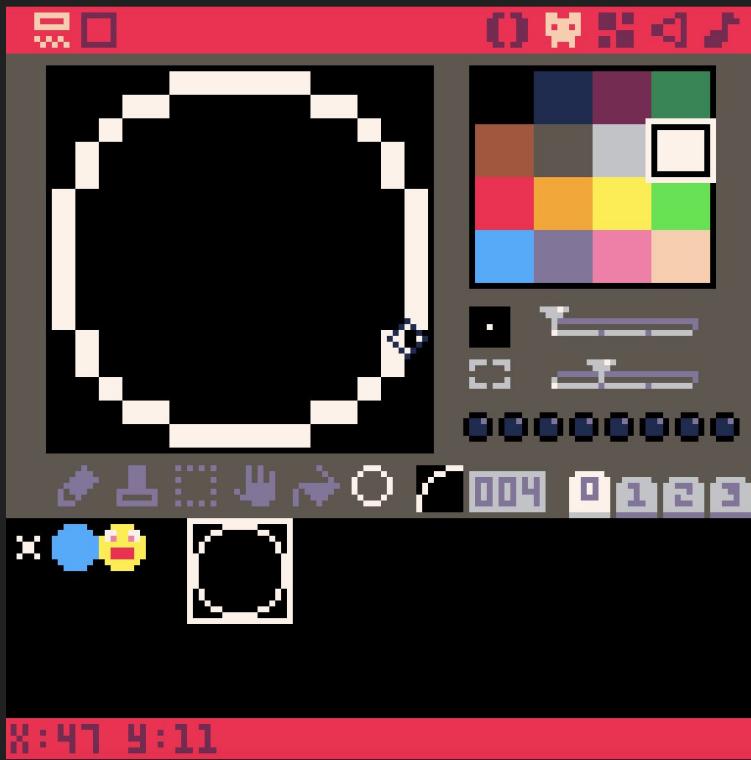
Drawing Sprites

- Use this slider to change the size of your canvas
- This is a 16 x 16px circle
- You can also make 32 or 64 pixel sprites



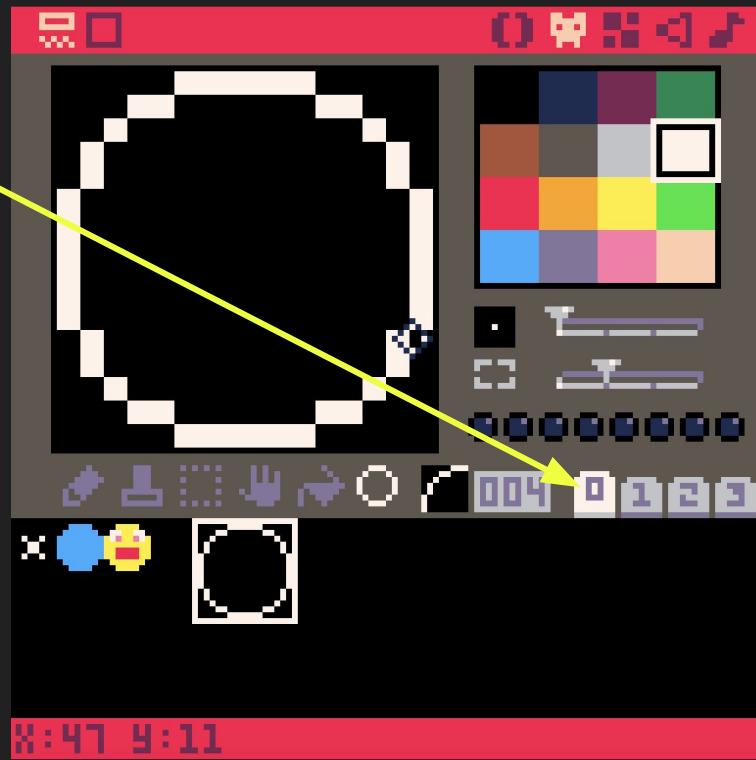
Drawing Sprites

- You have a limited amount of space in your sprite sheet, so larger sprites means fewer sprites



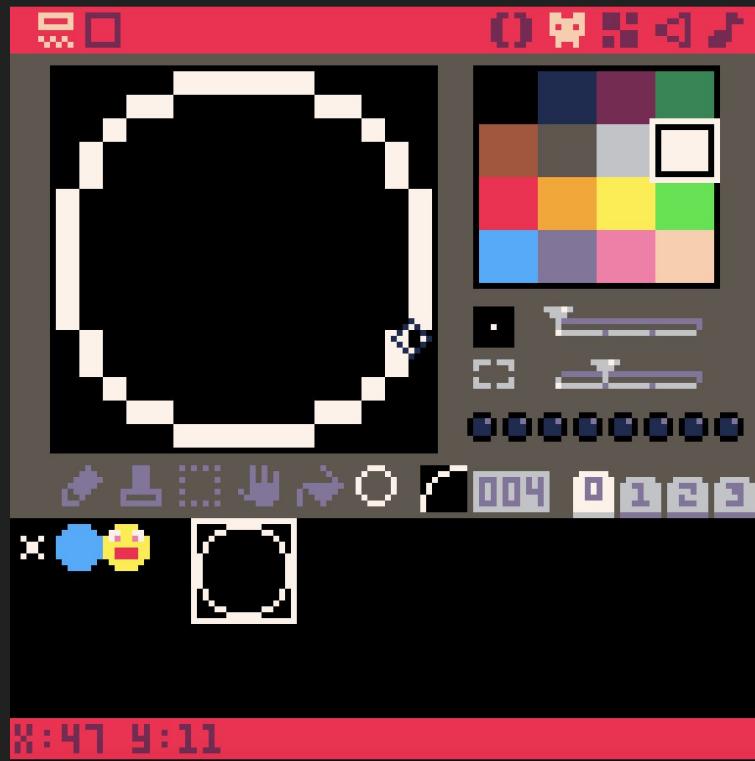
Drawing Sprites

- You have 4 tabs (labeled 0, 1, 2, 3) for drawing sprites, but it's *strongly recommended to only use Tabs 0 and 1*, because the sprite and map editors share some of the same memory



Drawing Sprites

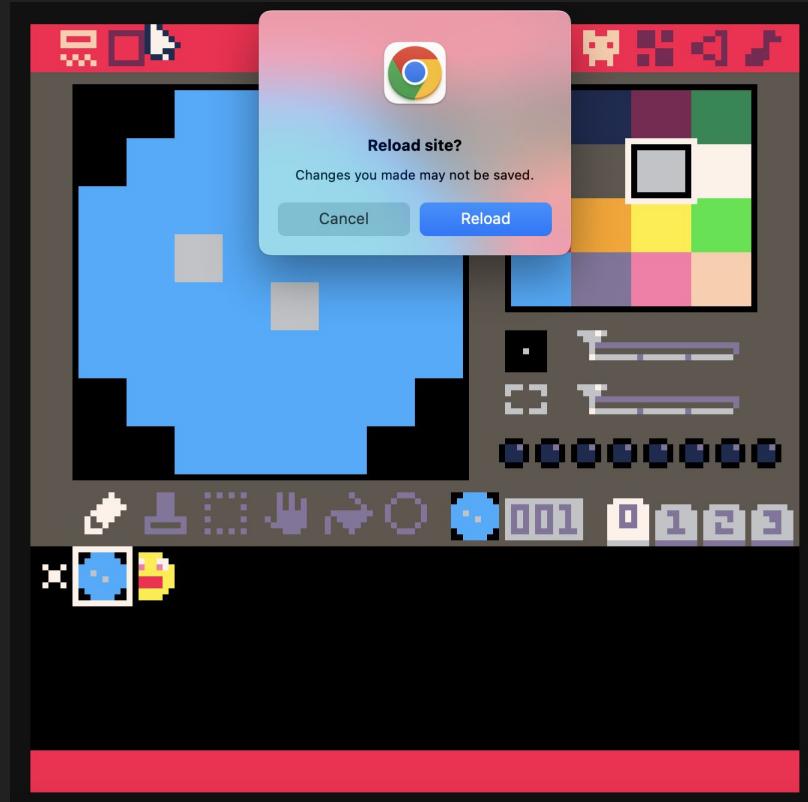
- Not all games require a map, so in that case you could use all 4 tabs to draw sprites
- Just be aware that the map and sprite editors share some memory



Saving Your Work

Saving Your Work

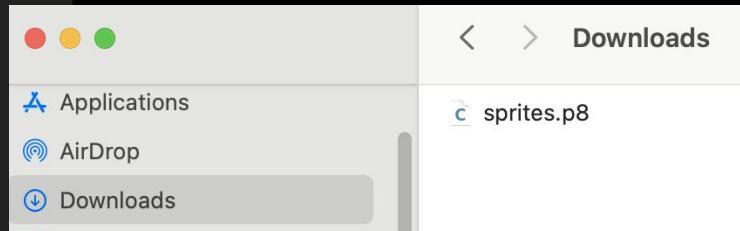
- Press **CTRL S** to save your work to the browser window
- But be aware that **refreshing your window** will erase the data saved in the browser



Saving Your Work

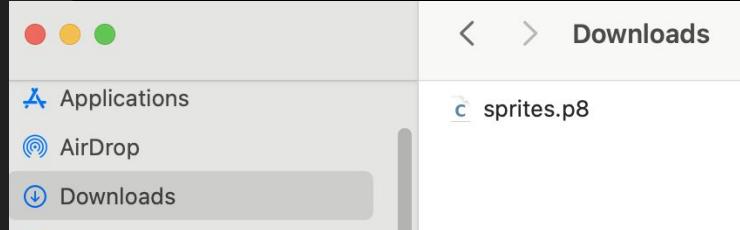
- Hit Esc and type the SAVE command to download your file
- You can load a file from your computer

I recommend saving or backing up your work for this course to Google Drive, so you can easily download the files anywhere



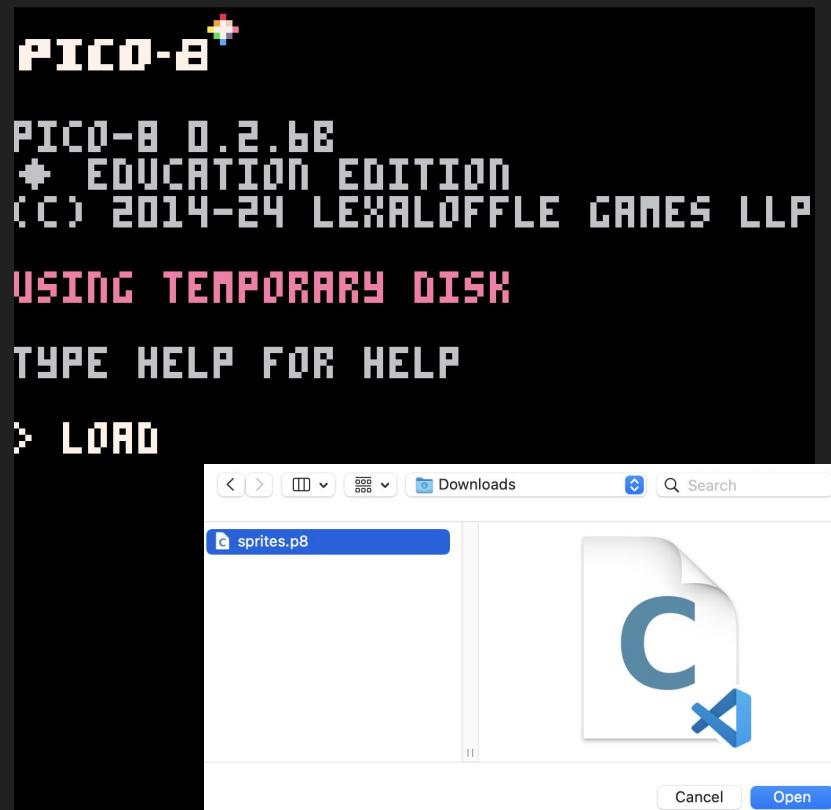
Saving Your Work

- You can provide a filename and extension after the command SAVE
- For example:
SAVE MYGAME.p8
- *PICO-8 files use the .P8 extension*
- *You'll type in all caps by default*
- Press Enter/Return to perform the command



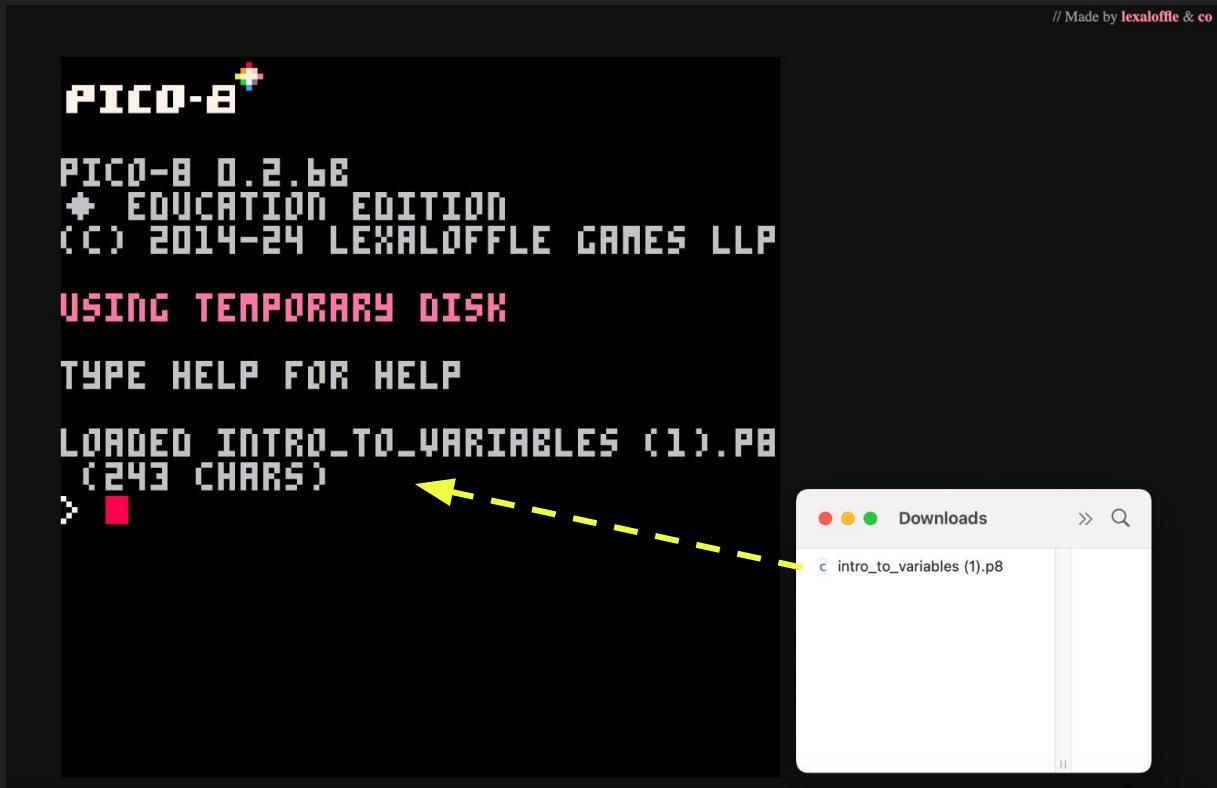
Loading a Saved P8 File

- Once you've saved, you can load a .P8 file using the LOAD command
- You'll be asked to browse for the file



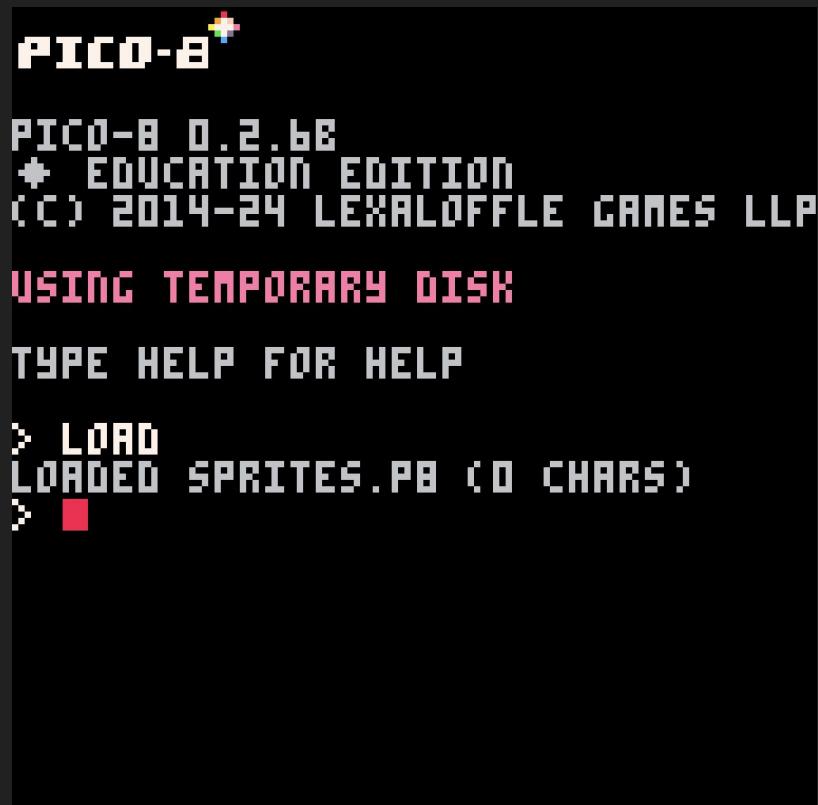
Loading a Saved P8 File

You can also
drag the file
into the PICO-8
window from
your desktop



Loading a Saved P8 File

- When you see a message that the file was loaded, you can press Esc to return to the editor and resume your work



PICO-8
PICO-8 0.2.68
★ EDUCATION EDITION
(C) 2014-24 LEXALOFFLE GAMES LLP
USING TEMPORARY DISK
TYPE HELP FOR HELP
> LOAD
LOADED SPRITES.P8 (0 CHARS)
> ■

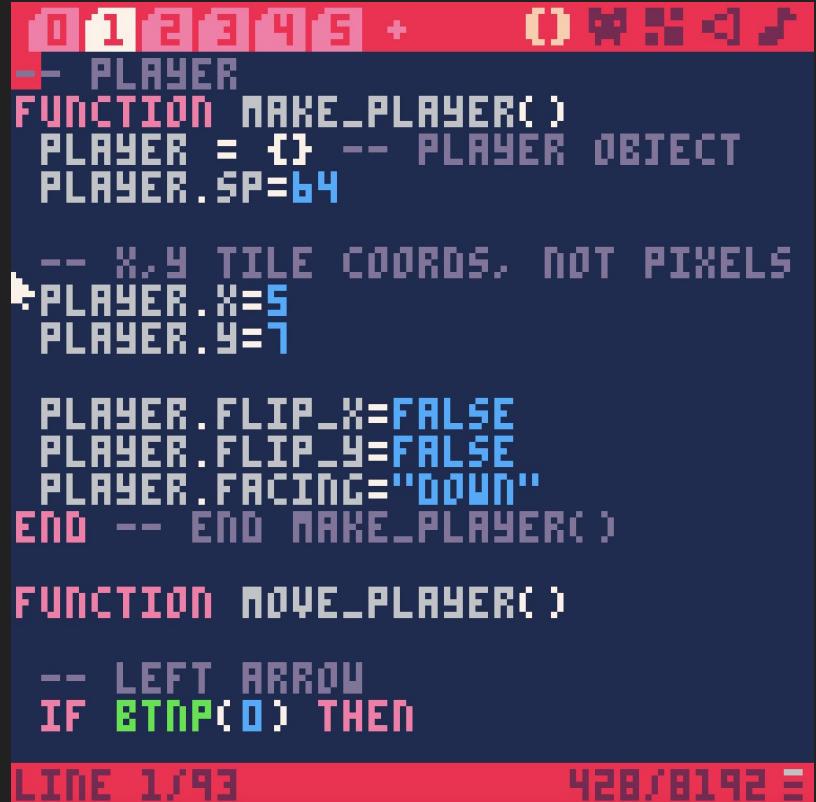
Loading a Saved P8 File

- Please note that *the desktop version of the software is a little different when it comes to loading saved files*
 - [https://www.reddit.com/r/pico8/comments/s0vy39/where can i find my save files/](https://www.reddit.com/r/pico8/comments/s0vy39/where_can_i_find_my_save_files/)

Setting up an External Code Editor

Coding in PICO-8

The font in the web editor is very stylized, so if you prefer an alternative code editor, I recommend [Microsoft Visual Studio Code](#) with this PICO-8 extension



A screenshot of the PICO-8 game editor interface. The top bar shows the title 'PICO-8' and the number '1'. The main area is a code editor with a dark blue background and white text. It displays a portion of BASIC script:

```
0 1 2 3 4 5 + 0 W H D Z  
-- PLAYER  
FUNCTION MAKE_PLAYER()  
PLAYER = {} -- PLAYER OBJECT  
PLAYER.SP=64  
  
-- X,Y TILE COORDS, NOT PIXELS  
PLAYER.X=5  
PLAYER.Y=7  
  
PLAYER.FLIP_X=False  
PLAYER.FLIP_Y=False  
PLAYER.FACING="DOWN"  
END -- END MAKE_PLAYER()  
  
FUNCTION MOVE_PLAYER()  
  
-- LEFT ARROW  
IF BTNP(0) THEN  
LINE 1/93 428/8192 E
```

