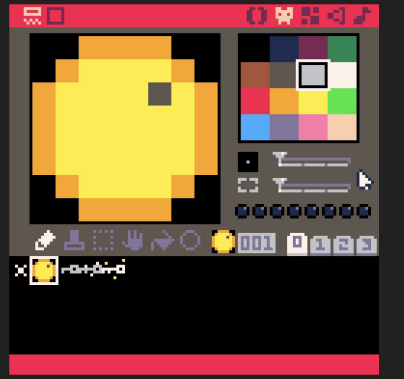
Intro to Game Programming with PICO-8

by Matthew DiMatteo



Contents

Part 1: Background Info and Getting Started

5) About This Lesson
10) About PICO-8
16) Using the PICO-8 Editor
28) Drawing Sprites
44) Saving Your Work
47) Loading a Saved File

Contents

Part 2: Your First PICO-8 Program

51) Coding in PICO-8
59) Displaying Sprites in Your Game
72) The PICO-8 Coordinate Plane
82) The PICO-8 Program Structure or “ Game Loop
89) Using Variables
94) How the Game Loop Affects Variable Values
106) Rules for Using Variables
117) Detecting and Responding to Input

Contents

Part 3: Adding Organization and Complexity

132) Keeping Your Code Organized
141) Writing Your Own Functions
158) Animating a Sprite
165) Collecting Items (Detecting Collision)
187) Recap

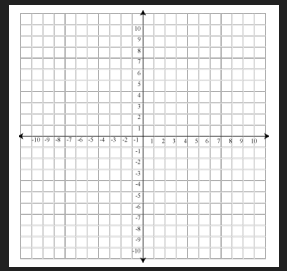
About This Lesson

Download Example Files:
_helloworld_00_intro_to_game_programming.zip

About This Lesson

This lesson assumes a working knowledge of:

- Basic arithmetic
- The cartesian plane



But no prior knowledge of programming is required

About This Lesson

In this lesson, you'll learn how to:

- Navigate the PICO-8 game engine
- Create pixel-art game graphics
- Program a basic interaction in a digital game (moving a character across a screen to collect an item)

You'll also understand fundamental concepts of programming that will transfer to other areas

About This Lesson

This lesson includes a series of step-by-step code examples

Download the example set from my GitHub:
_helloworld_00_intro_to_game_programming.zip

Code Examples

Download the full set:
_helloworld_00_intro_to_game_programming.zip

1.	intro_01_hello_world.p8	Your first PICO-8 program
2.	intro_02_coordinate_plane.p8	Understanding PICO-8's x,y coordinate system
3.	intro_03_text.p8	Positioning text on the screen
4.	intro_04_color.p8	Setting text color
5.	intro_05_layers.p8	Understanding stacking order
6.	intro_06_sprites.p8	Displaying sprites in your game
7.	intro_07_game_loop.p8	Understanding the PICO-8 program structure
8.	intro_08_variables.p8	Using values that can change over time
9.	intro_09_move_player.p8	Detecting and responding to key input
10.	intro_10_functions.p8	Organizing your code
11.	intro_11_animation.p8	Adding visual effects to graphics
12.	intro_12_collection.p8	Using collision detection to collect an item

About PICO-8

pico-8-edu.com

About PICO-8

PICO-8 is an all-in-one game engine that includes:

- A code editor
- A graphics editor
- A map editor
- A sound editor
- A music editor

About PICO-8

PICO-8 is a game engine built around *limitations*

- Limited number of sprites (graphics)
- Limited amount of code
- Limited map size
- Limited colors, sounds

About PICO-8

PICO-8 is a game engine built around limitations

- While *not necessarily the best choice for larger games*, its simplicity makes it a good choice for:
  - Educational exercises
  - Small games
  - Prototyping

Celeste was originally prototyped in PICO-8

About PICO-8

Embrace your limitations!

- Think of them as ***direction***
- Constraints can keep you focused and help you prioritize what's most important
- Sometimes they can even inspire you

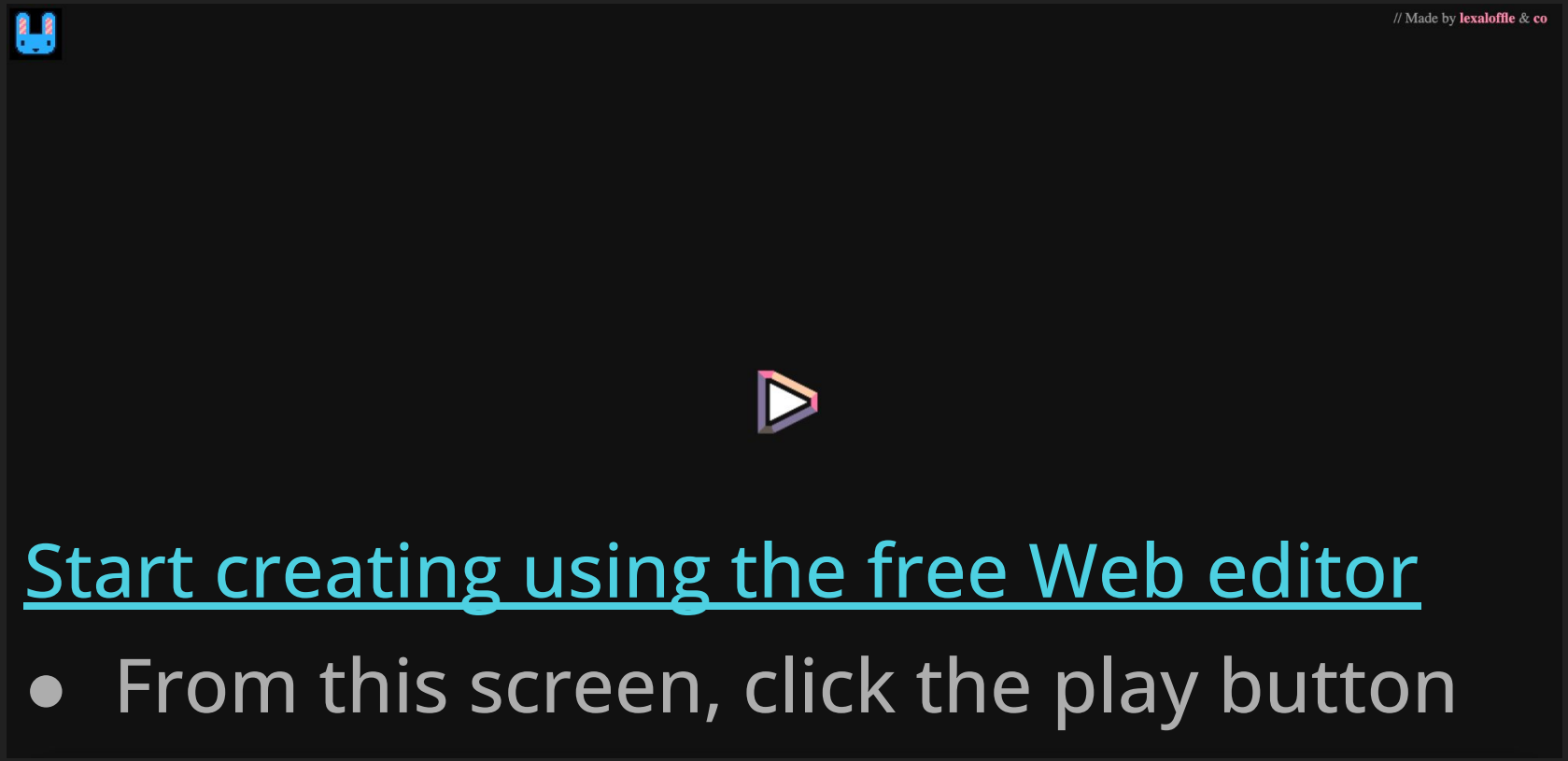
About PICO-8

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

Go to pico-8-edu.com

Using the PICO-8 Editor

Using the PICO-8 Editor



Start creating using the free Web editor

- From this screen, click the play button

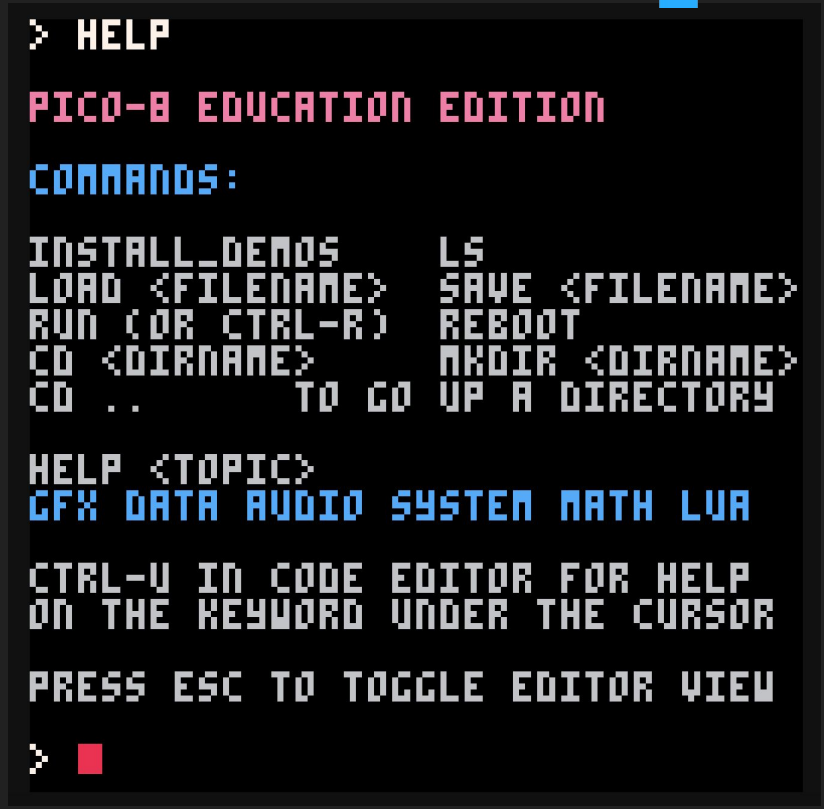
Using the PICO-8 Editor

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

Using the PICO-8 Editor

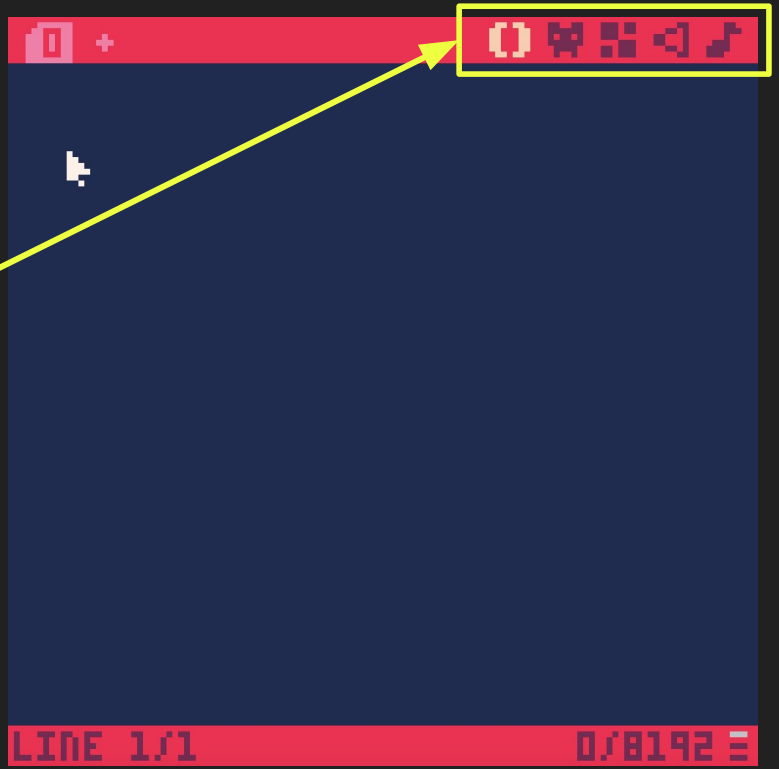
- If you type **HELP** and press **Enter/Return**, you'll see a list of commands
- Hit **Esc** to swap between this screen and the editor panels

Note that you'll automatically type in capital letters

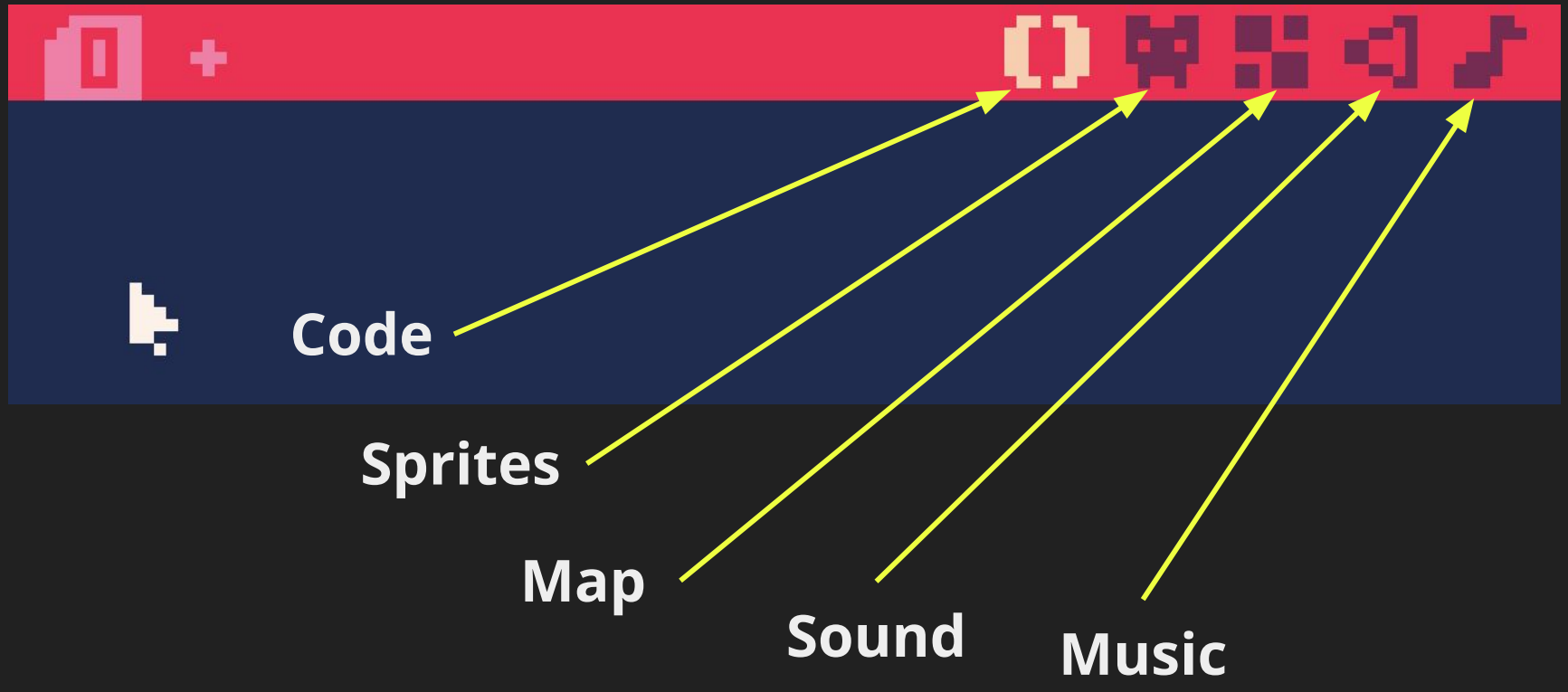


Using the PICO-8 Editor

- 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

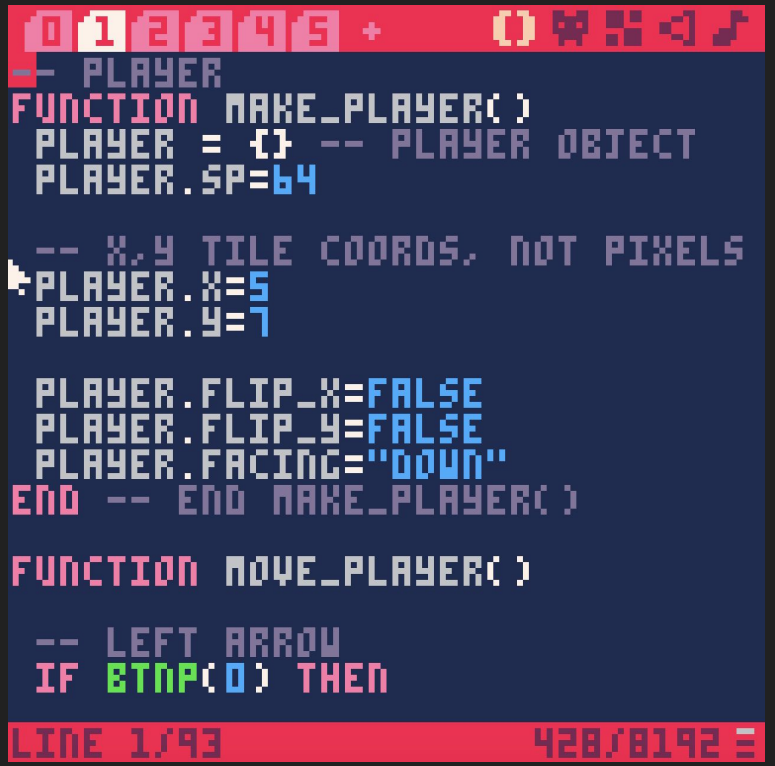


Using the PICO-8 Editor



Using the PICO-8 Editor

PICO-8 has a code editor for writing your code



Using the PICO-8 Editor

PICO-8 has a sprite editor for drawing graphics

You must draw all graphics in the editor; no uploading image files

Using the PICO-8 Editor

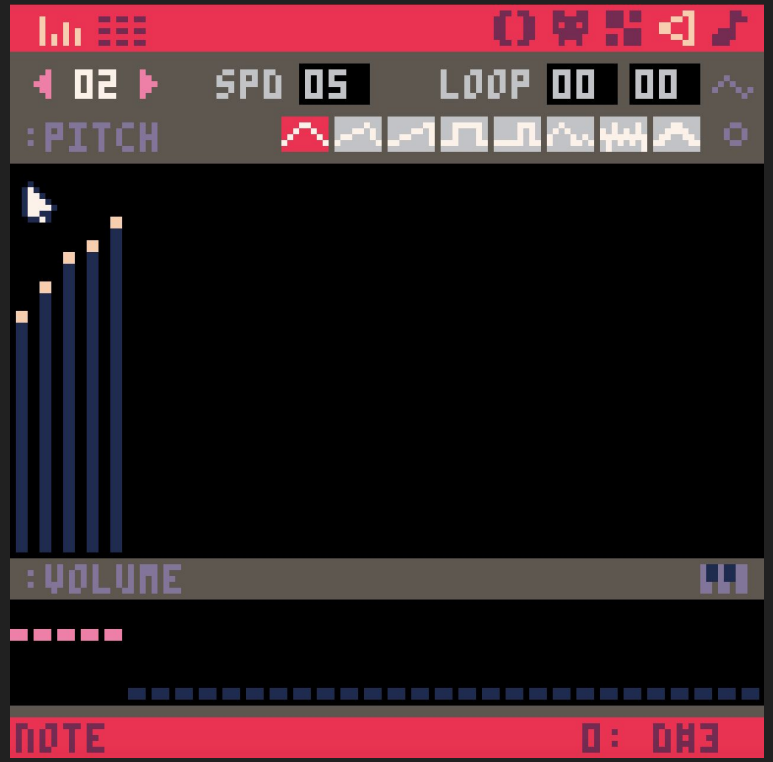
PICO-8 has a map editor for creating your game world

- 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)

Using the PICO-8 Editor

PICO-8 has an audio editor for creating sound effects

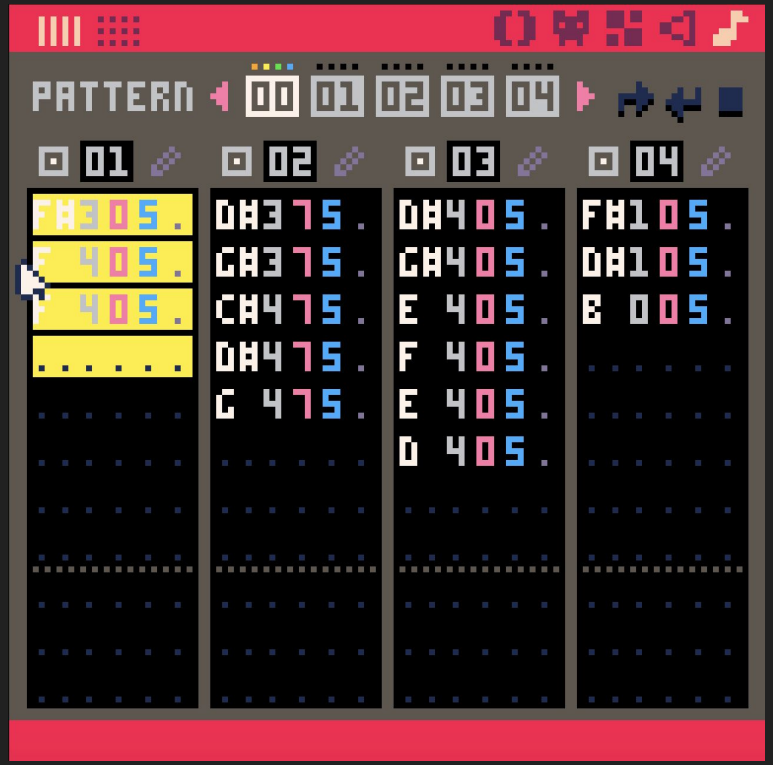
All audio must be created in the engine; no file uploads



