

# Making a Sidescrolling Platformer Game in PICO-8

## Complete Walkthrough

by Matthew DiMatteo

Assistant Professor, Game & Interactive Media Design at Rider University  
[mdimatteo@rider.edu](mailto:mdimatteo@rider.edu)

*as seen in GAM-120: Intro to Game Logic*



1865 RIDER  
UNIVERSITY

# Contents

- 4) Getting Started
- 10) Horizontal Movement
- 14) Physics-Based Movement
- 25) Applying Friction
- 30) Jumping
- 34) Applying Gravity
- 38) Map Collision
- 47) Jumping, Revisited

# Contents

- 53) Collision with Walls
- 60) Refining Collision and Movement
- 71) Camera
- 93) Pits and Respawning
- 111) Animating a Sprite
- 130) Switching Between Different Levels
- 157) Different Types of Terrain and Friction
- 165) Learning Resources

# Making a Side-Scrolling Platformer Game

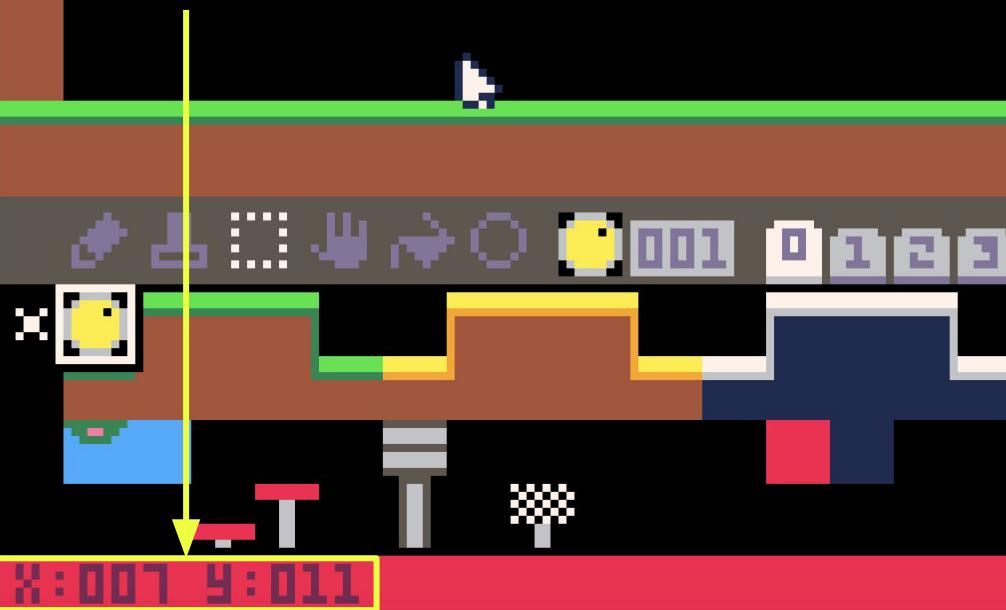
Download Assets File: [platformer\\_00\\_assets\\_only.p8](#)

# Step 01: Setting the Player Start Position

Download Example File: [platformer\\_01\\_starting\\_position.p8](#)



In the map editor, I can mouse over a tile that looks like a good place to start my player, and the tile coordinates are displayed in the bottom left of the screen





In the map editor, I can mouse over a tile that looks like a good place to start my player, and the tile coordinates are displayed in the bottom left of the screen



```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR={}
PLYR.N=1 --> SPRITE NUMBER
PLYR.X=7*8 -- X=56 PIXELS
PLYR.Y=11*8 -- Y=88 PIXELS
END -- END FUNCTION MAKE_PLYR()
```

Our platformer game will most likely work best with **pixel-based movement** (as opposed to the tile-based movement in the top-down game), so we want to express our **player's position** in terms of **pixels**

In the map editor, I can mouse over a tile that looks like a good place to start my player, and the tile coordinates are displayed in the bottom left of the screen

Finally, I can call **make\_plyr()** in my **\_init()** function and draw my player sprite using **spr()** in **\_draw()**, plugging in the player variables

X:001 Y:011

```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR={}
PLYR.n=1 -- SPRITE NUMBER
PLYR.X=7*8 -- X=56 PIXELS
PLYR.Y=11*8 -- Y=88 PIXELS
END -- END FUNCTION MAKE_PLYR()
```

```
FUNCTION _INIT()
MAKE_PLYR() -- TAB 1
END -- END FUNCTION _INIT()

FUNCTION _UPDATE()
END -- END FUNCTION _UPDATE()

FUNCTION _DRAW()
CLS()
MAP()
SPR(PLYR.n, PLYR.X, PLYR.Y)
END -- END FUNCTION _DRAW()
```

- **make\_plyr()** is a custom function I wrote on tab 1, where my player variables are declared (sprite #, x, y)
  - **cls()** clears the screen
  - **map()** draws the map
  - **spr()** draws my sprite – plug in the values for the **sprite number**, the **x coordinate**, and the **y coordinate**, separated by commas

```
FUNCTION _INIT()
MAKE_PLYR() -- TAB 1
END -- END FUNCTION _INIT()

FUNCTION _UPDATE()
END -- END FUNCTION _UPDATE()

FUNCTION _DRAW()
CLS()
MAP()
SPR(PLYR.N, PLYR.X, PLYR.Y)
END -- END FUNCTION _DRAW()
```

# Step 02: Moving Left/Right

Download Example File: [platformer\\_02\\_horizontal\\_movement.p8](#)

012 I made a new *move\_plyr()* function on tab 2

```
-- MOVE PLAYER  
FUNCTION MOVE_PLYR()
```

```
-- MOVE LEFT  
IF BTN(0) THEN  
    PLYR.X -= PLYR.XSPD  
END -- END IF BTN(0)  
  
-- MOVE RIGHT  
IF BTN(1) THEN  
    PLYR.X += PLYR.XSPD  
END -- END IF BTN(1)
```

```
END -- END FUNCTION MOVE_PLYR()
```

To move horizontally, we can use the **btn0** function to detect **left/right arrow** key presses and **add to or subtract from** the player's **x coordinate**

*Make sure to call this function in \_update()*

```
012 + 0 0 0 0 0  
-- TAB 0: GAME LOOP  
-- TAB 1: MAKE PLAYER  
-- TAB 2: MOVE PLAYER  
  
FUNCTION _INIT()  
    MAKE_PLYR() -- TAB 1  
END -- END FUNCTION _INIT()  
  
FUNCTION _UPDATE()  
    MOVE_PLYR() -- TAB 2  
END -- END FUNCTION _UPDATE()  
  
FUNCTION _DRAW()  
    CLS() -- CLEAR SCREEN  
    MAP() -- DRAW MAP  
    SPR(PLYR.N, PLYR.X, PLYR.Y)  
END -- END FUNCTION _DRAW()
```