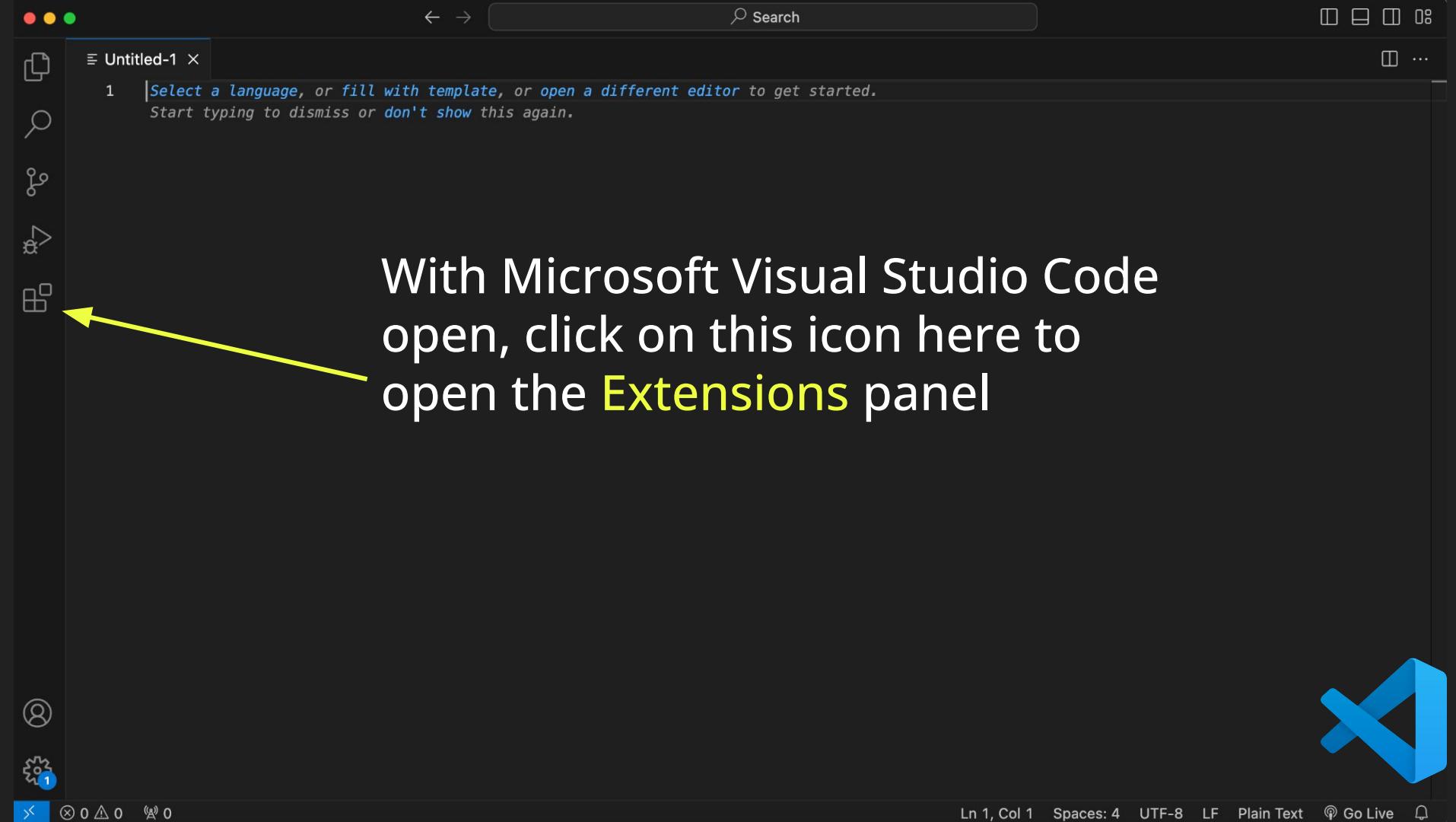
Using an External Code Editor

- Microsoft Visual Studio Code 
 - A professional-grade programming environment/code editor
 - Runs on Windows, MacOSX, and Linux
- Is installed on the lab computers
- Download on a personal computer

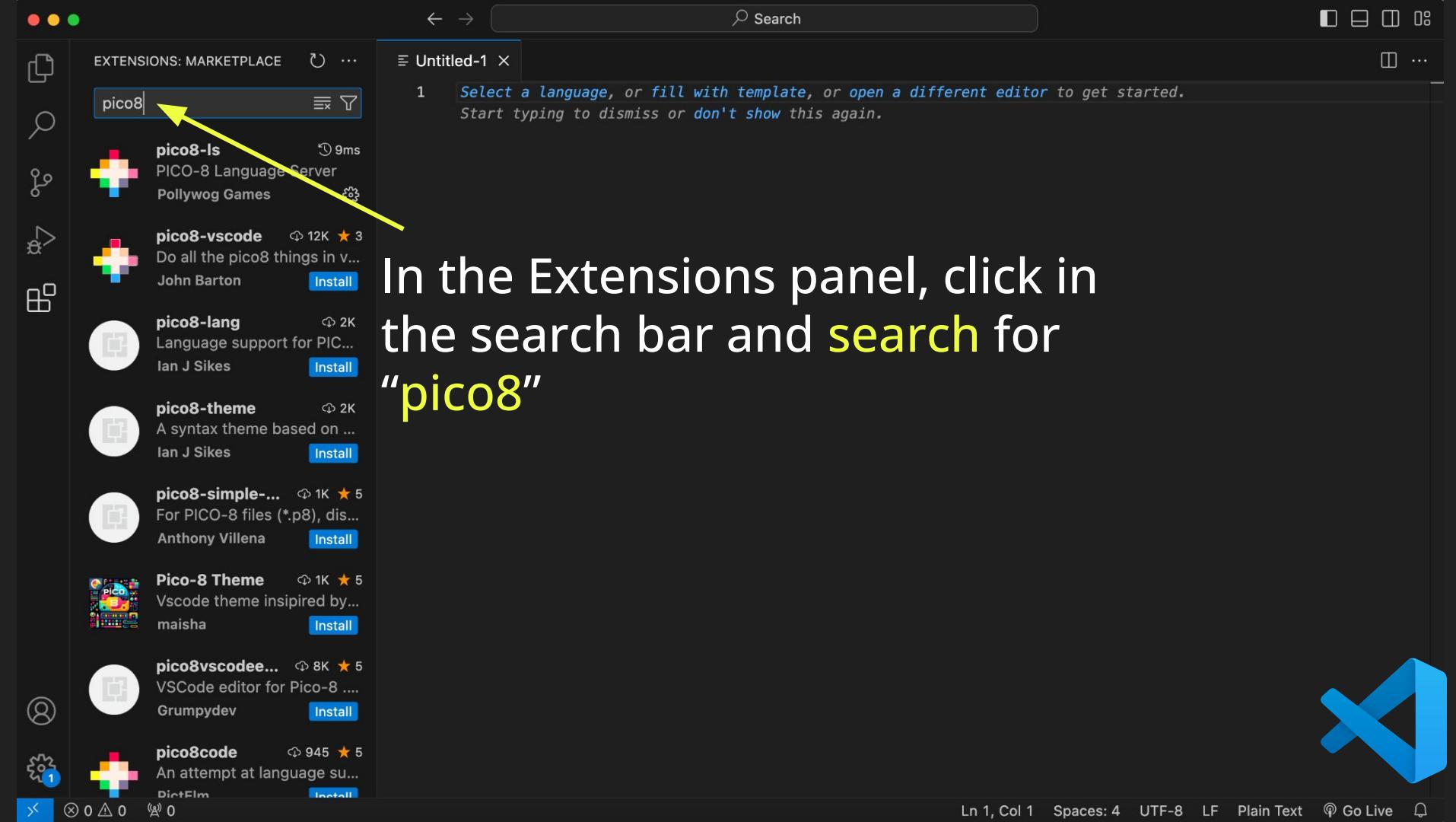
Using an External Code Editor

- Launch [Microsoft Visual Studio Code](#)
- Install the [pico8-ls Extension](#) in Visual Studio Code
 - Provides syntax highlighting and other Lua/PICO-8 support





With Microsoft Visual Studio Code open, click on this icon here to open the **Extensions** panel



In the Extensions panel, click in the search bar and **search** for
"pico8"

EXTENSIONS: MARKETPLACE

Search: pico8

Untitled-1

Select a language, or fill with template, or open a different editor to get started.
Start typing to dismiss or don't show this again.

pico8-Is PICO-8 Language Server Pollywog Games

pico8-vscode Do all the pico8 things in... John Barton **Install**

pico8-lang Language support for PIC... Ian J Sikes **Install**

pico8-theme A syntax theme based on... Ian J Sikes **Install**

pico8-simple-... For PICO-8 files (*.p8), dis... Anthony Villena **Install**

Pico-8 Theme Vscode theme inspired by... maisha **Install**

pico8vscodee... VSCode editor for Pico-8 ... Grumpydev **Install**

pico8code An attempt at language su... Dicfflm **Install**

Ln 1, Col 1 Spaces: 4 UTF-8 LF Plain Text Go Live

You should see a result called **pico8-Is: PICO-8 Language Server**

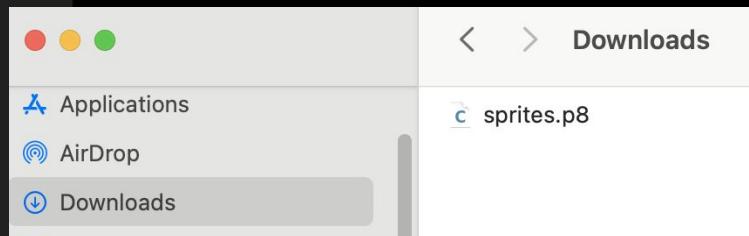
Click the blue Install button to enable the extension in VS Code

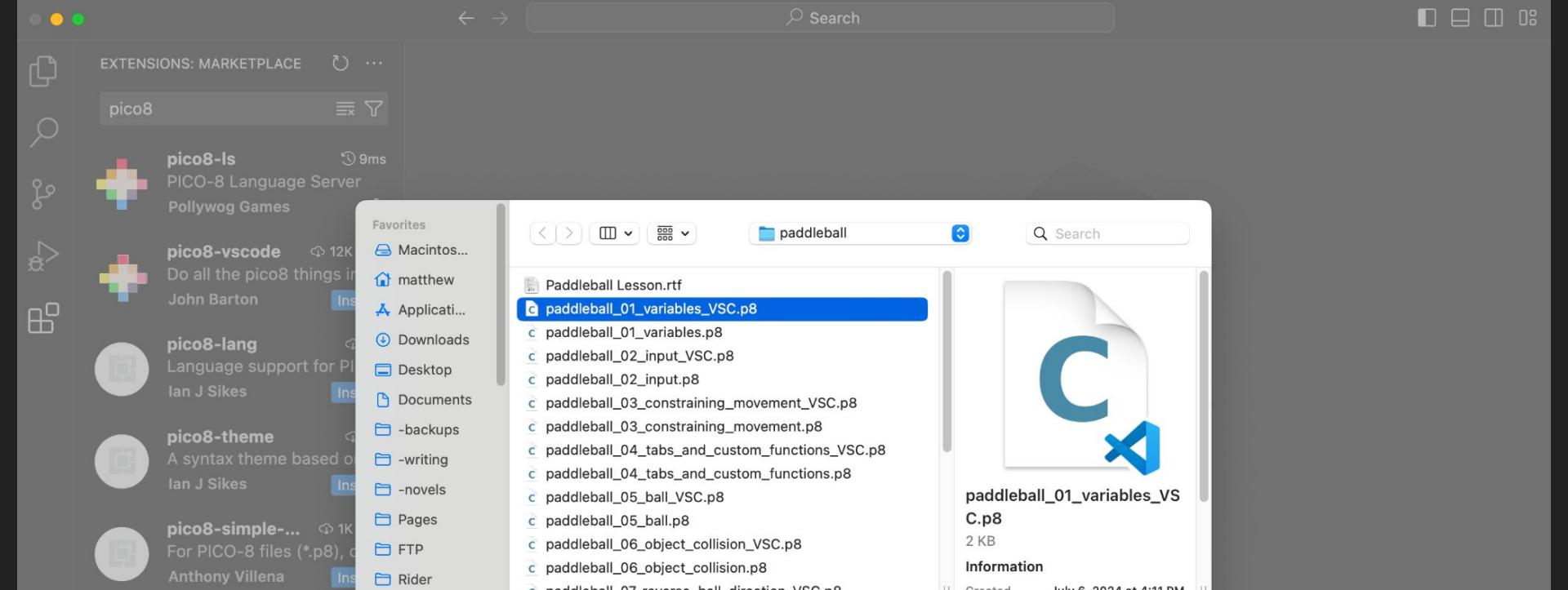
It should only take a few moments



Save Your Work

- In the PICO-8 editor,
Hit Esc and type the
SAVE command with
a filename to
download your file





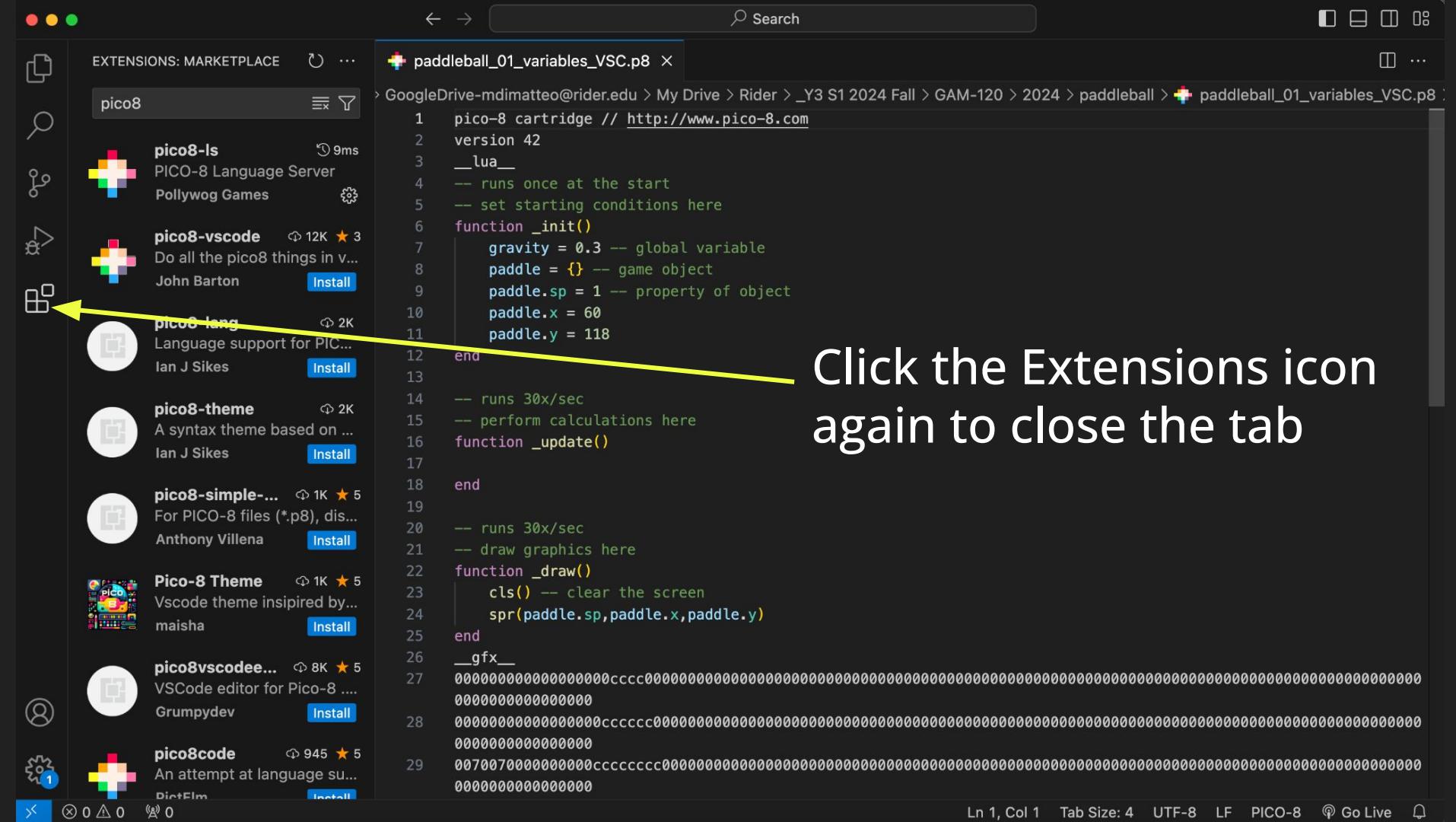
Back in VS Code, select **File > Open** or **CTRL O (Windows)** or **CMD O (Mac)** to browse for your PICO-8 file



It should have a .p8 file extension

The screenshot shows the Visual Studio Code interface. On the left, the Extensions Marketplace sidebar is open, displaying a list of extensions related to the pico8 language. The top extension listed is "pico8-is" by Pollywog Games, which has a rating of 9ms and 3 reviews. Other extensions shown include "pico8-vscode", "pico8-lang", "pico8-theme", "pico8-simple-", "Pico-8 Theme", "pico8vscodee...", and "pico8code". Most of these have an "Install" button next to them. The main workspace shows a code editor with a file named "paddleball_01_variables_VSC.p8". The code is written in a PICO-8 script language, featuring functions like _init() and _update(), and variables like paddle.sp and paddle.x. The code is syntax-highlighted in green, blue, and purple. The status bar at the bottom indicates "Ln 1, Col 1" and "Tab Size: 4".

With the pico8-is extension enabled, you'll see your code highlighted like this



With _lua_

```
1 pico-8 cartridge // http://www.pico-8.com
2 version 42
3 _lua_
4 -- runs once at the start
5 -- set starting conditions here
6 function _init()
7     gravity = 0.3 -- global variable
8     paddle = {} -- game object
9     paddle.sp = 1 -- property of object
10    paddle.x = 60
11    paddle.y = 118
12 end
13
14 -- runs 30x/sec
15 -- perform calculations here
16 function _update()
17
18
19
20
21
22
23
24 spr(paddle.sp,paddle.x,paddle.y)
```

Without _lua_

```
1 pico-8 cartridge // http://www.pico-8.com
2 version 42
3
4 -- runs once at the start
5 -- set starting conditions here
6 function _init()
7     gravity = 0.3 -- global variable
8     paddle = {} -- game object
9     paddle.sp = 1 -- property of object
10    paddle.x = 60
11    paddle.y = 118
12 end
13
14 -- runs 30x/sec
15 -- perform calculations here
16 function _update()
17
18
19
20
21
22
23
24 spr(paddle.sp,paddle.x,paddle.y)
```

Please note that you will need to type at least one character in the PICO-8 code editor before saving

this adds a _lua_ tag to the downloaded P8 file
that you need for syntax highlighting to work in
Visual Studio Code

You will need to install the pico8-ls extension whenever using a different lab computer or signing in with another account

If you return to the same computer you were using and log in with the same account, the extension should still be enabled

Loading a Saved P8 File

- You can load your game in PICO-8 by refreshing your window and using the LOAD command, or dragging the file in from your desktop



Coding in PICO-8

Coding in PICO-8

- In the code editor, we can type SPR() to display a sprite in our game
- SPR() is a “*function*”
- A *function* is a programming concept for a set of instructions for the computer to execute



Coding in PICO-8

- We need to tell the computer *which* sprite to draw
- Each sprite in our sprite sheet has a number associated with it
- We can type the number 1 inside the parentheses – **SPR(1)** to tell the program to draw sprite number 1



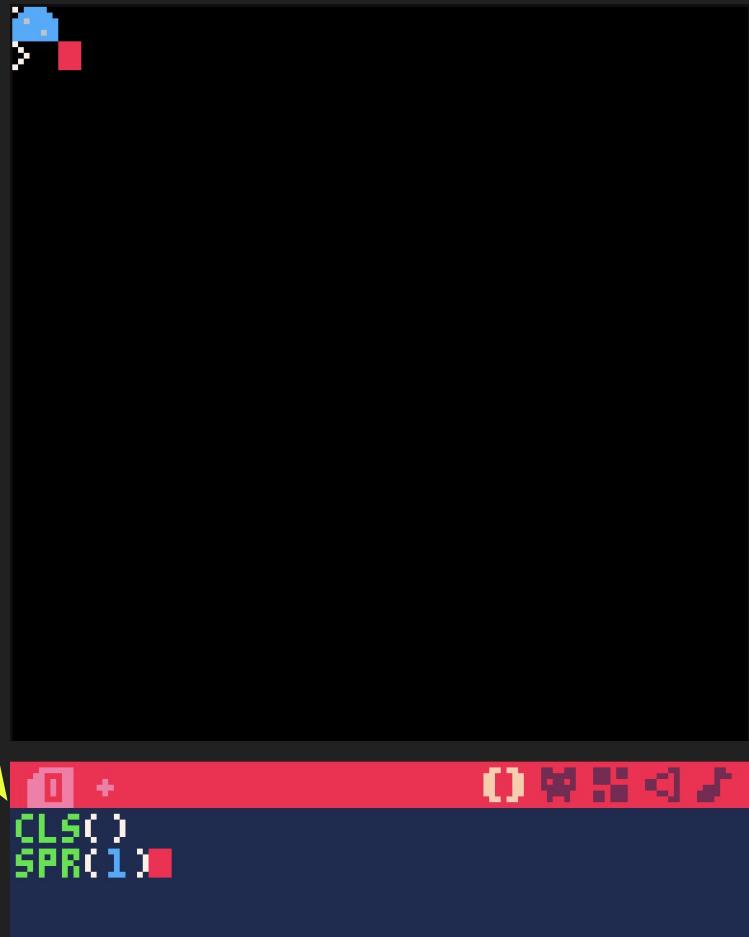
Coding in PICO-8

- Press CTRL R or hit Esc and use the RUN command to run the game
- Our sprite is drawn at the top left of the screen by default
- Also, it's drawn on top of our command line



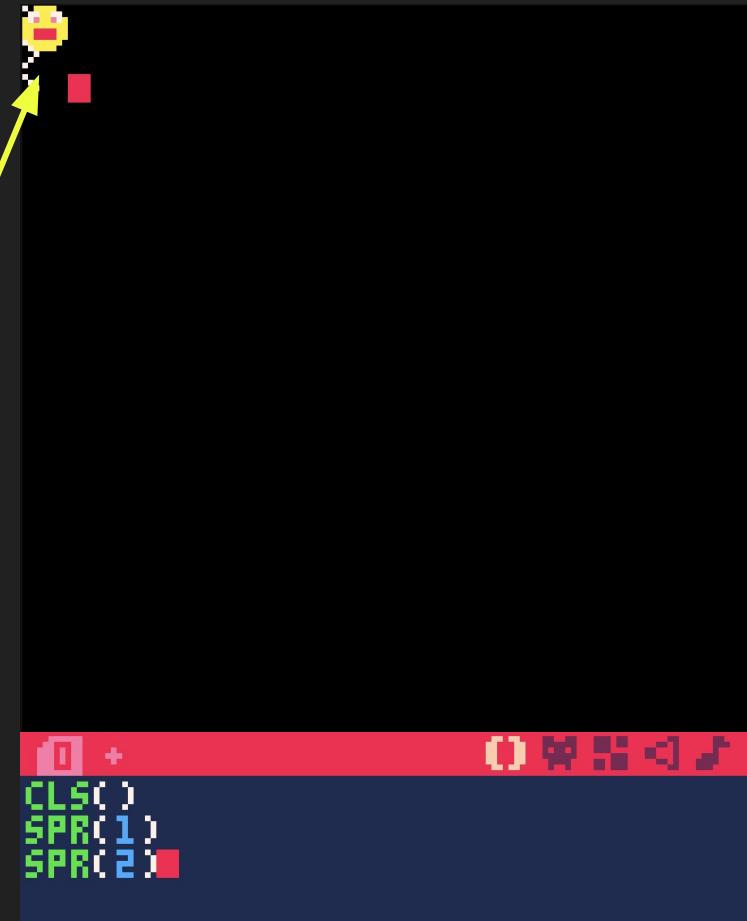
Coding in PICO-8

- We can get rid of the command line by typing CLS() before we type SPR(1)
- CLS() is a function that clears the screen