Using the PICO-8 Editor

PICO-8 has a music editor for creating chiptunes

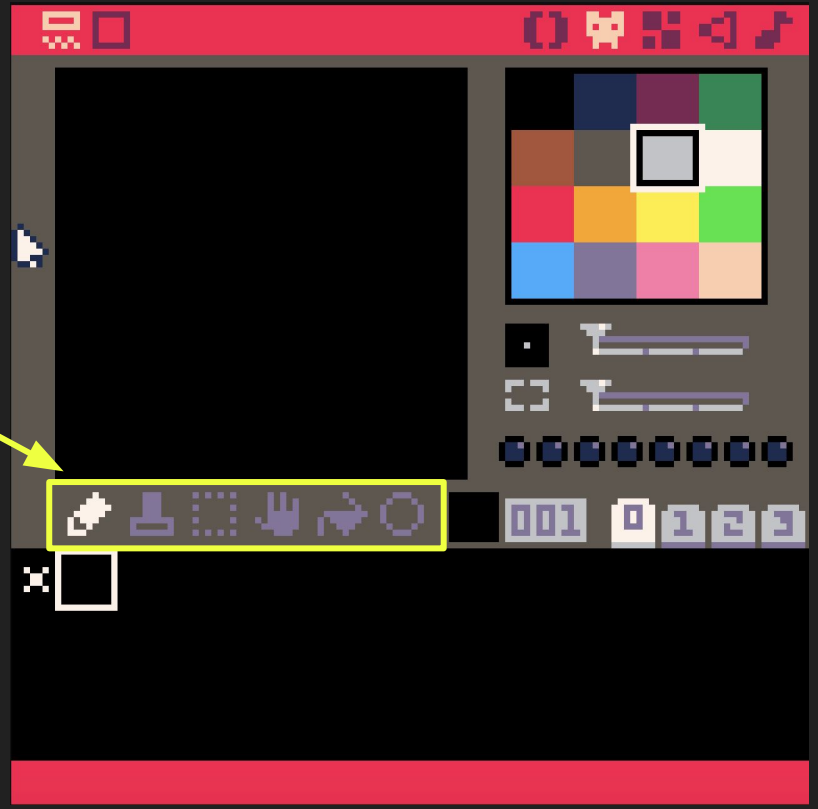
All audio must be created in the engine; no file uploads



Drawing Sprites

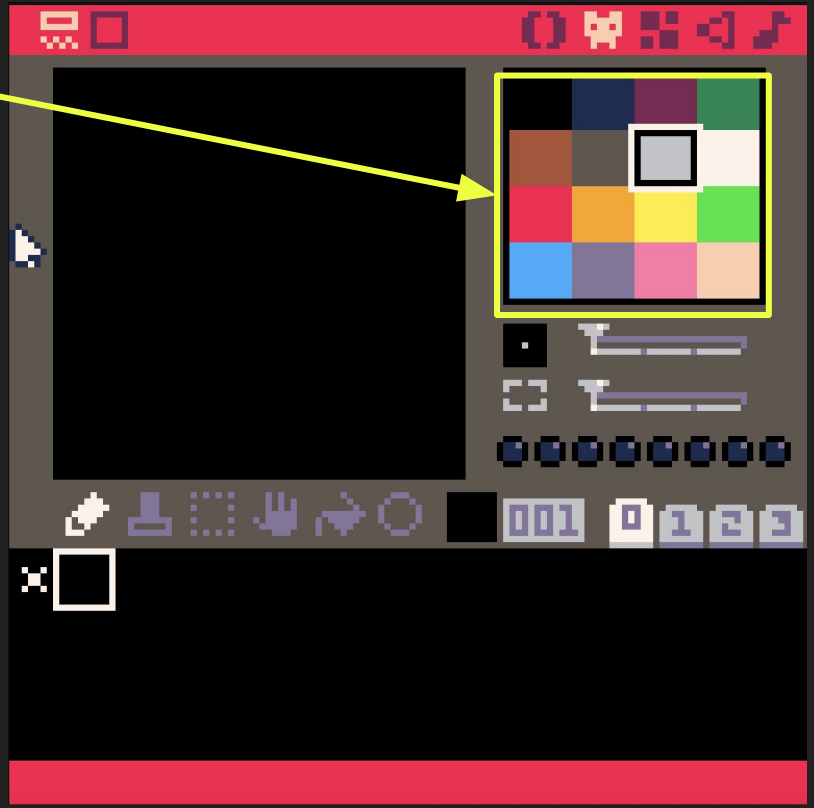
Drawing Sprites

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



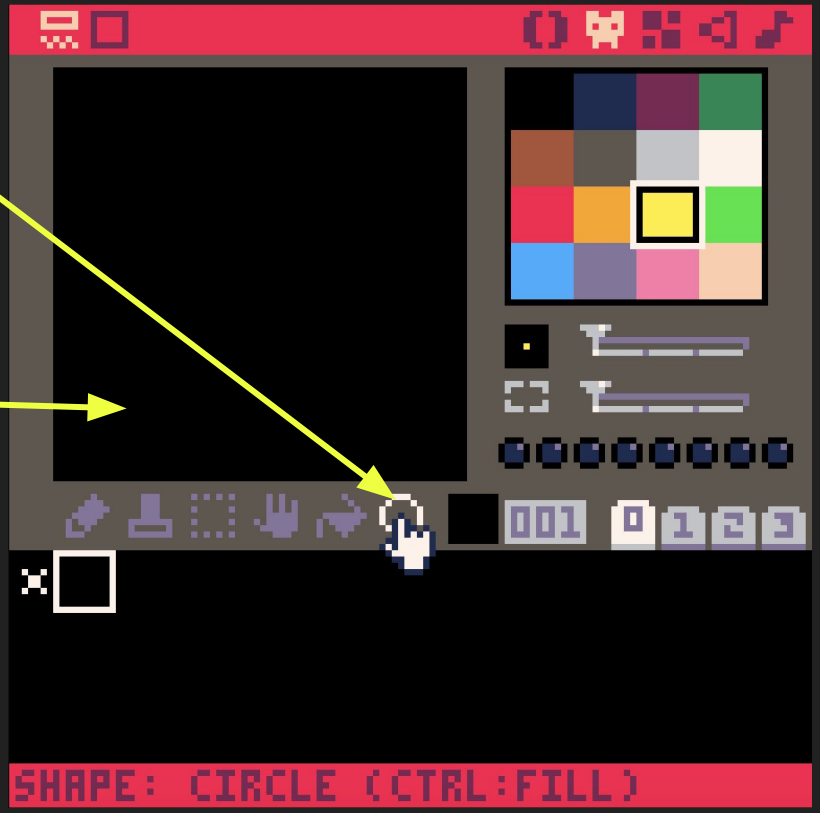
Drawing Sprites

- Select a color here



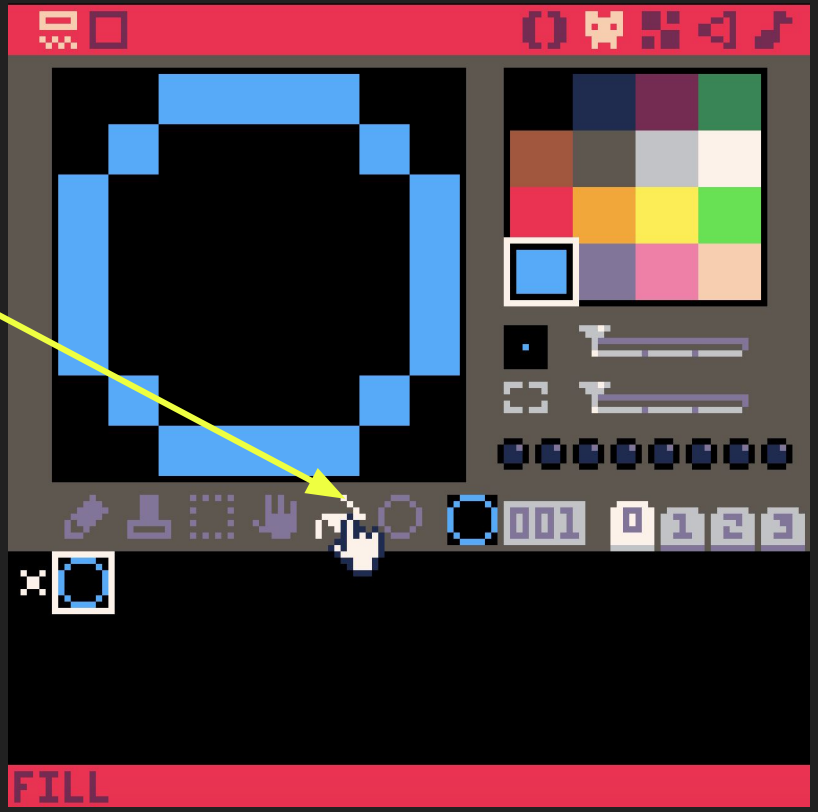
Drawing Sprites

- 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



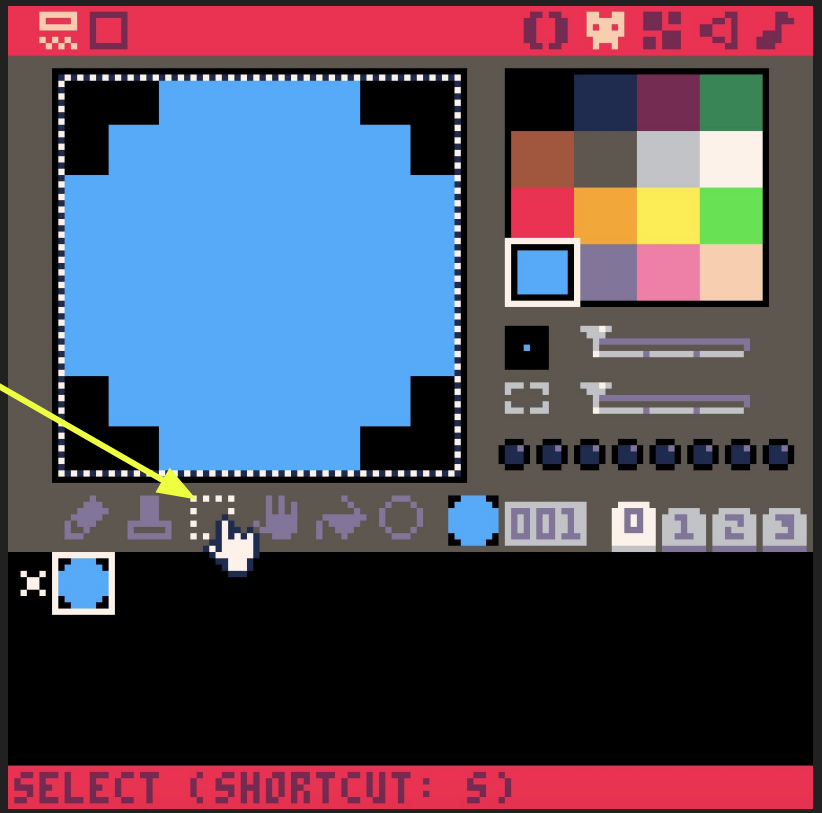
Drawing Sprites

- You can also fill an area using the Fill Tool



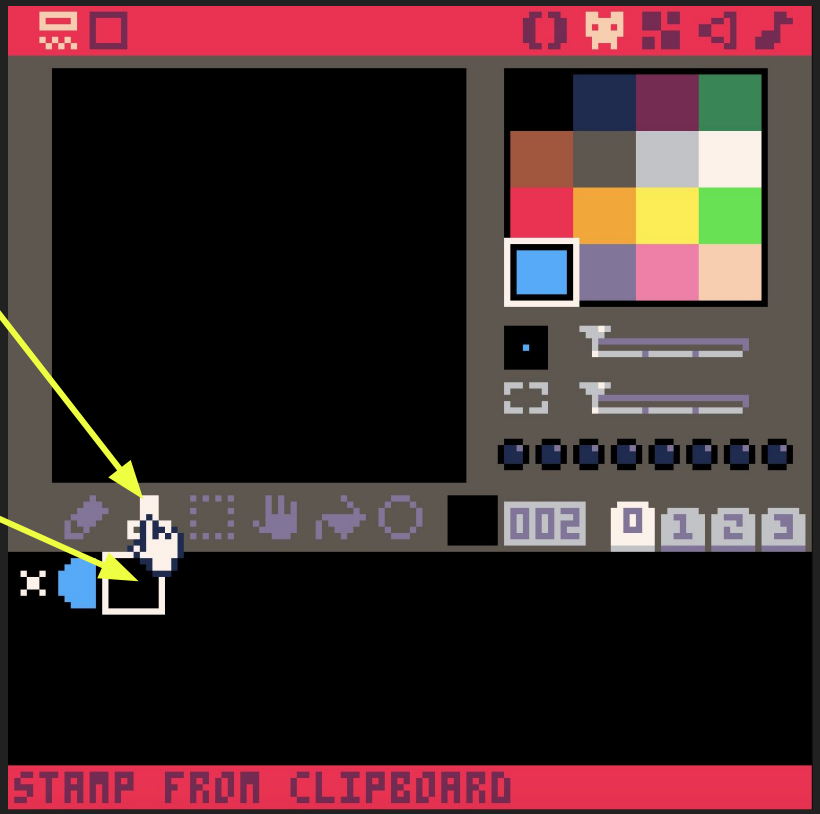
Drawing Sprites

- 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



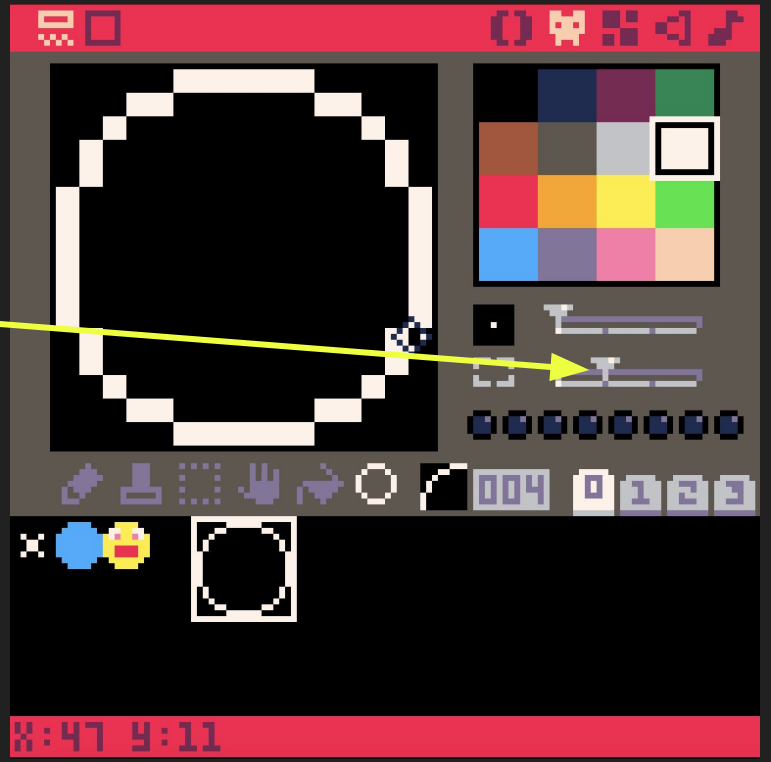
Drawing Sprites

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

Drawing Sprites

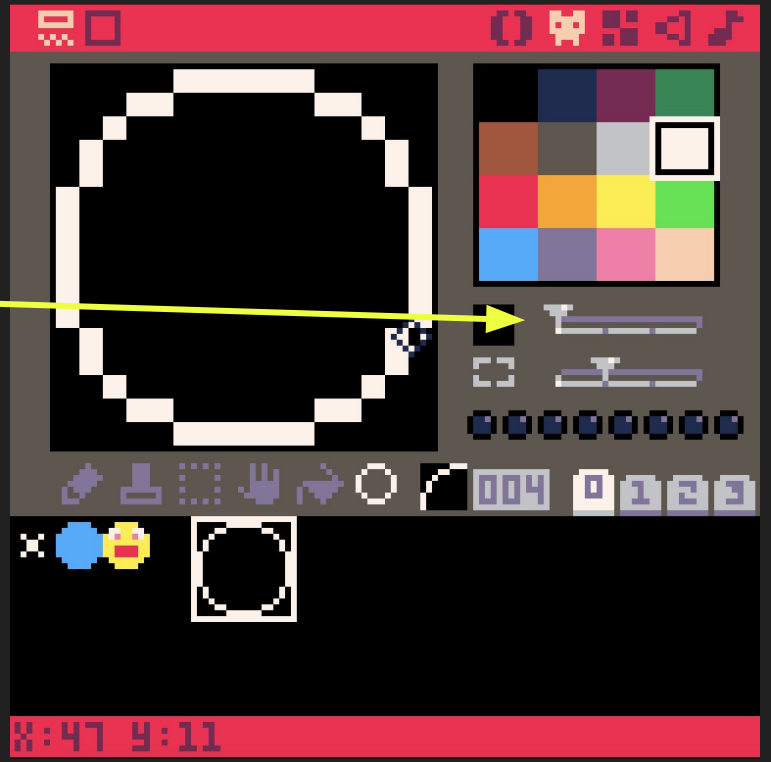
- Use this slider to change the size of your canvas

- This is a 16x16 pixel circle
- You can also make 32 or 64 pixel sprites



Drawing Sprites

- Use this slider to change the size of your brush



Drawing Sprites

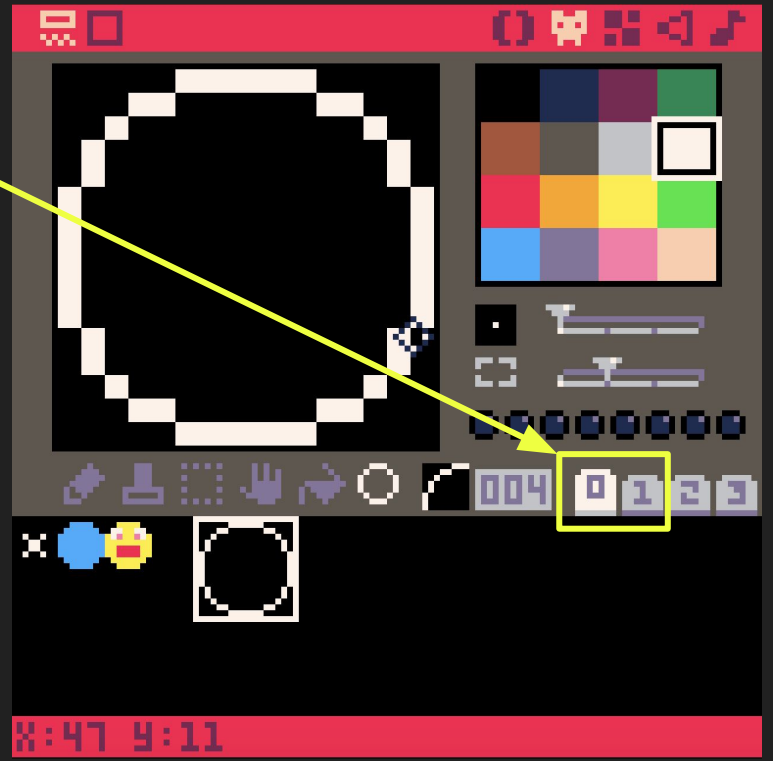
- All sprites you draw will occupy tiles on the sprite sheet

Drawing Sprites

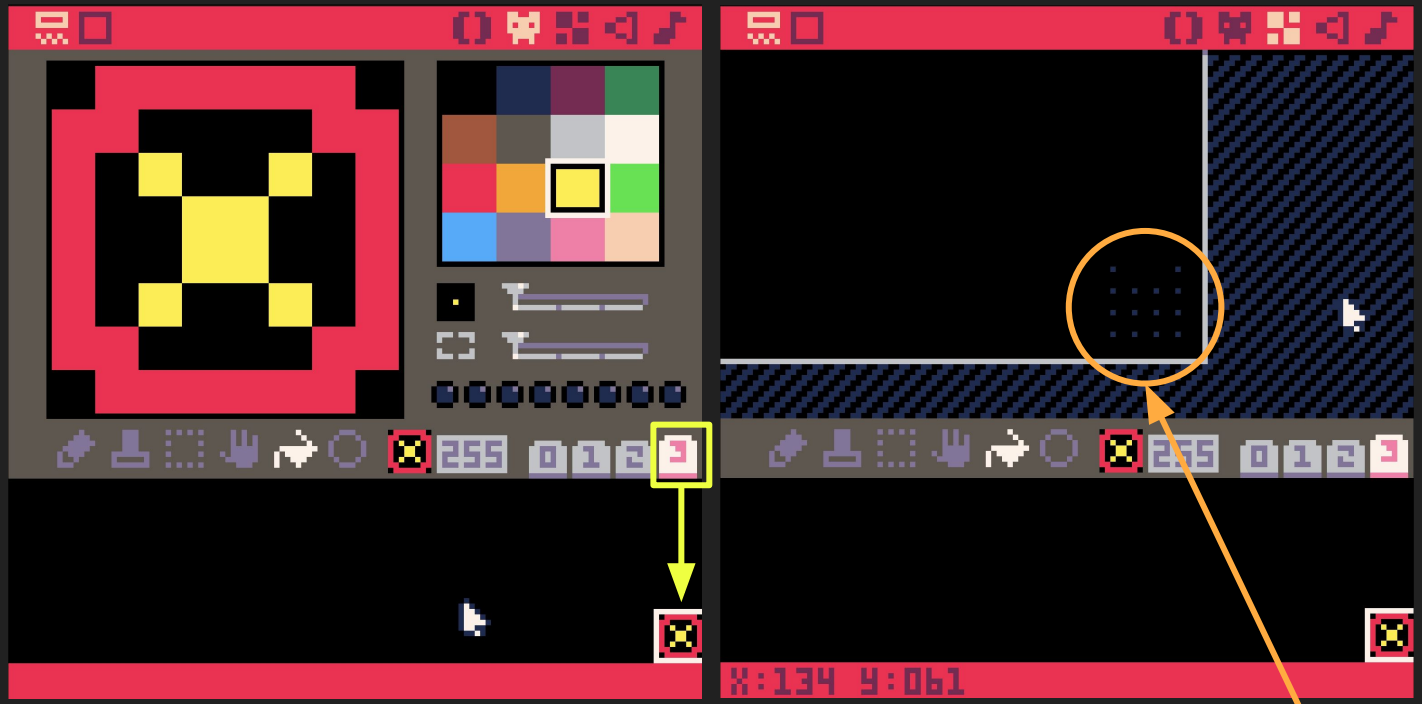
- You have a limited amount of space in your sprite sheet, so larger sprites means fewer sprites

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 Tabs 2 and 3 share memory with the map