0/1/2

I made a new *move\_plyr()* function on tab 2

()

```
-- MOVE PLAYER
FUNCTION MOVE_PLYR()

-- MOVE LEFT
IF BTn(0) THEN
  PLYR.X -= PLYR.XSPD
END -- END IF BTn(0)

-- MOVE RIGHT
IF BTn(1) THEN
  PLYR.X += PLYR.XSPD
END -- END IF BTn(1)

END -- END FUNCTION MOVE_PLYR()
```

To move horizontally, we can use the **btn0** function to detect **left/right arrow** key presses and **add to or subtract from** the player's **x coordinate**

```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR={} -- TABLE FOR PLYR OBJ
PLYR.N=1 -- SPRITE NUMBER

-- X,Y COORDINATES
PLYR.X=7*8 -- X=56 PIXELS
PLYR.Y=11*8 -- Y=88 PIXELS

PLYR.XSPD=4 -- X SPEED

END -- END FUNCTION MAKE_PLYR()
```

I added a **variable** for the player's horizontal speed in my **make\_plyr()** function

This is fine, but *the movement is a little boring* . . . we can **apply some basic physics** to make it feel more interesting

# Step 03: Physics-Based Movement

Download Example File: [platformer\\_03\\_delta\\_x.p8](#)

# Physics-Based Movement

TIME (Seconds)	ACCELERATION	SPEED (MPH)	TOTAL DIST.
1	10	10	10
2	10	20	30
3	10	30	60
4	10	40	100
5	10	50	150
6	10	60	210

Think of what happens when you press the accelerator pedal in your car to go from 0 to 60 MPH

# Physics-Based Movement

TIME (Seconds)	ACCELERATION	SPEED (MPH)	TOTAL DIST.
1	10	10	10
2	10	20	30
3	10	30	60
4	10	40	100
5	10	50	150
6	10	60	210

Here, acceleration is constant, but the **speed increases** over time because *acceleration continually adds to the speed*

# Physics-Based Movement

TIME (Seconds)	ACCELERATION	SPEED (MPH)	TOTAL DIST.
1	10	10	10
2	10	20	30
3	10	30	60
4	10	40	100
5	10	50	150
6	10	60	210

The speed is then how much distance is traveled over time, which increases total distance

# Physics-Based Movement

TIME (Seconds)	ACCELERATION	SPEED (MPH)	TOTAL DIST.
1	10	10	10
2	10	20	30
3	10	30	60
4	10	40	100
5	10	50	150
6	10	60	210

In game programming, it's conventional to name variables **dx** and **dy** to represent **speed**

# Physics-Based Movement

TIME (Seconds)	ACCELERATION	SPEED (MPH)	TOTAL DIST.
1	10	10	10
2	10	20	30
3	10	30	60
4	10	40	100
5	10	50	150
6	10	60	210

**dx**, for example, stands for delta-x (*delta* = *change*), so dx means the change in x

# Physics-Based Movement

YOUR CAR			
TIME (Seconds)	ACCELERATION	SPEED (MPH)	TOTAL DIST.
1	10	10	10
2	10	20	30
3	10	30	60
4	10	40	100
5	10	50	150
6	10	60	210

THE PLAYER			
FRAME	GRAVITY	DY	Y
1	0.3	0.3	0.3
2	0.3	0.6	0.9
3	0.3	0.9	1.8
4	0.3	1.2	3
5	0.3	1.5	4.5
6	0.3	1.8	6.3

In our game,  
acceleration will be a  
*constant force* like  
friction or gravity

# Physics-Based Movement

YOUR CAR			
TIME (Seconds)	ACCELERATION	SPEED (MPH)	TOTAL DIST.
1	10	10	10
2	10	20	30
3	10	30	60
4	10	40	100
5	10	50	150
6	10	60	210

THE PLAYER			
FRAME	GRAVITY	DY	Y
1	0.3	0.3	0.3
2	0.3	0.6	0.9
3	0.3	0.9	1.8
4	0.3	1.2	3
5	0.3	1.5	4.5
6	0.3	1.8	6.3

And we can use **dx** and **dy** to represent *how much the player is moving each frame* (their current speed)

We can use “delta time” ( $dx/dy$ ) to create the effect of ***acceleration***

I added a variable for the **player's dx** (*the change in the x position*) in my **make\_plyr()** function



```
012 + 0 0 0 0 0 0
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR={} -- TABLE FOR PLYR OBJ
PLYR.n=1 -- SPRITE NUMBER

-- X,Y COORDINATES
PLYR.X=788 -- X=56 PIXELS
PLYR.Y=1188 -- Y=88 PIXELS

PLYR.XSPD=4 -- X SPEED
PLYR.DX=0 -- CHANGE IN X

END -- END FUNCTION MAKE_PLYR()
```

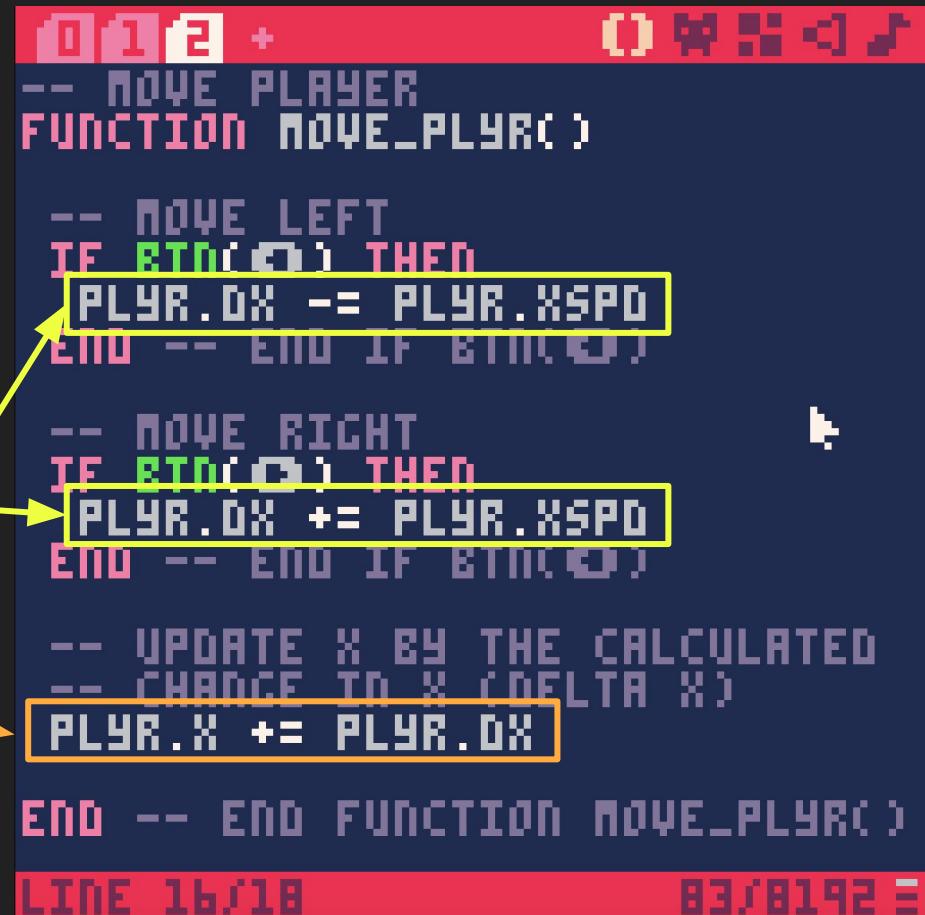
*dx will usually start at 0 because the player is not moving until an input is made*