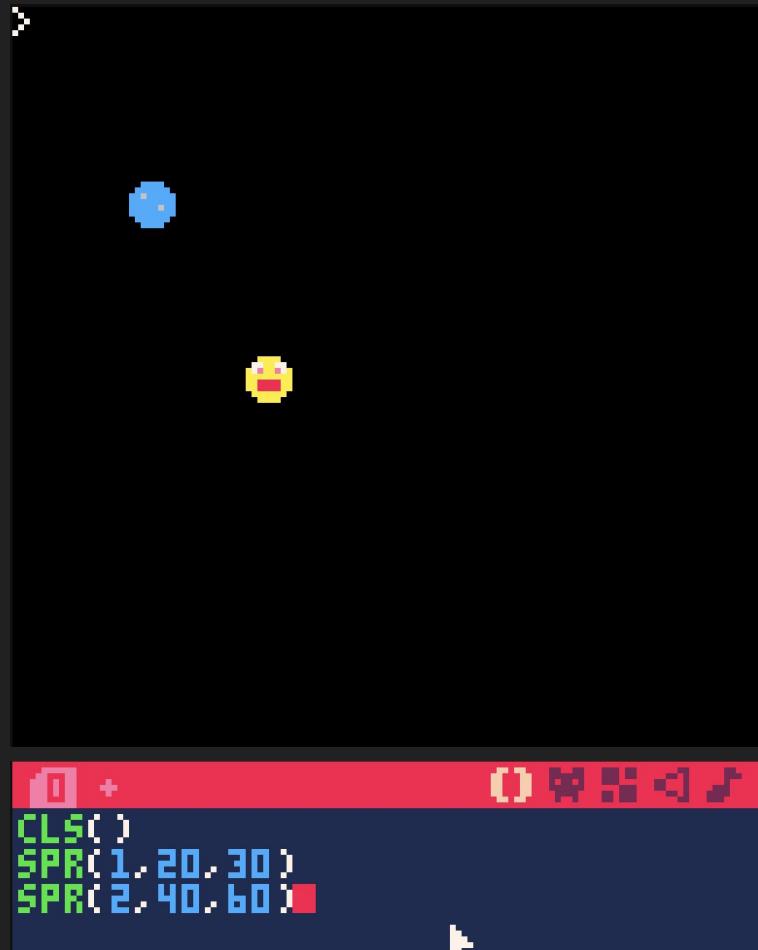
Coding in PICO-8

- Notice what happens if I try to draw sprite 2 after sprite 1
 - *Sprite 2 covers up sprite 1 because both are being drawn in the same location*



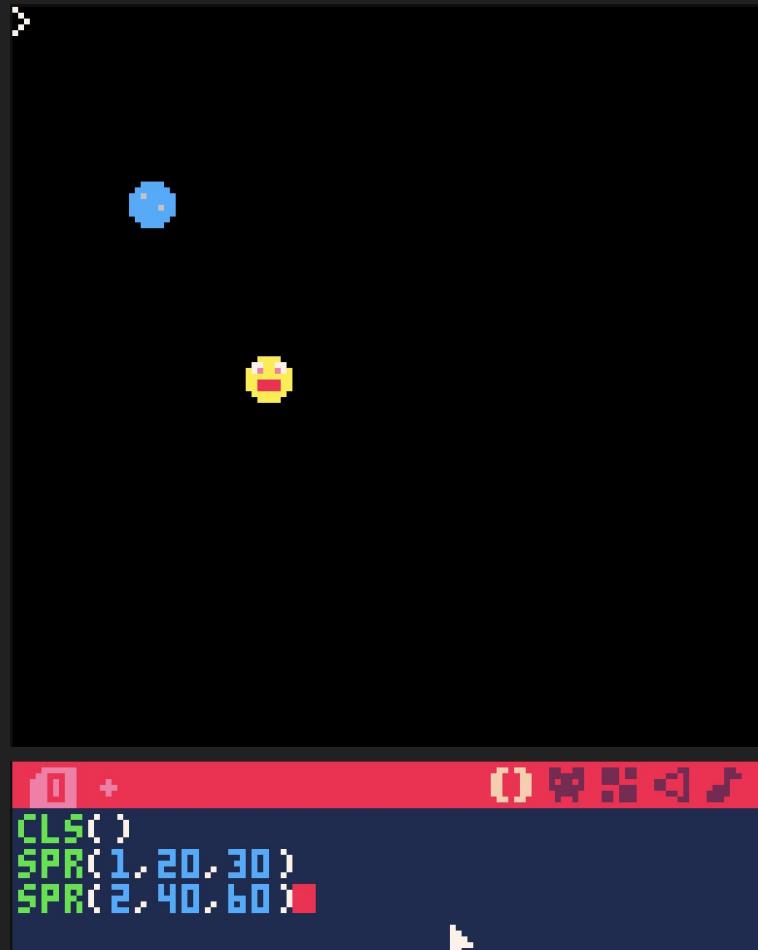
Coding in PICO-8

- We can add other values to the SPR() function to tell the program *where* to draw each sprite on the screen



Coding in PICO-8

- The SPR() function can be given values inside the parentheses for custom behavior
- These values must be given in the proper order



Coding in PICO-8

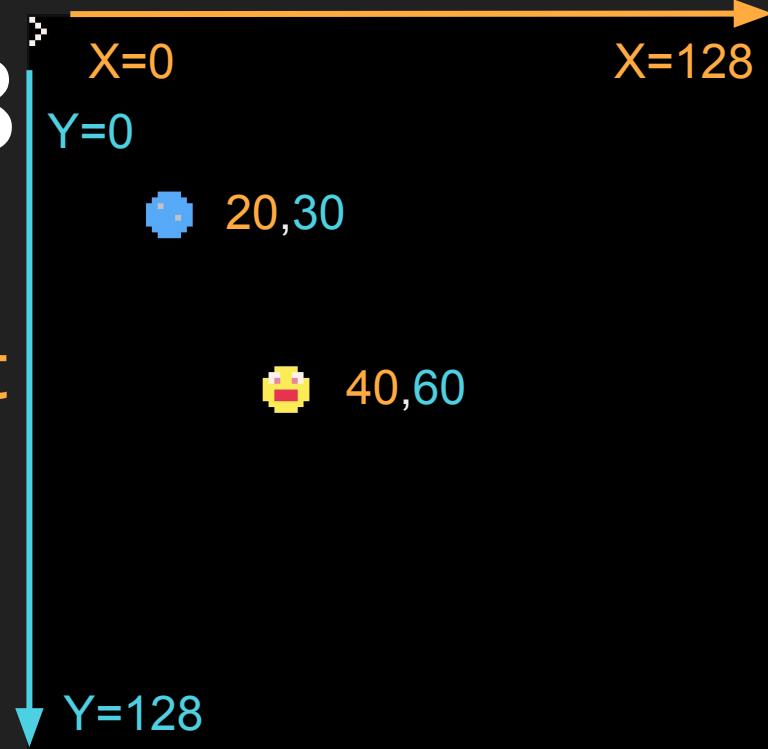
- 1st value: sprite #
- 2nd value: x coordinate
- 3rd value: y coordinate

There are other values we can provide, but these 3 will suffice for now



Coding in PICO-8

- The X,Y coordinate plane is drawn from **left to right** and **top to bottom**



```
SPR(1,20,30)  
SPR(2,40,60)
```



in: Reference, API



Spr



EDIT



```
spr( n, [x,] [y,] [w,] [h,] [flip_x,] [flip_y] )
```

Draws a sprite, or a range of sprites, on the screen.

n

The sprite number. When drawing a range of sprites, this is the upper-left corner.

x

The x coordinate (pixels). The default is 0.

y

The y coordinate (pixels). The default is 0.

You can look up any PICO-8 function on the [PICO-8 Wiki](#), and it will show you the exact order of the values that function can use



in: Reference, API



Spr



EDIT



```
spr( n, [x,] [y,] [w,] [h,] [flip_x,] [flip_y] )
```

Draws a sprite, or a range of sprites, on the screen.

n

The sprite number. When drawing a range of sprites, this is the upper-left corner.

x

The x coordinate (pixels). The default is 0.

y

The y coordinate (pixels). The default is 0.

Values with [square brackets]
around them are optional values



in: Reference, API



Spr



EDIT



```
spr( n, [x,] [y,] [w,] [h,] [flip_x,] [flip_y] )
```

Draws a sprite, or a range of sprites, on the screen.

n

The sprite number. When drawing a range of sprites, this is the upper-left corner.

x

The x coordinate (pixels). The default is 0.

y

The y coordinate (pixels). The default is 0.

The wiki also explains what each value means and what its default value is

Coding in PICO-8

- Another tradition when learning a new programming language is to begin by printing the words HELLO WORLD on the screen

HELLO WORLD!

> █



```
CLS()
PRINT("HELLO WORLD!")
```

Coding in PICO-8

- We can use PICO-8's PRINT() function to do this
- Inside the parentheses, type "HELLO WORLD" inside quotes

```
HELLO WORLD!
```

```
> █
```

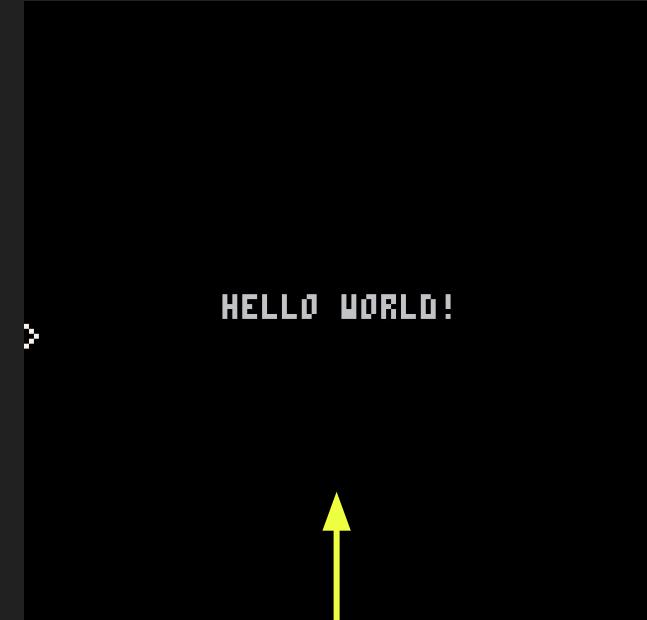


```
CLS()  
PRINT("HELLO WORLD!")
```

PRINT() is also useful for displaying a game HUD

Coding in PICO-8

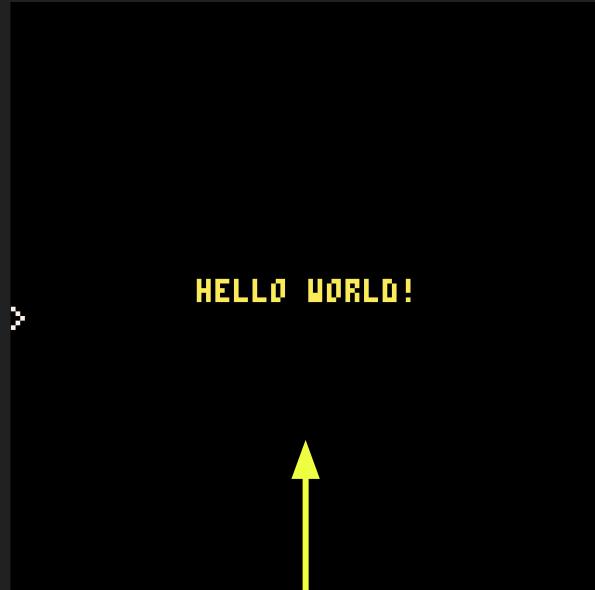
- We can add values after the quotes, separated by commas, to set the position of the text
- Here, **40** is the X coordinate and **60** is the Y coordinate



```
CLS()  
PRINT("HELLO WORLD!", 40, 60)
```

Coding in PICO-8

- We can even add another value to set the **color** of the text
- Here, 10 corresponds with the color **yellow**



```
CLS()
PRINT("HELLO WORLD!", 40, 60, 10)
```

Coding in PICO-8

- Each of the 16 available colors in PICO-8 corresponds with a number
- Refer to the [PICO-8 Cheat Sheet](#)

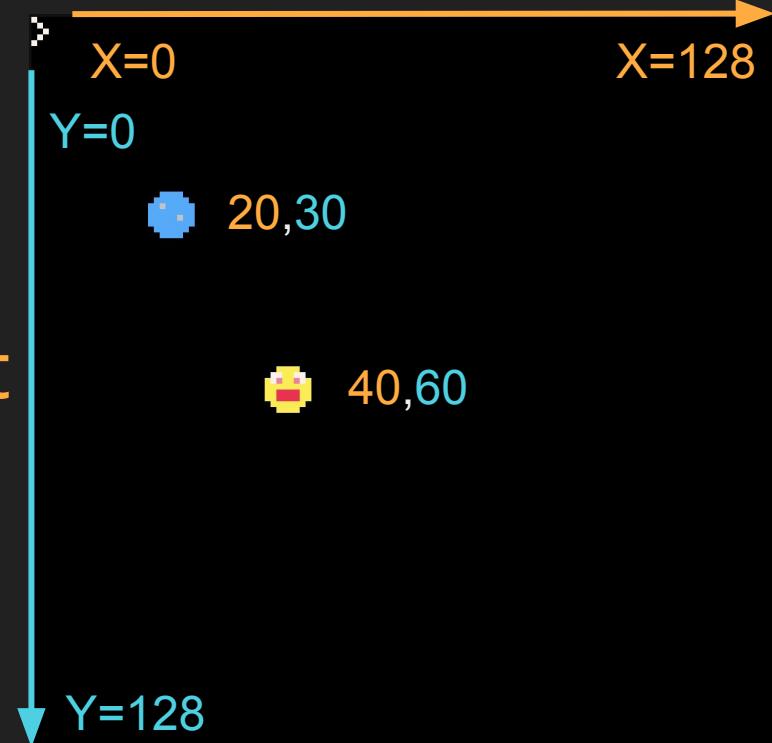
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
CLS()
PRINT("HELLO WORLD!", 40, 60, 10)
```

The PICO-8 Coordinate Plane

Coordinate Plane

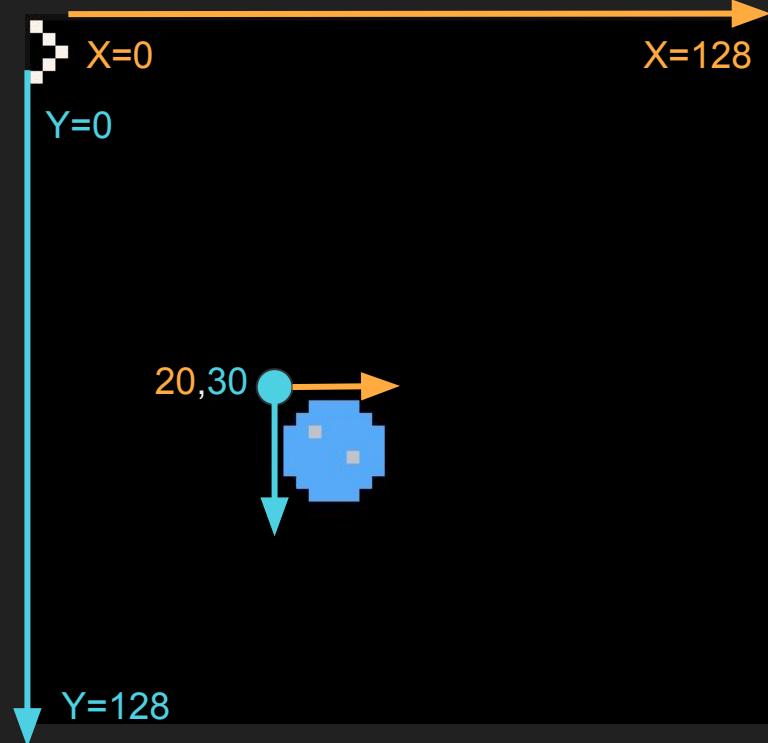
- The X,Y coordinate plane is drawn from **left to right** and **top to bottom**
- This means higher Y values are lower on the screen***



```
SPR(1, 20, 30)  
SPR(2, 40, 60)
```

Coordinate Plane

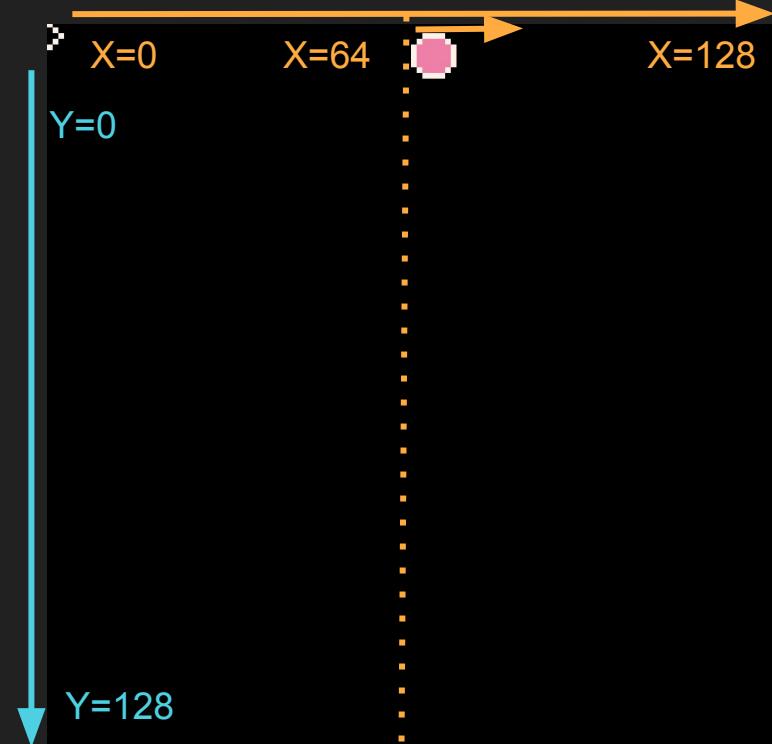
- PICO-8 *sprites are drawn from the top-left corner*
- Meaning the X,Y coordinates you give a sprite coincide with its top-left corner



```
|SPR(1, 20, 30)|
```

Coordinate Plane

- Because sprites are drawn from the top left, *setting the x value to half of the screen width ($128/2=64$) results in the sprite being slightly off-center to the right*



```
CLS( )  
SPR(1, 64, 2)
```

Coordinate Plane

- When trying to center a sprite, you must account for half its width or height because of the top-left origin



```
CLS( )  
SPR(1,64,2)
```

Coordinate Plane

- When trying to center a sprite, you must account for half its width or height because of the top-left origin
- $64 - (8/2) = 64 - 4 = 60$



Next, we'll want to get
these sprites **moving**
around the screen
and responding to
player **input**

Before we can move our
sprites in the game, we
need to understand the
way PICO-8 programs
are structured

The PICO-8 Program Structure

Download Example File: [basics_02_gameloop.p8](#)

The PICO-8 Program Structure

- These three functions comprise the main structure of your PICO-8 program:
 - `_init()`, `_update()`, `_draw()`
- This is sometimes called the PICO-8 “game loop”

The PICO-8 Program Structure

- `_init()`
 - Runs once at the start
 - Use for setting initial conditions
 - Declaring variables
- `_update()`
 - Loops 30x per second
 - Use for calculations, movement
- `_draw()`
 - Loops 30x per second
 - Use to draw sprites, print text

The PICO-8 Program Structure

- We can set our initial variable values in `_init()`

```
FUNCTION _INIT()
    n=1
    x=60
    y=2
END
```

- Then we can change those values in `_update()`

```
FUNCTION _UPDATE()
    y=y+2
END
```

- And apply those values in `_draw()`

```
FUNCTION _DRAW()
    CLS()
    SPR(n,x,y)
END
```

The PICO-8 Program Structure

- All functions must be “closed”
- You close a function using the keyword **END**

If you don't close a function, it can result in an error or unexpected behavior



```
FUNCTION _INIT()
    n=1
    x=60
    y=2
END

FUNCTION _UPDATE()
    y=y+2
END

FUNCTION _DRAW()
    CLS()
    SPR(n,x,y)
END
```

Using Variables

Download Example File: [paddleball_basic_02_variables.p8](#)

Using Variables

- *If we only use numbers to position our sprites, they won't be able to respond to player input and move*
- Using fixed numbers is called “hard-coding”



Using Variables

- We can set our initial variable values in `_init()`

```
FUNCTION _INIT()
  n=1
  x=60
  y=2
END
```

- Then we can change those values in `_update()`

```
FUNCTION _UPDATE()
  y=y+2
END
```

- And apply those values in `_draw()`

```
FUNCTION _DRAW()
  CLS()
  SPR(n,x,y)
END
```

Using Variables



This results in the ball moving lower on the screen

```
FUNCTION _INIT()
```

```
n=1
```

```
x=60
```

```
y=2
```

```
END
```

```
FUNCTION _UPDATE()
```

```
y=y+2
```

```
END
```

```
FUNCTION _DRAW()
```

```
CLS()
```

```
SPR(n,x,y)
```

```
END
```

Understanding the Game Loop

The PICO-8 Program Structure

- `_init()` only runs *once*, as soon as the game starts
- *If these values need to change, that will be done in `_update()`*

```
FUNCTION _INIT()
n=1
x=60
y=2
END
```

```
FUNCTION _UPDATE()
y=y+2
END
```

```
FUNCTION _DRAW()
CLS()
SPR(n,x,y)
END
```

The PICO-8 Program Structure

- `_update` and `_draw()` both run in a *loop*, 30 times per second (*or once every 1/30th of a second*)

```
FUNCTION _INIT()
```

```
    n=1  
    x=60  
    y=2
```

```
END
```

```
FUNCTION _UPDATE()
```

```
    y=y+2
```

```
END
```

```
FUNCTION _DRAW()
```

```
    CLS()
```

```
    SPR(n,x,y)
```

```
END
```

The PICO-8 Program Structure

Frame	y
0	2
1	4
2	6
3	8

At frame 0, y is what it's set at in `_init()`, or 2

```
FUNCTION _INIT()
    n=1
    x=60
    y=2
END

FUNCTION _UPDATE()
    y=y+2
END

FUNCTION _DRAW()
    CLS()
    SPR(n,x,y)
END
```

The PICO-8 Program Structure

Frame	y
0	2
1	4
2	6
3	8

Because _update() runs continuously, the value of y increases each frame by the amount specified in the code (2)

```
FUNCTION _INIT()
    n=1
    x=60
    y=2
END

FUNCTION _UPDATE()
    y=y+2
END

FUNCTION _DRAW()
    CLS()
    SPR(n,x,y)
END
```

The PICO-8 Program Structure

Frame	y
0	2
1	4
2	6
3	8
30 (1 second)	62

After 1 second (30 frames), the ball will have moved 60 pixels beyond its initial position (from 2 to 62)

```
FUNCTION _INIT()
    n=1
    x=60
    y=2
END

FUNCTION _UPDATE()
    y=y+2
END

FUNCTION _DRAW()
    CLS()
    SPR(n,x,y)
END
```

The PICO-8 Program Structure

- It's important to begin the `_draw()` block with `cls()` to clear the screen
- This resets the screen each frame



```
FUNCTION _DRAW()
CLS()
SPR(n,x,y)
END
```

```
FUNCTION _DRAW()
CLS()
SPR(n,X,Y)
END
```



```
FUNCTION _DRAW()
SPR(n,X,Y)
END
```



Without cls(), the ball at its previous position is not wiped from the screen, resulting in this effect

The PICO-8 Program Structure

- It helps to think through what is happening each frame
- *The code executes in the order it is written, and it does this each frame*

```
FUNCTION _DRAW()
    CLS()
    SPR(n,x,y)
END
```

The PICO-8 Program Structure

Frame	Effect
1 (Line 1 of _draw)	clear the screen
1 (Line 2 of _draw)	draw the ball
2 (Line 1 of _draw)	clear the screen
2 (Line 2 of _draw)	draw the ball
3 (Line 1 of _draw)	clear the screen
3 (Line 2 of _draw)	draw the ball

And so on . . .