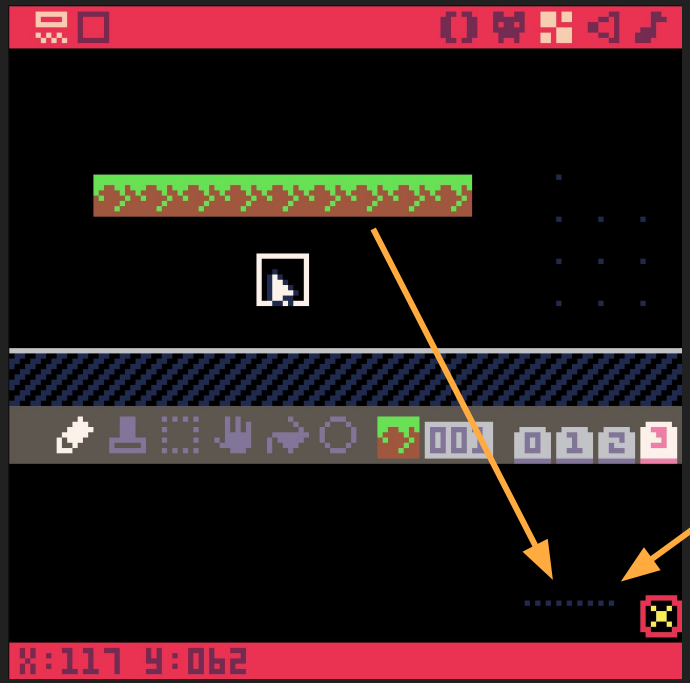


Drawing Sprites



If I draw a sprite on the very last spot on Tab 3 of the sprite sheet, you'll see the corresponding location in the map editor get corrupted

Drawing Sprites



Likewise, if I draw map tiles in the lower half of the map, the corresponding location in the sprite sheet (somewhere on Tabs 2 and 3) will get corrupted

Drawing Sprites

- If your game does not require a map, you could use all 4 tabs to draw sprites
- Just be aware that the map and sprite editors share some memory

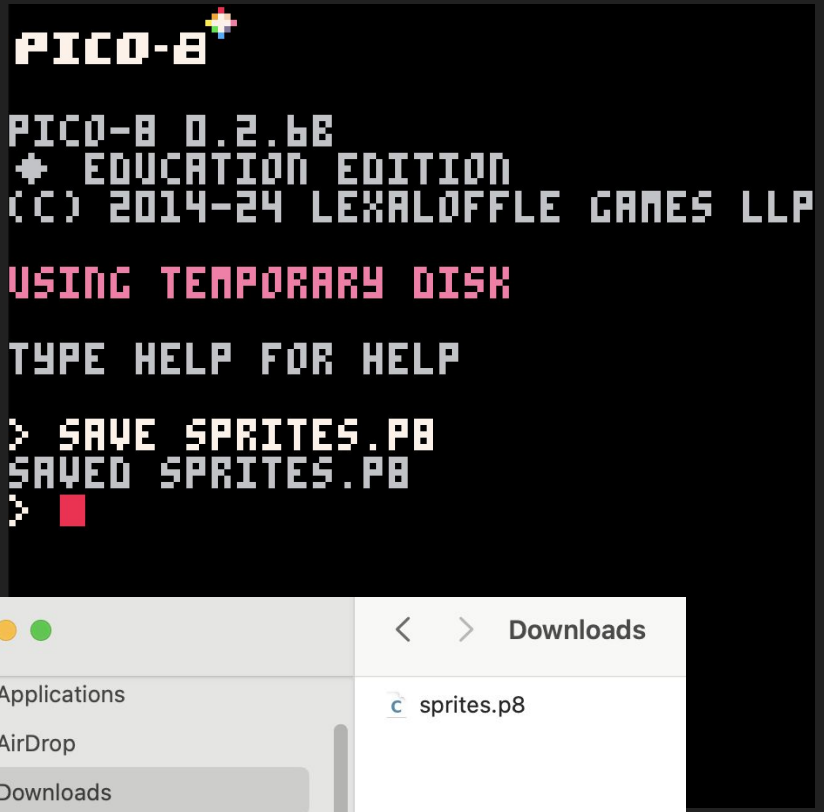
Saving Your Work

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

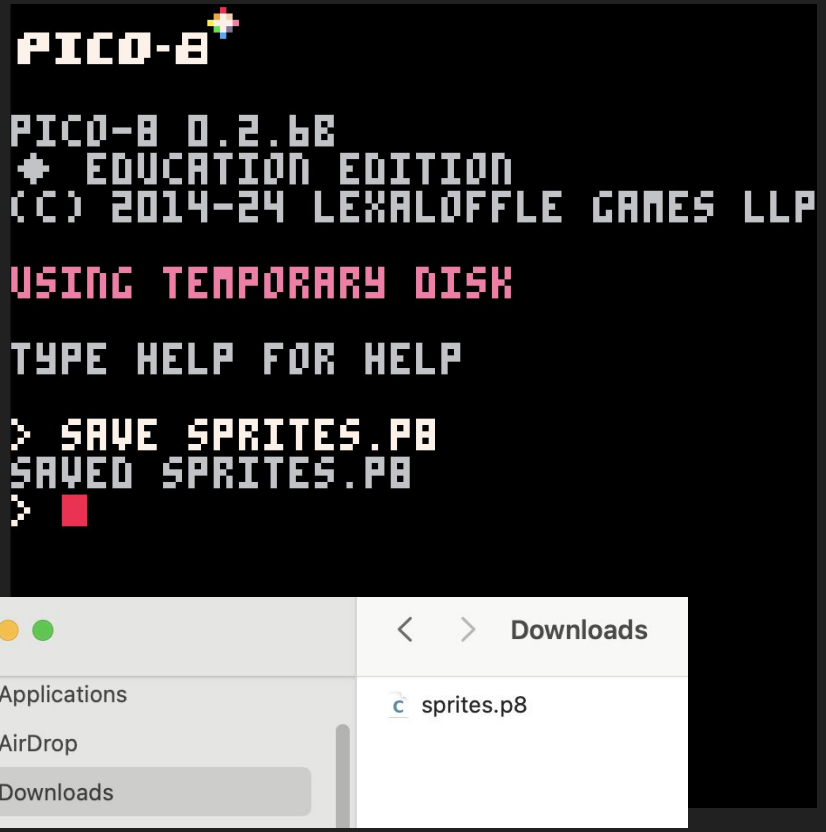
- Hit **Esc** and type the **SAVE** command to download your file
- You can load a file from your computer



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

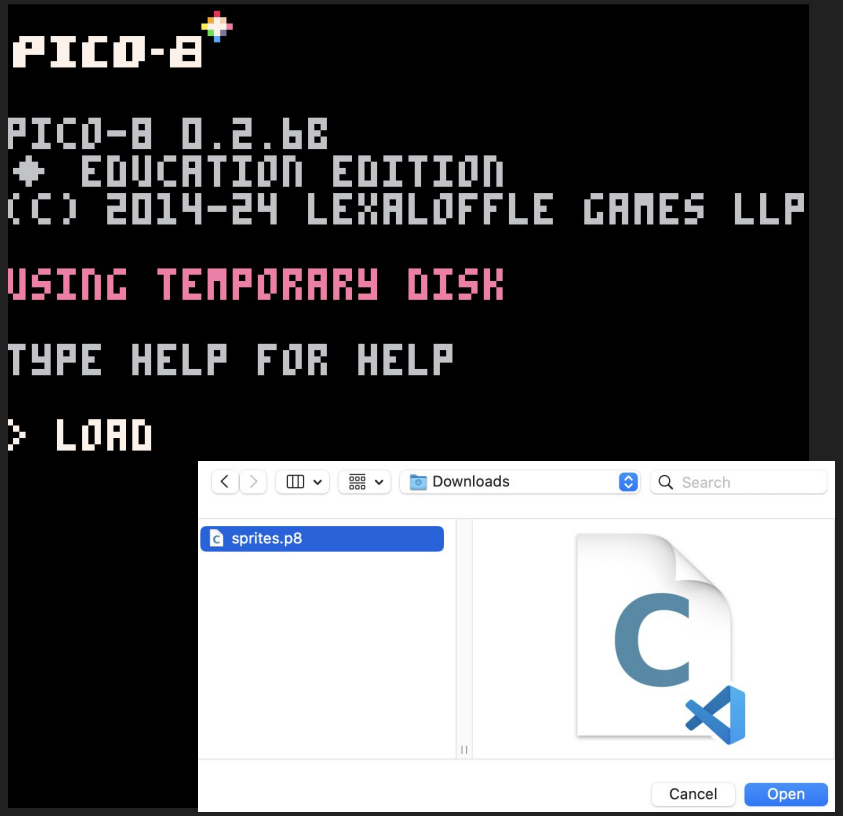
Saving Your Work

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



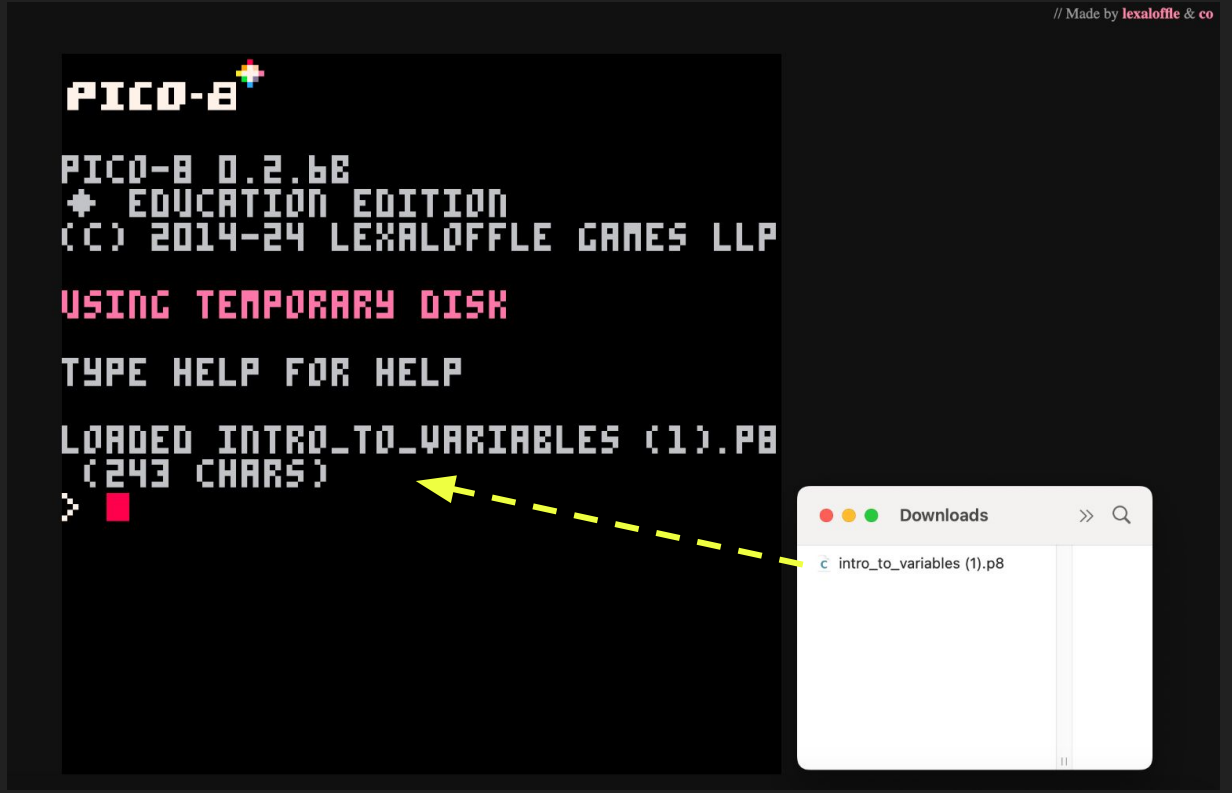
Loading a Saved P8 File

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



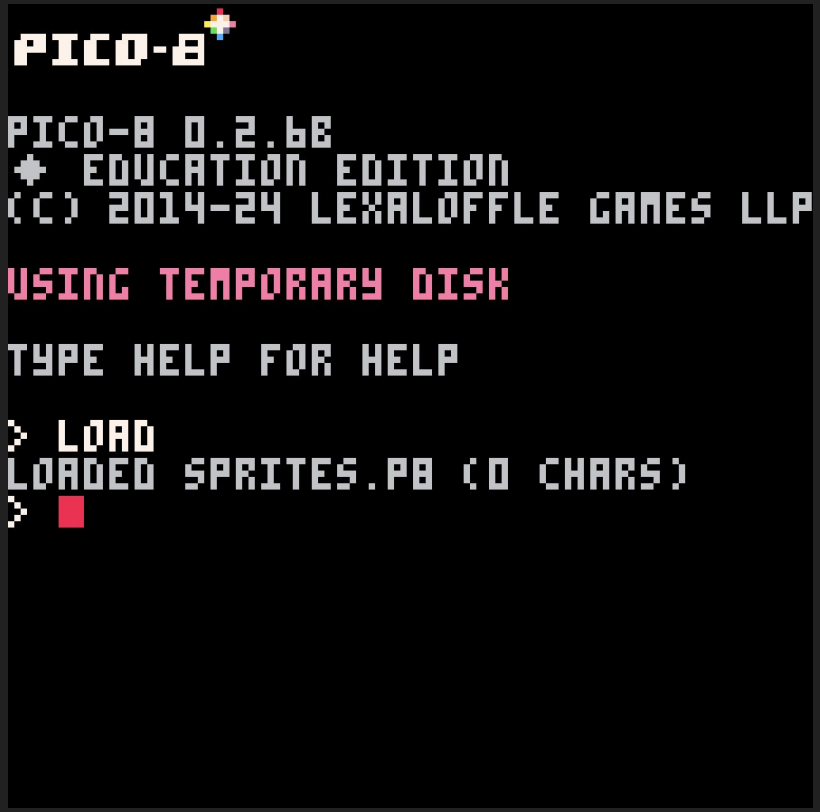
Loading a Saved P8 File

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



Loading a Saved P8 File

- *If you're using the desktop version of the software, use the* **FOLDER** *command to go to the location your computer where PICO-8 saves files*

Coding in PICO-8

Download Example Files:
_helloworld_00_intro_to_game_programming.zip

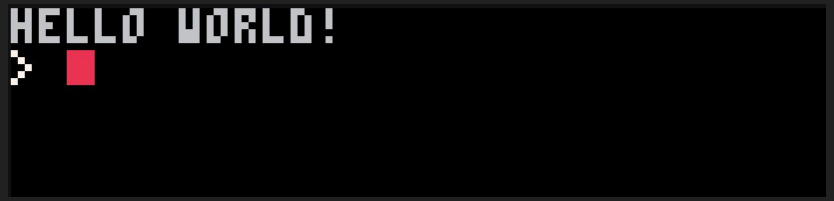
Coding in PICO-8

- PICO-8 uses the **Lua** programming language
- It's a bit lightweight, but it's comparable to other programming languages used in game development (and software development in general) – *everything you learn will transfer to other types of programming*

Download Demo File: intro_01_hello_world.p8

Coding in PICO-8

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



Download Demo File: intro_01_hello_world.p8

Coding in PICO-8

We can use PICO-8's **print()** function to do this

1. Type ***PRINT()***
2. Inside the parentheses, type ***"HELLO WORLD"*** *(include the quotations)*

You'll type in all caps automatically

Download Demo File: intro_01_hello_world.p8

Coding in PICO-8

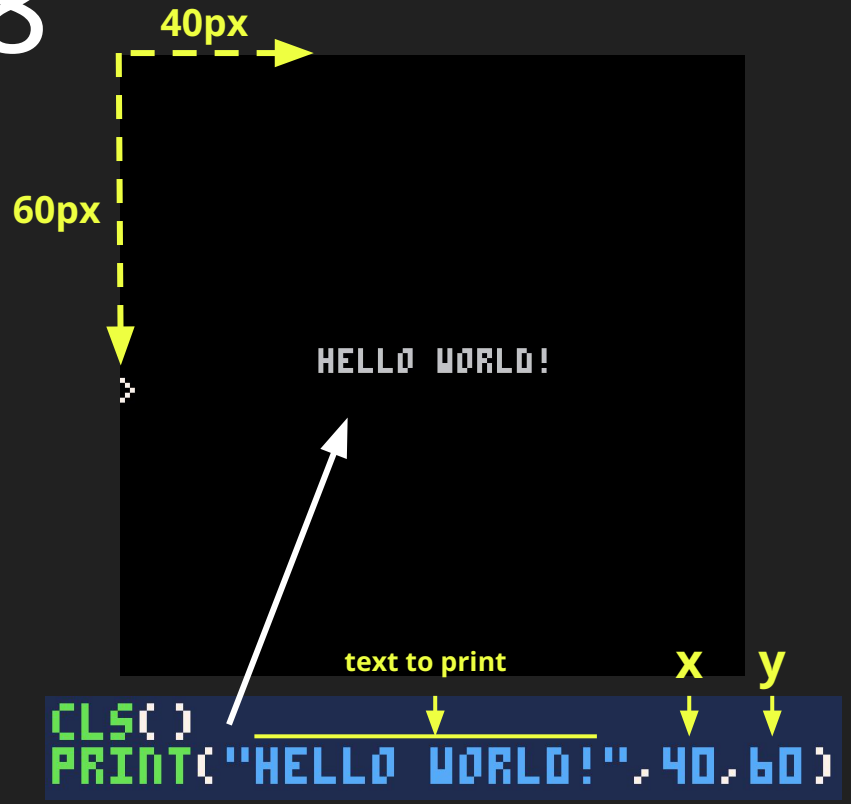
- If you'd like to hide the rest of the command line, type **CLS()** to clear the screen before printing your text

print() ***is also useful for displaying a game HUD***

Download Demo File: intro_01_hello_world.p8

Coding in PICO-8

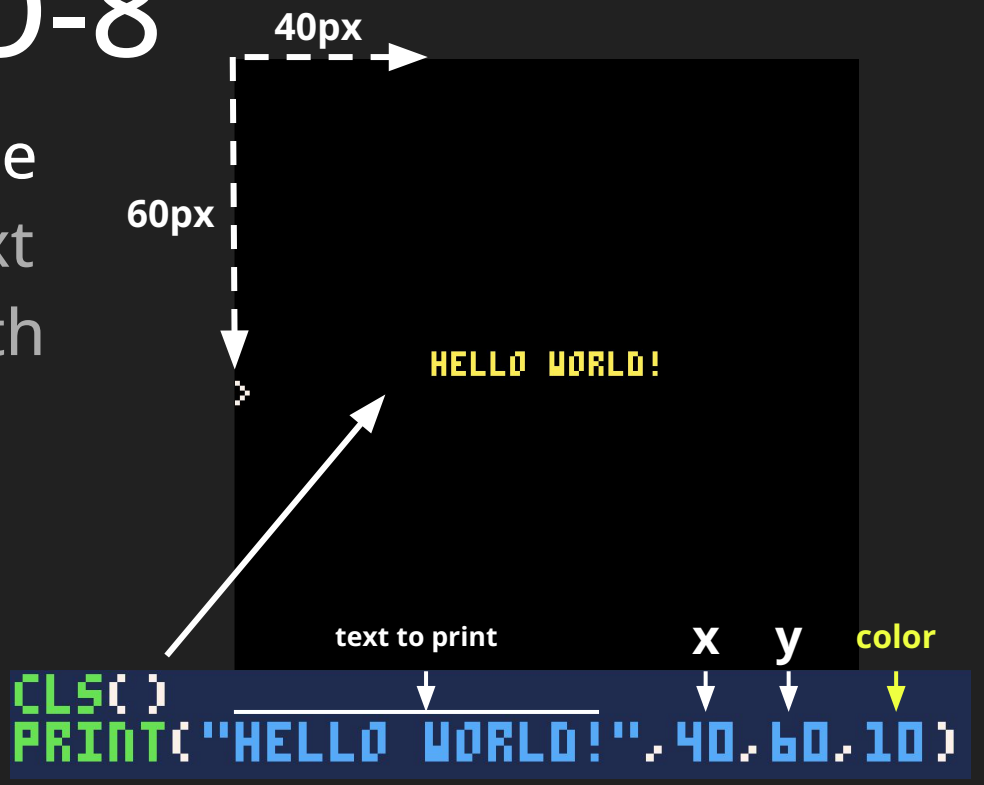
- We can add values after the quotes in our **print()** function, separated by commas, to set the position of the text
- Here, **40** is the **X** coordinate and **60** is the **Y** coordinate



Download Demo File: intro_02_coordinate_plane.p8

Coding in PICO-8

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



Download Demo Files: intro_03_text.p8 | intro_04_color.p8

Coding in PICO-8

- Each of the 16 available colors in PICO-8 corresponds with a number
- Refer to the PICO-8 Cheat Sheet

color

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

Download Demo Files: intro_03_text.p8 | intro_04_color.p8

Displaying Sprites in Your Game

Download Demo File: intro_06_sprites.p8

Displaying Sprites in Your Game

- 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

Displaying Sprites in Your Game

- A function is comprised of related lines of code
- You can write your own functions
- The PICO-8 engine has many predefined functions you can use, like **print()** and **spr()**

Displaying Sprites in Your Game

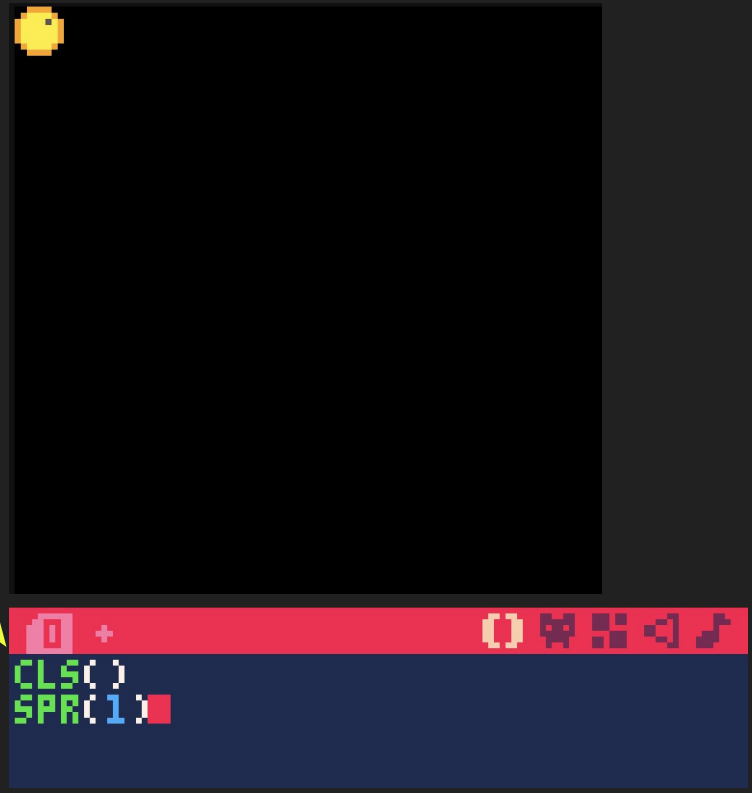
- We need to tell the computer *which* sprite to draw
- Each sprite in our sprite sheet has a number associated with it
- We can type the number 1 inside the parentheses – **spr(1)** to tell the program to draw sprite number 1

Displaying Sprites in Your Game

- 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

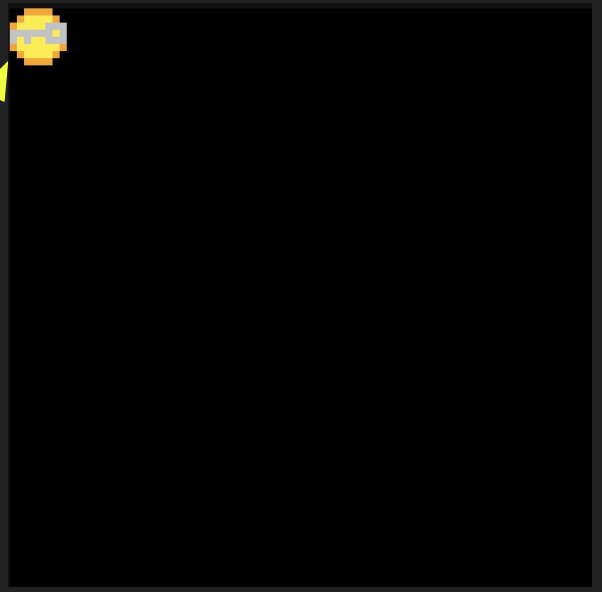
Displaying Sprites in Your Game

- We can hide the command line by typing **cls()** before we type **spr(1)**
- **cls()** is a function that clears the screen



Displaying Sprites in Your Game

- Notice what happens if I try to draw sprite 2 after sprite 1
- *Sprite 2 (the key) is drawn over sprite 1 (the player) because both are being drawn in the same location*



Displaying Sprites in Your Game

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

Displaying Sprites in Your Game

- The **spr()** function can be given values inside the parentheses to specify the sprite's position
- *These values must be given in the proper order*

Displaying Sprites in Your Game

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

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

PICO-8 Wiki

EXPLORE POPULAR PAGES COMMUNITY

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

PICO-8 Wiki

EXPLORE POPULAR PAGES COMMUNITY

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

PICO-8 Wiki

EXPLORE POPULAR PAGES COMMUNITY

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.