We can use “delta time” ( $dx/dy$ ) to create the effect of *acceleration*

I added a variable for the player’s  $dx$  (*the change in the x position*) in my `make_plyr()` function

Instead of changing **x** when the L/R arrow keys are pressed, we change **dx**

Then we **add dx to x** at the end of the `move_plyr()` function



The screenshot shows a game development environment with assembly-like code. The code defines a function `MOVE_PLYR()` which handles player movement based on key inputs. It uses variables `PLYR.DX` and `PLYR.X` to calculate movement. The code is annotated with arrows pointing to specific lines:

- An arrow points from the text "Instead of changing x when the L/R arrow keys are pressed, we change dx" to the line `PLYR.DX -= PLYR.XSPD`.
- An arrow points from the text "Then we add dx to x at the end of the move\_plyr() function" to the line `PLYR.X += PLYR.DX`.

```
012 +
-- MOVE PLAYER
FUNCTION MOVE_PLYR()
-- MOVE LEFT
IF BT0(0) THEN
    PLYR.DX -= PLYR.XSPD
END -- END IF BT0(0)
-- MOVE RIGHT
IF BT0(1) THEN
    PLYR.DX += PLYR.XSPD
END -- END IF BT0(1)
-- UPDATE X BY THE CALCULATED
-- CHANGE IN X (DELTA X)
    PLYR.X += PLYR.DX
END -- END FUNCTION MOVE_PLYR()
LINE 16/18          83/8192 E
```

You'll notice that *the player moves way too fast and never comes to a full stop*, even after the arrow keys are released - we'll need to *add friction to slow and stop the player*

# Step 04: Applying Friction

Download Example File: [platformer\\_04\\_friction.p8](#)

```
012 + 00000000  
-- MAKE PLAYER  
FUNCTION MAKE_PLYR()  
PLYR={} -- TABLE FOR PLYR OBJ  
PLYR.n=1 -- SPRITE NUMBER  
  
-- X,Y COORDINATES  
PLYR.X=7*8 -- X=56 PIXELS  
PLYR.Y=11*8 -- Y=88 PIXELS  
  
PLYR.XSPD=4 -- X SPEED  
PLYR.DX=0 -- CHANGE IN X  
END -- END FUNCTION MAKE_PLYR()
```

```
012 + 00000000  
-- MAKE PLAYER  
FUNCTION MAKE_PLYR()  
PLYR={} -- TABLE FOR PLYR OBJ  
PLYR.n=1 -- SPRITE NUMBER  
  
-- X,Y COORDINATES  
PLYR.X=7*8 -- X=56 PIXELS  
PLYR.Y=11*8 -- Y=88 PIXELS  
  
PLYR.XSPD=0.4 -- X SPEED  
PLYR.DX=0 -- CHANGE IN X  
END -- END FUNCTION MAKE_PLYR()
```

For starters, I **reduced my player's x speed by a factor of 10**

With x being increased exponentially each frame (because dx is growing larger), the value gets higher much more quickly

*Note: I eventually settled on an xspd of 0.5 – feel free to fine-tune as necessary*

```
0 1 2 3 + 0 1 2 3 ↵  
-- PLATFORNER  
-- LESSON 06: GRAVITY  
-- BY MATTHEW DINATTED  
  
-- TAB 0: GAME LOOP  
-- TAB 1: MAKE PLAYER  
-- TAB 2: MOVE PLAYER  
  
FUNCTION _INIT()  
    FRIC = 0.85 -- FRICTION  
    MAKE_PLAYER() -- TAB 1  
END -- END FUNCTION _INIT() ↵  
  
FUNCTION _UPDATE()  
    MOVE_PLAYER() -- TAB 2  
END -- END FUNCTION _UPDATE()  
  
FUNCTION _DRAW()  
    CLS() -- CLEAR SCREEN  
LINE 11/22 287/8192 ↵
```

In my `_init()` function, I added new variable for friction, called **fric**, and set it to a value of **0.85**

If we *multiply the player's dx by a value less than 1.0, it will gradually get smaller*, slowing the player to a stop once the arrow keys are released

012 +

0 9 8 7 6 5 4

## FUNCTION MOVE\_PLYR()

```
-- APPLY FRICTION SO THE PLYR  
-- EVENTUALLY STOPS MOVING
```

```
PLYR.DX *= FRIC
```

```
-- MOVE LEFT
```

```
IF BTN(0) THEN
```

```
PLYR.DX -= PLYR.XSPD
```

```
END -- END IF BTN(0)
```

```
-- MOVE RIGHT
```

```
IF BTN(1) THEN
```

```
PLYR.DX += PLYR.XSPD
```

```
END -- END IF BTN(1)
```

```
-- UPDATE X BY THE CALCULATED  
-- CHANGE IN X (DELTA X)
```

```
PLYR.X += PLYR.DX
```

LINE 20/22

90/8192

In my **move\_plyr()** function, I **multiply the player's dx by the friction value of 0.85** specified in `_init()`

If we *multiply the player's dx by a value less than 1.0, it will gradually get smaller*, slowing the player to a stop once the arrow keys are released

Horizontal movement now  
feels pretty good - let's add  
the ability to **jump** next

# Step 05: Jumping

Download Example File: [platformer\\_05\\_jumping.p8](#)

In order to allow the player to jump, I'll need to **add a variable** for its **y speed** (*how much to move it up by* when the jump button is pressed)

I'll also **add a dy** to go along with dx, so that we can have our vertical movement incorporate physics as well

```
012 +
0987654321

-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR={} -- TABLE FOR PLYR OBJ
PLYR.n=1 -- SPRITE NUMBER

-- X, Y COORDINATES
PLYR.X=738 -- X=56 PIXELS
PLYR.Y=1138 -- Y=88 PIXELS

PLYR.XSPD=0.5 -- X SPEED
PLYR.YSPD=4 -- Y SPEED

PLYR.DX=0 -- CHANGE IN X
PLYR.DY=0 -- CHANGE IN Y

END -- END FUNCTION MAKE_PLYR()
```

Then, in my player movement function, I'll:

- **Check for a key press** (I'll allow the player to press either the up arrow OR the X key)
- **Reduce the player's dy by its y speed** (which I set to 4) - we subtract because y gets smaller as we move up on the screen
- **Apply the change in y (dy, or delta y)** to the player's y coordinate by **adding dy to y** (like we did with x and dx)

The image shows a Scratch script with the following code:

```
012 + 0123456789
IF BTn(0) THEN
  PLYR.DX -= PLYR.XSPD
END -- END IF BTn(0)

-- MOVE RIGHT
IF BTn(0) THEN
  PLYR.DX += PLYR.XSPD
END -- END IF BTn(0)

-- JUMP
IF BTnP(0) OR BTnP(8) THEN
  PLYR.DY -= PLYR.YSPD
END -- END IF BTnP(0)/8

-- UPDATE X,Y BY THE CALCULATED
-- CHANGE (DELTA X, DELTA Y)
PLYR.X += PLYR.DX
PLYR.Y += PLYR.DY
```

The code handles player movement based on button presses. It checks for the left arrow key (BTn(0)) to move left and the right arrow key (BTn(1)) to move right. For jumping, it checks for the up arrow key (BTnP(0)) or the space bar (BTnP(8)). It reduces the player's vertical speed (dy) by the y speed (YSPD) when jumping upwards. Finally, it updates the player's x and y coordinates by adding the calculated changes (delta x and delta y).



You'll notice that *the player can keep pressing the jump button to move up indefinitely*

We'll need to **add gravity** so that what goes up comes back down

```
012 + 0 0 0 0 0 0
IF BTn(0) THEN
  PLYR.DX -= PLYR.XSPD
END -- END IF BTn(0)

-- MOVE RIGHT
IF BTn(0) THEN
  PLYR.DX += PLYR.XSPD
END -- END IF BTn(0)

-- JUMP
IF BTnP(0) OR BTnP(8) THEN
  PLYR.DY -= PLYR.YSPD
END -- END IF BTnP(0)/8

-- UPDATE X,Y BY THE CALCULATED
-- CHANGE (DELTA X, DELTA Y)
PLYR.X += PLYR.DX
PLYR.Y += PLYR.DY
```

# Step 06: Applying Gravity

Download Example File: [platformer\\_06\\_gravity.p8](#)

```
0 1 2 + ( ) [ ] < > ⏪  
-- TAB 0: GAME LOOP  
-- TAB 1: MAKE PLAYER  
-- TAB 2: MOVE PLAYER  
  
FUNCTION _INIT()  
    FRIC = 0.85 -- FRICTION  
    GRAY = 0.3 -- GRAVITY  
    MAKE_PLAYER() -- TAB 1  
END -- END FUNCTION _INIT()  
  
FUNCTION _UPDATE()  
    MOVE_PLAYER() -- TAB 2  
END -- END FUNCTION _UPDATE()  
  
FUNCTION _DRAW()  
    CLS() -- CLEAR SCREEN  
    MAP() -- DRAW MAP  
    SPR(PLYR.N, PLYR.X, PLYR.Y)  
END -- END FUNCTION _DRAW()  
LINE 11/23 124/8192 ⏪
```

Like with friction, I'll declare a variable for **gravity** in my `_init()` function

I'll set **gravity** (abbreviated as **grav**) to **0.3**, which worked well in our paddleball game

012 +

0 0 0 0 0 0

```
-- EVENTUALLY STOPS MOVING  
PLYR.DX *= FRIC
```

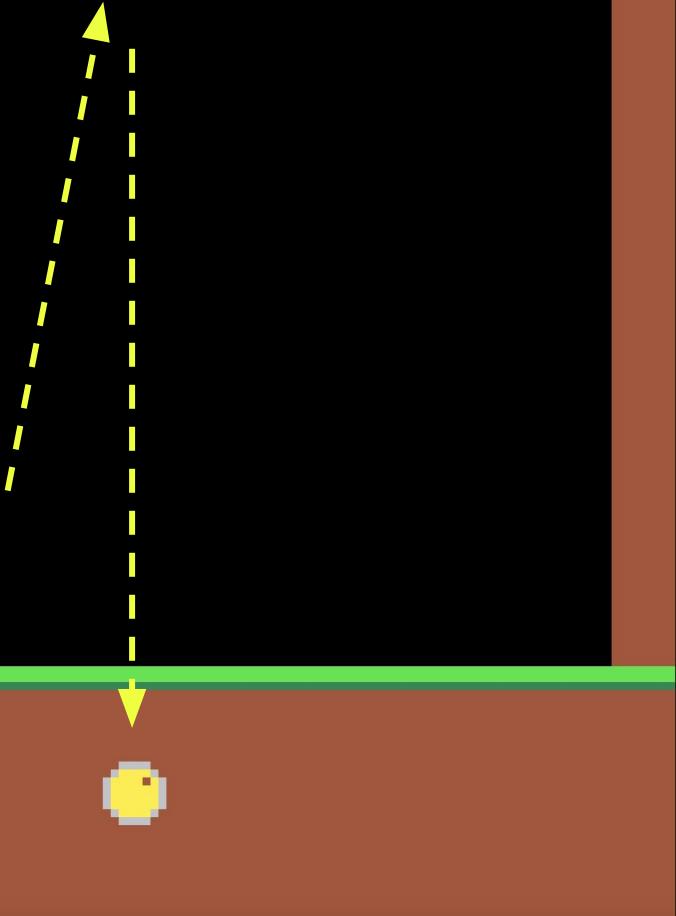
```
-- APPLY GRAVITY SO THE PLYR  
-- DOES NOT FLOAT ENDLESSLY  
PLYR.DY += GRAY
```

```
-- MOVE LEFT  
IF BTn(0) THEN  
    PLYR.DX -= PLYR.XSPD  
END -- END IF BTn(0)
```

```
-- MOVE RIGHT  
IF BTn(1) THEN  
    PLYR.DX += PLYR.XSPD  
END -- END IF BTn(1)
```

```
-- JUMP  
IF BTn(2) OR BTn(3) THEN  
LINE 2/32
```

Then, in my **move\_plyr()** function, I'll **add gravity to dy** at the beginning of the function (*similarly to friction and dx*)



This works a little too well ...