```
FUNCTION _DRAW()
    CLS()
    SPR(1,X,Y)
END
```

The PICO-8 Program Structure

Frame	Effect
1 (Line 1 of _draw)	clear the screen
1 (Line 2 of _draw)	draw the ball at $y=4$
2 (Line 1 of _draw)	clear the screen
2 (Line 2 of _draw)	draw the ball at $y=6$
3 (Line 1 of _draw)	clear the screen
3 (Line 2 of _draw)	draw the ball at $y=8$

The diagram illustrates the flow of the variable y from its modification in the `_UPDATE()` function to its use in the `_DRAW()` function. A yellow oval encloses the `Y=Y+2` line in the `_UPDATE()` function. A yellow arrow points from this oval to the `SPR(n, x, y)` line in the `_DRAW()` function, indicating that the updated value of y is being passed to the drawing function.

```
FUNCTION _UPDATE()
    Y=Y+2
END

FUNCTION _DRAW()
    CLS()
    SPR(n, x, y)
END
```

Even though the spr() code in _draw() isn't changing, the ball is being drawn in a different position because the value of y is being changed in _update, and the updated value is being used in _draw()

The PICO-8 Program Structure

- `_update()` is where the value of `y` is changed
- Whatever the current value of `y` is, that's the value being fed into the `spr()` function in `_draw()` - *this is what creates motion*

```
FUNCTION _UPDATE()
    y=y+2
END

FUNCTION _DRAW()
    CLS()
    SPR(n,x,y)
END
```

More about variables . . .

Using Variables

- Variables must be “declared” before they can be used
- When we type **X=60** in **_INIT()**, we are declaring a variable named **X** and assigning it a value of **60**

```
FUNCTION _INIT()
```

```
n=1
```

```
x=60
```

```
y=2
```

```
END
```

```
FUNCTION _UPDATE()
```

```
y=y+2
```

```
END
```

```
FUNCTION _DRAW()
```

```
CLS()
```

```
SPR(n,x,y)
```

```
END
```

Using Variables

- If I fail to declare the variable **Y**, and then try to change its value, I will get an **error**

```
FUNCTION _INIT()
    n=1
    x=60
END

FUNCTION _UPDATE()
    y=y+2
END
```

```
RUNTIME ERROR LINE 7 TAB 0
y=y+2
ATTEMPT TO PERFORM ARITHMETIC ON
GLOBAL 'y' (A NIL VALUE)
IN _UPDATE LINE 7 (TAB 0)
AT LINE 0 (TAB 0)
```

Using Variables

If you ever see a reference to “A NIL VALUE” in your error message, *it’s likely that you did not define the variable (or function) before trying to do something with it*

```
RUNTIME ERROR LINE 7 TAB 0
  Y=Y+2
ATTEMPT TO PERFORM ARITHMETIC ON
  GLOBAL `Y` (A NIL VALUE)
IN _UPDATE LINE 7 (TAB 0)
AT LINE 0 (TAB 0)
```

```
FUNCTION _INIT()
    n=1
    x=60
```

```
END
```

```
FUNCTION _UPDATE()
```

```
    y=y+2
END
```

This is also a common error if you **change** a variable **name** in one place but not another

Troubleshooting Errors

Most runtime errors are caused by simple **typos**; if you get an error, check:

- Are there any unclosed parentheses?
- Are there any missing **END** keywords?
- Are all your **variables** and **function names** spelled correctly and consistently?

Variables and Input

Download Example File: [paddleball_basic_04_input_and_movement.p8](#)

Variables & Input

- We can use variables to make the paddle move when the player inputs a keypress
- We'll do this inside the `_update()` function

```
FUNCTION _UPDATE()
IF BTn(0) THEN
    PAD_X = PAD_X - 1
END
END
```

FUNCTION _INIT()

```
-- BALL  
BAL_h=1  
BAL_x=60  
BAL_y=2  
  
-- PADDLE  
PAD_h=2  
PAD_x=60  
PAD_y=118  
  
END
```

1. We can set variables for both the ball and paddle in _INIT()
2. We can then use the BTN() function to check for a button press in _UPDATE()

```
FUNCTION _UPDATE()  
IF BTN(0) THEN  
    PAD_x = PAD_x - 1  
END  
END
```

3. Finally, apply those variables in the SPR() function in _DRAW()

```
FUNCTION _DRAW()  
CLS()  
SPR(BAL_h,BAL_x,BAL_y)  
SPR(PAD_h,PAD_x,PAD_y)  
END
```

Some **rules** for
naming variables . . .

FUNCTION _INIT()

```
-- BALL
BAL_h=1
BAL_x=60
BAL_y=2

-- PADDLE
PAD_h=2
PAD_x=60
PAD_y=118

END
```

There are a few rules when naming variables:

- The variable name *cannot be another recognized word like spr or _draw*
- The variable name must be **one word** and **cannot have spaces**
- Underscores **ARE** acceptable, so use these instead of spaces, or just make the variable name one unbroken word

PICO-8 allows you to use **symbols** as variable names

But many other languages require that the variable name:

- *must NOT contain symbols*
- *must NOT begin with a number*

Detecting Input

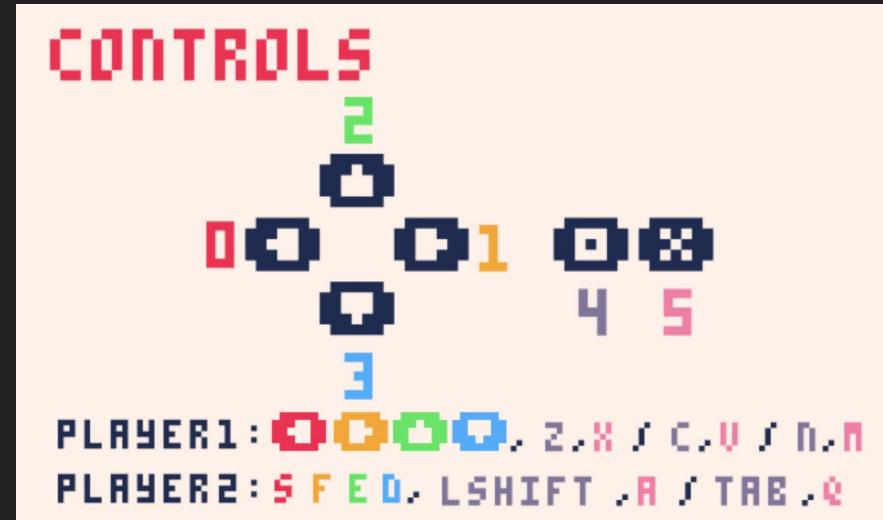
Download Example File: [paddleball_basic_04_input_and_movement.p8](#)

- Because the **_UPDATE()** function is used for **calculations**, this is where we will move our paddle sprite when a key is pressed
- We can use the keywords **IF** and **THEN** to create a “conditional statement”
- If the condition following **IF** is met, the code written after **THEN** will be executed

```
FUNCTION _UPDATE()
  IF BTn(0) THEN
    PAD_X = PAD_X - 1
  END
END
```

- The **BTn()** function checks whether a particular key was pressed
 - You’ll specify *which* key you’re checking for inside the parentheses
- Type **SHIFT L** to create a *left arrow icon* to represent the **left arrow key**

- You can type SHIFT and the first letter of a directional key to produce a symbol for that arrow
 - SHIFT L = left arrow
 - SHIFT R = right arrow
 - SHIFT U = up arrow
 - SHIFT D = down arrow
 - SHIFT O = O key (the letter O)
 - SHIFT X = X key
- Each of these arrow keys also has a numeric code associated with it
 - 0 = left arrow
 - 1 = right arrow
 - 2 = up arrow
 - 3 = down arrow
 - 4 = Z / C
 - 5 = X

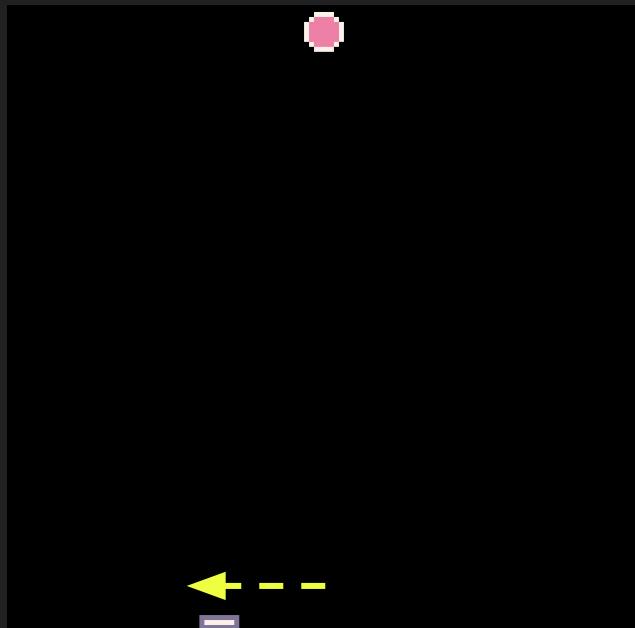


Refer to the [PICO-8 Cheat Sheet](#) for key codes

```
FUNCTION _UPDATE()
IF BTN(0) THEN
  PAD_X = PAD_X - 1
END
END
```

- To change the paddle's x position, we can take the variable PAD_X (or whatever you named your variable) and add or subtract from its value
- To move left, we'll subtract 1 from the value of PAD_X

```
FUNCTION _UPDATE()
IF BTn(0) THEN
    PAD_X = PAD_X - 1
END
END
```



- We can shorten this by using a minus sign followed by an equals sign (`-=`) to represent the same operation
- And we can use `+=` to do the same thing but add to the value instead of subtract
- This process is called incrementing (increasing) or decrementing (decreasing)

```
FUNCTION _UPDATE()
IF BTn(0) THEN
    PRO_X = PRO_X - 1
END
END
```

```
FUNCTION _UPDATE()
IF BTn(0) THEN
    PRO_X -= 1
END
IF BTn(0) THEN
    PRO_X += 1
END
END
```

```
PAO_X = PAO_X - 1
```

Each expression means the **same thing**
(subtract one from the variable's value)

```
PAO_X -= 1
```

“Closing” functions
and if-statements

- Whenever we define a function using the keyword **FUNCTION**, we need to “close” the function by typing the keyword **END** at the end of the block of code
- A “block” is just several related lines of code

```
FUNCTION _UPDATE()
IF BTn(0) THEN
    PRO_X = PRO_X - 1
END
END
```

- We also need to “close” our **IF/THEN** blocks using the keyword **END**

```
FUNCTION _UPDATE()
IF BTn(0) THEN
    PRO_X = PRO_X - 1
END
END
```

*Some languages, such as JavaScript, use **curly brackets { }** instead of **END***

If you don't close a function or if-statement, it can result in an error or unexpected behavior

- Notice how the IF/THEN block fits “between” the FUNCTION/END block
- We call this “nesting” (like the Russian nesting dolls)
- It’s good practice to indent any nested lines of code

This way, we can see which ENDS are attached to which FUNCTIONS, IFs, etc.

```
FUNCTION _UPDATE()
| IF BTn(0) THEN
| | PRO_X = PRO_X - 1
| END
END
```



- Visual Studio Code displays a vertical line connecting the beginnings and ends of each block, which can help you visualize the nesting structure

```
function _update()
  if btn(⬅️) then
    pad_x = pad_x - 1
  end
end
```

Make sure you use the **tab** key to create an indentation (spaces may not produce this effect)



- If you get an **error** when you run your game, often it is due to an **unclosed block**
- I prefer to use VS Code to troubleshoot because it more clearly illustrates the beginnings and ends of each block

```
function _update()
    if btn(⬅️) then
        pad_x = pad_x - 1
    end
end
```

Make sure you use the **tab** key to create an indentation (spaces may not produce this effect)



- In VS Code, make sure you use the **tab** key to create an indentation (spaces may not produce this effect)
- *Also, indentation between VS Code and the PICO-8 editor may not be the same - choose one editor and stick with it*

You can also use **CTRL/CMD]** to indent a selected block and **CTRL/CMD [** to outdent (move left)

```
function _update()
    if btn(⬅️) then
        pad_x = pad_x - 1
    end
end
```



- Sometimes, I'll even go so far as to add comments after each END statement to remind myself what they are closing
- Write a comment in PICO-8 using two dashes (the minus-sign key) --

```
function _update()

    -- move left
    if btn(←) then
        pad_x -= pad_speed
    end -- end if btn(←)

    -- move right
    if btn(→) then
        pad_x += pad_speed
    end -- end if btn(→)

end -- end function _update()
```

VS Code is also helpful in that it highlights the end statements that close each block



- **Comments are an underrated aspect of programming – use them as much as you want to provide reminders for yourself; you'll forget why you wrote something the way you did after some time has passed!**

```
function _update()

    -- move left
    if btn(←) then
        pad_x -= pad_speed
    end -- end if btn(←)

    -- move right
    if btn(→) then
        pad_x += pad_speed
    end -- end if btn(→)

end -- end function _update()
```

Comments also help other people understand your code



- You can also use comments to “deactivate” lines of code without permanently deleting them
- This comment will stop the game from drawing the ball
- You can remove the comment to “reactivate” the code

```
function _draw()
    cls()
    --spr(bal_n, bal_x, bal_y)
    spr(pad_n, pad_x, pad_y)
end
```



Our full program so far . . .

Variables & Input

This code:

- Defines variables for the sprite number and x,y position of both the ball and paddle
- Changes the value of the paddle's x coordinate when the left or right arrow keys are pressed
- Draws the ball and paddle at whatever value their x and y coordinates are

```
1 pico-8 cartridge // http://www.pico-8.com
2 version 42
3 lua
4 function _init()
5     -- ball
6     bal_n=1
7     bal_x=60
8     bal_y=2
9
10    -- paddle
11    pad_n=2
12    pad_x=60
13    pad_y=118
14 end
15
```

```
16 function _update()
17     -- move left
18     if btn(←) then
19         pad_x -= 1
20     end
21
22     -- move right
23     if btn(→) then
24         pad_x += 1
25     end
26 end
27
```

```
28 function _draw()
29     cls()
30     spr(bal_n,bal_x,bal_y)
31     spr(pad_n,pad_x,pad_y)
32 end
```

Variables & Input

- The paddle is probably moving too slowly to catch up to the ball in all instances
- We could simply change `+= 1` and `-= 1` to `+= 3` and `-= 3`
- But what if we need to keep tweaking this value? It would be more convenient to only have to change it in one place instead of two
- In cases like this, it makes sense to create a new variable – we can call it SPEED

```
1 pico-8 cartridge // http://www.pico-8.com
2 version 42
3 _lua_
4 function _init()
5     -- ball
6     bal_n=1
7     bal_x=60
8     bal_y=2
9
10    -- paddle
11    pad_n=2
12    pad_x=60
13    pad_y=118
14 end
15
16 function _update()
17     -- move left
18     if btn(←) then
19         pad_x -= 1
20     end
21
22     -- move right
23     if btn(→) then
24         pad_x += 1
25     end
26 end
27
28 function _draw()
29     cls()
30     spr(bal_n,bal_x,bal_y)
31     spr(pad_n,pad_x,pad_y)
32 end
```

Variables & Input

- Here, I created a variable in `_init()` called `pad_speed` (for the paddle's speed) and set it equal to 3
- Then I applied that variable inside `_update()` in both the left and right movement blocks
- If I decide 2, 4, or some other value will work better, I only need to change the variable value once, in `_init()`

```
function _init()
    -- ball
    bal_n=1
    bal_x=60
    bal_y=2

    -- paddle
    pad_n=2
    pad_x=60
    pad_y=118
    pad_speed=3
end

function _update()
    -- move left
    if btn(⬅️) then
        pad_x -= pad_speed
    end

    -- move right
    if btn(➡️) then
        pad_x += pad_speed
    end
end
```

Variables & Input

- When you assign a variable a value and then refer to that variable later in your code, you can *think of that variable as standing for the value given to it*
- Here, **pad_speed** means the same as 3 because I assigned pad_speed the value of 3

```
function _init()
    -- ball
    bal_n=1
    bal_x=60
    bal_y=2

    -- paddle
    pad_n=2
    pad_x=60
    pad_y=118
    pad_speed=3
end

function _update()
    -- move left
    if btn(<) then
        pad_x -= pad_speed
    end

    -- move right
    if btn(>) then
        pad_x += pad_speed
    end
end
```

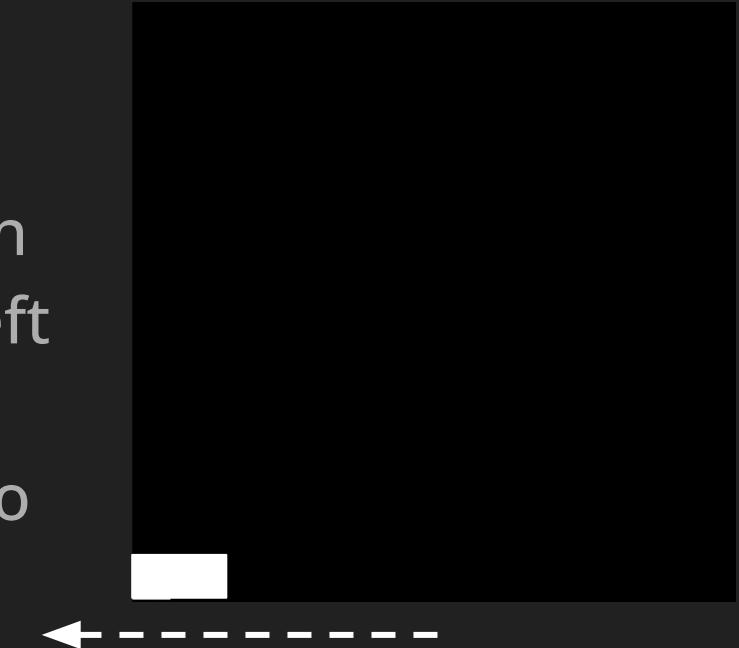
The diagram illustrates the concept of variables representing values. It shows two code snippets: `_init()` and `_update()`. In the `_init()` function, `pad_speed` is assigned the value `3`. In the `_update()` function, there are two conditional statements: one for moving left and one for moving right. Both conditions involve subtracting or adding `pad_speed` from `pad_x` respectively. Dashed yellow arrows point from the explanatory text below to the assignment statement in `_init()` and to the subtraction and addition statements in `_update()`, visually connecting the variable to its definition and its use in the program logic.

Keeping the Paddle on Screen

Download Example File: [paddleball_basic_05_conditional_logic.p8](#)

Keeping the Paddle on Screen

- You may have noticed that the player can continue moving the paddle with the arrow keys, even after it's gone *off screen* to the left or right
- We can use if/then statements to determine when the paddle has gone off screen and then correct its position



Keeping the Paddle on Screen

- *What do we **know** about the paddle when it's off screen?*
- We know its **x position must be less than 0**, since the left edge of the screen is at 0
- So we know we only need to adjust its position if its **x is less than 0**



Keeping the Paddle on Screen

- We know we only need to adjust the paddle's position if:
its **x** is **less than 0**
- We can express this condition in the code as:
if padx < 0

```
IF PADX < 0 THEN  
END
```



Keeping the Paddle on Screen

- If the condition **padx < 0** is met, we need to adjust the paddle's position to keep it on screen
- We can simply **reset padx to 0** so that it never gets less than that
- This is called "*constraining*" a value

```
IF PADX < 0 THEN  
    PADX = 0  
END
```



Keeping the Paddle on Screen

- We'll place this code in our `_update()` function, **AFTER** the code that moves the paddle left when the key is pressed

```
-- LEFT ARROW MOVES PAD LEFT
-- TYPE SHIFT L FOR ⌘
IF BTN(0) THEN
    PADX = PADX - PADSPD
END

-- RIGHT ARROW MOVES PAD RIGHT
-- TYPE SHIFT R FOR ⌘
IF BTN(1) THEN
    PADX = PADX + PADSPD
END

-- KEEP PAD ON SCREEN LEFT
IF PADX < 0 THEN
    PADX = 0
END
```

Keeping the Paddle on Screen

- If **padx** becomes less than 0 here . . .