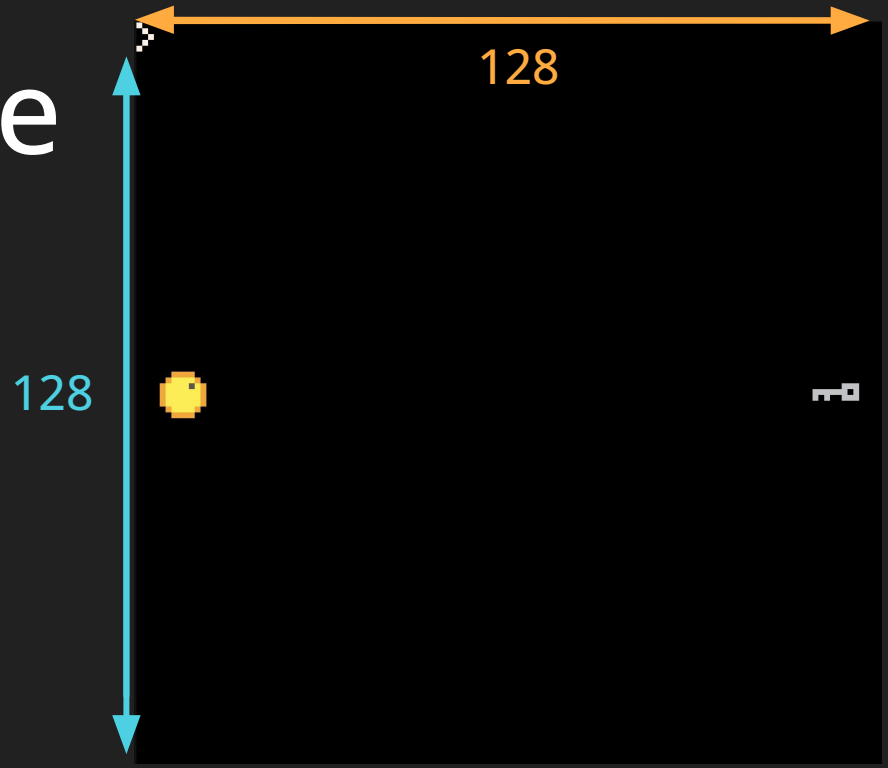
The wiki also explains what each value means and what its default value is

The PICO-8 Coordinate Plane

Download Demo File: intro_02_coordinate_plane.p8

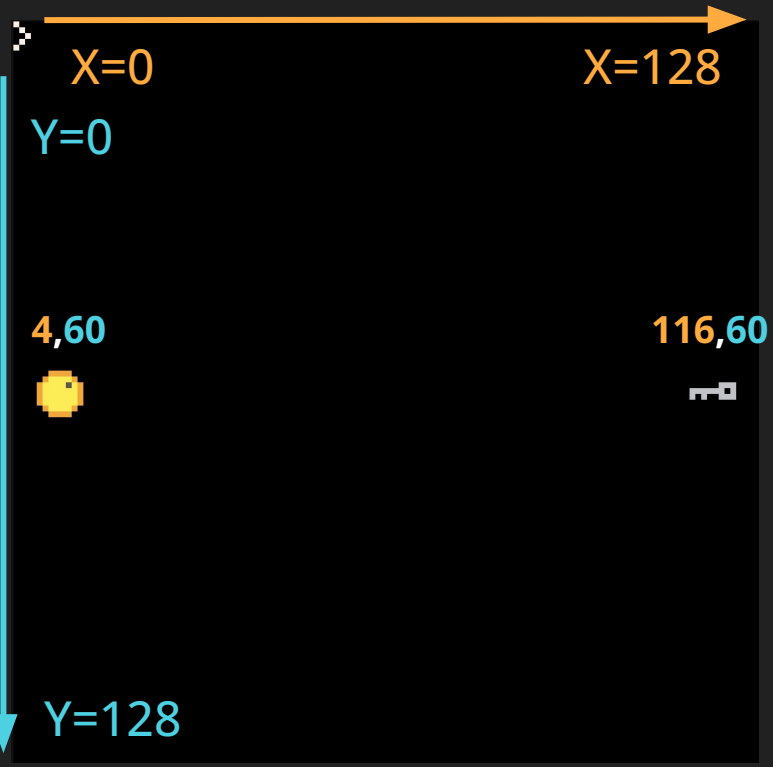
Coordinate Plane

- The PICO-8 game screen is **128 pixels square**



Coordinate Plane

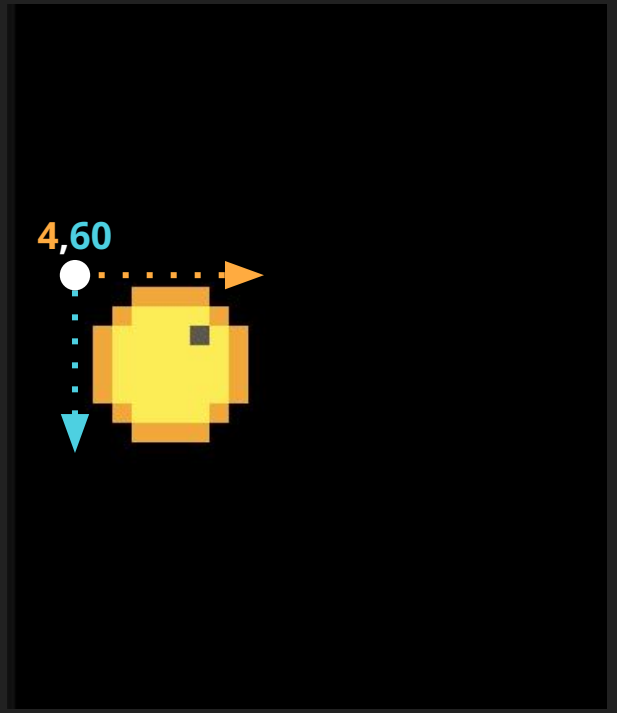
- The X,Y coordinate plane is drawn from left to right and top to bottom
- ***This means higher Y values are lower on the screen***

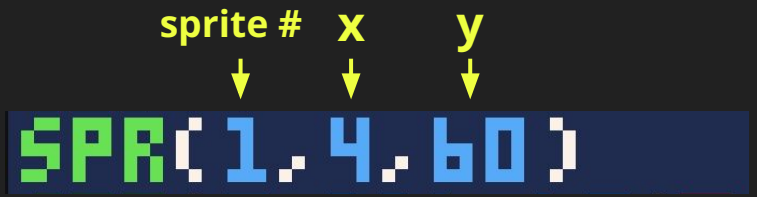


Coordinate Plane

- ***Graphics (such as text and sprites) are drawn from the top-left corner***

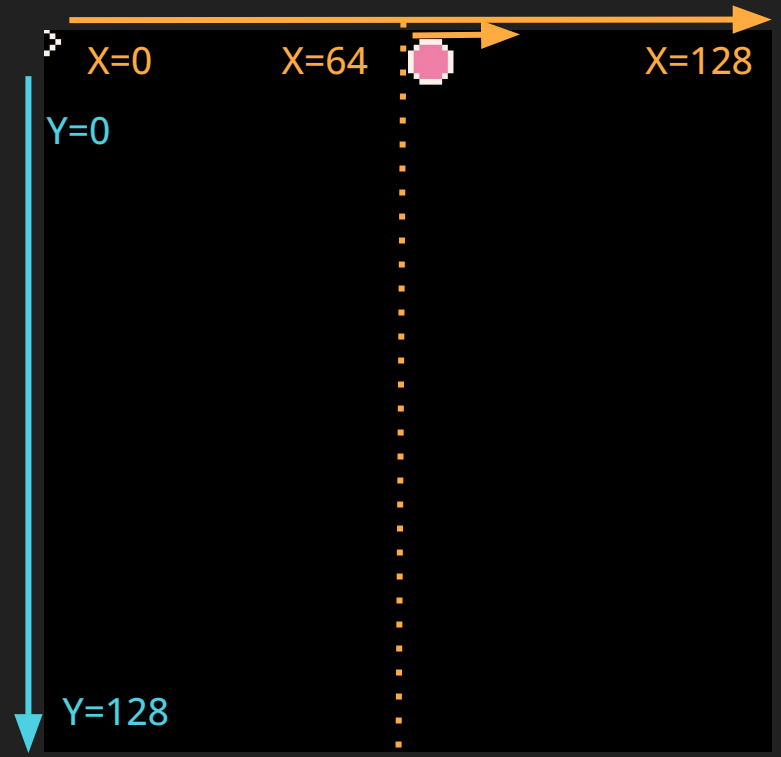
- Meaning the X,Y coordinates you assign an object coincide with its top-left corner





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)
```

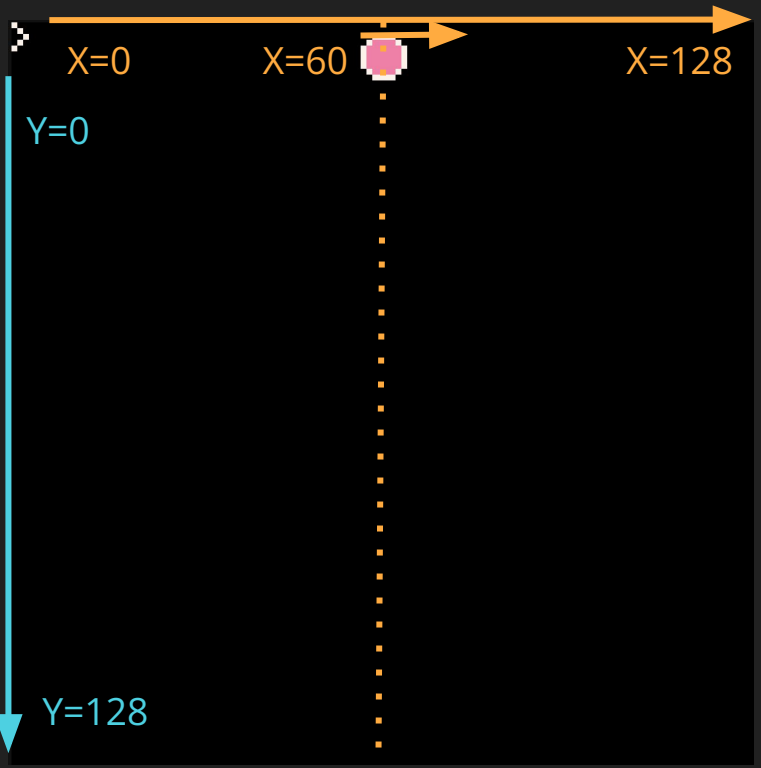
Coordinate Plane

- 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

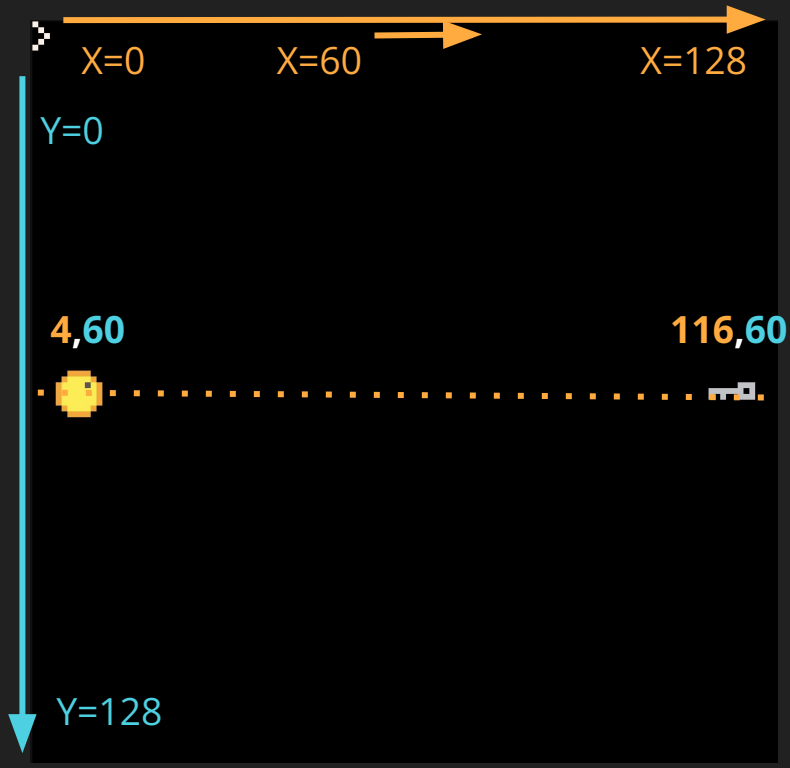
- When trying to center a sprite, you must account for half its width or height because of the top-left origin
- 64 - (8/2) = 64 - 4 = 60

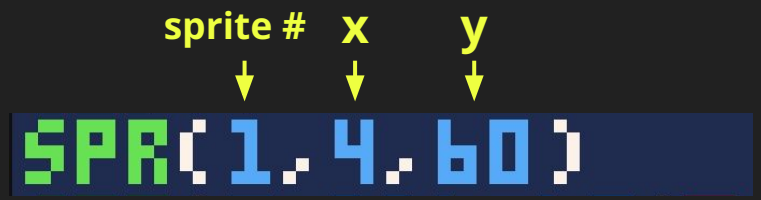



```
CLS()
SPR(1,60,2)
```

Coordinate Plane

- Likewise, a **Y** value of **60** (for an 8px sprite) instead of 64 will center the sprite vertically on the screen
- 64 - (8/2) = 64 - 4 = 60





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

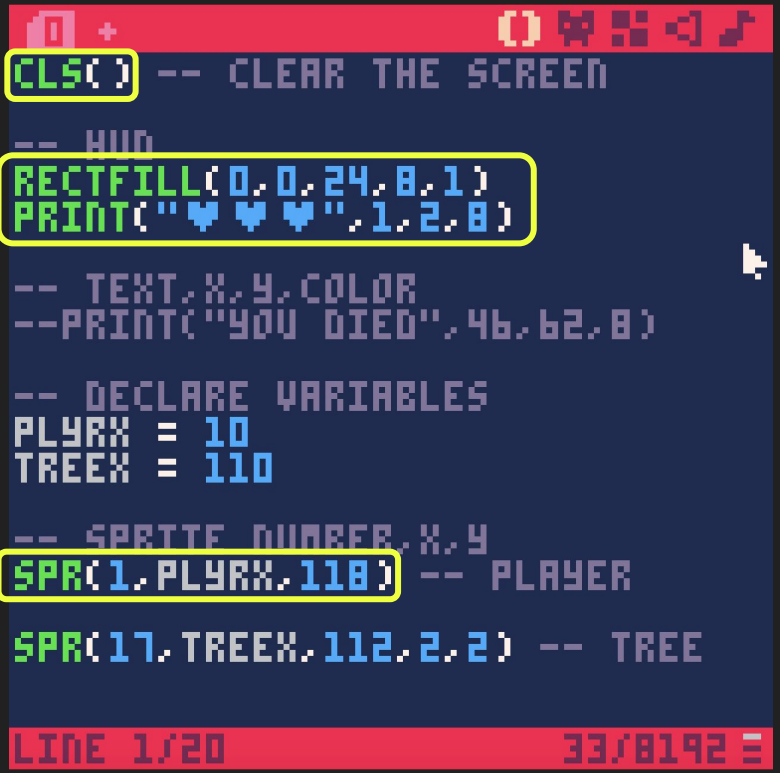
Before we can move our sprites in the game, we need to understand the way PICO-8 programs are structured

The PICO-8 Program Structure

Download Example File: intro_07_game_loop.p8

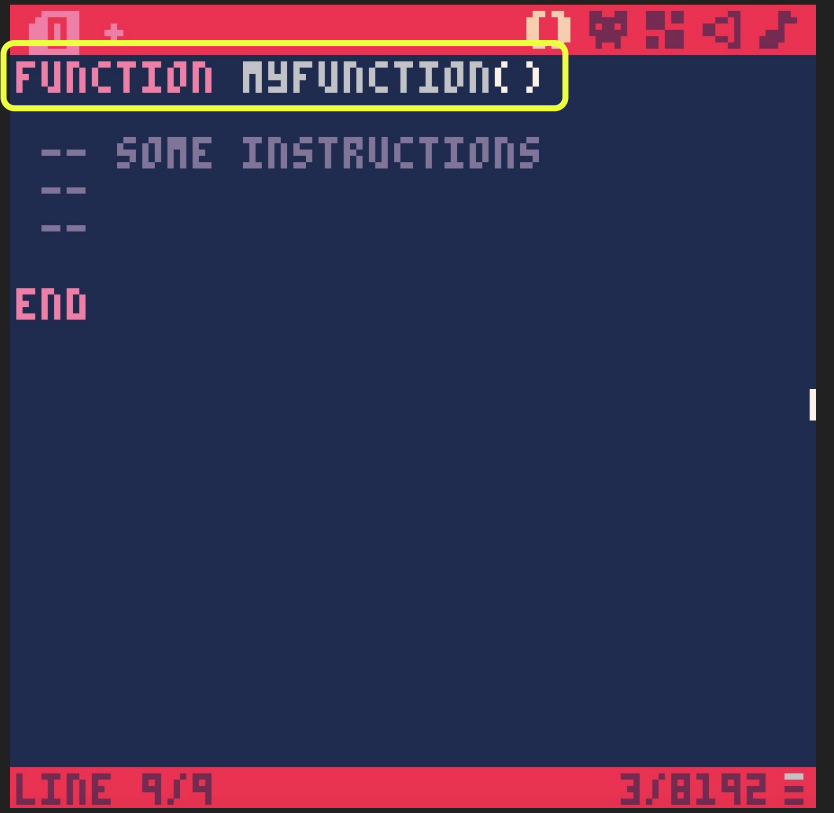
The PICO-8 Program Structure

- Programs often consist of sets of instructions called functions
- Some functions in PICO-8 have already been created for us – these are the keywords that turn green when you type them, like **cls()** and **spr()**



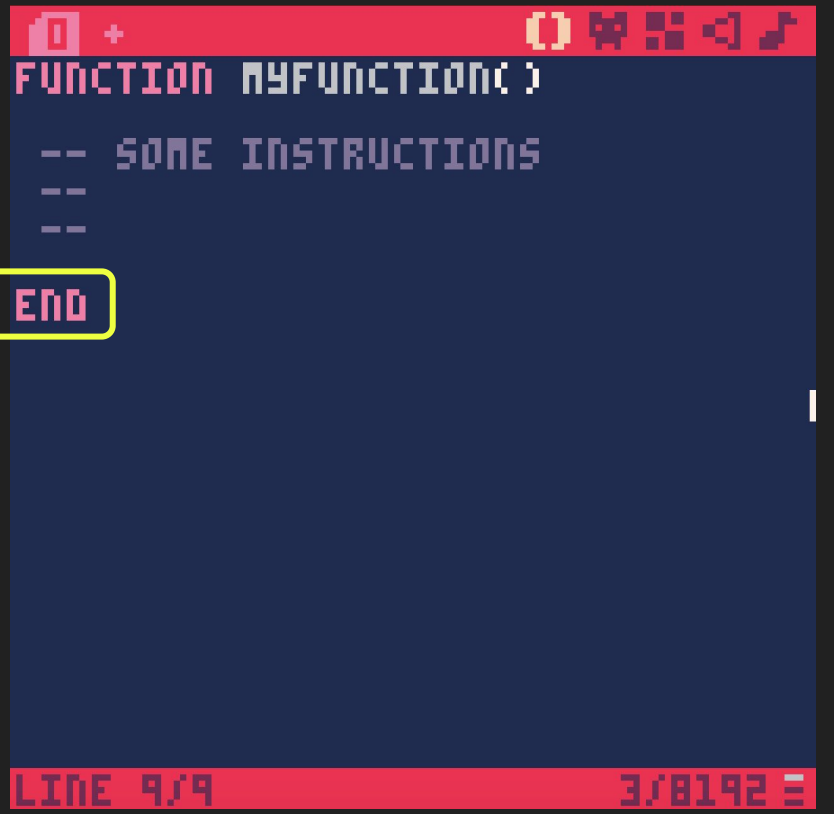
The PICO-8 Program Structure

- You can create your own functions using the keyword **function**
- Follow this with a name and a pair of parentheses *(the name should not contain any spaces, though you can use underscores)*



The PICO-8 Program Structure

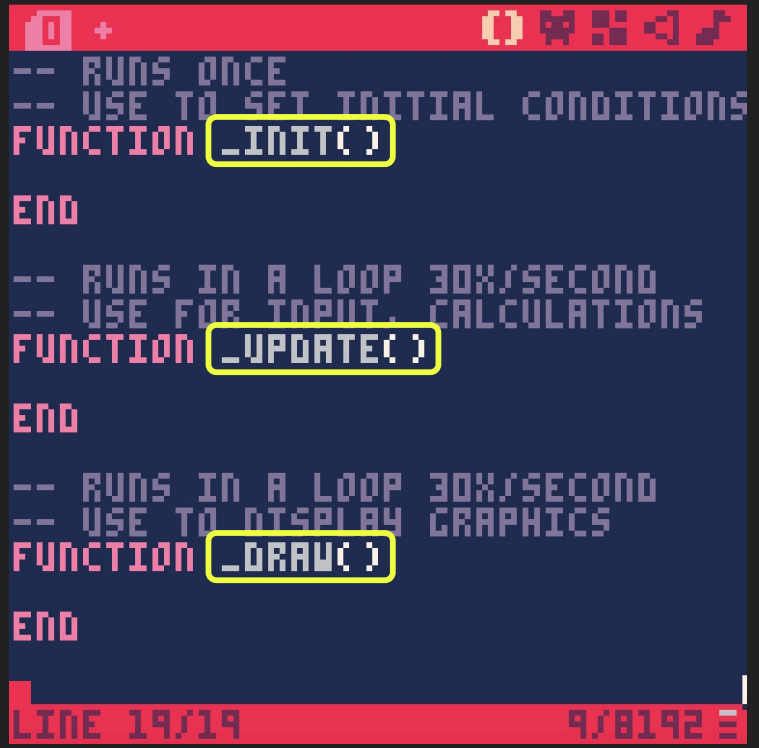
- All functions must be “closed” – do this using the keyword **end**
- Put all the instructions – the code that the function should execute – between the **function** line and the **end** line



*The Lua programming language uses **end** to close functions; other languages use different syntax, such as curly brackets { }*

The PICO-8 Program Structure

- There are some special function names that PICO-8 will recognize
- These function names are **_init()**, **_update()**, and **_draw()**
- *These three functions comprise the core structure of your PICO-8 program, called the "game loop"*



The PICO-8 Program Structure

- **_init()** Runs once at the start
  Use for setting initial conditions
  Defining variables & objects

- **_update()** Loops 30x per second
  Use for input & calculations

- **_draw()** Loops 30x per second
  Use to draw graphics

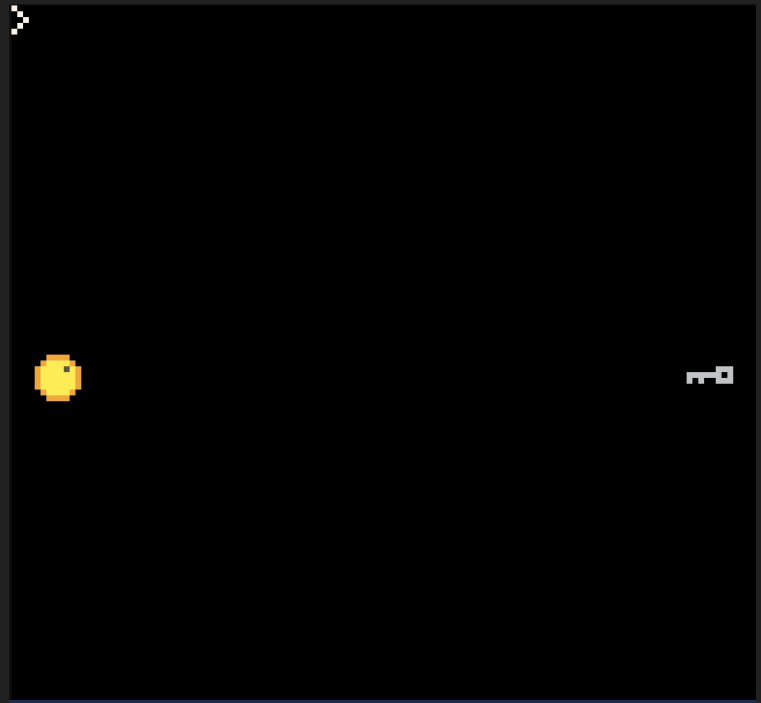
Once we have our game loop set up, we can make our sprites move using variables

Using Variables

Download Example File: intro_08_variables.p8

Using Variables

- *If we only use **fixed numbers** to position our sprites, they'll **never be able to move** from those places*
- This practice is called *"**hard-coding**"* (not advised)



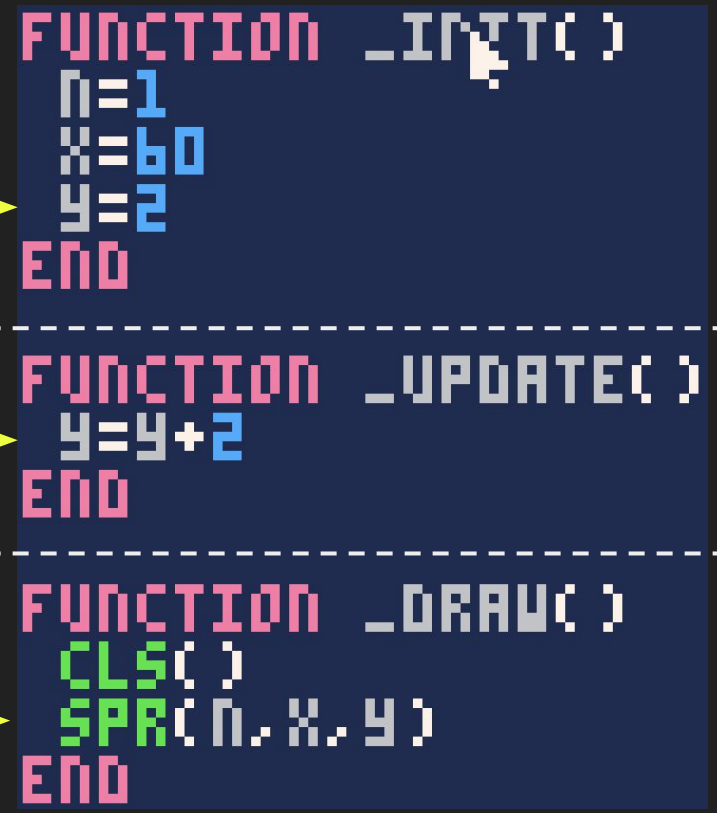
```
-- DRAW SPRITE NUMBER 1 (PLYR)
-- AT POSITION X=4, Y=60
SPR(1,4,60)