The player falls down after jumping up, but *they keep falling through the rest of the map*

We'll need to **detect** when there's a solid tile below the player and **stop** them from moving any further down in that case

# Step 07: Map Collision

Download Example File: [platformer\\_07\\_map\\_collision.p8](#)

I wrote a “**map collision**” function to detect tiles adjacent to the player, which you can use to *stop the player from falling when there's a solid tile beneath them*

It also works to prevent movement through solid tiles in the other directions

And you can detect other types of tiles, such as pits or hazards

Because it's a bit more complex than other concepts we've covered, I'll allow you to [borrow it](#) (copy and paste it into your code) for now



The screenshot shows a game development environment with assembly code. At the top, there are two sets of colored squares: red, green, blue, and yellow. Below the code, there are status indicators for memory usage: LINE 1/69, 290/8192 E.

```
0 1 2 3 + 0 1 2 3 +
-- MAP COLLISION FUNCTION
FUNCTION MCOLLIDE(OBJ,DIR,FLAG)

-- THIS FUNCTION CHECKS TWO
-- POINTS ON THE TILE ADJACENT
-- TO THE PLAYER: HX1,HY1 AND
-- HX2,HY2 -- WE CAN THEN USE
-- THESE COORDINATES TO LOOK UP
-- THE ADJACENT TILE'S SPRITE
-- NUMBER AND WHETHER IT HAS
-- A FLAG TURNED ON

-- POSITION OF TILE TO LEFT
IF DIR=="LEFT" THEN
    HX1=OBJ.X-1
    HY1=OBJ.Y

    HX2=HX1
    HY2=OBJ.Y+OBJ.H-1
```

-- MMAP COLLISION FUNCTION  
**FUNCTION** MCOLLIDE(OBJ,DIR,FLAG)  
  
-- THIS FUNCTION CHECKS TWO  
-- POINTS ON THE TILE ADJACENT  
-- TO THE PLAYER: HX1,HY1 AND  
-- HX2,HY2 -- WE CAN THEN USE  
-- THESE COORDINATES TO LOOK UP  
-- THE ADJACENT TILE'S SPRITE  
-- NUMBER AND WHETHER IT HAS  
-- A FLAG TURNED ON  
  
-- POSITION OF TILE TO LEFT  
**IF** DIR=="LEFT" **THEN**  
  HX1=OBJ.X-1  
  HY1=OBJ.Y  
  
  HX2=HX1  
  HY2=OBJ.Y+OBJ.H-1

I've copied and pasted the function  
**mcollide()** into my in-progress file

Notice that there are **3 required values** in the parentheses:

- **obj** (the object being used)
  - **dir** (the direction it's facing)
  - **flag** (the flag number of the tile to check for)

By making each of these values variables, this function *can be used in a variety of ways* – we'll start by using it to stop the player from falling through the ground

To use the map collision function (called **mcollide** for short), after pasting the function on a separate tab, go into your **player movement function**

After the code for jumping, **add an if statement that calls the mcollide function**

Plug in these values in the parentheses:

- **plyr** for the object
- “**down**” for the direction
- **0** for the sprite flag

```
0123+ 012345
PLYR.DX += PLYR.XSPD
END -- END IF BTn(0)

-- JUMP
IF BTn(0) OR BTnP(0) THEN
    PLYR.DY -= PLYR.YSPD
END -- END IF BTnP(0)/0

-- STOP FALLING WHEN A SOLID
-- TILE IS BELOW THE PLAYER
IF MCOLLIDE(PLYR,"DOWN",0) THEN
    PLYR.DY = 0
END -- END IF MCOLLIDE

-- UPDATE X,Y BY THE CALCULATED
-- CHANGE (DELTA X, DELTA Y)
PLYR.X += PLYR.DX
PLYR.Y += PLYR.DY
```

LINE 31/38 301/8192 E

Plug in these values in the parentheses:

- **plyr** for the object
- “**down**” for the direction
- 0 for the sprite flag

In the event that there is a map tile beneath the player with sprite flag 0 turned on, mcollide will **return** a value of **true**

Writing:

```
if mcollide(plyr,"down",0)
```

is shorthand for writing:

```
if mcollide(plyr,"down",0) == true
```



```
0123 + 0 0 0 0 0 0
PLYR.DX += PLYR.XSPD
END -- END IF BTn(0)

-- JUMP
IF BTn(0) OR BTnP(0) THEN
    PLYR.DY -= PLYR.YSPD
END -- END IF BTnP(0)/0

-- STOP FALLING WHEN A SOLID
-- TILE IS BELOW THE PLAYER
IF MCOLLIDE(PLYR,"DOWN",0) THEN
    PLYR.DY = 0
END -- END IF MCOLLIDE

-- UPDATE X,Y BY THE CALCULATED
-- CHANGE (DELTA X, DELTA Y)
PLYR.X += PLYR.DX
PLYR.Y += PLYR.DY
```

LINE 31/38 301/8192 E

If the condition (a solid tile is below the player) is met, we want to stop the player from falling

We can do this by setting the player's **dy** (change in y) to **0**

This means we're *adding 0 to the player's y coordinate* each frame

Effectively negating gravity



```
0123+ 0123+
PLYR.DX += PLYR.XSPD
END -- END IF BTn(0)

-- JUMP
IF BTn(0) OR BTnP(0) THEN
    PLYR.DY -= PLYR.YSPD
END -- END IF BTnP(0)/0

-- STOP FALLING WHEN A SOLID
-- TILE IS BELOW THE PLAYER
IF MCOLLIDE(PLYR,"DOWN",0) THEN
    PLYR.DY = 0
END -- END IF MCOLLIDE

-- UPDATE X,Y BY THE CALCULATED
-- CHANGE (DELTA X, DELTA Y)
PLYR.X += PLYR.DX
PLYR.Y += PLYR.DY
```

LINE 31/38 301/8192 E

> RUNTIME ERROR LINE 40 TAB 3

HY1=OBJ.Y+OBJ.H  
ATTEMPT TO PERFORM ARITHMETIC ON  
FIELD 'H' (A NIL VALUE)  
IN MCOLLIDE LINE 40 (TAB 3)  
IN MOVE\_PLYR LINE 29 (TAB 2)  
IN \_UPDATE LINE 16 (TAB 0)  
AT LINE 0 (TAB 0)

> SAVE

SAVED PLATFORMER\_07\_MAP\_COLLISION  
n.PB \*\*\*TEMP DISK\*\*\*

>

*Try running the game, and you'll likely see the following error*

This is because the `mcollide` function **requires the object plugged in (in our case, `plyr`)**, to have variables named **w** and **h** (for the width and height)

These values are necessary for calculating the position of the map tile adjacent to the player

*The error is saying that the variable h has not been defined*

so we can fix this by giving our player variables named **w** and **h**

0 1 2 3 +

0 1 2 3 < >

```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR={} -- TABLE FOR PLYR OBJ
PLYR.n=1 -- SPRITE NUMBER

-- X, Y COORDINATES
PLYR.X=7*8 -- X=56 PIXELS
PLYR.Y=11*8 -- Y=88 PIXELS
```

```
-- WIDTH AND HEIGHT IN PIXELS
-- NEEDED FOR MAP COLLISION
PLYR.W=8
PLYR.H=8
```