```
-- LEFT ARROW MOVES PAD LEFT
-- TYPE SHIFT L FOR ☐
IF BTN(☐) THEN
    PADX = PADX - PADSPD
END

-- RIGHT ARROW MOVES PAD RIGHT
-- TYPE SHIFT R FOR ☐
IF BTN(☐) THEN
    PADX = PADX + PADSPD
END

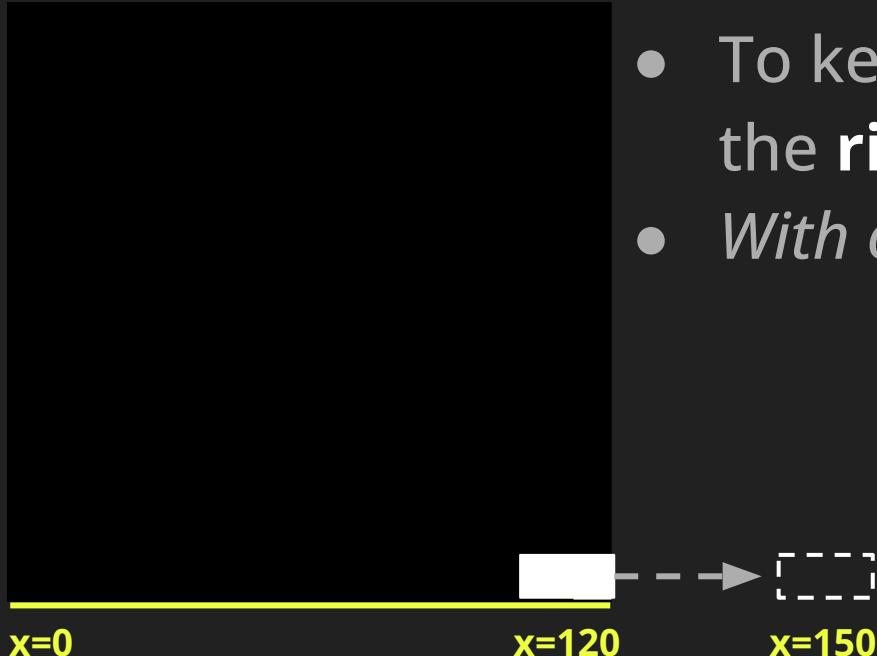
-- KEEP PAD ON SCREEN LEFT ☐
IF PADX < 0 THEN
    PADX = 0
END
```

- It gets **reset back to 0** here



Keeping the Paddle on Screen

- To keep the paddle on screen to the **right**, it's a similar process
- *With one difference . . .*



Keeping the Paddle on Screen

*Because sprites are drawn from **left** to **right**, when **x** is **128**, the paddle will be offscreen (**x** is the **left** side of the paddle)*

x=0



x=128

*This means the last pixel where the paddle is still fully on screen will be **one paddle-width to the left of the screen edge***

x=0



x=120

Keeping the Paddle on Screen

- 128 is the right edge of the screen
- If the paddle is **8px wide**, we subtract 8 from 128 to get the last point where the paddle will be fully on screen
- $128 - 8 = \mathbf{120}$

*The last pixel where the paddle is still fully on screen will be **one paddle-width to the left of the screen edge***



Keeping the Paddle on Screen

- This is our full code for constraining the paddle to the screen bounds

```
-- KEEP PAD ON SCREEN LEFT
IF PADX < 0 THEN
  PADX = 0
END

-- KEEP PAD ON SCREEN RIGHT
IF PADX > 120 THEN
  PADX = 120
END
```

Keeping the Paddle on Screen

- An even better solution would be to use a **variable** for the paddle's width, so that the calculation still works if the paddle's size changes due to a bonus item or some other factor

```
-- KEEP PAD ON SCREEN LEFT
IF PADX < 0 THEN
    PADX = 0
END

-- KEEP PAD ON SCREEN RIGHT
IF PADX > 128-PADW THEN
    PADX = 128-PADW
END
```

Keeping the Paddle on Screen

- An even better solution would be to use a **variable** for the paddle's width, so that the calculation still works if the paddle's size changes due to a bonus item or some other factor

```
-- RUNS ONCE AT START
FUNCTION _INIT()
    -- DECLARE VARIABLES FOR PADDLE
    PADN = 1 -- SPRITE NUMBER
    PADX = 60 -- X COORDINATE
    PADY = 118 -- Y COORDINATE
    PADW = 8 -- WIDTH
    PADSPD = 3 -- SPEED
```

We would just need to declare this variable first, in _init()

```
-- KEEP PAD ON SCREEN LEFT
IF PADX < 0 THEN
    PADX = 0
END

-- KEEP PAD ON SCREEN RIGHT
IF PADX > 128-PADW THEN
    PADX = 128-PADW
END
```

Next, we'll work
more with
if-statements and
start creating our
own **functions**

Writing Custom Functions

Download Example File: [paddleball_basic_06_functions.p8](#)

Writing Custom Functions

- We can write our *own* functions to do whatever we want – use the function keyword, followed by a name and parentheses

```
FUNCTION MAKE_PLAYER()
END
```

Functions need to be “*closed*” by including the keyword end at the end

Writing Custom Functions

- We can write a custom function to define all our paddle variables

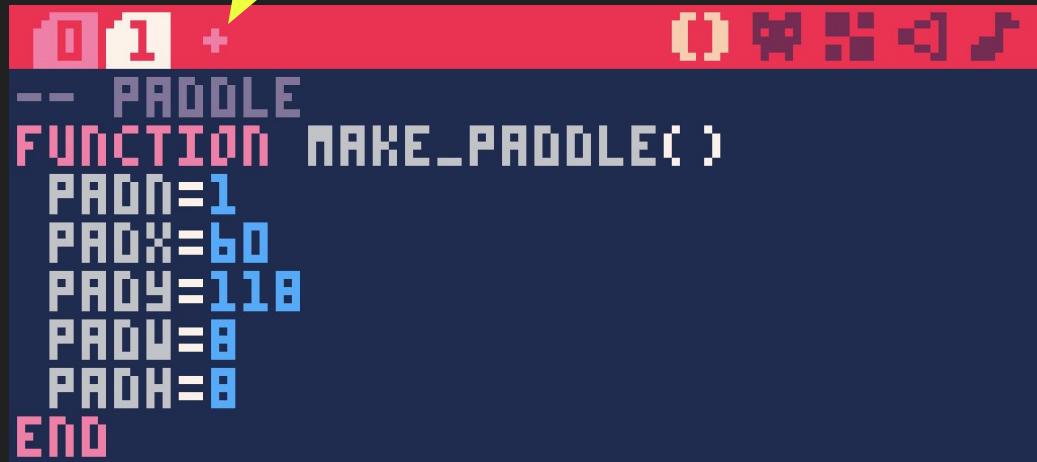
```
FUNCTION MAKE_PADDLE()
PADn=1
PADx=60
PADy=110
PADv=0
PADh=0
END
```

I named mine `make_paddle()`, but you can name yours whatever you want, as long as there are **no spaces** in the name (I used an underscore `_` to connect compound words) and the function name begins with a **letter**

Writing Custom Functions

- I can write this custom function on a separate tab in the PICO-8 editor to save space on my main tab (0)

Click the + icon to add a tab



The screenshot shows the PICO-8 editor interface. At the top, there are two tabs labeled '0' and '1'. A yellow arrow points to the '+' icon between the tabs, indicating where to click to add a new tab. Below the tabs is a toolbar with various icons. The main workspace is dark blue and contains the following code:

```
-- PADDLE
FUNCTION MAKE_PADDLE()
PADH=1
PADX=60
PADY=118
PADU=8
PADL=8
END
```

Writing Custom Functions

- *This code will not execute on its own*
- The function needs to be “called” elsewhere in the program

```
FUNCTION MAKE_PADDLE()
PADH=1
PADX=60
PADY=110
PADU=8
PADH=8
END
```

You might also say the function must be “run,” “triggered,” or “executed” – calling a function means the same thing

Writing Custom Functions

- Think of a function like an item in your inventory
- It doesn't do anything until you tell it to

```
FUNCTION MAKE_PADDLE()
PADH=1
PADX=60
PADY=118
PADU=8
PADH=8
END
```



Writing Custom Functions

- We need to “call” our function in either `_init()`, `_update()`, or `_draw()`
- Here, I “call” `make_paddle()` in `_init()` since that’s where I declare variables



```
FUNCTION _INIT()
    MAKE_PADDLE()
END

FUNCTION MAKE_PADDLE()
    PADL=1
    PADX=60
    PADY=118
    PADU=8
    PADH=8
END
```

Writing Custom Functions

- The code inside a function runs right when/where you call it
- Think of it as a shortcut or portal

A Scratch script illustrating the concept of functions. At the top, a blue hat block labeled "FUNCTION _INIT()" contains the blocks "MAKE_PADDLE()", "END", and a yellow arrow pointing to the start of another function definition below. Below this, a blue hat block labeled "FUNCTION MAKE_PADDLE()" contains five white blocks: "PADDL = 1", "PADDX = 60", "PADDY = 110", "PADDU = 8", and "PADDH = 8". At the bottom, a blue hat block labeled "END" contains a yellow arrow pointing to the end of the function definition.

```
FUNCTION _INIT()
  MAKE_PADDLE()
END ↑

FUNCTION MAKE_PADDLE()
  PADDL = 1
  PADDX = 60
  PADDY = 110
  PADDU = 8
  PADDH = 8
END ↓
```

Writing Custom Functions

```
FUNCTION _INIT()
PADON=1
PADX=60
PADY=118
PADU=8
PADH=8
END
```

```
= FUNCTION _INIT()
    MAKE_PADDLE()
END
FUNCTION MAKE_PADDLE()
PADON=1
PADX=60
PADY=118
PADU=8
PADH=8
END
```

Instead of declaring my variables directly in `_init()`, I can do that in a custom function and then “call” that function in `_init()`

Writing Custom Functions

```
FUNCTION _INIT()
PADH=1
PADX=60
PADY=118
PADU=8
PADH=8
END
```

=

```
FUNCTION _INIT()
MAKE_PADDLE()
END

FUNCTION MAKE_PADDLE()
PADH=1
PADX=60
PADY=118
PADU=8
PADH=8
END
```

The code on the left will do the same thing as the code on the right; the only difference is how it's organized

This may seem like a lot
of work for something
that doesn't change the
program's behavior ...

*but **the result is more
readable code***

```
1 pico-8 cartridge // http://www.pico-8.com
2 version 42
3 __lua__          With custom functions
4 function _init()
5     make_paddle()
6     make_ball()
7 end
8
9 function _update()
10    move_paddle()
11    move_ball()
12 end
13
14 function _draw()
15    cls()
16    draw_paddle()
17    draw_ball()
18 end
19 -->8
20 -- functions defined on other tabs below
```

```
1 pico-8 cartridge // http://www.pico-8.com
2 version 42
3 __lua__          Without custom
4 function _init()
5     pad_n = 1
6     pad_x = 60
7     pad_y = 118
8     pad_spd = 3
9
10    bal_n = 2
11    bal_x = 60
12    bal_y = 2
13    bal_spd = 3
14 end
15
16 function _update()
17    -- left arrow
18    if btn(0) then
19        pad_x -= pad_spd
20    end -- end if btn(0)
21
22    -- right arrow
23    if btn(1) then
24        pad_x += pad_spd
25    end -- end if btn(1)
26
27    -- keep on screen left
28    if pad_x < 0 then
29        pad_x = 0
30    end -- end if paddle.x < 0
```

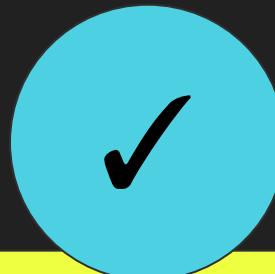
Next, let's *detect when the ball and paddle collide*, so we can *make the ball change direction* when this happens

Collision Detection

Download Example File: [paddleball_basic_07_collision.p8](#)

Collision Detection

- *Collision detection means determining when two objects in your game are touching*



Ball and paddle are colliding



Ball and paddle are NOT colliding

Collision Detection

- Collision detection is a bit more involved than anything we've done so far
- Many game engines will do collision detection for you
- But it helps to understand how it works

Collision Detection

- Basically, we want to compare the position of one object (the paddle) with the position of another object (the ball)



Ball and paddle are colliding



Ball and paddle are NOT colliding

Collision Detection

- We can use the ball's and the paddle's **x** and **y** values to compare their positions

Collision Detection

- We can use the ball's and the paddle's **x** and **y** values to compare their positions
- We'll also need to account for their width and height, so ***we'll need variables for the width and height***

Collision Detection

- I've added **variables** for the paddle's and ball's **width** and **height** (**PADW**, **PADH**, **BALW**, **BALH**)
- The values are equal to the sprite dimensions*

```
0 1 2 3 + 0 1 2 3 +
-- DECLARE VARIABLES FOR PADDLE
FUNCTION MAKE_PADDLE()
PADN = 1 -- SPRITE NUMBER
PADX = 60 -- X COORDINATE
PADY = 118 -- Y COORDINATE
PADSPD = 3 -- SPEED
PADW = 8 -- WIDTH
PADH = 2 -- HEIGHT
END

-- DECLARE VARIABLES FOR BALL
FUNCTION MAKE_BALL()
BALN = 2 -- SPRITE NUMBER
BALX = 60 -- X COORDINATE
BALY = 2 -- Y COORDINATE
BALSPD = 3 -- SPEED
BALW = 8 -- WIDTH
BALH = 8 -- HEIGHT
END
```

Collision Detection

- Next, we'll write code to determine whether the ball and paddle are touching
- I'll create a new function called **move_ball()** and write the code there, because we'll want the ball to move in a different direction when it touches the paddle

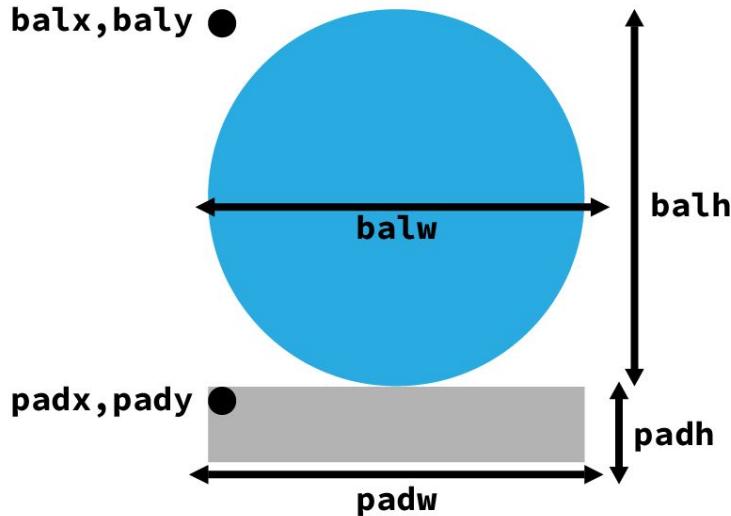
```
0 1 2 3 +      0 0 0 < > 
-- MOVE BALL AND COLLIDE
FUNCTION MOVE_BALL()
    -- COLLIDE WITH PADDLE
    IF BALX + BALW >= PADX
    AND BALX <= PADX + PADW
    AND BALY + BALH >= PADY
    AND BALY <= PADY + PADH
    THEN
        BALY = BALY + 0 -- STOP MOVING
    ELSE
        -- ONLY MOVE IF NOT COLLIDING
        BALY = BALY + BALSPD
    END
END -- END FUNCTION MOVE_BALL()
```

Collision Detection

Basically, we *compare the position of opposite sides of each object against one another*

```
0 1 2 3 +      0 9 8 < > 
-- MOVE BALL AND COLLIDE
FUNCTION MOVE_BALL()
    -- COLLIDE WITH PADDLE
    IF BALX + BALW >= PADX
    AND BALX <= PADX + PADW
    AND BALY + BALH >= PADY
    AND BALY <= PADY + PADH
    THEN
        BALY = BALY + 0 -- STOP MOVING
    ELSE
        -- ONLY MOVE IF NOT COLLIDING
        BALY = BALY + BALSPD
    END
END -- END FUNCTION MOVE_BALL()
```

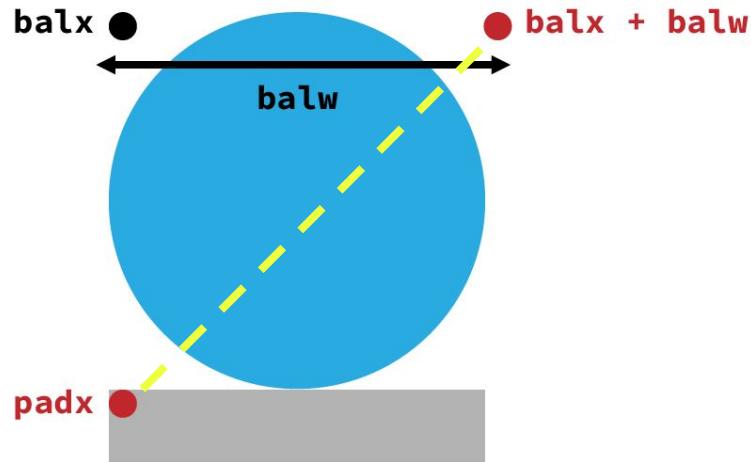
```
if balx + balw >= padx  
and balx <= padx + padw  
and baly + balh >= pady  
and baly <= pady + padh  
then there's collision!
```



We'll need **multiple conditions** to be met in order for the objects to be considered colliding

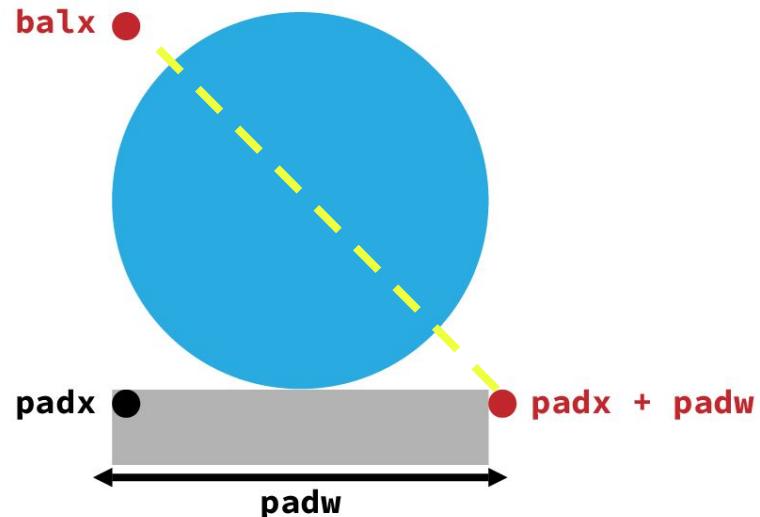
Let's break this down, one condition at a time . . .

balx + balw >= padx



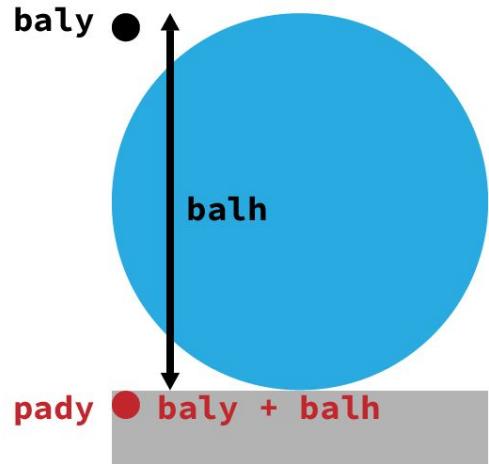
The right side of the ball (*measured as **balx + balw***) must have an **x** value greater than or equal to (\geq) the left side of the paddle (*measured as **padx***)

balx <= padx + padw



The left side of the ball (*measured as balx*) must have an **x** value **less than or equal to** (\leq) the right side of the paddle (*measured as padx + padw*)

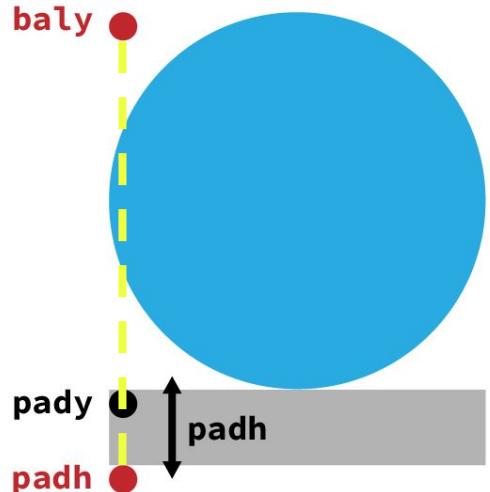
baly + balh \geq pady



The bottom of the ball (*measured as baly + balh*) must have a **y** value greater than or equal to (\geq) the top of the paddle (*measured as pady*)

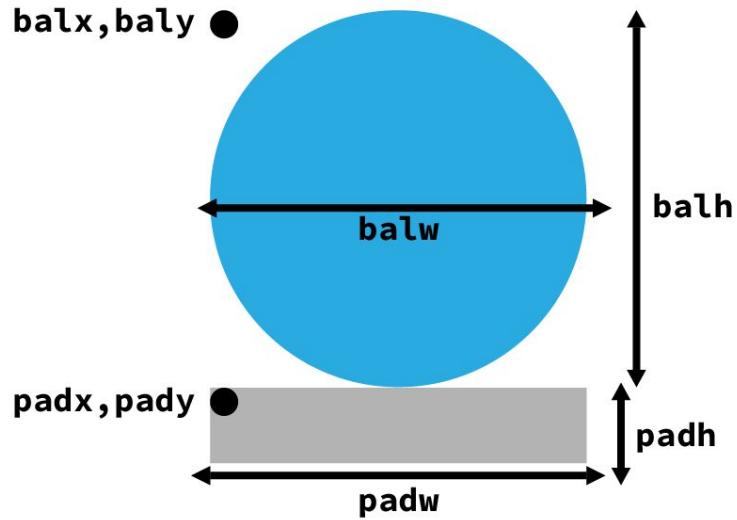
Here, they're equal

baly <= pady + padh



The top of the ball
(*measured as baly*)
must have a **y** value
less than or equal to
(\geq) the **bottom** of
the paddle
(*measured as pady +*
padh)