-- DRAW SPRITE NUMBER 2 (KEY)
-- AT POSITION X=116, Y=60
SPR(2,116,60)
```

Using Variables

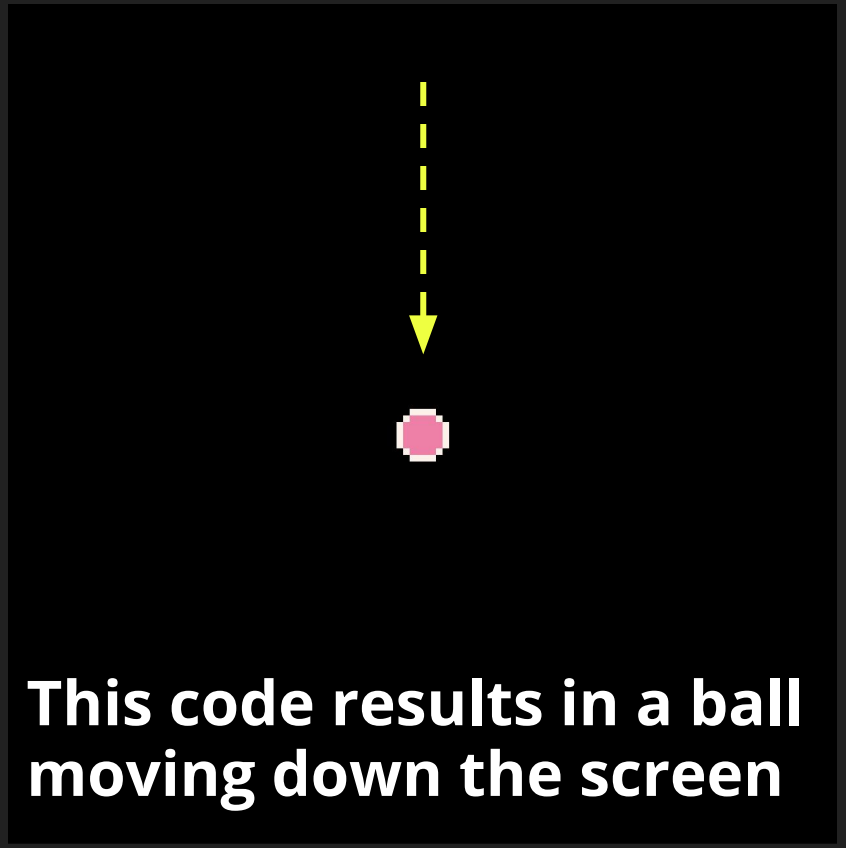
- We can use *variables* to represent numbers or other types of *values that can change over time*

Using Variables

- We can set our initial variable values in **_init()**
- Then we can change those values in **_update()**
- And apply those values in **_draw()**



Using Variables



This code results in a ball moving down 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
```

How the Game Loop Affects Variable Values

Variables and the Game Loop

- **_init()** only runs *once*, as soon as the game starts
- This is where you define the starting value for each of your variables

- If variable values need to change, that will be done in **_update()**

Variables and the Game Loop

- **_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
```

Variables and the Game Loop

Frame	y
0	2
1	4
2	6
3	8

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

Variables and the Game Loop

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 (2px)*

```
function _init()
 n=1
 x=60
 y=2
end

function _update()
 y=y+2
end

function _draw()
 cls()
 spr(n,x,y)
end
```

Variables and the Game Loop

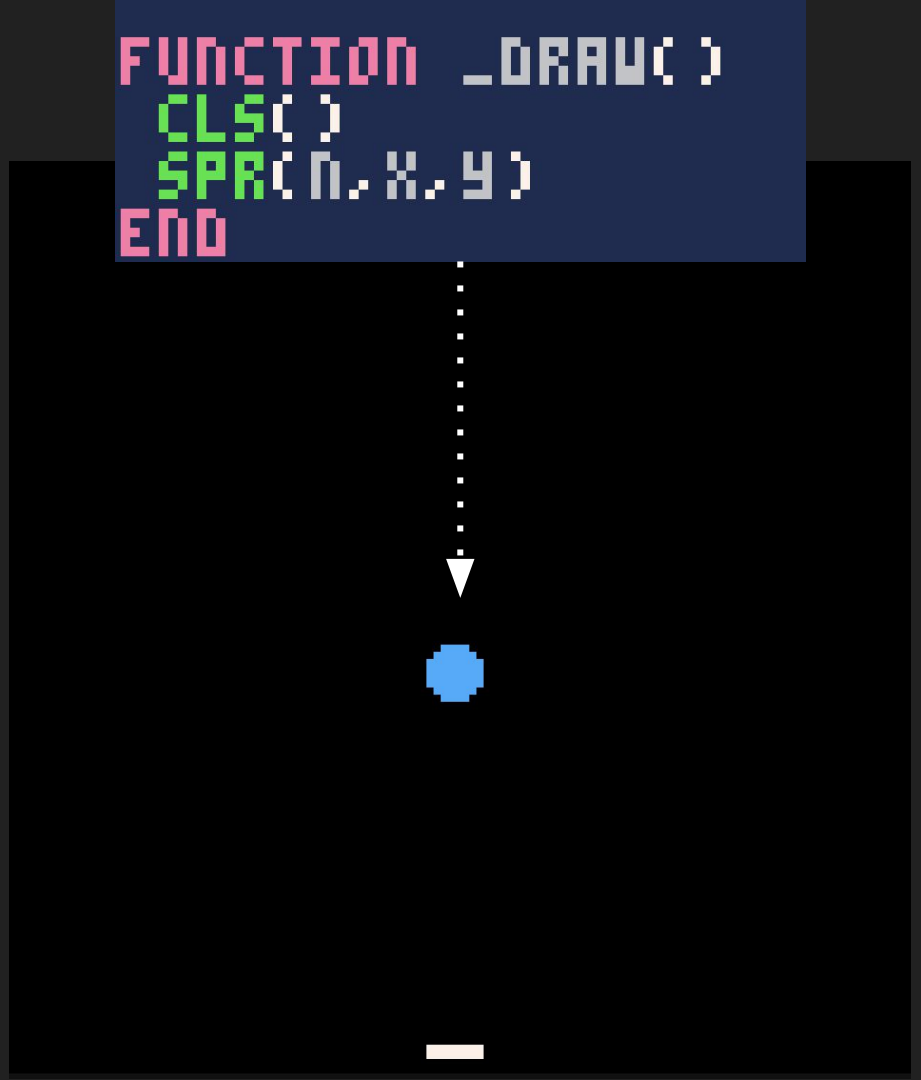
Frame	y
0	2
1	4
2	6
3	8
30 (1 second)	62

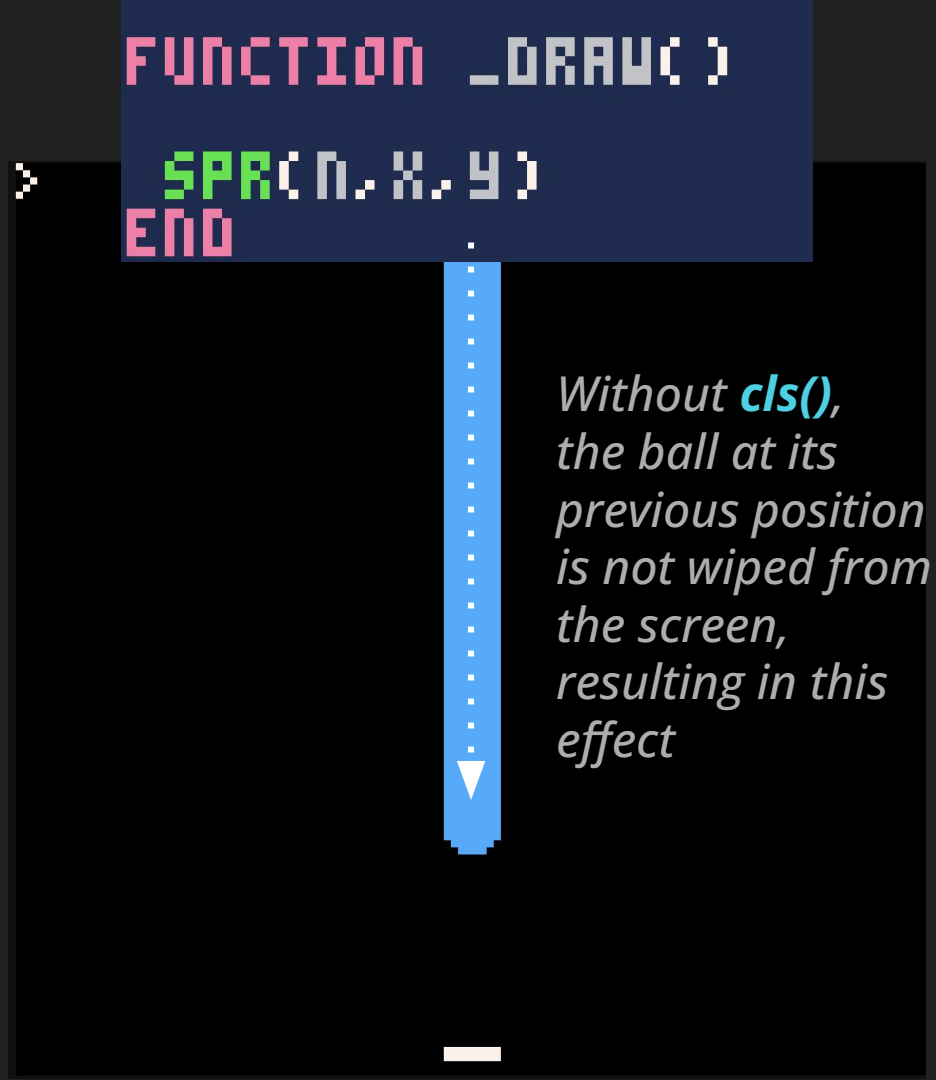
After 1 second (30 frames), the ball will have moved 60 pixels beyond its initial position (from 2 to 62)

```
function _init()
 n=1
 x=60
 y=2
end

function _update()
 y=y+2
end

function _draw()
 cls()
 spr(n,x,y)
end
```

Variables and the Game Loop

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





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

Variables and the Game Loop

- 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*

Variables and the Game Loop

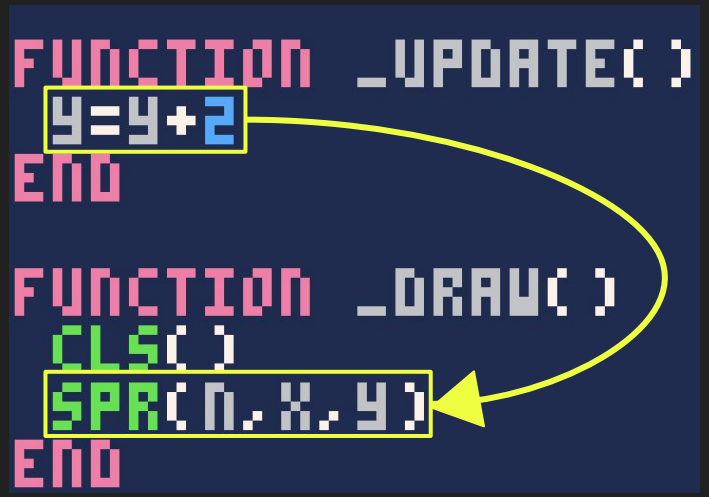
Frame	Effect
1 (Line 1 of _draw)	clear the screen
1 (Line 2 of _draw)	**draw the ball**
2 (Line 1 of _draw)	clear the screen
2 (Line 2 of _draw)	**draw the ball**
3 (Line 1 of _draw)	clear the screen
3 (Line 2 of _draw)	**draw the ball**

And so on . . .

```
FUNCTION _DRAW()
  CLS()
  SPR(0,X,Y)
END
```

Variables and the Game Loop

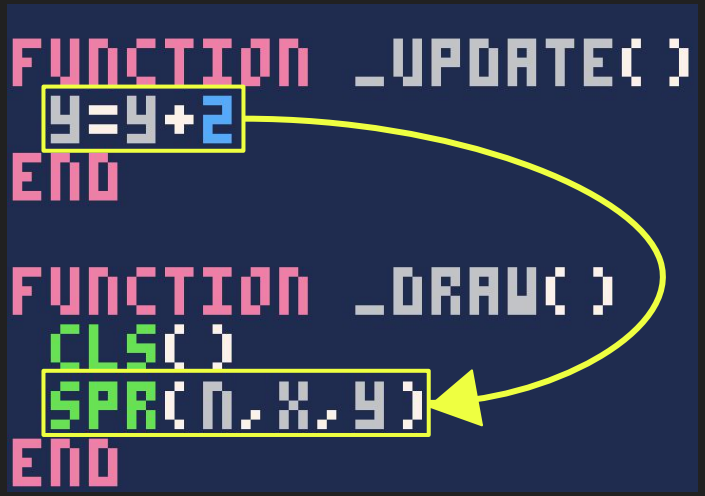
Frame	Effect
1 (Line 1 of _draw)	clear the screen
1 (Line 2 of _draw)	draw the ball at y=4
2 (Line 1 of _draw)	clear the screen
2 (Line 2 of _draw)	draw the ball at y=6
3 (Line 1 of _draw)	clear the screen
3 (Line 2 of _draw)	draw the ball at y=8



*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()***

Variables and the Game Loop

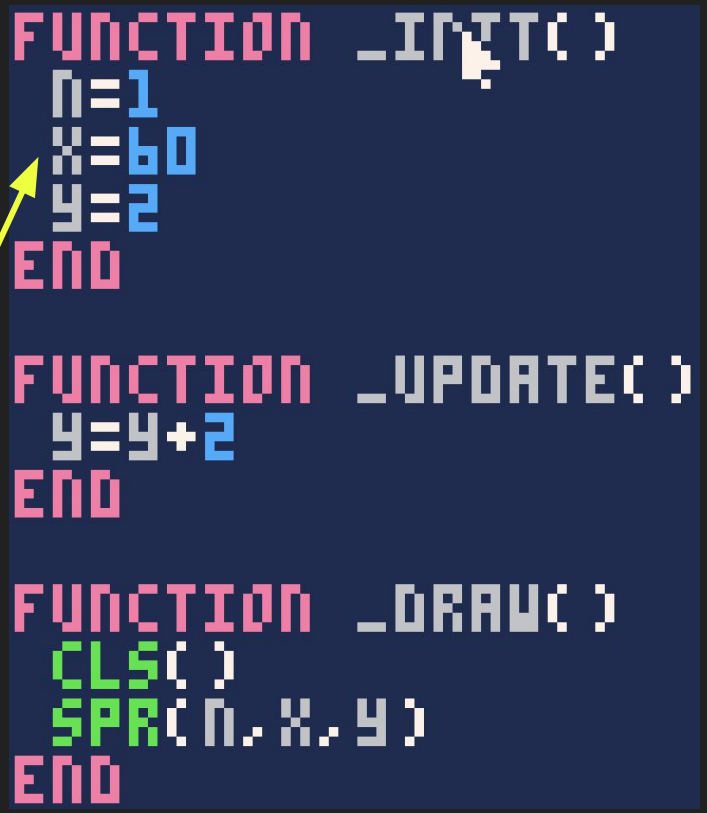
- **_update()** is where the value of **y** is changed
- Whatever the current value of **y** is, that's the value being fed into the **spr()** function in **_draw()** – ***this is what creates motion***



Rules for Using Variables

Rules for 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**



Rules for 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)
```

The program won't recognize **y** because *it was never defined in the first place*

Rules for 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

Rules for Using Variables

- Imagine if you were asked to explain what a ***zxcvbplkj*** is – you have never heard of this term before, so you don't recognize it
- It's the same with undefined variables in a computer program

```
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
```

The program won't recognize y because *it was never defined in the first place*

Rules for Using Variables

```
function _init()

 -- ball
 bal_n=1
 bal_x=60
 bal_y=2

 -- paddle
 pad_n=2
 pad_x=60
 pad_y=118

end
```

There are a few rules for 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***