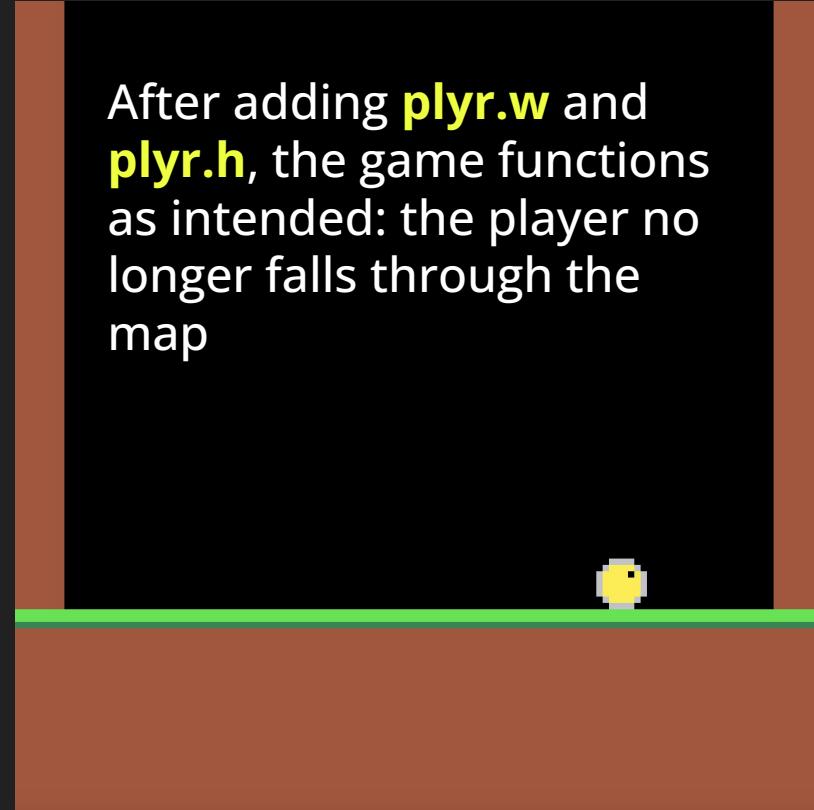
```
PLYR.XSPD=0.5 -- X SPEED
PLYR.YSPD=4 -- Y SPEED
```

```
PLYR.DX=0 -- CHANGE IN X
PLYR.DY=0 -- CHANGE IN Y
```

LINE 11/21

309/8192 E

After adding **plyr.w** and **plyr.h**, the game functions as intended: the player no longer falls through the map



But you'll notice *a side effect*  
*is that the player can no  
longer jump*

The player's dy is always  
being set to 0 now, so our  
*map collision is overriding the  
jump*

# Step 08: Jumping, Part 2

Download Example File: [platformer\\_08\\_jumping2.p8](#)

0 1 2 3 +

0 1 2 3 < >

```
-- MOVE RIGHT
IF BTn(0) THEN
    PLR.X += PLR.XSPD
END -- END IF BTn(0)

-- JUMP
IF BTnP(0) OR BTnP(8) THEN
    PLR.DY -= PLR.YSPD
END -- END IF BTnP(0/8)

-- STOP FALLING WHEN A SOLID
-- TILE IS BELOW THE PLAYER
IF MCOLLIDE(PLR, "DOWN", 0)
    AND PLR.DY > 0 THEN
    PLR.DY = 0
END -- END IF MCOLLIDE

-- UPDATE X, Y BY THE CALCULATED
-- CHANGE (DELTAX, DELTAY)
LINE 30/39
```

317/8192 E

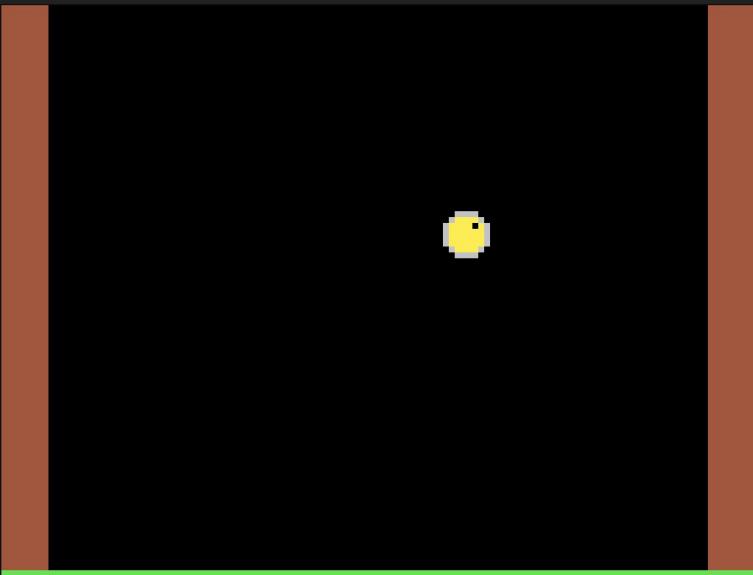
Our problem was that *our map collision function was preventing the player from jumping*, because *dy is always being set to 0 once the player is touching the ground*

We can fix this and allow jumping again by **adding a condition** to our if statement for map collision:

Use the keyword **AND** to stack conditions

We want to specify that *we only want to stop the player from moving if they are already moving down* (in other words, **if dy > 0**)

This allows the player to jump again, but *there's another issue: they can still jump indefinitely* (essentially floating like in Flappy Bird)



We'll need to track whether the player is in the air or not, and *only allow them to jump if they're not already in the air*

0 1 2 3 +

0 1 2 3 < >

```
-- WIDTH AND HEIGHT IN PIXELS
-- NEEDED FOR MAP COLLISION
PLYR.W=8
PLYR.H=8

PLYR.XSPD=0.5 -- X SPEED
PLYR.YSPD=4 -- Y SPEED

PLYR.DX=0 -- CHANGE IN X
PLYR.DY=0 -- CHANGE IN Y

-- PLAYER STATE
PLYR.LANDED=False

END -- END FUNCTION MAKE_PLYR()
```

We'll need to track whether the player is in the air or not, and **only** allow them to jump if they're not already in the air – I added a variable called `plyr.landed` to my `make_plyr()` function, and set it to **false** initially

```
0 1 2 3 +
```

```
0 1 2 3 < >
```

```
-- WIDTH AND HEIGHT IN PIXELS  
-- NEEDED FOR MAP COLLISION
```

```
PLYR.W=8  
PLYR.H=8
```

```
PLYR.XSPD=0.5 -- X SPEED  
PLYR.YSPD=4 -- Y SPEED
```

```
PLYR.DX=0 -- CHANGE IN X  
PLYR.DY=0 -- CHANGE IN Y
```

```
-- PLAYER STATE  
PLYR.LANDED=FALSE
```

```
END -- END FUNCTION MAKE_PLYR()
```