```
if balx + balw >= padx  
and balx <= padx + padw  
and baly + balh >= pady  
and baly <= pady + padh  
then there's collision!
```



If all four conditions are met, the ball and paddle must be touching (colliding)

Let's look at this a
different way . . .

Collision Detection

balx + balw >= padx

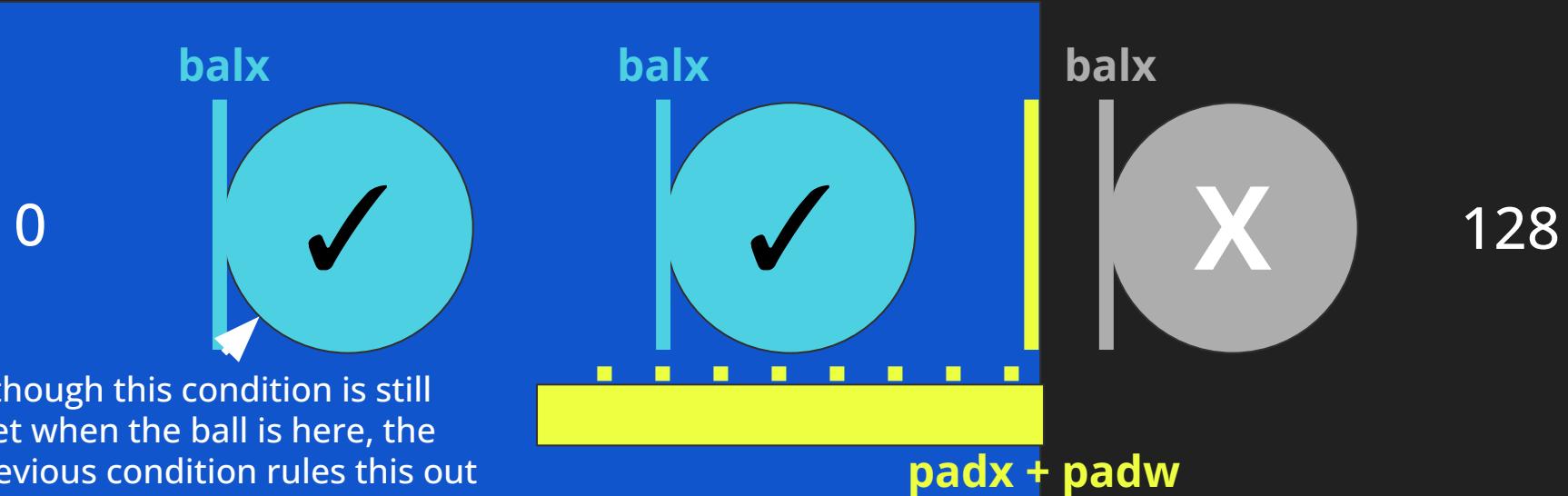
The right side of the ball ($balx + balw$) must be at or to the right of (\geq) the left side of the paddle ($padx$)



Collision Detection

$$\text{balx} \leq \text{padx} + \text{padw}$$

The left side of the ball (`balx`) must be at or to the left of (\leq) the right side of the paddle (`padx + padw`)



Collision Detection

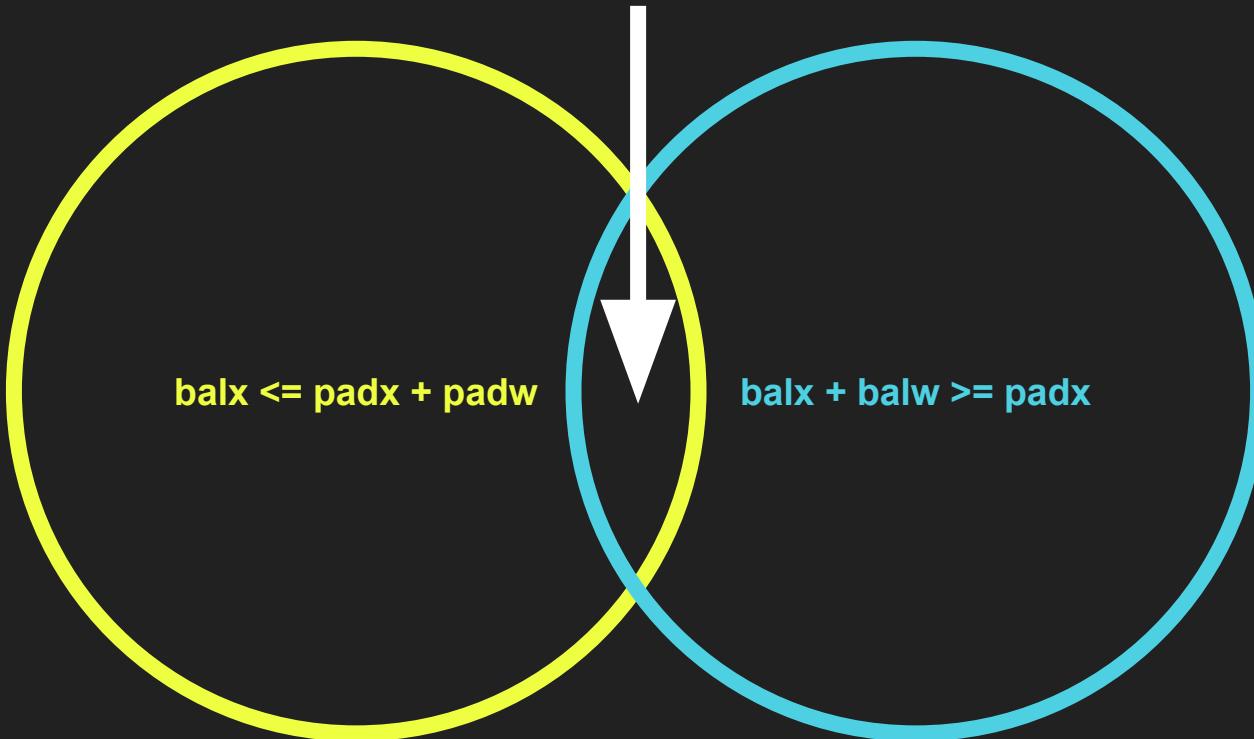
$\text{balx} \leq \text{padx} + \text{padw}$

If both conditions are met, the ball is here

$\text{balx} + \text{balw} \geq \text{padx}$



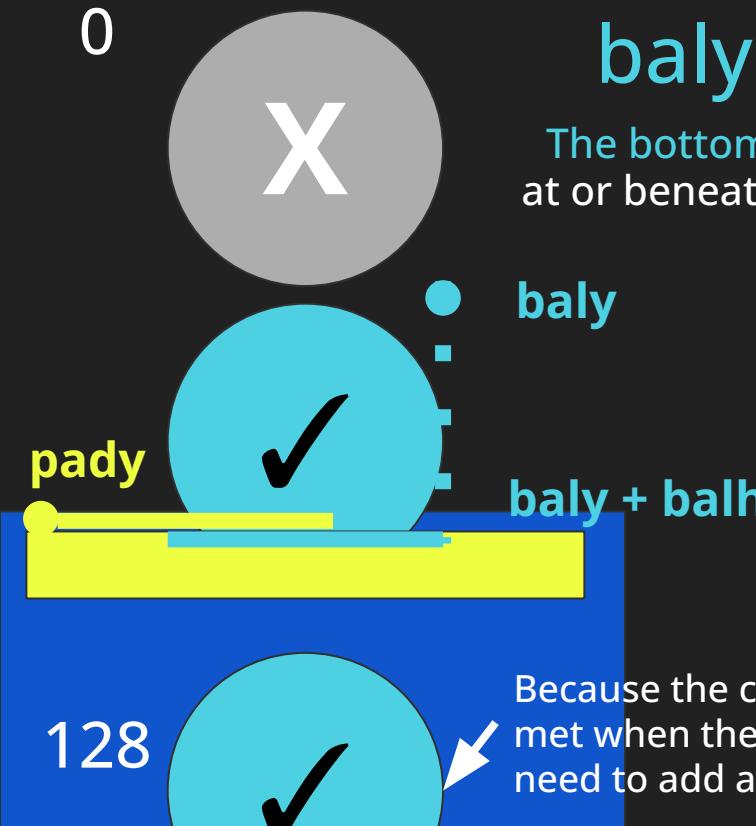
If both conditions are met, the ball is horizontally aligned with the paddle



Venn diagram

Collision Detection

0



$$\text{baly} + \text{balh} \geq \text{pady}$$

The bottom of the ball (**baly + balh**) must be at or beneath (\geq) the top of the paddle (**pady**)

Because the condition is still met when the ball is here, we'll need to add another condition

Collision Detection

0



$baly <= pady + padh$

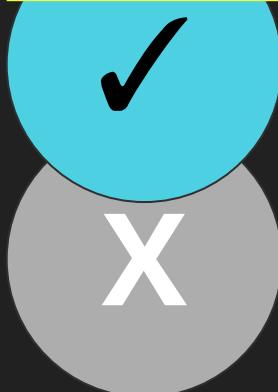
The top of the ball ($baly$) must be at or above (\leq)
the bottom of the paddle ($pady + padh$)

Although this condition is still
met when the ball is here, the
previous condition rules this out

$baly$

$pady + padh$

128

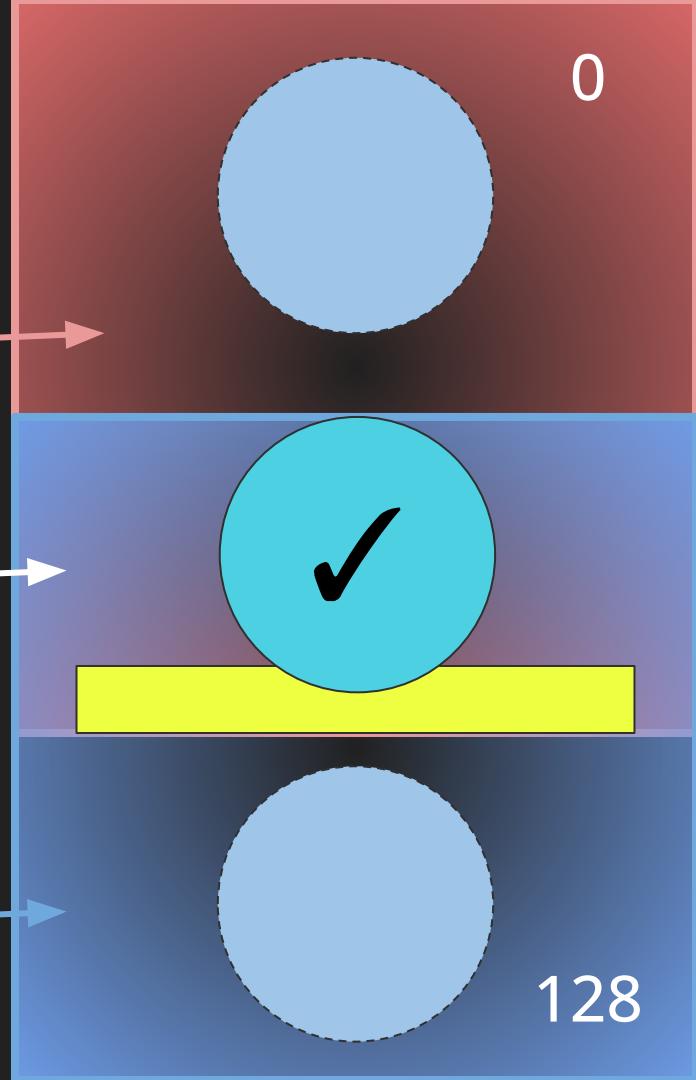


Collision Detection

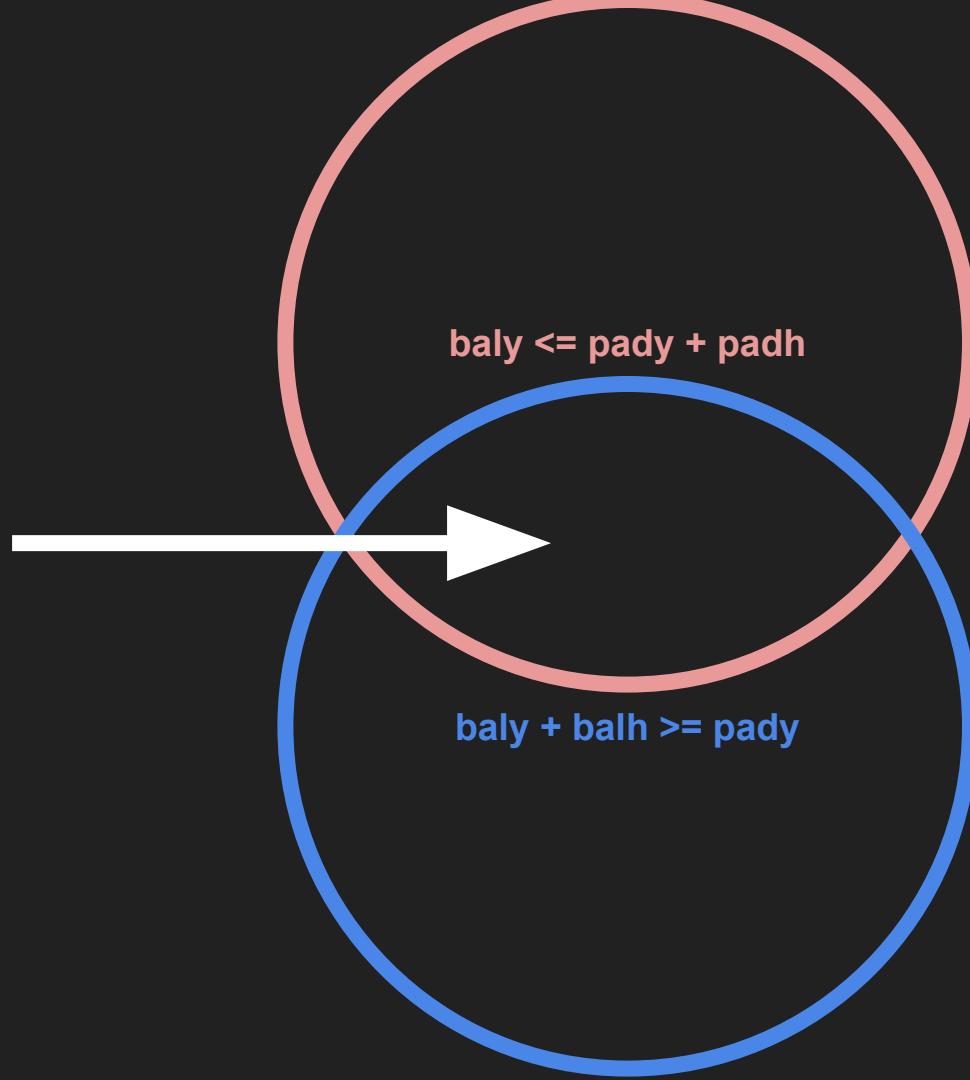
$baly \leq pady + padh$

If both conditions are met, the ball is here

$baly + balh \geq pady$



If both conditions are met, the ball is vertically aligned with the paddle



Venn diagram

balx <= padx + padw

baly <= pady + padh

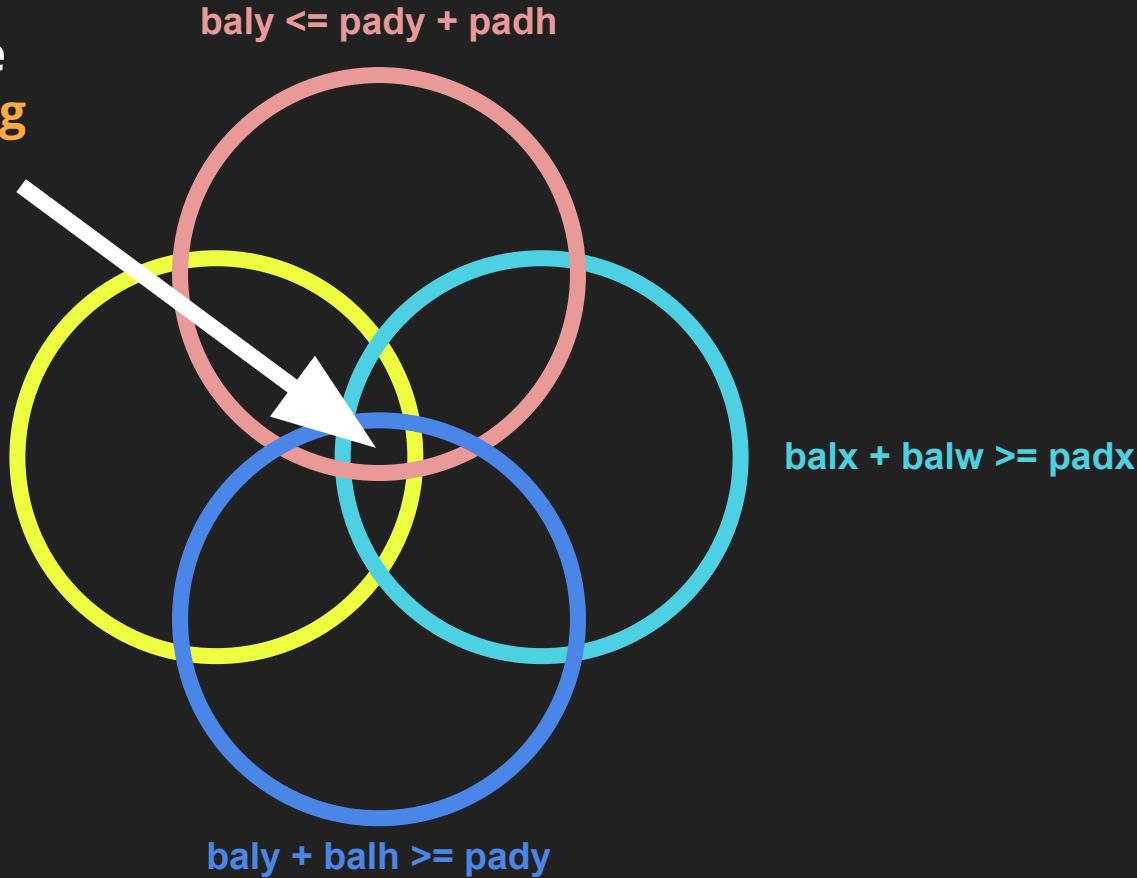
balx + balw >= padx

If ALL conditions are met, the ball is here



baly + balh >= pady

If **ALL 4** conditions are met, the ball is **colliding** with the paddle



Venn diagram

Collision Detection

If all four conditions are met, the ball is in collision with the paddle



The screenshot shows a BASIC interpreter interface with the following code:

```
0 1 2 3 +      0 9 8 < > 
-- MOVE BALL AND COLLIDE
FUNCTION MOVE_BALL()
    -- COLLIDE WITH PADDLE
    IF BALX + BALW >= PROX
    AND BALX <= PROX + PROW
    AND BALY + BALH >= PROY
    AND BALY <= PROY + PROH
    THEN
        BALY = BALY + 0 -- STOP MOVING
    ELSE
        -- ONLY MOVE IF NOT COLLIDING
        BALY = BALY + BALSPD
    END
END -- END FUNCTION MOVE_BALL()
```

The condition for collision with the paddle is highlighted with a yellow box.

Collision Detection

We can use the keyword **AND** to stack conditions
(require multiple conditions to be met)



The image shows a Scratch script with the stage background set to "dark blue". At the top, there are two costumes: one with a pink gradient and another with a dark blue gradient. The script starts with a "when green flag clicked" hat. It then branches into two parallel loops: "repeat [ball] []" and "repeat [paddle] []". Inside the ball loop, it checks if the ball has hit the paddle ("if <[touching something?]) then"). If so, it changes the ball's x position ("set x to (-1) * (x - (x of [paddle v]))") and ends the paddle loop ("end"). If not, it moves the ball ("move (10 steps)" right). Inside the paddle loop, it checks if the paddle has hit the ball ("if <[touching something?]) then"). If so, it changes the paddle's y position ("set y to (-1) * (y - (y of [ball v]))") and ends the ball loop ("end"). If not, it moves the paddle ("move (10 steps)" up). Both loops end with a "end" block.

```
-- MOVE BALL AND COLLIDE
FUNCTION MOVE_BALL()
    -- COLLIDE WITH PADDLE
    IF BALX + BALW >= PADX
    AND BALX <= PADX + PADW
    AND BALY + BALH >= PADY
    AND BALY <= PADY + PADH
    THEN
        BALY = BALY + 0 -- STOP MOVING
    ELSE
        -- ONLY MOVE IF NOT COLLIDING
        BALY = BALY + BALSPD
    END
END -- END FUNCTION MOVE_BALL()
```

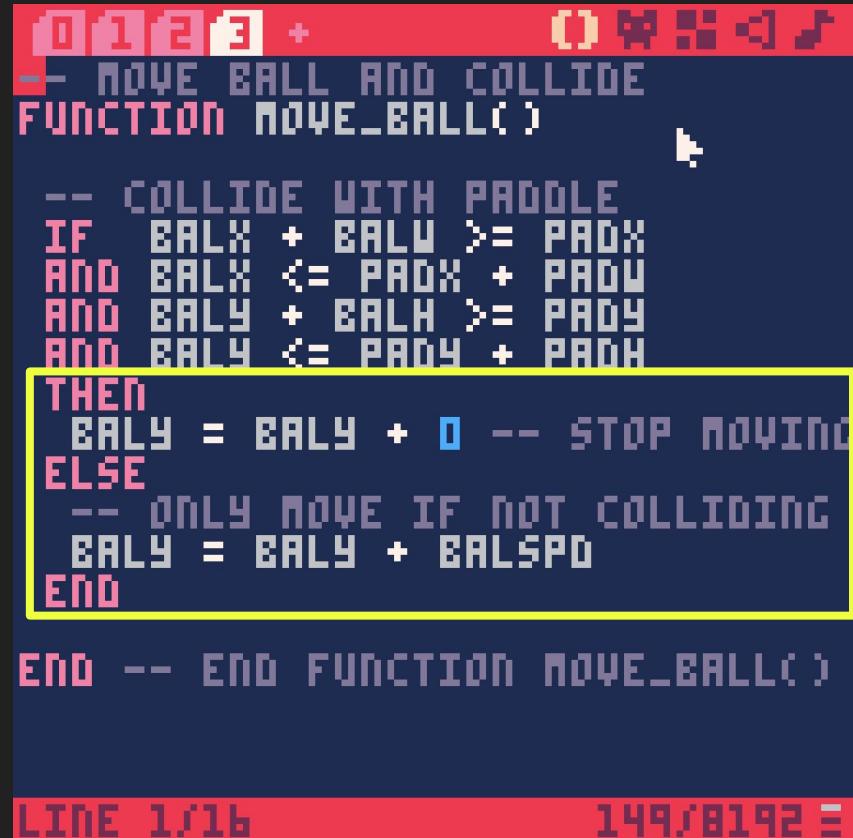
LINE 1/16 149/8192 ▶

What do we do if all
conditions are met
and the ball is
colliding with the
paddle?

Collision Detection

To test whether your code is detecting collision correctly, you could choose to only move the ball if there is NOT collision

You can use the keyword **ELSE** to specify what should happen if the *opposite* of the condition(s) occurs



The image shows a Scratch script titled "MOVE BALL AND COLLIDE". It contains a function named "MOVE_BALL()". Inside the function, there is a conditional loop that checks for collisions with the paddle. If a collision is detected (BALX + BALW >= PROX and BALX <= PROX + PROW and BALY + BALH >= PROY and BALY <= PROY + PROH), the ball's y-position is set to its current position plus zero, effectively stopping it from moving. If no collision is detected (BALX + BALW < PROX or BALX > PROX + PROW or BALY + BALH < PROY or BALY > PROY + PROH), the ball's y-position is increased by its speed ("BALSPD"). The script ends with an "END" block.

```
-- MOVE BALL AND COLLIDE
FUNCTION MOVE_BALL()
    -- COLLIDE WITH PADDLE
    IF BALX + BALW >= PROX
        AND BALX <= PROX + PROW
        AND BALY + BALH >= PROY
        AND BALY <= PROY + PROH
    THEN
        BALY = BALY + 0 -- STOP MOVING
    ELSE
        -- ONLY MOVE IF NOT COLLIDING
        BALY = BALY + BALSPD
    END
END -- END FUNCTION MOVE_BALL()
```

LINE 1/16 149/8192 ↻

Collision Detection

Another way to stop the ball would be to set its speed to zero, so that nothing is being added to the y coordinate value

```
BALY = BALY + BALSPD
```

```
-- COLLIDE WITH PADDLE
IF BALX + BALU >= PADX
AND BALX <= PADX + PADU
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
    BALSPD = 0
END
```

But what we really
want is for the ball to
bounce back up
when it hits the
paddle

Making the Ball Bounce

Download Example File: [paddleball_basic_08_reverse_direction.p8](#)

Making the Ball Bounce

I'm going to create a new variable for the ball's vertical (y-axis) direction (*is it moving up or down?*)

I'll name it **BALYDIR**

And I'll set its value to "**DOWN**"

You'll need to include quotation marks for a text value



```
0 1 2 3 +      ○ □ △ ▹ ▸
END

-- DECLARE VARIABLES FOR BALL
FUNCTION MAKE_BALL()
BALN = 2 -- SPRITE NUMBER
BALX = 60 -- X COORDINATE
BALY = 2 -- Y COORDINATE
BALSPD = 3 -- SPEED
BALW = 8 -- WIDTH
BALH = 8 -- HEIGHT

-- WE NEED VARIABLES TO TRACK
-- THE DIRECTION OF THE BALL
BALXDIR = "" -- HORIZONTAL
[BALYDIR = "DOWN" -- VERTICAL
END

LINE 1/26          175/8192 ⏴
```

The image shows a Scratch script with a red title bar at the top. The script begins with an 'END' block, followed by a function 'MAKE_BALL()'. Inside the function, variables are defined: 'BALN' (sprite number), 'BALX' (x coordinate), 'BALY' (y coordinate), 'BALSPD' (speed), 'BALW' (width), and 'BALH' (height). Below the function, comments indicate the need for variables to track ball direction ('BALXDIR' for horizontal and 'BALYDIR' for vertical). The 'BALYDIR' assignment is highlighted with a yellow box. The script concludes with an 'END' block. At the bottom, there is a status bar with 'LINE 1/26' and '175/8192 ⏴'.

Making the Ball Bounce

While I'm at it, I'll create a variable for the horizontal direction called **BALXDIR**

I'll leave its value as just empty quotes for now, since the ball just drops straight down at the start (not moving left/right yet)

The image shows a Scratch script on a dark blue background. At the top, there's a red banner with the numbers 0, 1, 2, 3, +, and a green banner with the letters O, M, H, C, J. Below these are the words END and SPRITE. The main script starts with a function definition:

```
-- DECLARE VARIABLES FOR BALL
FUNCTION MAKE_BALL()
  BALN = 2 -- SPRITE NUMBER
  BALX = 60 -- X COORDINATE
  BALY = 2 -- Y COORDINATE
  BALSPD = 3 -- SPEED
  BALW = 8 -- WIDTH
  BALH = 8 -- HEIGHT
```

Below this, there are comments and two variable assignments:

```
-- WE NEED VARIABLES TO TRACK
-- THE DIRECTION OF THE BALL
BALXDIR = "" -- HORIZONTAL
BALYDIR = "DOWN" -- VERTICAL
```

At the bottom, there's another END keyword and some status indicators:

```
END
LINE 1/26 175/8192 E
```

Making the Ball Bounce

When the ball and paddle collide, I'll switch **BALYDIR** from “DOWN” to “UP”

```
0123 + 0 00000000  
BALY = BALY - BALSPD  
END  
  
-- COLLIDE WITH PADDLE  
IF BALX + BALW >= PRDX  
AND BALX <= PRDX + PRDW  
AND BALY + BALH >= PRDY  
AND BALY <= PRDY + PRDH  
THEN  
    -- REVERSE DIRECTION  
    BALYDIR = "UP"  
END  
  
-- BOUNCE OFF CEILING  
IF BALY < 0 THEN  
    BALYDIR = "DOWN"  
END  
  
END -- END FUNCTION MOVE_BALL()
```

LINE 29/29 175/8192 E

Making the Ball Bounce

If the ball hits the ceiling, I'll switch its direction back to **"DOWN"** so it will bounce off the ceiling and start falling again

We know the ceiling is at **Y=0**, so we can compare the ball's y position against 0

```
0123 + 0WHD
BALY = BALY - BALSPD
END

-- COLLIDE WITH PADDLE
IF BALX + BALW >= PRDX
AND BALX <= PRDX + PRDW
AND BALY + BALH >= PRDY
AND BALY <= PRDY + PRDH
THEN
    -- REVERSE DIRECTION
    BALYDIR = "UP"
END

-- BOUNCE OFF CEILING
IF BALY < 0 THEN
    BALYDIR = "DOWN"
END
END -- END FUNCTION MOVE_BALL()
LINE 29/29 175/8192 E
```

Making the Ball Bounce

Finally, we can control which way the ball is moving (up or down) based on the value of **BALLYDIR**

Add the speed to **BALY** to move down; subtract to move up

```
0123+00000000  
-- MOVE BALL AND COLLIDE  
FUNCTION MOVE_BALL()  
  
-- MOVE BALL LOWER ON SCREEN  
IF BALLYDIR == "DOWN" THEN  
    BALY = BALY + BALSPD  
END  
  
-- MOVE BALL HIGHER ON SCREEN  
IF BALLYDIR == "UP" THEN  
    BALY = BALY - BALSPD  
END  
  
-- COLLIDE WITH PADDLE  
IF BALX + BALW >= PADX  
AND BALX <= PADX + PADW  
AND BALY + BALH >= PADY  
AND BALY <= PADY + PADH  
THEN  
LINE 29/29 175/8192 E
```

Making the Ball Bounce Sideways

Download Example File: [paddleball_basic_09_bounce.p8](#)

Making the Ball Bounce Sideways

If the paddle is moving when it hits the ball, we want the ball to move in the same direction (left or right) as well as up

This means *we'll have to combine if/then statements*

```
0 1 2 3 + 0 9 8 7 6 5
-- COLLIDE WITH PADDLE
IF BALX + BALW >= PADX
AND BALX <= PADX + PADW
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
    -- REVERSE DIRECTION
    BALXDIR = "UP"

    -- HIT BALL TO LEFT
    IF BTn(0) THEN
        BALXDIR = "LEFT"
    END

    -- HIT BALL TO RIGHT
    IF BTn(1) THEN
        BALXDIR = "RIGHT"
    END -- END IF BTn(1)
END -- END IF COLLIDE
LINE 42/59 227/8192 E
```

Making the Ball Bounce Sideways

If the paddle is moving, that means a button is being pressed

So we can use **BTN()** for the left and right arrow keys as our condition to measure whether the paddle is moving

```
0 1 2 3 + 0 4 5 6 < >
-- COLLIDE WITH PADDLE
IF BALX + BALW >= PADX
AND BALX <= PADX + PADW
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
    -- REVERSE DIRECTION
    BALXDIR = "UP"

    -- HIT BALL TO LEFT
    IF BTN(0) THEN
        BALXDIR = "LEFT"
    END

    -- HIT BALL TO RIGHT
    IF BTN(1) THEN
        BALXDIR = "RIGHT"
    END -- END IF BTN(1)
END -- END IF COLLIDE
LINE 42/59 227/8192 E
```

Making the Ball Bounce Sideways

If a button is being pressed,
we'll set **BALXDIR** to "LEFT"
or "RIGHT" accordingly

```
0/1/2/3 +      () * * < > 
-- COLLIDE WITH PADDLE
IF BALX + BALW >= PADX
AND BALX <= PADX + PADW
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
    -- REVERSE DIRECTION
    BALXDIR = "UP"

-- HIT BALL TO LEFT
IF BTn(0) THEN
    BALXDIR = "LEFT"
END

-- HIT BALL TO RIGHT
IF BTn(1) THEN
    BALXDIR = "RIGHT"
END -- END IF BTn(1)
END -- END IF COLLIDE
LINE 42/59          227/8192 E
```

Making the Ball Bounce Sideways

The button-press IF/THEN code must be *nested* within the IF/THEN block for collision

Because we need **TWO conditions to be met** for the ball to bounce at an angle:

- 1) Collision with paddle
- 2) An arrow keypress



```
-- COLLIDE WITH PADDLE
IF BALX + BALW >= PADX
AND BALX <= PADX + PADW
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
    -- REVERSE DIRECTION
    BALYDIR = "UP"

-- HIT BALL TO LEFT
IF BTn(0) THEN
    BALXDIR = "LEFT"
END

-- HIT BALL TO RIGHT
IF BTn(1) THEN
    BALXDIR = "RIGHT"
END -- END IF BTn(1)
END -- END IF COLLIDE
```

The screenshot shows a BASIC interpreter window with a pink header and footer. The main area contains a program for ball collision detection and direction reversal. A yellow box highlights the nested IF/THEN block for left-side collision, and another yellow box highlights the nested IF/THEN block for right-side collision. The footer displays 'LINE 42/54' and '22/08/1923'.

Making the Ball Bounce Sideways

Think of each IF/THEN block like slices of bread in a sandwich

An IF/THEN within an IF/THEN is like a double-decker sandwich

The image shows a Scratch script with a yellow border. The script uses variables for ball position (BALX, BALY) and paddle position (PADX, Pady). It checks for collisions with the paddle and then handles ball direction based on which side it was hit from.

```
-- COLLIDE WITH PADDLE
IF [BALX + BALW] ≥ [PADX]
AND [BALX] ≤ [PADX + PADW]
AND [BALY + BALH] ≥ [Pady]
AND [BALY] ≤ [Pady + PADH]
THEN
    -- REVERSE DIRECTION
    [BALYDIR] := "UP"

-- HIT BALL TO LEFT
IF [BTn(1)] THEN
    [BALXDIR] := "LEFT"
END

-- HIT BALL TO RIGHT
IF [BTn(2)] THEN
    [BALXDIR] := "RIGHT"
END -- END IF BTn(2)
END -- END IF COLLIDE
```

LINE 42/54 22/8192 E

Why can't we just
use **AND** to stack
conditions here?

Making the Ball Bounce Sideways

We can use **AND** to stack conditions when what happens when all conditions are met is the *same*

But in this instance, if collision is occurring, we will then **do 1 of 2 things:** move the ball left or move it right



```
-- COLLIDE WITH PADLE
IF BALX + BALW >= PADX
AND BALX <= PADX + PADW
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
-- REVERSE DIRECTION
BALYDIR = "UP"

-- HIT BALL TO LEFT
IF BTn(0) THEN
BALXDIR = "LEFT"
END

-- HIT BALL TO RIGHT
IF BTn(1) THEN
BALXDIR = "RIGHT"
END -- END IF BTn(1)
END -- END IF COLLIDE
```

The image shows a Scratch script with a yellow border. It starts with a comment '-- COLLIDE WITH PADLE' followed by an IF block. Inside the IF block, there are four AND conditions: 'BALX + BALW >= PADX', 'BALX <= PADX + PADW', 'BALY + BALH >= PADY', and 'BALY <= PADY + PADH'. Below the IF block is a THEN section containing a comment '-- REVERSE DIRECTION' and the assignment 'BALYDIR = "UP"'. The script then branches into two separate IF blocks, each enclosed in a yellow box. The first IF block checks for 'BTn(0)' (button A pressed), which sets 'BALXDIR = "LEFT"'. The second IF block checks for 'BTn(1)' (button B pressed), which sets 'BALXDIR = "RIGHT"'. Both of these sections have an associated END block. Finally, the entire IF block has another END block at the bottom, preceded by a comment '-- END IF COLLIDE'.

COLLISION BTWN PADDLE & BALL

IF BTN(R)
THEN
MOVE
RIGHT

IF BTN(L)
THEN
MOVE
LEFT

Think of this as a Venn diagram

```
-- COLLIDE WITH PADDLE
IF BALX + BALW >= PADX
AND BALX <= PADX + PADW
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
    -- REVERSE DIRECTION
    BALXDIR = "UP"

-- HIT BALL TO LEFT
IF BTN(0) THEN
    BALXDIR = "LEFT"
END

-- HIT BALL TO RIGHT
IF BTN(1) THEN
    BALXDIR = "RIGHT"
END -- END IF BTN(0)
END -- END IF COLLIDE
```

LIN 42/54

22/8192 E

Truth Tables

Another way to look at it

AND (Both conditions must be met)

X	Y	OUTCOME
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

AND: In Paddleball (2 Conditions Must be Met to Bounce at an Angle)

Collision	Arrow Key Pressed	OUTCOME	RESULT
FALSE	FALSE	FALSE	No action taken
FALSE	TRUE	FALSE	Paddle moves
TRUE	FALSE	FALSE	Ball bounces straight up
TRUE	TRUE	TRUE	Ball bounces at an angle

Making the Ball Bounce Sideways

A common error is that the ball moves left/right whenever an arrow key is pressed, regardless of whether there's collision

If this is happening, you likely placed the BTN IF/THEN statements **outside** the collision IF/THEN block

Place these **within** the collision IF/THEN block instead >>

The image shows a Scratch script with a yellow border. It starts with a collision sensor for a sprite named 'Paddle'. Inside the collision loop, it checks if the ball has collided with the paddle. If so, it reverses the ball's horizontal direction and sets its vertical direction to 'UP'. Then, it checks if the ball has hit the left edge of the stage. If yes, it sets the ball's horizontal direction to 'LEFT'. Similarly, it checks if the ball has hit the right edge and sets the horizontal direction to 'RIGHT' if true. Finally, it ends the collision loop and ends the script if no collision occurred.

```
-- COLLIDE WITH PADDLE
IF [COLLIDE WITH PADDLE v] THEN
    -- REVERSE DIRECTION
    BALXDIR = "UP"

    -- HIT BALL TO LEFT
    IF [BTN (L) v] THEN
        BALXDIR = "LEFT"
    END

    -- HIT BALL TO RIGHT
    IF [BTN (R) v] THEN
        BALXDIR = "RIGHT"
    END
END -- END IF COLLIDE
```

LINE 42/54 22/18192 E

Making the Ball Bounce Sideways

Finally, we need to adjust the ball's X position based on its horizontal direction

```
0 1 2 3 + 0 9 8 7 6 5
-- MOVE BALL AND COLLIDE
FUNCTION MOVE_BALL()

    -- MOVE BALL LEFT
    IF BALXDIR == "LEFT" THEN
        BALX = BALX - BALSPD
    END

    -- MOVE BALL RIGHT
    IF BALXDIR == "RIGHT" THEN
        BALX = BALX + BALSPD
    END

    -- MOVE BALL LOWER ON SCREEN
    IF BALYDIR == "DOWN" THEN
        BALY = BALY + BALSPD
    END

    -- MOVE BALL HIGHER ON SCREEN
END FUNCTION
```

LINE 42/59 227/8192 E

Making the Ball Bounce Sideways

You can't see these cases all at once in PICO-8's editor, so I'm showing it in VS Code

There's an IF/THEN to define how the ball should move for each of the 4 directions

```
-- move ball left
if balxdir == "left" then
    balx = balx - balspd
end

-- move ball right
if balxdir == "right" then
    balx = balx + balspd
end

-- move ball lower on screen
if balydir == "down" then
    baly = baly + balspd
end

-- move ball higher on screen
if balydir == "up" then
    baly = baly - balspd
end
```

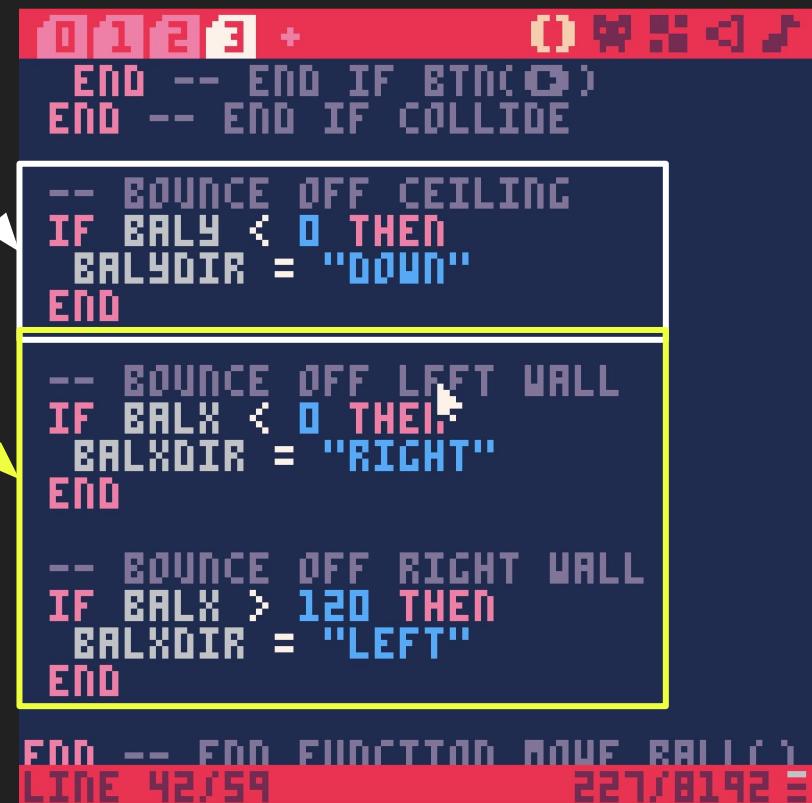
Bouncing off the Walls & Ceiling

Bouncing off the Walls and Ceiling

We already added this to make the ball switch direction when reaching the ceiling

We can do the same thing for the walls

A more complete solution would be to use 128 minus the ball's width to check against the right edge of the screen



The screenshot shows a BASIC interpreter interface with the following code:

```
0123 + 0
END -- END IF BTN(0)
END -- END IF COLLIDE
-- BOUNCE OFF CEILING
IF BALY < 0 THEN
    BALYDIR = "DOWN"
END
-- BOUNCE OFF LEFT WALL
IF BALX < 0 THEN
    BALXDIR = "RIGHT"
END
-- BOUNCE OFF RIGHT WALL
IF BALX > 120 THEN
    BALXDIR = "LEFT"
END
END -- END FUNCTION MOVE BALL(1)
```

Line 42/59 227/8192 E

Annotations with arrows point from the text "We can do the same thing for the walls" to the code blocks for "BOUNCE OFF LEFT WALL" and "BOUNCE OFF RIGHT WALL".

Bouncing off the Walls and Ceiling

Make sure to place these IF/THEN statements ***OUTSIDE*** the collision IF/THEN block

Otherwise the ball would only switch direction if BOTH collision is true and the other conditions are met

The image shows a Scratch script with the following code:

```
0123 + [0 v] 
END -- END IF BTn([0])
END -- END IF COLLIDE
-- BOUNCE OFF CEILING
IF BALY < 0 THEN
  BALYDIR = "DOWN"
END
-- BOUNCE OFF LEFT WALL
IF BALX < 0 THEN
  BALXDIR = "RIGHT"
END
-- BOUNCE OFF RIGHT WALL
IF BALX > 120 THEN
  BALXDIR = "LEFT"
END
END -- END FUNCTION MOVE BALL ( )
LINE 42/59 227/8192 =
```

The script consists of several nested loops and conditionals. It includes logic for bouncing off the ceiling, left wall, and right wall, all placed outside the main collision detection loop. The code uses variables like BALY, BALX, and BALYDIR/BALXDIR to track the ball's position and direction.

*A Venn
diagram of
our IF/THEN
logic so far*

COLLISION BTWN PADDLE & BALL

IF BTN(R)
THEN
MOVE
RIGHT

IF BTN(L)
THEN
MOVE
LEFT

IF X >=120
THEN
SWITCH
HORIZONTAL
DIRECTION

IF X <=0
THEN
SWITCH
HORIZONTAL
DIRECTION

IF Y <=0
THEN
SWITCH
VERTICAL
DIRECTION

The entire move_ball() function so far:

Change X/Y based on direction

Collision and bouncing at an angle

Bouncing off walls and ceiling

```
-- move ball and collide
function move_ball()

    -- move ball left
    if balxdir == "left" then
        balx = balx - balspd
    end

    -- move ball right
    if balxdir == "right" then
        balx = balx + balspd
    end

    -- move ball lower on screen
    if balydir == "down" then
        baly = baly + balspd
    end

    -- move ball higher on screen
    if balydir == "up" then
        baly = baly - balspd
    end

    -- collide with paddle
    if balx + balw >= padx
    and balx <= padx + padw
    and baly + balh >= pady
    and baly <= pady + padh
    then
        -- reverse direction
        balydir = "up"

        -- hit ball to left
        if btn(⬅️) then
            balxdir = "left"
        end

        -- hit ball to right
        if btn(➡️) then
            balxdir = "right"
        end
    end -- end if collide

    -- bounce off ceiling
    if baly < 0 then
        balydir = "down"
    end

    -- bounce off left wall
    if balx < 0 then
        balxdir = "right"
    end

    -- bounce off right wall
    if balx > 120 then
        balxdir = "left"
    end
```

Resetting the Ball After a Miss

Resetting the Ball After a Miss

If the player misses the ball and it falls off the bottom of the screen, we need to move the ball back to its initial position

We also need to reset its direction

All the values being set here are the same as the initial values used in make_ball()

```
0123 + 0000< )
-- BOUNCE OFF LEFT WALL
IF BALX < 0 THEN
  BALXDIR = "RIGHT"
END

-- BOUNCE OFF RIGHT WALL
IF BALX > 120 THEN
  BALXDIR = "LEFT"
END

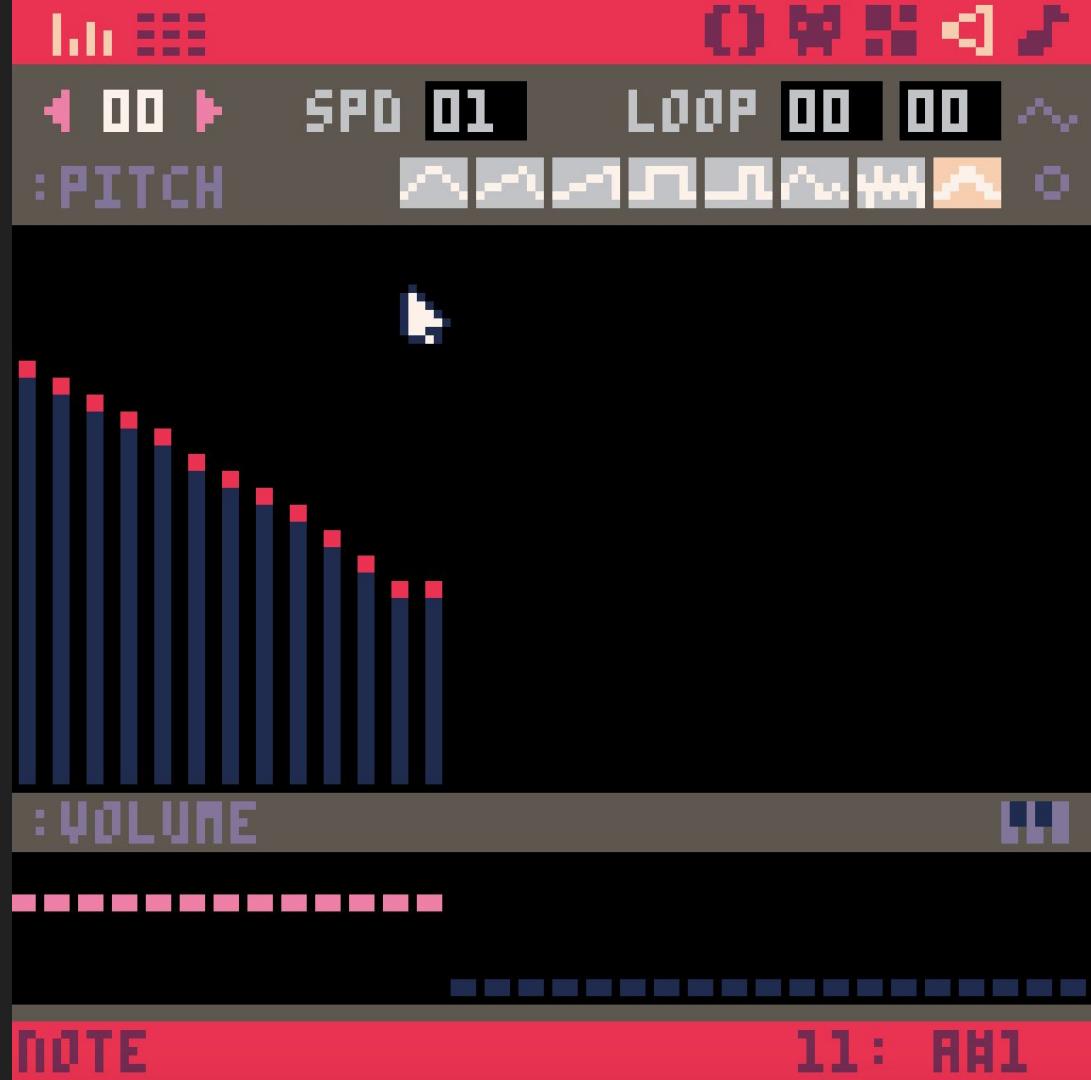
-- RESET BALL WHEN MISSED
IF BALY > 128 THEN
  BALX = 60
  BALY = 2
  BALXDIR = ...
  BALYDIR = "DOWN"
END

END -- END FUNCTION MOVE_BALL()
LINE 59/75 275/8192 E
```

Adding Sounds

Adding Sounds

I created a cool squishy laser boinky sound for when the ball bounces off the paddle, walls, or ceiling



Adding Sounds

Also a sad chime for
when the ball goes off
the bottom of the screen



Adding Sounds

You can use the SFX() function, with the number for the sound inside the parentheses, to play a sound effect

I'll add SFX(0) anywhere the ball bounces

```
-- BOUNCE OFF CEILING
IF BALY < 0 THEN
  BALYDTR = "BALLUP"
  SFX(0) -- PLAY BOUNCE SOUND
END

-- BOUNCE OFF LEFT WALL
IF BALX < 0 THEN
  BALYDTR = "RIGHT"
  SFX(0) -- PLAY BOUNCE SOUND
END

-- BOUNCE OFF RIGHT WALL
IF BALX > 120 THEN
  BALYDTR = "LEFT"
  SFX(0) -- PLAY BOUNCE SOUND
END
```

LINE 67/79 289/8192 E

Adding Sounds

You can use the SFX() function, with the number for the sound inside the parentheses, to play a sound effect

I'll add SFX(1) in the if/then block for handling a miss



The screenshot shows a game development interface with a code editor and a preview window.

Code Editor:

```
01 [2] + 0 [WASD]
-- BOUNCE OFF RIGHT WALL
IF BALX > 120 THEN
  BALXDIR = "LEFT"
  SFX(0) -- PLAY BOUNCE SOUND
END

-- RESET BALL WHEN MISSED
IF BALY > 128 THEN
  BALX = 60
  BALY = 2
  BALXDIR = ""
  BALYDIR = "down"
  SFX(1) -- PLAY FAILURE SOUND
END

END -- END FUNCTION MOVE_BALL()
```

Preview Window:

The preview window shows a waveform for a sound effect. A yellow box highlights the SFX(1) line in the code, which corresponds to the waveform shown in the preview. The waveform has a distinct downward-sweeping pattern, representing a failure sound.

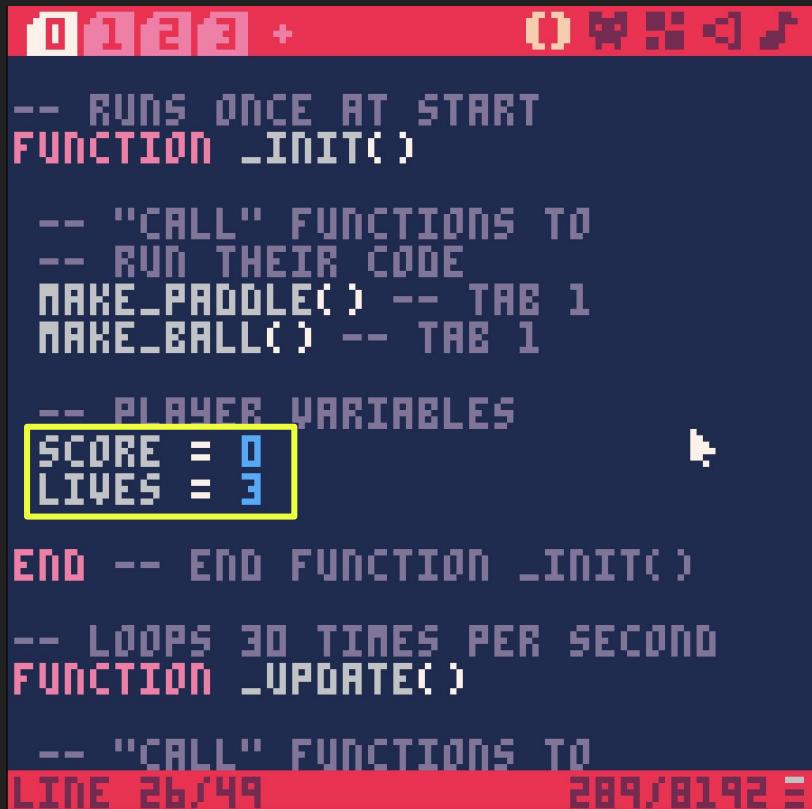
Adding Score & Lives

Download Example File: [paddleball_basic_10_score_and_lives.p8](#)

Adding Score and Lives

We'll need to introduce some new variables for the player's **score** and remaining **lives**

I'll do this in `_init()`



The screenshot shows a game development environment with Python code for a Pong-like game. The code includes functions for initializing the game and updating it, and defines player variables for score and lives.