Troubleshooting Errors

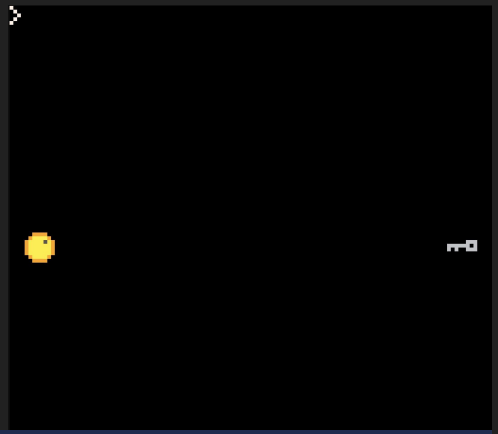
Most runtime errors are caused by simple typos; if you get an error, check:

- Are there any unclosed parentheses?
- Are there any missing **end** keywords?
- Are all your variables and function names spelled correctly and consistently?
- *Is the PICO-8 font disguising any typos?*

Let’s use variables to define our player sprite’s properties

Using Variables

- We can define the player's properties *(sprite number, x coordinate, y coordinate)* as variables in the **_init()** function (which runs once when the game starts)
- And then use those variables in our **spr()** function later on to refer to those values



```
function _init()
 n=1 -- sprite number
 x=4 -- x coordinate
 y=60 -- y coordinate
end
```

```
function _draw()
 cls()
 spr(n,x,y) -- player
 spr(2,116,60) -- key
end
```

Using Variables

- But **we'll need variables for the key in our game too**
- So we should ***name our variables for the player and the key in such a way that we can tell them apart***

If you change the name of a variable in one place (such as _init), you must also change the name accordingly anyplace else you use that variable (such as in _draw)

```
function _init()

 -- player variables
 plyr_n=1 -- sprite number
 plyr_x=4
 plyr_y=60

 -- key variables
 key_n=2 -- sprite number
 key_x=116
 key_y=60

end
```

```
function _draw()
 cls() -- refresh screen

 -- player
 spr(plyr_n,plyr_x,plyr_y)

 -- key
 spr(key_n,key_x,key_y)
end
```

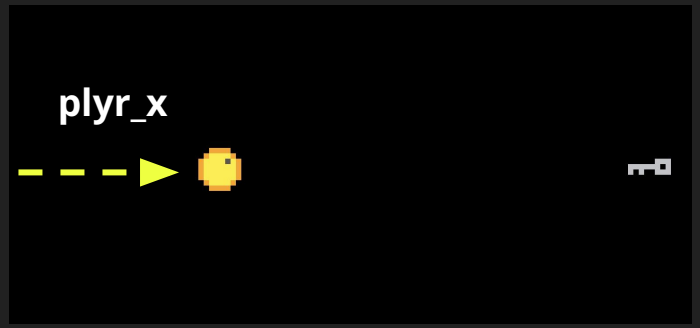
Detecting and Responding to Input

Download Example File: intro_09_move_player.p8

Detecting and Responding to Input

We can use **variables** to change the player sprite's position (making it move) when the player presses an arrow key

```
-- RUNS 30X/SEC
FUNCTION _UPDATE()

 -- MOVE RIGHT
 IF BTN(➡️) THEN
  PLYR_X = PLYR_X + 1
 END

END
```



*We'll do this in the **_update()** function, because it's always running*

Detecting and Responding to Input

The function **btn()** is used to check whether a button is being pressed currently

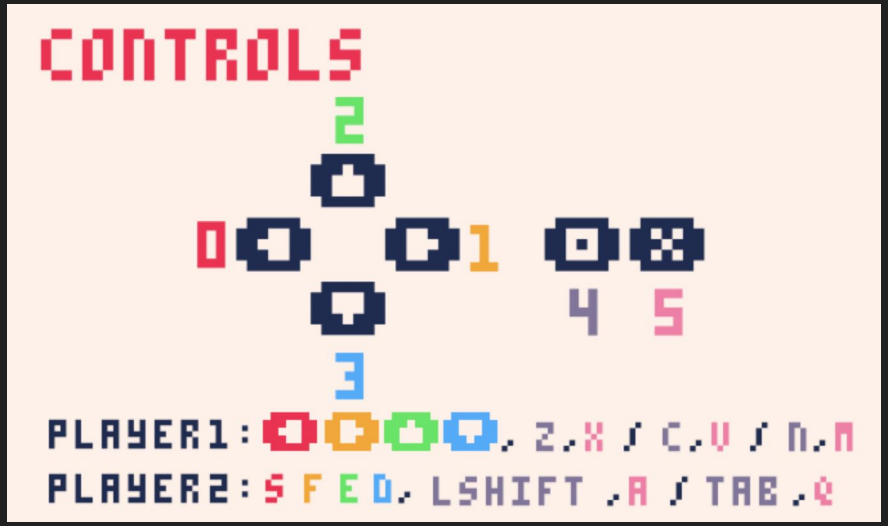
*There's also the function **btnp()**, which checks whether a key was pressed and released*

Detecting and Responding to Input

- Inside the parentheses for the **btn()** function, include the symbol for the button you're checking for
- You can create a left-arrow symbol by typing **SHIFT L**

SHIFT R = right arrow
SHIFT U = up arrow
SHIFT D = down arrow

- 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

```
-- RUNS 30X/SEC
FUNCTION _UPDATE()

 -- MOVE RIGHT
 IF BTN() THEN
  PLYR_X = PLYR_X + 1
 END

END
```

Detecting and Responding to Input

- **btn()** is used within what's called an *"if-statement"*
- Use the keyword **if** to begin an if-statement
- Follow it with the condition that you are checking for
  - *(in this case, that the left arrow key has been pressed)*

Detecting and Responding to Input

- After the condition (*the button press*), use the keyword **then**
- Followed by the effect that should happen if the condition is met
- *Finally, "close" the if-statement using the keyword **end***

*The **effect** in this case is that the **plyr_x** variable is **increased by 1**, moving the sprite to the right*

Anatomy of an If-Statement

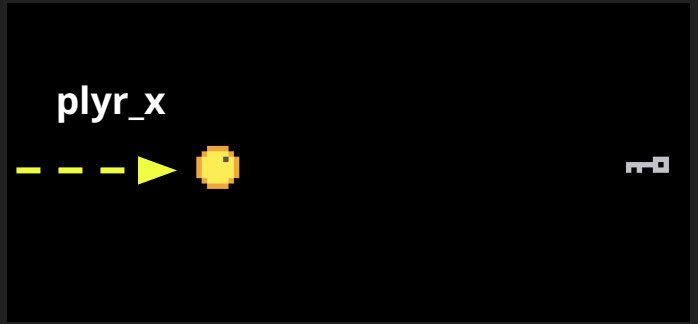
1. **if** (begins the if-statement)
2. the condition
3. **then** (separates the condition from the effect)
4. the effect (only happens if the condition is met)
5. **end** (closes the if-statement)

*You'll get a **syntax error** if you try to run your program without the keywords **then** or **end***

Detecting and Responding to Input

- This code will increase the player's x position by 1, ONLY if the right arrow key is pressed
- *If the condition specified after **if** is NOT met, the code between **then** and **end** will NOT execute*

```
-- RUNS 30X/SEC
FUNCTION _UPDATE()

 -- MOVE RIGHT
 IF BTN(➡️) THEN
  PLYR_X = PLYR_X + 1
 END

END
```

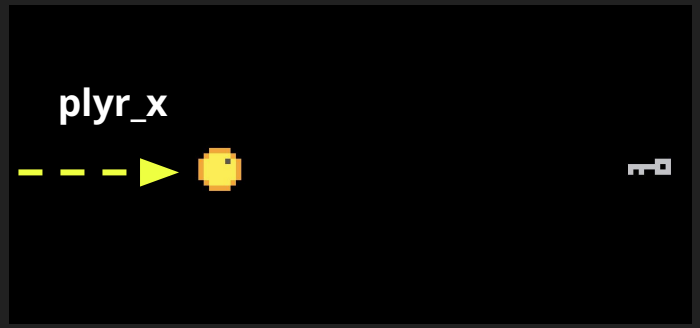


Detecting and Responding to Input

- This is how we write the expression that changes the value of plyr_x, adding 1 to it

```
-- RUNS 30X/SEC
FUNCTION _UPDATE()

 -- MOVE RIGHT
 IF BTN(➡️) THEN
  PLYR_X = PLYR_X + 1
 END

END
```

- *If plyr_x was equal to 4 at the start of the game, it will change to 5 after the right arrow key is pressed for 1 frame*
- *_update() runs 30 times per second, so this adds up over time*

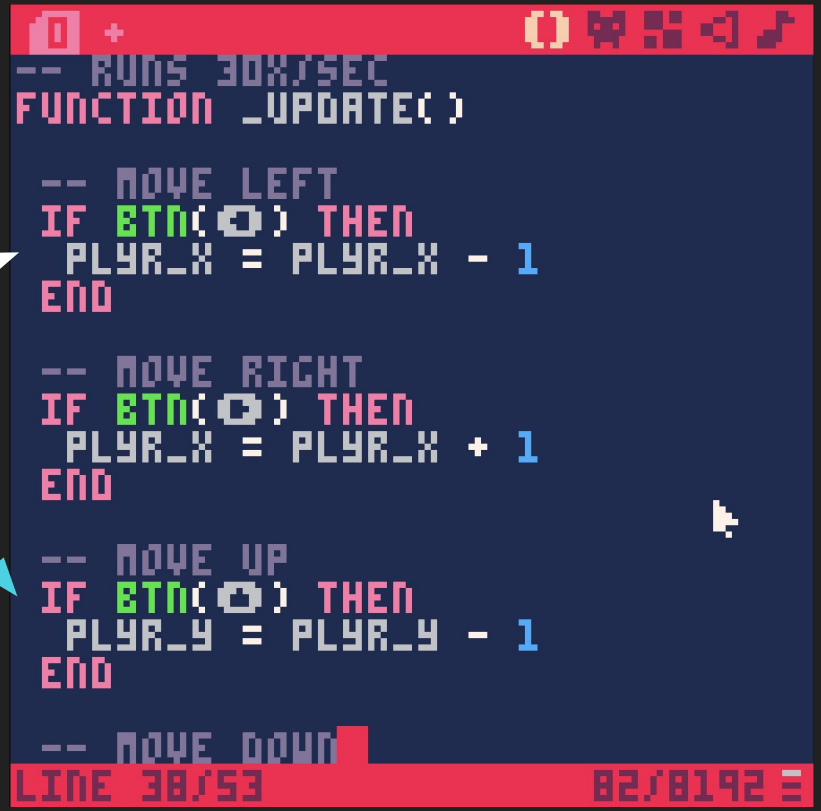


Detecting and Responding to Input

We can extrapolate this to the **other directions**:

- **Subtract one from the player's x coordinate** when the **left key** is pressed

- **Subtract one from the player's y coordinate** when the **up key** is pressed
- And **add one to the player's y coordinate** when the **down key** is pressed



Detecting and Responding to Input

- A **shorter way** of changing variable values is to use the syntax below:
- The arithmetic symbol (plus/minus) followed by an equals sign: **+=** or **-=**
- ***Both versions of this code will achieve the same result***

```
-- MOVE RIGHT
IF BTN(➡) THEN
 PLYR_X = PLYR_X + 1
END
```

```
-- MOVE RIGHT
IF BTN(➡) THEN
 PLYR_X += 1
END
```

Detecting and Responding to Input

- This practice is called incrementing a value
- And decrementing when decreasing a value

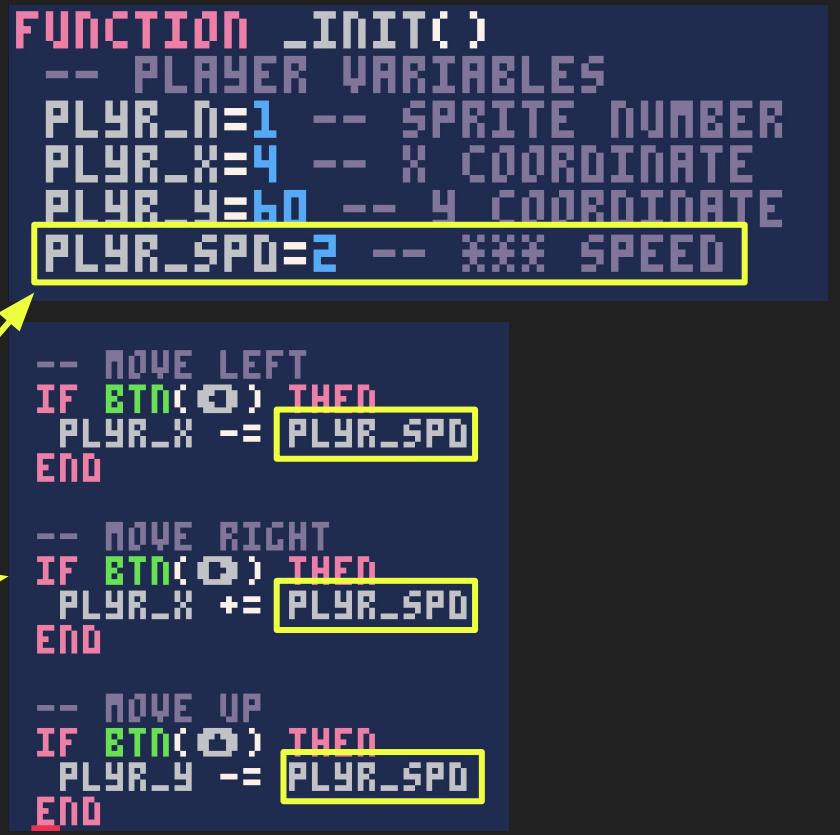
```
-- MOVE RIGHT
IF BTN(➡) THEN
 PLYR_X = PLYR_X + 1
END
```

- *You could also use asterisk-equals* ****=*** *for multiplication*
- *and slash-equals* ***/=*** *for division*

```
-- MOVE RIGHT
IF BTN(➡) THEN
 PLYR_X += 1
END
```

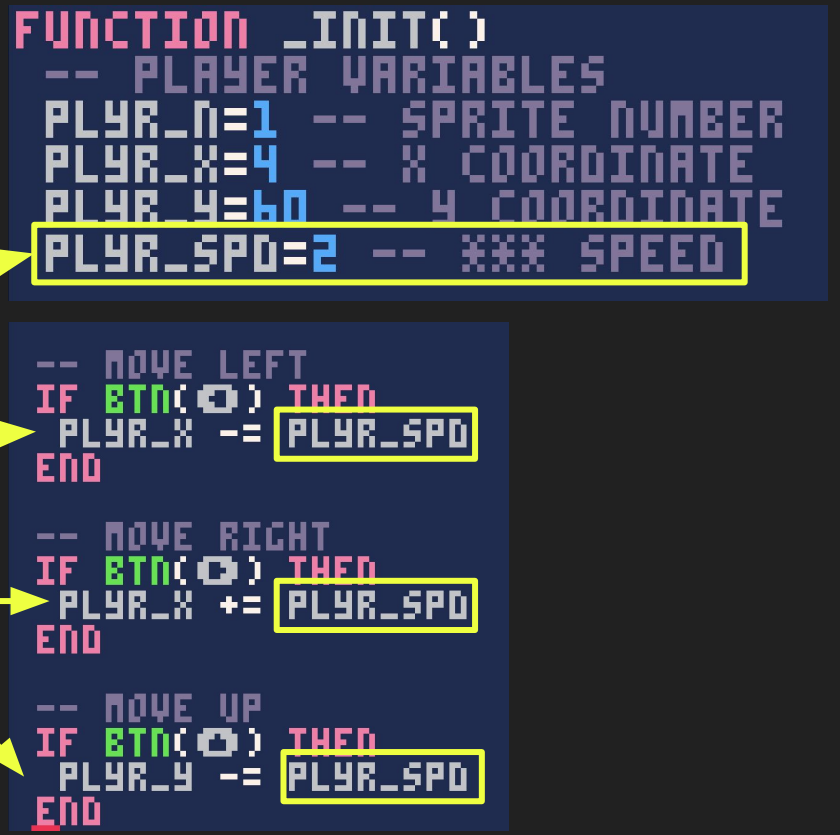
Detecting and Responding to Input

- If you want to try different values for the player's **speed**, it makes sense to *make that number a **variable***
- Define the player speed variable in **_init()**
- And use that variable to add to or subtract from the player's position



Detecting and Responding to Input

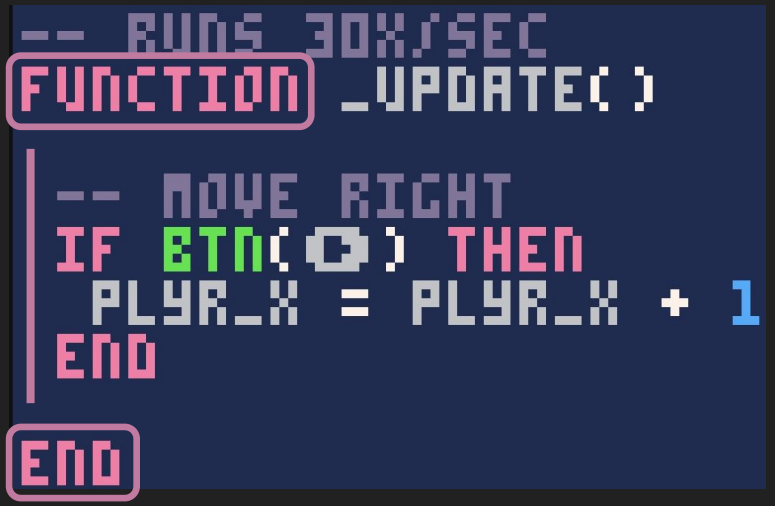
- *If you need to repeatedly change the speed value, you only have to change it in one place: in* ***_init()***
- And all the places you use that variable (such as in **_update**) refer back to that value



Keeping Your Code Organized

Keeping your Code Organized

- A good practice is to indent any code that's in a "block"
- A block is *several lines of related code*, such as a function or if-statement
- Blocks usually begin with a keyword (like **function** or **if**) and are closed by the keyword **end**



*Everything between **function** and **end** is a block*

Keeping your Code Organized

- Sometimes, you can have a *block within a block*, like this if-statement within a function
- This is called “nesting”
  - Named after nesting dolls

```
-- RUNS 30X/SEC
FUNCTION _UPDATE()

 -- MOVE RIGHT
 IF BTN(➡️) THEN
  PLYR_X = PLYR_X + 1
 END

END
```

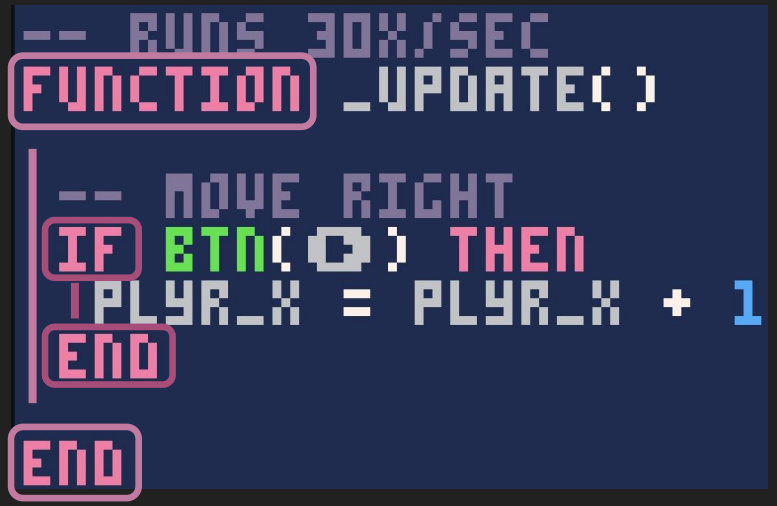
Keeping your Code Organized

- You can even *nest* an if-statement within another if-statement

```
IF BTN(➡) THEN
 PLYR_X += PLYR_SPD

 IF BTN(❎) THEN
  PLYR_SPD += 0.2
 END

END -- END IF
```

Keeping your Code Organized

When you have a block *nested* within another block, *indentation helps indicate where one block ends and another begins, making code more readable*



Keeping your Code Organized

- This is the same code, formatted with and without indentation
- You can see in the lower version how it becomes tricky to see which **end** closes which block
- Making troubleshooting difficult

```
-- RUNS 30X/SEC
FUNCTION _UPDATE()

 -- MOVE RIGHT
 IF BTN(➡️) THEN
  PLYR_X = PLYR_X + 1
 END

END
```

```
FUNCTION _UPDATE()
IF BTN(➡️) THEN
PLYR_X += 1
END
END
```

Keeping your Code Organized

- It can also be helpful to put comments after each **end** statement, indicating which block they close

*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

Keeping your Code Organized

- Comments are started with two dashes: --
  - or two slashes //
- Any text after the dashes or slashes is treated as separate from the executable code, so you can type whatever you want and write notes for yourself

Keeping your Code Organized

- You can even put a comment in front of a line of code to disable it and prevent it from executing

- This is called “**commenting out**” a line of code
- And is helpful if you are working with experimental code that you don’t want to delete altogether just yet

Writing Custom Functions

Download Example File: intro_10_functions.p8

Writing Custom Functions

- We've been using several built-in PICO-8 functions:
  - **cls()**
  - **print()**
  - **spr()**
  - **btn()**

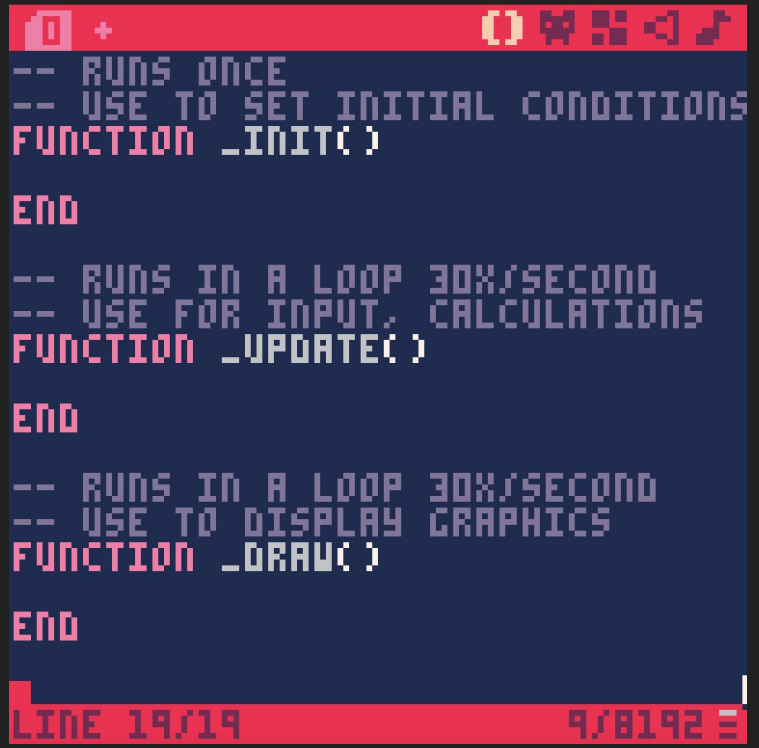
Functions are sets of instructions for a program to perform, written as multiple lines of code

These functions have been written into the game engine

But we can write our own functions, too!