We'll need to **track whether the player is in the air or not, and *only* allow them to jump if they're not already in the air** – I added a variable called `plyr.landed` to my `make_plyr()` function, and set it to **false** initially

```
0 1 2 3 +
```

```
0 1 2 3 < >
```

```
PLYR.DX += PLYR.XSPD  
END -- END IF BTNP(1)
```

```
-- JUMP
```

```
IF BTNP(2) OR BTNP(3) THEN  
PLYR.DY -= PLYR.YSPD  
PLYR.LANDED=FALSE  
END -- END IF BTNP(2/3)
```

```
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER
```

```
IF NCOLLIDE(PLYR,"DOWN",0)  
AND PLYR.DY > 0 THEN  
PLYR.DY = 0
```

```
PLYR.LANDED=TRUE  
END -- END IF NCOLLIDE
```

```
-- UPDATE X,Y BY THE CALCULATED  
-- CHANGE (DELTAX, DELTAY)
```

```
LINE 33/41 326/8192 E
```

We can then **set landed to true when collision is detected with the ground, and back to false again when the jump key is pressed**

Finally, we need to add one more condition to allow the player to jump:

The player can only jump if they are not already in the air – in other words:

**plyr.landed** must be true

(we could express this as  
player.landed == true, or simply  
player.landed for short)

*If you wanted to allow a double jump, you could include a variable tracking the number of jumps since last landing*

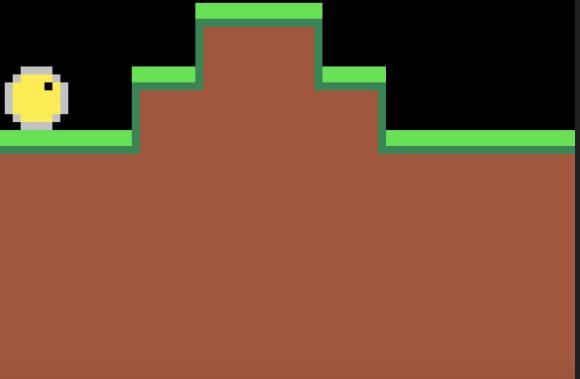
```
0123 + 04567890
PLYR.DX += PLYR.XSPD
END -- END IF BTNP(0)

-- JUMP
IF BTNP(0) OR BTNP(8) THEN
  IF PLYR.LANDED THEN
    PLYR.DY -= PLYR.YSPD
    PLYR.LANDED=FALSE
  END -- END IF PLYR.LANDED
END -- END IF BTNP(0/8)

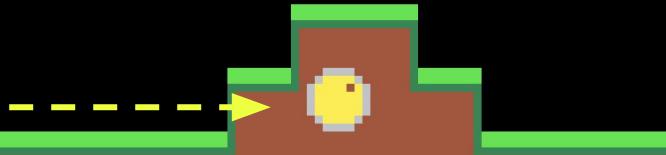
-- STOP FALLING WHEN A SOLID
-- TILE IS BELOW THE PLAYER
IF MCOLLIDE(PLYR,"DOWN",0)
AND PLYR.DY > 0 THEN
  PLYR.DY = 0
  PLYR.LANDED=TRUE
END -- END IF MCOLLIDE
```

# Step 09: Walls

Download Example File: [platformer\\_09\\_walls.p8](#)



We added collision with the ground, but what if the player approaches a solid tile *horizontally*?



*Right now, they would move right through walls as if they don't exist*

Let's fix that by using our map collision function **mcollide** to check for flag 0 to the player's **left** and **right**

0 1 2 3 +

( ) [ ] { }

```
IF NOT COLLIDE(PLYR, "DOWN", 0)
AND PLYR.DY > 0 THEN
    PLYR.DY = 0
    PLYR.LANDED=TRUE
END -- END IF NOT COLLIDE DOWN
```

```
-- PREVENT MOVEMENT THROUGH
-- WALLS TO THE LEFT
IF NOT COLLIDE(PLYR, "LEFT", 0)
AND PLYR.DX < 0 THEN
    PLYR.DX = 0
END -- END IF NOT COLLIDE LEFT
```

```
-- PREVENT MOVEMENT THROUGH
-- WALLS TO THE RIGHT
IF NOT COLLIDE(PLYR, "RIGHT", 0)
AND PLYR.DX > 0 THEN
    PLYR.DX = 0
END -- END IF NOT COLLIDE RIGHT
```

After we check for solid tiles below the player, we can **check to the left and right**

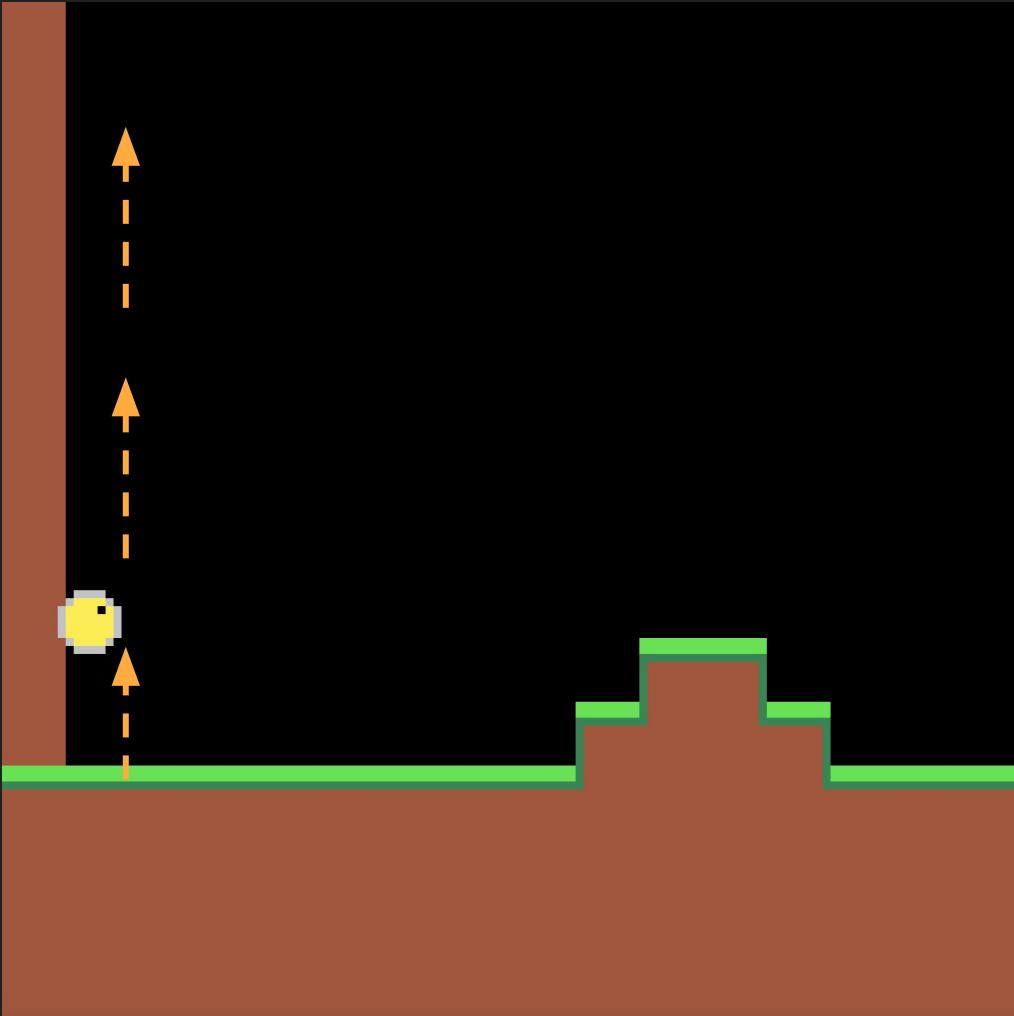
Learning from what happened when our jumping was accidentally disabled, we'll make sure we **only apply this collision check (and stop the player's horizontal movement) if they are moving in that direction to begin with**