```
0 1 2 3 + 0 0 0 < >
-- RUNS ONCE AT START
FUNCTION _INIT()
    -- "CALL" FUNCTIONS TO
    -- RUN THEIR CODE
    MAKE_PADDLE() -- TAB 1
    MAKE_BALL() -- TAB 1
    -- PLAYER VARIABLES
    SCORE = 0
    LIVES = 3
END -- END FUNCTION _INIT()

-- LOOPS 30 TIMES PER SECOND
FUNCTION _UPDATE()
    -- "CALL" FUNCTIONS TO
LINE 26/49 289/8192 E
```

Adding Score and Lives

We can add a quick HUD by using the PRINT() function in _draw()

```
print( text, [x,] [y,] [color] )
```



```
-- LOOPS 30 TIMES PER SECOND
FUNCTION _DRAW()
    CLS() -- CLEARS THE SCREEN
    -- DRAW PADDLE SPRITE
    SPR(PADn, PADX, PADY)
    -- DRAW BALL SPRITE
    SPR(BALn, BALX, BALY)
    -- PRINT SCORE, LIVES
    -- PLACE TEXT IN QUOTES
    -- COMBINE TEXT WITH NUMBERS
    -- BY TYPING .. BETWEEN THEM
    PRINT("SCORE: "..SCORE, 2, 2)
    PRINT("LIVES: "..LIVES, 2, 10)
END
```

LINE 26/49 289/8192 E

Adding Score and Lives

- Remember, we can put plain text inside quotes
- But to **combine text with a numeric value** (the variable values for score, lives), we can **put two periods .. between the text in quotes and the variable**

Text values are called
“strings” in programming

Combining two “strings”
together is called
“concatenation”

```
FUNCTION _DRAW()  
CLS()  
PRINT("SCORE: ".SCORE,2,2,7)  
PRINT("LIVES: ".LIVES,2,10,8)  
print( text, [x,] [y,] [color] )
```

Adding Score and Lives

- After the first value (the text to print), we can add values for the x and y coordinates of the text
- The **last value** is the numeric code for the color

COLORS			
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
FUNCTION _DRAW()
CLS()
PRINT("SCORE: ".SCORE,2,2,7)
PRINT("LIVES: ".LIVES,2,10,8)
print( text, [x,] [y,] [color] )
```

Adding Score

To change the score when the player hits the ball, we'll add to the score variable's value when collision between the ball and paddle is true, in the move_ball() function

```
-- COLLIDE WITH PADDLE
IF BALX + BALW >= PADX
AND BALX <= PADX + PADW
AND BALY + BALH >= PADY
AND BALY <= PADY + PADH
THEN
    -- REVERSE DIRECTION
    BALYDIR = "UP"

-- ADD TO SCORE
SCORE = SCORE + 1

-- PLAY BOUNCE SOUND
SFX(0)

-- HIT BALL TO LEFT
IF BTB(0) THEN
    BALXDIR = "LEFT"
LINE 78/79          289/8192 E
```

Managing Lives

Inside the **IF/THEN** block for resetting the ball, we can subtract a life when the ball goes off screen

```
0 1 2 3 + 0 9 8 7 6 5 4 3 2 1 0
BALXDIR = "LEFT"
SFX(0) -- PLAY BOUNCE SOUND
END

-- RESET BALL WHEN MISSED
IF BALY > 128 THEN
BALX = 60
BALY = 2
BALXDIR = "..."
BALYDIR = "DOWN"
LIVES = LIVES - 1 -- LOSE LIFE
SFX(1) -- PLAY FAILURE SOUND
END

END -- END FUNCTION MOVE_BALL()
```

Managing Lives

But look what happens if we keep losing lives . . .

The value turns
negative, but we
can still keep
playing



Implementing a Game Over Screen

Download Example File: [paddleball_basic_11_gameover.p8](#)

Implementing a Game Over Screen

If the player's lives reach zero,
we can stop the game

We can approach this
backwards, by **only** running the
game as long as **lives are above**
zero



```
0 1 2 3 + 0 9 8 7 6 5
LIVES = 3

END -- END FUNCTION _INIT()
-- LOOPS 30 TIMES PER SECOND
FUNCTION _UPDATE()

-- STOP THE GAME IF NO LIVES
IF LIVES > 0 THEN
    MOVE_PADDLE() -- TAB 2
    MOVE_BALL() -- TAB 3
END

END -- END FUNCTION _UPDATE()

-- LOOPS 30 TIMES PER SECOND
FUNCTION _DRAW()
CLS() -- CLEARS THE SCREEN

LINE 1/56 305/8192 E
```

Implementing a Game Over Screen

Likewise, we can use **IF/THEN** statements in our **_DRAW()** loop to **display a game over** message



There's no text alignment feature in PICO-8, so you have to find the right coordinates by trial & error

```
0 1 2 3 + 0 9 8 7 < >
SPR(PADn, PADx, PADy)
-- DRAW BALL SPRITE
SPR(BALn, BALx, BALy)

-- PRINT SCORE, LIVES
-- PLACE TEXT IN QUOTES
-- COMBINE TEXT WITH NUMBERS
-- BY TYPING .. BETWEEN THEM
PRINT("SCORE: "..SCORE, 2, 2)
PRINT("LIVES: "..LIVES, 2, 10)

-- SHOW GAME OVER MESSAGE
IF LIVES < 1 THEN
    PRINT("GAME OVER", 4b, b4, 8)
END

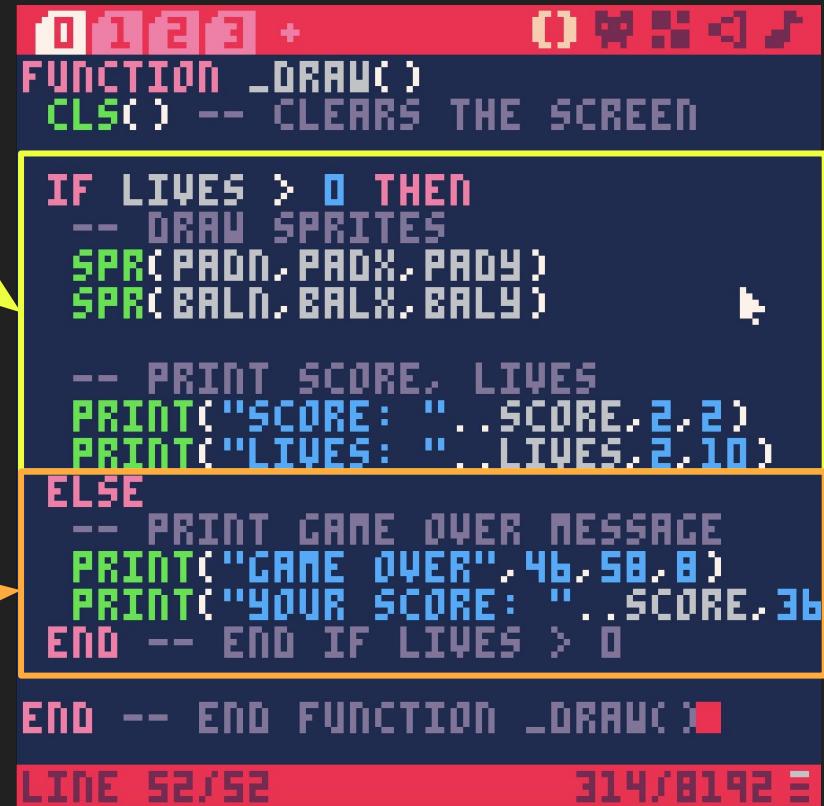
END -- END FUNCTION _DRAW()
```

LINE 1/56 305/8192 E

Implementing a Game Over Screen

If you want the game to disappear and be replaced by a **game over screen**, you can require that the player have lives left to draw the sprites and HUD

Use **ELSE** to display a game over screen when lives reach zero



The screenshot shows a game development environment with assembly-like pseudocode. A yellow arrow points from the text "If you want the game to disappear and be replaced by a game over screen, you can require that the player have lives left to draw the sprites and HUD" to the conditional logic in the code. Another yellow arrow points from the text "Use ELSE to display a game over screen when lives reach zero" to the "ELSE" block in the code.

```
FUNCTION _DRAW()
CLS() -- CLEARS THE SCREEN
IF LIVES > 0 THEN
    -- DRAW SPRITES
    SPR(PADD, PADX, PADY)
    SPR(BALD, BALX, BALY)
    -- PRINT SCORE, LIVES
    PRINT("SCORE: ".SCORE, 2, 2)
    PRINT("LIVES: ".LIVES, 2, 10)
ELSE
    -- PRINT GAME OVER MESSAGE
    PRINT("GAME OVER", 46, 58, 8)
    PRINT("YOUR SCORE: ".SCORE, 36)
END -- END IF LIVES > 0
END -- END FUNCTION _DRAW()
```

LINE 52/52 314/8192 E

Implementing a Game Over Screen

GAME OVER
YOUR SCORE: 17



```
0/1/2/3 + 0/8/8/8/8
FUNCTION _DRAW()
CLS() -- CLEARS THE SCREEN

IF LIVES > 0 THEN
-- DRAW SPRITES
SPR(PADDN, PADDX, PADDY)
SPR(BALN, BALX, BALY)

-- PRINT SCORE, LIVES
PRINT("SCORE: ".SCORE, 2, 2)
PRINT("LIVES: ".LIVES, 2, 10)

ELSE
-- PRINT GAME OVER MESSAGE
PRINT("GAME OVER", 46, 58, 8)
PRINT("YOUR SCORE: ".SCORE, 36
END -- END IF LIVES > 0

END -- END FUNCTION _DRAW()
LINE 52/52          314/8192 E
```

[Download example file](#)

Restarting the Game

Download Example File: [paddleball_basic_12_restart.p8](#)

Restarting the Game

```
0 1 2 3 +      0 9 8 7 6 5  
LIVES = 3  
END -- END FUNCTION _INIT()  
-- LOOPS 30 TIMES PER SECOND  
FUNCTION _UPDATE()  
    -- STOP THE GAME IF NO LIVES  
    IF LIVES > 0 THEN  
        MOVE_PADDLE() -- TAB 2  
        MOVE_BALL() -- TAB 3  
    END  
END -- END FUNCTION _UPDATE()  
-- LOOPS 30 TIMES PER SECOND  
FUNCTION _DRAW()  
    CLS() -- CLEARS THE SCREEN  
LINE 1/56          305/8192 E
```

In **_UPDATE()**, we are already stopping the game when lives reach zero

But we should also give the player a way to restart the game after getting a game over

We can add an **ELSE** case to this IF/THEN block to specify what inputs the player can perform after a game over – if they press the **X** key, we can call the **_INIT()** function, which *reverts the game to its starting conditions*

```
0 1 2 3 + 0 9 8 7 6 5 4 3 2 1
LIVES = 3

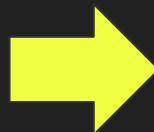
END -- END FUNCTION _INIT()

-- LOOPS 30 TIMES PER SECOND
FUNCTION _UPDATE()

    -- STOP THE GAME IF NO LIVES
    IF LIVES > 0 THEN
        MOVE_PADDLE() -- TAB 2
        MOVE_BALL() -- TAB 3
    END

    END -- END FUNCTION _UPDATE()

    -- LOOPS 30 TIMES PER SECOND
    FUNCTION _DRAW()
        CLS() -- CLEARS THE SCREEN
LINE 1/56          305/8192 E
```



```
0 1 2 3 + 0 9 8 7 6 5 4 3 2 1
END -- END FUNCTION _INIT()

-- LOOPS 30 TIMES PER SECOND
FUNCTION _UPDATE()

    -- STOP THE GAME IF NO LIVES
    IF LIVES > 0 THEN
        MOVE_PADDLE() -- TAB 2
        MOVE_BALL() -- TAB 3
    ELSE
        -- PRESS ⚡ TO RESTART
        IF BTNP(⚡) THEN
            _INIT()
        END -- END IF BTNP(⚡)

    END -- END IF/ELSE LIVES > 0

    END -- END FUNCTION _UPDATE()

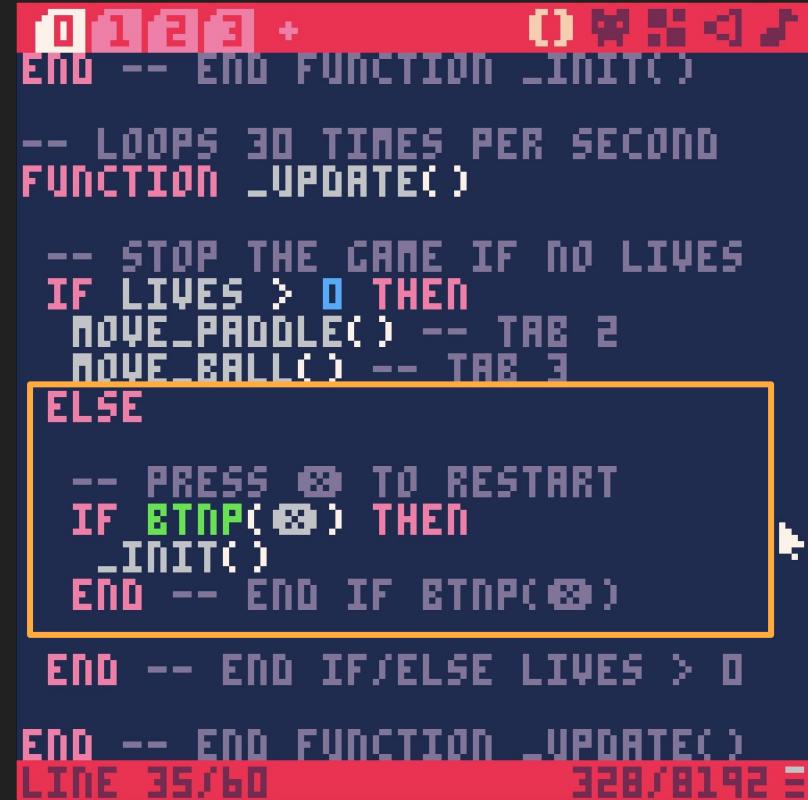
LINE 35/60          328/8192 E
```

Restarting the Game

BTNP() is a function similar to
BTN()

But while **BTN()** requires the player to press and hold a key, **BTNP()** will trigger as long as a key is pressed

Type **Shift X** in the PICO-8 editor to generate an X-key symbol – *or* use the key code **5**, as in **BTNP(5)**



The screenshot shows the PICO-8 game editor with the following source code:

```
0 1 2 3 + 0 0 0 0 0 0 0 0
End -- END FUNCTION _INIT()

-- LOOPS 30 TIMES PER SECOND
FUNCTION _UPDATE()

-- STOP THE GAME IF NO LIVES
IF LIVES > 0 THEN
    MOVE_PADDLE() -- TAB 2
    MOVE_BALL() -- TAB 3
ELSE
    -- PRESS ⚡ TO RESTART
    IF BTNP(5) THEN
        _INIT()
    End -- END IF BTNP(5)
END -- END IF/ELSE LIVES > 0
End -- END FUNCTION _UPDATE()
LINE 35/60 328/8192 E
```

The code implements a game loop that calls `_INIT()` at the start. It then enters a loop that calls `MOVE_PADDLE()` and `MOVE_BALL()` every frame. An `IF` statement checks if there are lives left. If there are, it moves the paddle and ball. If there are none, it checks if the X key is pressed (via `BTNP(5)`). If the X key is pressed, it calls `_INIT()` to restart the game. Finally, it exits the `IF` block and ends the `_UPDATE()` function.

Restarting the Game

Finally, we should provide instructions to the player that they can press X to restart from the game over screen

I'll add a **PRINT()** statement in **_DRAW()** after displaying the game over message and score



```
0 1 2 3 + 0 0 0 < > 
-- LOOPS 30 TIMES PER SECOND
FUNCTION _DRAW()
CLS() -- CLEARS THE SCREEN

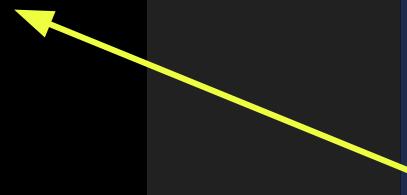
IF LIVES > 0 THEN
-- DRAW SPRITES
SPR(PADn, PADx, PADy)
SPR(BALn, BALx, BALy)

-- PRINT SCORE, LIVES
PRINT("SCORE: ", SCORE, 2, 2)
PRINT("LIVES: ", LIVES, 2, 10)
ELSE
-- PRINT GAME OVER MESSAGE
PRINT("GAME OVER", 46, 58, 8)
PRINT("YOUR SCORE: ", SCORE, 36)
PRINT("PRESS ⚡ TO RESTART", 28
END -- END IF LIVES > 0      
```

LINE 34/57 328/8192 E

Restarting the Game

GAME OVER
YOUR SCORE: 7
PRESS ⚡ TO RESTART



The complete PRINT() statement:

```
PRINT("PRESS ⚡ TO RESTART", 28, 74, 7)
```

```
0 1 2 3 + 0 9 8 7 6
-- LOOPS 30 TIMES PER SECOND
FUNCTION _DRAW()
    CLS() -- CLEARS THE SCREEN

    IF LIVES > 0 THEN
        -- DRAW SPRITES
        SPR(PADn, PADx, PADy)
        SPR(BALn, BALx, BALy)

        -- PRINT SCORE, LIVES
        PRINT("SCORE: ", SCORE, 2, 2)
        PRINT("LIVES: ", LIVES, 2, 10)
    ELSE
        -- PRINT GAME OVER MESSAGE
        PRINT("GAME OVER", 46, 58, 8)
        PRINT("YOUR SCORE: ", SCORE, 36)
        PRINT("PRESS ⚡ TO RESTART", 28, 74, 7)
    END -- END IF LIVES > 0
```

LINE 34/57

328/8192 E

New Game Plus: Paddleball w/Physics

Download Example Files: [_helloworld_01a_paddleball_physics.zip](#)

Learning Resources

PICO-8 Resources

- [PICO-8 Home](#)
- [Official Resources](#)
- [Cheat Sheet](#)
- [Forum](#)
- [Games](#)

[My GitHub Repository, helloworld](#)

Code Examples

- basics_00_helloworld.p8
- basics_01_shapes.p8
- basics_02_gameloop.p8
- basics_03_variables.p8
- basics_04_coordinates.p8

Code Examples

1. [paddleball_basic_01_sprites.p8](#)
2. [paddleball_basic_02_variables.p8](#)
3. [paddleball_basic_03_variables2.p8](#)
4. [paddleball_basic_04_input_and_movement.p8](#)
5. [paddleball_basic_05_conditional_logic.p8](#)
6. [paddleball_basic_06_functions.p8](#)
7. [paddleball_basic_07_collision.p8](#)
8. [paddleball_basic_08_reverse_direction.p8](#)
9. [paddleball_basic_09_bounce.p8](#)
10. [paddleball_basic_10_score_and_lives.p8](#)
11. [paddleball_basic_11_gameover.p8](#) | [paddleball_basic_11a_gameover_screen.p8](#)
12. [paddleball_basic_12_restart.p8](#)

Download all paddleball examples: [_helloworld_01_paddleball_basic.zip](#)

PICO-8 Function Reference

PICO-8 Wiki – Functions Reference

- `_init()` Define starting conditions for your game
 - `_update()` For input, calculations, and updating variable values
 - `_draw()` For displaying text and graphics
-
- `cls()` Clears the screen
 - `spr()` Draws a sprite to the screen
 - `print()` Prints text to the screen
 - `btn()` Checks whether a button is being held down
 - `btnp()` Checks whether a button was pressed
 - `sfx()` Plays a sound effect



Please note that this lesson is licensed under a [Creative Commons](#)
[Attribution-NonCommercial-NoDerivatives 4.0 International License](#). This
lesson may not be used for commercial purposes or be distributed as part of
any derivative works without my (Matthew DiMatteo's) written permission.

</lesson>

Questions? Email me at mdimatteo@rider.edu anytime!