Writing Custom Functions

- Technically, we've already been doing this with **_init()**, **_update()**, and **_draw()**
- But these function names just happen to be recognized by the game engine for specific purposes



Writing Custom Functions

- We can follow a similar practice and use the keyword **function** to write our *own* functions
- For example, we could write a function called *make_plyr()* that declares all of the player variables

```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
 PLYR_N=1 -- SPRITE NUMBER
 PLYR_X=4 -- X COORDINATE
 PLYR_Y=60 -- Y COORDINATE
 PLYR_SPD=1 -- SPEED
END -- /FUNCTION MAKE_PLYR()
```

We could name this custom function (almost) anything we want

We could even name it *yourmom()*

But it makes sense to name it in a way that describes accurately what it does

Writing Custom Functions

There are some **rules** for writing functions:

- The **function** keyword must be used first to *declare* the function
- The function name must be all one word (use underscores instead of spaces)
- Often, the first character of the name must not be a number or symbol (though PICO-8 gives us some flexibility with symbols)
- The name must end with parentheses (no spaces between the rest of the name and the parentheses)
- The function must be *closed* using the keyword **end**
- A function should not be declared within another function

Anatomy of a Function

1. **function** (declares the function)
2. the name
3. parentheses at the end of the name

```
FUNCTION MAKE_PLYR()

  PLYR_N=1 -- SPRITE NUMBER
  PLYR_X=4 -- X COORDINATE
  PLYR_Y=60 -- Y COORDINATE
  PLYR_SPD=1 -- SPEED

END -- CLOSE THE FUNCTION
```

4. the lines of code that the function will execute
5. **end** (closes the function)

Writing Custom Functions

But our code was working just fine! Why do we need to change it and write our own functions?

- For a simple program like the one we've written so far, it's true that custom functions are not strictly necessary
- But as your game grows, it will become more useful to have code separated and organized by what it does
- This will make the overall program more readable
- Troubleshooting becomes compartmentalized – you only need to edit a specific block instead of having to disentangle code that performs various purposes

Writing Custom Functions

- Another reason to write your own functions is that you can run them whenever you want, as many times as you need
- *When you find yourself writing the same code over and over again, it may be a sign that the code should be written as a function*
- Similarly to our plyr_spd variable – we can insert the same functionality in multiple places but only need to define that functionality once

Writing Custom Functions

***Where** do you write your custom functions?*

- Anywhere that's outside the main game loop (as in, not in the **_init()**, **_update()**, or **_draw()** functions
- And not inside any other function

*One place you can write your custom functions is **after** the **_draw()** function*

Writing Custom Functions

***Where** do you write your custom functions?*

- Another option is to use the **code tabs** in the editor
- Click the **+** next to the tab list to add a new tab
- *Anything you type is treated as written after the last line on the previous tab*

Writing Custom Functions

- *It's important to understand that **the code in a function will not run on its own***
- The function needs to be "*called*" or referred to elsewhere in the program (in the main game loop)

```
function make_plyr()

 plyr_n=1 -- sprite number
 plyr_x=4 -- x coordinate
 plyr_y=60 -- y coordinate
 plyr_spd=1 -- speed

end -- close the function
```

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_PLYR()

 PLYR_N=1  -- SPRITE NUMBER
 PLYR_X=4  -- X COORDINATE
 PLYR_Y=60 -- Y COORDINATE
 PLYR_SPD=1

END -- CL
```

Writing Custom Functions

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

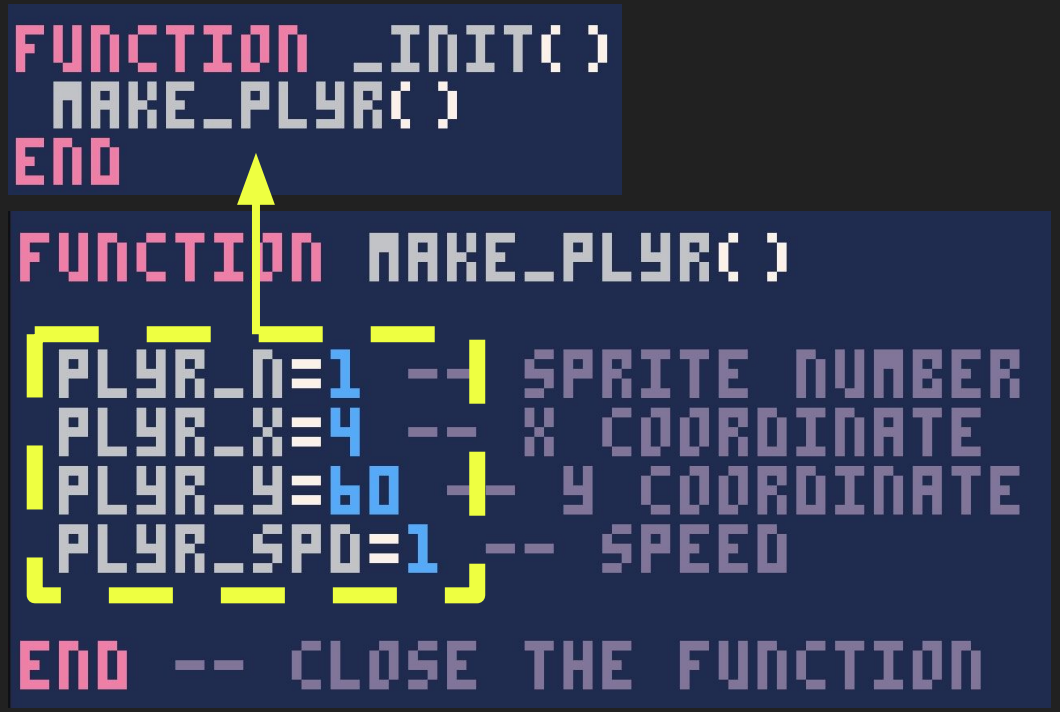
```
function _init()
 make_plyr()
end
```

```
function make_plyr()

 plyr_n=1 -- sprite number
 plyr_x=4 -- x coordinate
 plyr_y=60 -- y coordinate
 plyr_spd=1 -- speed

end -- close the function
```

Writing Custom Functions

The code inside a function runs right where you call it

Think of it as a shortcut or portal



Writing Custom Functions

```
function _init()
 plyr_n=1  -- sprite number
 plyr_x=4  -- x coordinate
 plyr_y=60 -- y coordinate
 plyr_spd=1 -- speed
end
```

=

```
function _init()
 make_plyr()
end
```

```
function make_plyr()

 plyr_n=1  -- sprite number
 plyr_x=4  -- x coordinate
 plyr_y=60 -- y coordinate
 plyr_spd=1 -- speed

end -- close the function
```

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()
 plyr_n=1 -- sprite number
 plyr_x=4 -- x coordinate
 plyr_y=60 -- y coordinate
 plyr_spd=1 -- speed
end
```

=

```
function _init()
 make_plyr()
end
```

```
function make_plyr()

 plyr_n=1 -- sprite number
 plyr_x=4 -- x coordinate
 plyr_y=60 -- y coordinate
 plyr_spd=1 -- speed

end -- close the function
```

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*

Animating a Sprite

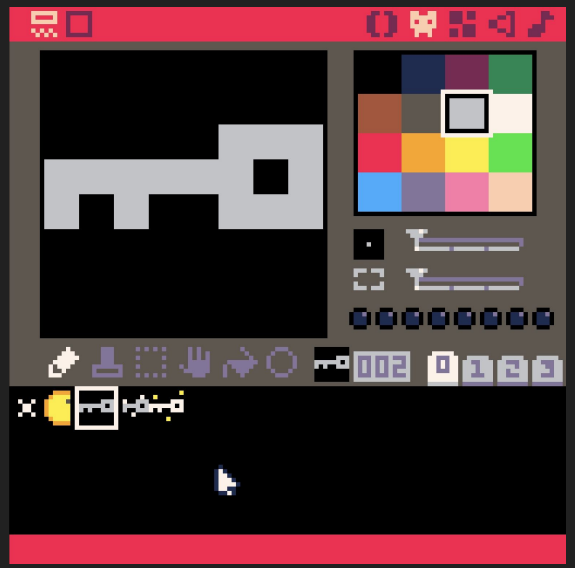
Download Example File: intro_11_animation.p8

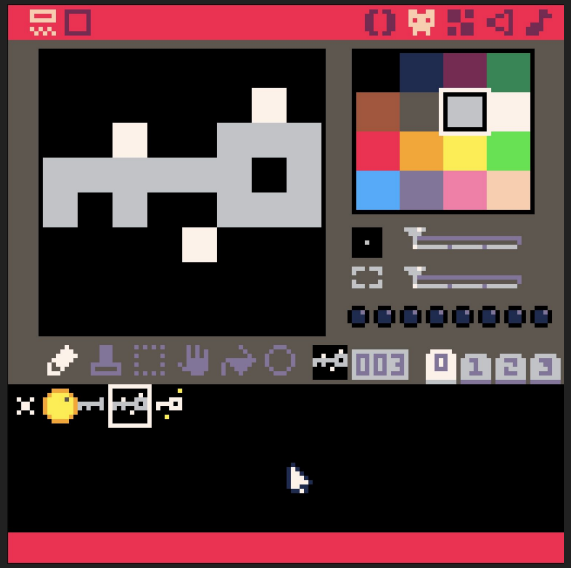
Animating a Sprite

- Animating a sprite simply involves changing which sprite is displaying over time
- When planning an animation, it's helpful to draw all frames (all sprites) for the animation on **consecutive spots** in the sprite sheet

Animating a Sprite

- I’ve drawn three frames for an animation of a key sparkling – on sprite slots 2, 3, and 4





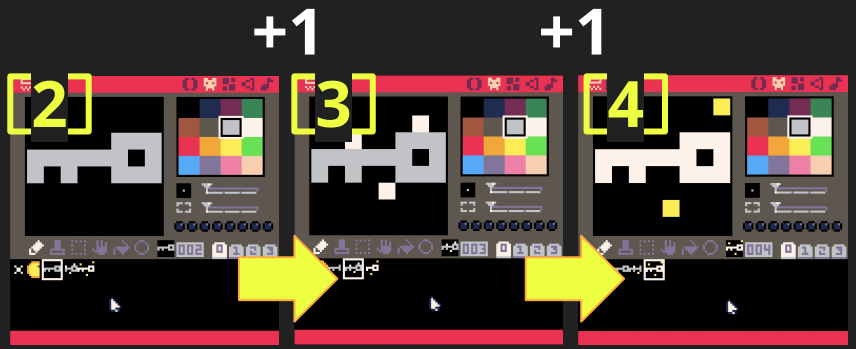
Animating a Sprite

Since the key's sprite number is a variable, **key_n**, I can **increase the sprite number by 1** to go from sprite 2 to sprite 3 to sprite 4

This code would go in _update() or in a custom animation function that's called in _update()

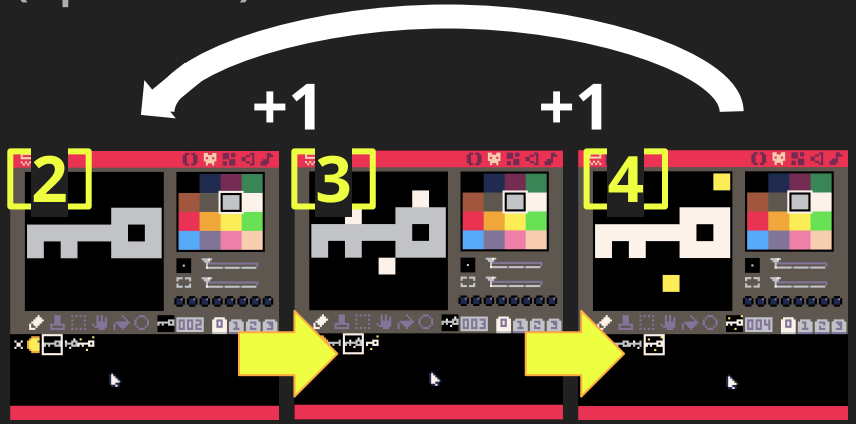
```
-- GO TO NEXT SPRITE
KEY_N = KEY_N + 1
```

```
-- OUTPUT/GRAPHICS
FUNCTION _DRAW()
 CLS() -- REFRESH SCREEN

 -- DRAW PLAYER
 SPR(PLYR_N,PLYR_X,PLYR_Y)

 -- DRAW KEY
 SPR(KEY_N,KEY_X,KEY_Y)

END -- /FUNCTION _DRAW()
```



Animating a Sprite

Then, once the sprite number goes beyond the range of the animation (past sprite 4), I can reset it back to the first frame (sprite 2)

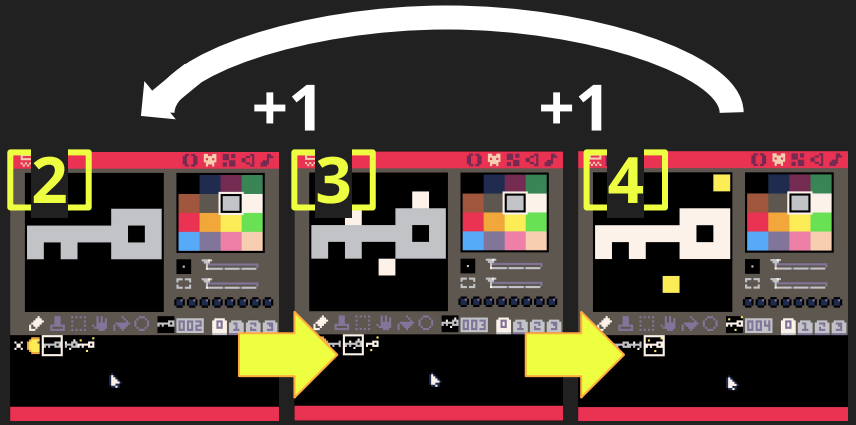


This code would go in _update() or in a custom animation function that's called in _update()

```
-- GO TO NEXT SPRITE
KEY_N = KEY_N + 1

-- IF KEY SPRITE REACHES END
-- OF LOOP, GO BACK TO START
IF KEY_N > 4 THEN
 KEY_N = 2
END
```

Animating a Sprite

This will be a bit too fast, so we'll use a timer to stagger the sprite swapping

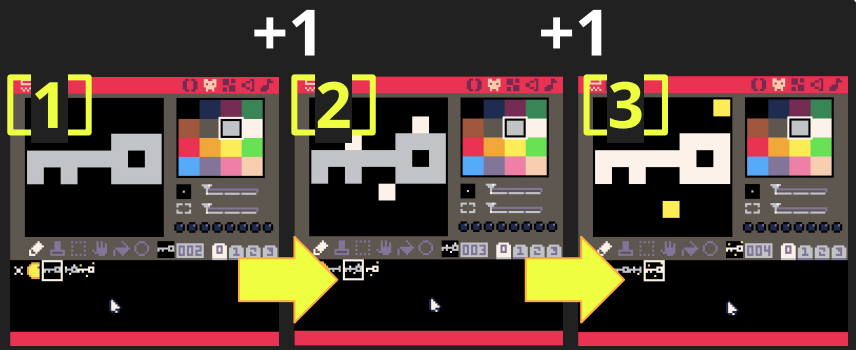


This code would go in _update() or in a custom animation function that's called in _update()

```
-- START TIMER
TIMER = TIMER + 1
-- EVERY FEW FRAMES, SWAP
-- THE KEY'S SPRITE
IF TIMER >= 9 THEN

 -- GO TO NEXT SPRITE
 KEY_N = KEY_N + 1

 -- IF KEY SPRITE REACHES END
 -- OF LOOP, GO BACK TO START
 IF KEY_N > 4 THEN
  KEY_N = 2
 END

 -- RESET TIMER
 TIMER = 0
END -- /IF TIMER >= RATE
```

Animating a Sprite

1. Start the timer
2. Only go to next sprite once the timer reaches its limit
3. Reset the timer so the animation can repeat



This code would go in _update() or in a custom animation function that's called in _update()

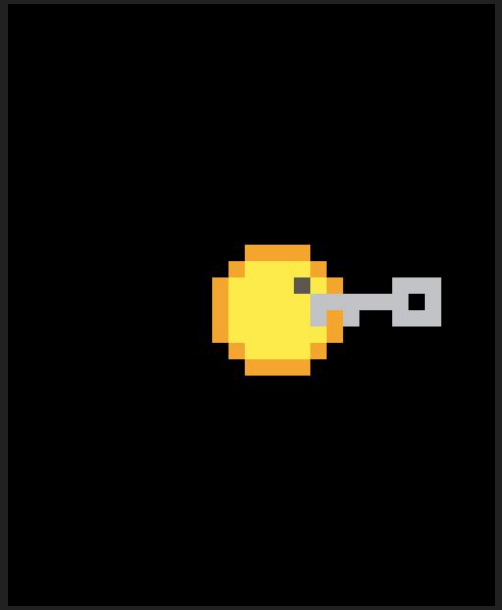
```
-- START TIMER
TIMER = TIMER + 1

-- EVERY FEW FRAMES, SWAP
-- THE KEY'S SPRITE
IF TIMER >= 9 THEN

 -- GO TO NEXT SPRITE
 KEY_N = KEY_N + 1

 -- IF KEY SPRITE REACHES END
 -- OF LOOP, GO BACK TO START
 IF KEY_N > 4 THEN
  KEY_N = 2
 END

 -- RESET TIMER
 TIMER = 0
END -- /IF TIMER >= RATE
```

Collecting Items (Detecting Collision)

Download Example File: intro_12_collection.p8

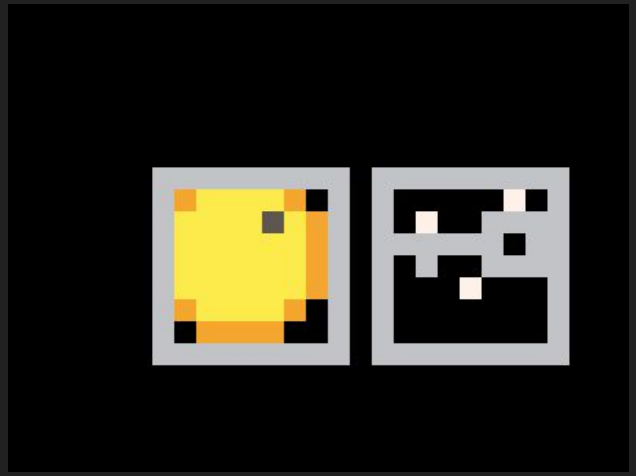
Detecting Collision Between Objects

- To implement item collection, the first step will be to *determine when the **player** and the **item** are **overlapping** on the screen*
- We can use the player's and key's **x,y coordinates** as a starting point



Detecting Collision Between Objects

- For a simple collision-detection function, we can **treat both objects (player and key) as boxes**
- You can even use PICO-8's **rect()** function to draw and visualize these hitboxes



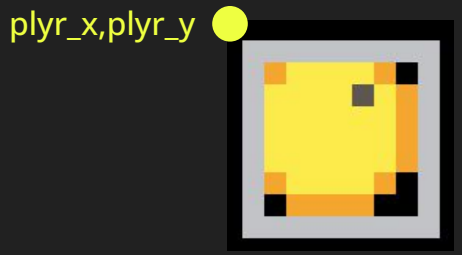
Rect

```
rect( x0, y0, x1, y1, [col] )
```

Draws an empty rectangle shape.

Detecting Collision Between Objects

We know the point at the top-left of the player corresponds to its x,y location:



Rect

SIGN IN TO EDIT

`rect( x0, y0, x1, y1, [col] )`

Draws an empty rectangle shape.

`x0`

The x coordinate of the upper left corner.

`y0`

The y coordinate of the upper left corner.

`x1`

The x coordinate of the lower right corner.

`y1`

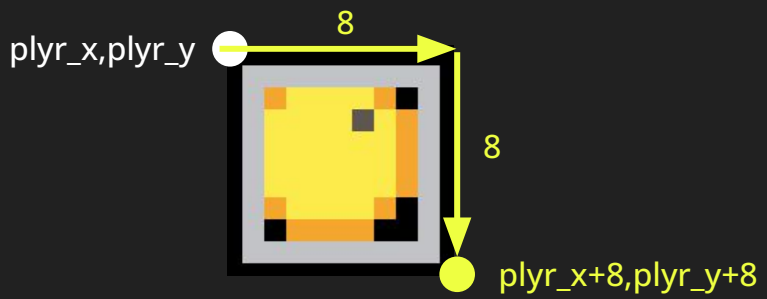
The y coordinate of the lower right corner.

`col`

The color of the rectangle border. If omitted, the color from the draw state is used.

Detecting Collision Between Objects

And we know the player sprite is 8px wide and tall, so we can add 8 to the player's x,y to get the opposite point:



Rect

SIGN IN TO EDIT

`rect( x0, y0, x1, y1, [col] )`

Draws an empty rectangle shape.

`x0`

The x coordinate of the upper left corner.

`y0`

The y coordinate of the upper left corner.

`x1`

The x coordinate of the lower right corner.

`y1`

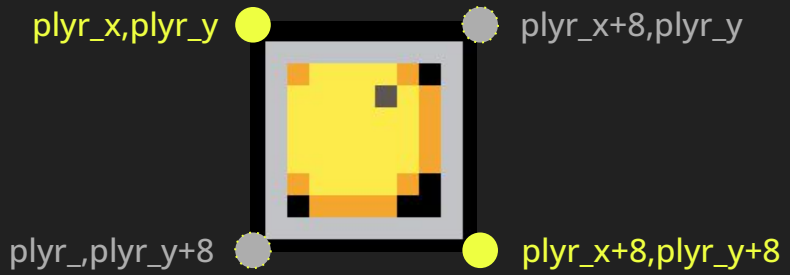
The y coordinate of the lower right corner.

`col`

The color of the rectangle border. If omitted, the color from the draw state is used.

Detecting Collision Between Objects

With these two points, the **rect()** function infers the remaining two



Rect

SIGN IN TO EDIT

```
rect( x0, y0, x1, y1, [col] )
```

Draws an empty rectangle shape.

`x0`

The x coordinate of the upper left corner.

`y0`

The y coordinate of the upper left corner.

`x1`

The x coordinate of the lower right corner.

`y1`

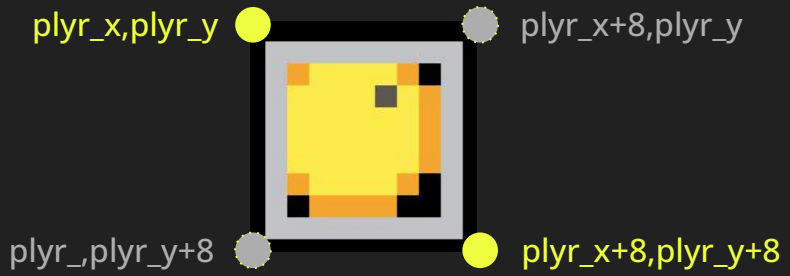
The y coordinate of the lower right corner.

`col`

The color of the rectangle border. If omitted, the color from the draw state is used.

Detecting Collision Between Objects

We can use the following code in **_draw()** to draw the player's hitbox




```
RECT(PLYR_X,PLYR_Y,
PLYR_X+8,PLYR_Y+8)
```

Rect

SIGN IN TO EDIT

```
rect( x0, y0, x1, y1, [col] )
```

Draws an empty rectangle shape.

`x0`

The x coordinate of the upper left corner.

`y0`

The y coordinate of the upper left corner.

`x1`

The x coordinate of the lower right corner.

`y1`

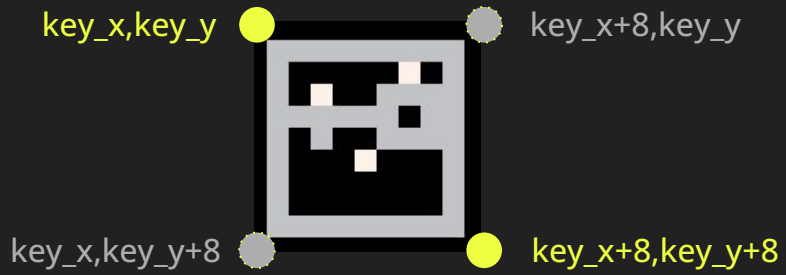
The y coordinate of the lower right corner.

`col`

The color of the rectangle border. If omitted, the color from the draw state is used.

Detecting Collision Between Objects

And we can do the same for the key:




```
RECT( KEY_X, KEY_Y,
KEY_X+8, KEY_Y+8 )
```

Rect

SIGN IN TO EDIT

```
rect( x0, y0, x1, y1, [col] )
```

Draws an empty rectangle shape.

`x0`

The x coordinate of the upper left corner.

`y0`

The y coordinate of the upper left corner.

`x1`

The x coordinate of the lower right corner.

`y1`

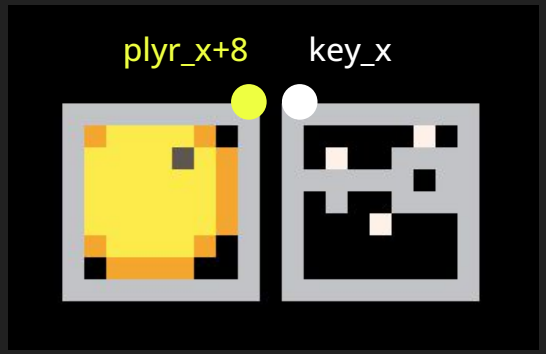
The y coordinate of the lower right corner.

`col`

The color of the rectangle border. If omitted, the color from the draw state is used.

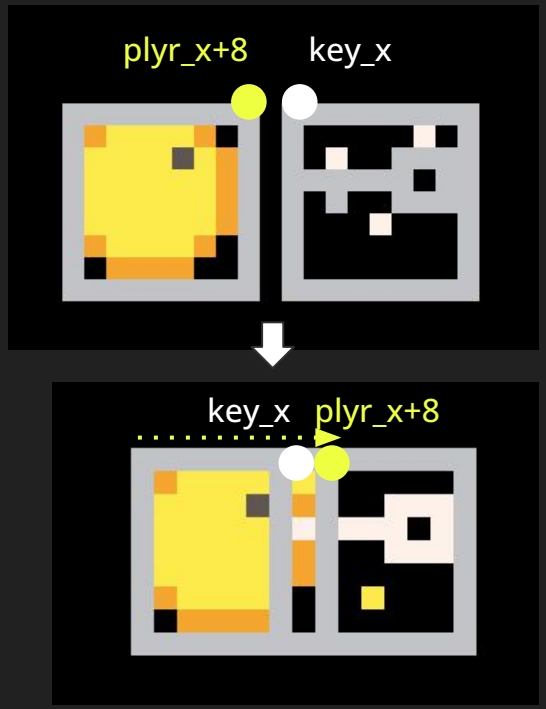
Detecting Collision Between Objects

- To actually detect collision, we'll **compare the player's x with the key's x**, and **compare the player's y with the key's y**



Detecting Collision Between Objects

- To actually detect collision, we'll **compare the player's x with the key's x**, and **compare the player's y with the key's y**
- For instance, we know that **the player's right edge** must be ***farther to the right*** than **the key's left edge**

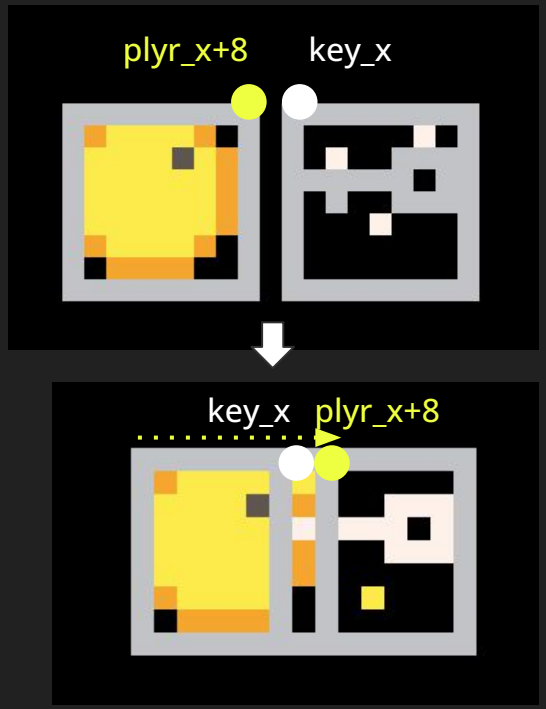


Detecting Collision Between Objects

- If **the player's right edge** is ***farther to the right*** than the **key's left edge**, we can express this as an equation:

plyr_x+8 >= key_x

*Use **greater than or equals** to allow for the possibility that both points are exactly the same*



We can use this equation as the condition for an if-statement:

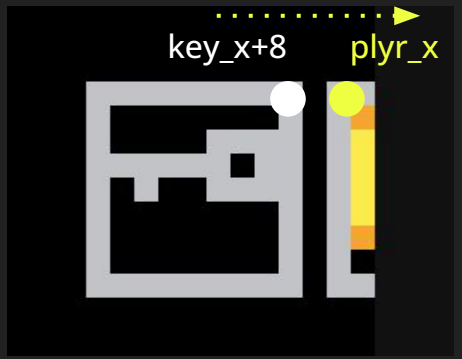
```
IF PLYR_X+8 >= KEY_X
```

Detecting Collision Between Objects

- However, we must also make sure the player hasn't passed the key entirely; in other words, **the player's left edge** must be ***farther to the left*** than the **key's right edge**:

  plyr_x <= key_x+8

We must rule out this possibility



We can use AND to stack multiple conditions in our if-statement:

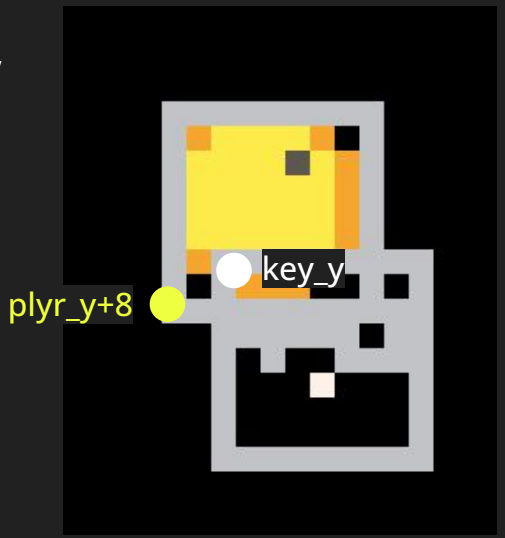
```
IF PLYR_X+8 >= KEY_X
AND PLYR_X <= KEY_X+8
```

Detecting Collision Between Objects

Now we can **compare the Y positions** of both objects:

- The **bottom of the player** must be ***below*** the **top of the key**

plyr_y+8 >= **key_y**




```
IF PLYR_X+8 >= KEY_X
AND PLYR_X <= KEY_X+8
AND PLYR_Y+8 >= KEY_Y
```

Detecting Collision Between Objects

Now we can **compare the Y positions** of both objects:

- The **top of the player** must be ***above*** the **bottom of the key**

plyr_y <= key_y +8


```
IF PLYR_X+8 >= KEY_X
AND PLYR_X <= KEY_X+8
AND PLYR_Y+8 >= KEY_Y
AND PLYR_Y <= KEY_Y+8
```

Detecting Collision Between Objects

```
function _update()
 move_plyr()
 anim_key()
 collect()
end
```

We can place this if-statement with its four conditions in a custom function and call that function in **_update()**

```
function collect()

 -- if plyr is touching key
 if plyr_x+8 >= key_x
 and plyr_x <= key_x+8
 and plyr_y+8 >= key_y
 and plyr_y <= key_y+8
 then



 end -- /if

end -- /function collect()
```

Detecting Collision Between Objects

But what should we ***do*** when the player touches the key?

- To collect the key, ***its sprite should disappear***
- And there should be ***some indication of our inventory in the HUD***

Detecting Collision Between Objects

We can ***increase a variable that counts the number of keys a player has***

But we'll need to ***create that variable first***, in **_init()**

```
FUNCTION _INIT()
 MAKE_PLYR()
 MAKE_KEY()

 -- INVENTORY
 KEYS = 0

 -- ANIMATION TIMER
 TIMER = 0
END
```

```
FUNCTION COLLECT()

 -- IF PLYR IS TOUCHING KEY
 IF PLYR_X+8 >= KEY_X
 AND PLYR_X <= KEY_X+8
 AND PLYR_Y+8 >= KEY_Y
 AND PLYR_Y <= KEY_Y+8
 THEN
  KEYS = KEYS + 1
 END -- /IF

END -- /FUNCTION COLLECT()
```

Detecting Collision Between Objects

```
function _init()
 make_plyr()
 make_key()

 -- inventory
 keys = 0

 -- animation timer
 timer = 0
end
```

Declare variable for number of keys collected

```
function _update()
 move_plyr()
 anim_key()
 collect()
end
```

Call the collect() function

```
function _draw()
 cls() -- refresh screen

 -- draw player
 spr(plyr_n,plyr_x,plyr_y)

 -- draw key
 spr(key_n,key_x,key_y)

 -- print number of keys
 print(keys,2,2,7)
end -- /function _draw()
```

Print the number of keys

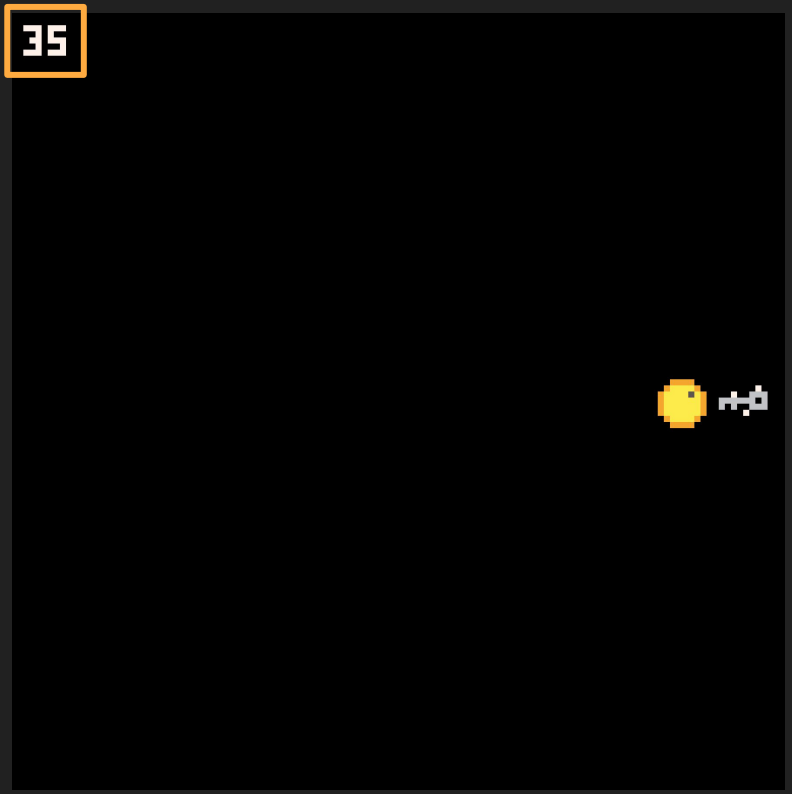
```
function collect()

 -- if plyr is touching key
 if plyr_x+8 >= key_x
 and plyr_x <= key_x+8
 and plyr_y+8 >= key_y
 and plyr_y <= key_y+8
 then
  keys = keys + 1
 end -- /if

end -- /function collect()
```

Increase the number of keys when the player collides with the key

Detecting Collision Between Objects

You'll notice that ***the player can collect keys indefinitely*** as long as they are touching the key

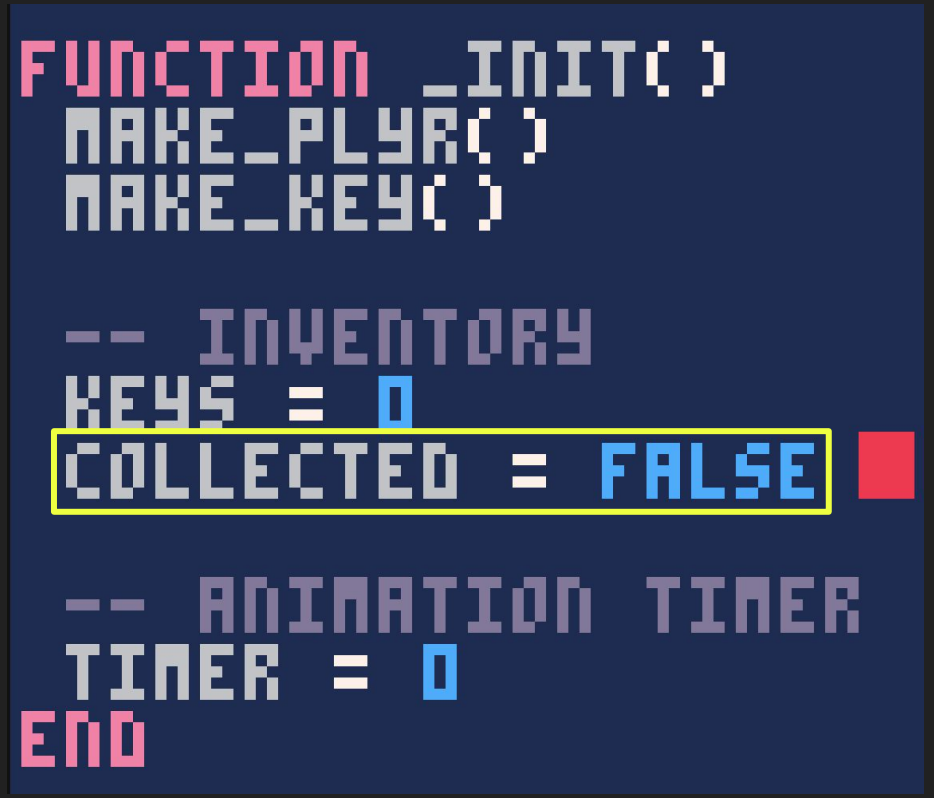
So we'll need to **track whether the player has already collected the key**, and ***only allow this number to go up if the key hasn't been collected yet***



Detecting Collision Between Objects

We can use a ***special type of variable*** to track whether the key has been collected

A variable whose value can be only **true** or **false** is called a ***boolean*** variable



Detecting Collision Between Objects

```
FUNCTION _INIT()
 MAKE_PLYR()
 MAKE_KEY()

 -- INVENTORY
 KEYS = 0
 COLLECTED = FALSE

 -- ANIMATION TIMER
 TIMER = 0
END
```

1. Declare the ***collected*** variable in **_init()**

```
FUNCTION COLLECT()

 -- IF PLYR IS TOUCHING KEY
 IF PLYR_X+8 >= KEY_X
 AND PLYR_X <= KEY_X+8
 AND PLYR_Y+8 >= KEY_Y
 AND PLYR_Y <= KEY_Y+8
 AND COLLECTED == FALSE
 THEN
  KEYS = KEYS + 1
  COLLECTED = TRUE
 END -- /IF

END -- /FUNCTION COLLECT()
```

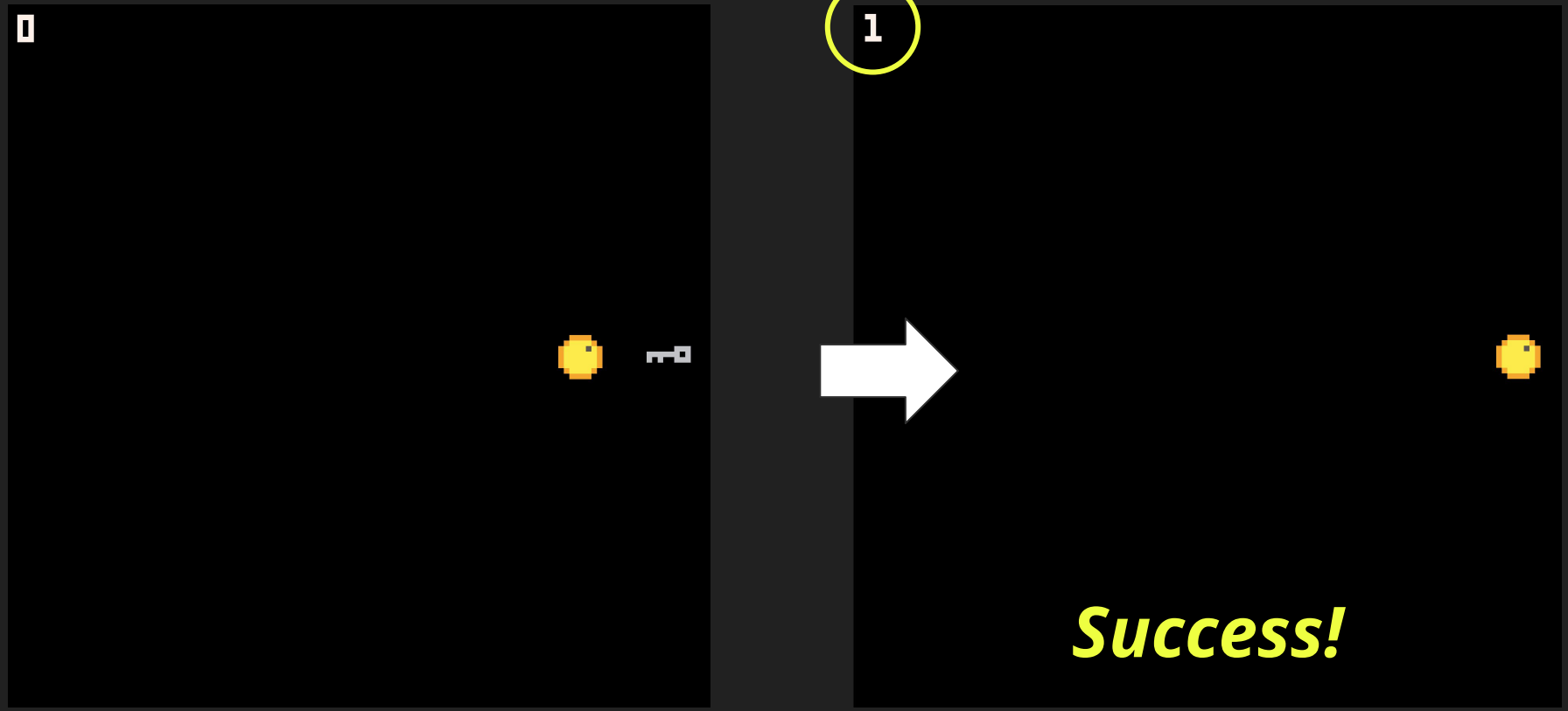
2. Add ***another condition*** to ***only allow collection if the key hasn't been collected yet***

3. Set ***collected*** to ***true*** when the player gets the key

```
FUNCTION _DRAW()
 CLS() -- REFRESH SCREEN

 -- DRAW PLAYER
 SPR(PLYR_N,PLYR_X,PLYR_Y)

 -- DRAW KEY
 IF COLLECTED == FALSE THEN
  SPR(KEY_N,KEY_X,KEY_Y)
 END

 -- PRINT NUMBER OF KEYS
 PRINT(KEYS,2,2,7)

END -- /FUNCTION _DRAW()
```

4. ***Only draw the key if it hasn't been collected yet***

Detecting Collision Between Objects



Congratulations!

In this lesson, you:

1. Wrote your first program in the PICO-8 game engine using **print()**
2. **Understood how the coordinate plane works** in PICO-8
3. **Wrote code to display a sprite in your game** using **spr()**
4. **Used variables to track changing values** for the player's x,y coordinates
5. **Moved the player in response to keyboard input** by using if-statements and **btn()**
6. **Organized your code using custom functions**, calling those functions from the main PICO-8 game loop
7. **Animated a sparkling key** by using a timer to control the changing key sprites
8. **Implemented item collection** by writing a simple collision-detection function and using boolean (true/false) logic

These skills will transfer to other game engines and programming languages

Code Examples

Download the full set:
_helloworld_00_intro_to_game_programming.zip

1.	intro_01_hello_world.p8	Your first PICO-8 program
2.	intro_02_coordinate_plane.p8	Understanding PICO-8's x,y coordinate system
3.	intro_03_text.p8	Positioning text on the screen
4.	intro_04_color.p8	Setting text color
5.	intro_05_layers.p8	Understanding stacking order
6.	intro_06_sprites.p8	Displaying sprites in your game
7.	intro_07_game_loop.p8	Understanding the PICO-8 program structure
8.	intro_08_variables.p8	Using values that can change over time
9.	intro_09_move_player.p8	Detecting and responding to key input
10.	intro_10_functions.p8	Organizing your code
11.	intro_11_animation.p8	Adding visual effects to graphics
12.	intro_12_collection.p8	Using collision detection to collect an item

</lesson>

Questions? Email matt.dimatteo@gmail.com