In other words, if dx is less than 0 (moving left) or greater than 0 (moving right)

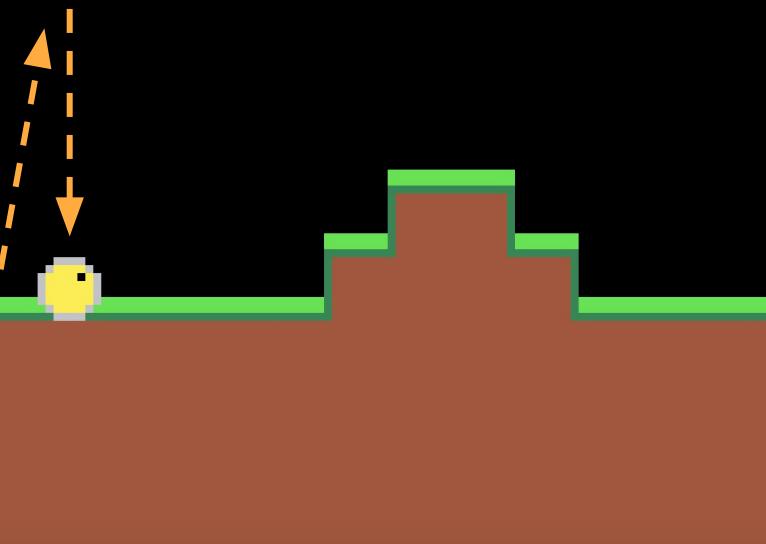


Now the player is prevented from moving through walls, but *sometimes they get stuck a few pixels into the walls*

This is because the player's speed occasionally allows them to still move between frames where collision is being checked



The player can *exploit this glitch* by getting stuck in the wall on the left or right side and then *jumping up the wall indefinitely*



And sometimes *after jumping, the player gets stuck a few pixels into the floor*

You'll notice *this prevents horizontal movement*, because technically the wall tile is being detected adjacent to the player in this case

# Step 10: Movement Refined

Download Example File: [platformer\\_10\\_movement\\_refined.p8](#)

This YouTube tutorial by [NerdyTeachers](#) (Video #5 in the playlist) provides a solution by calculating the number of extra pixels the player has fallen (into the floor) and correcting the player's position by removing that number of pixels



NerdyTeachers.com

```
IF TIP( )  
PLAYER.LANDED THEN  
AYER.DY-=PLAYER.BOOST  
AYER.LANDED=FALSE  
  
ECK COLLISION_UP AND DOWN  
AYER.DY>0 THEN  
AYER.FALLING=TRUE  
AYER.LANDED=FALSE  
AYER.JUMPING=FALSE  
  
COLLIDE_MAP(PLAYER,"DOWN",0) THEN  
PLAYER.LANDED=TRUE  
PLAYER.FALLING=FALSE  
PLAYER.DY=0  
PLAYER.Y-=(PLAYER.Y+PLAYER.H)/28  
0  
LINE 47/56
```

0 1 2 3 +

() [] {} &lt; &gt;

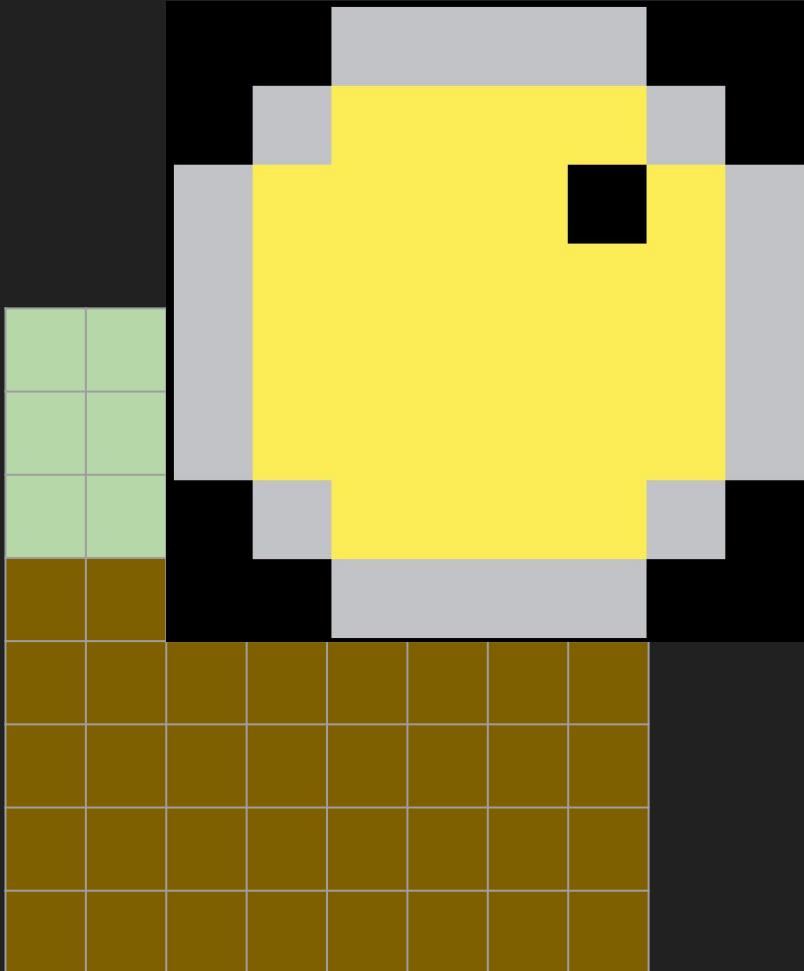
```
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER  
IF MCOLLIDE(PLYR,"DOWN",0)  
AND PLYR.DY > 0 THEN  
PLYR.DY = 0  
PLYR.LANDED=TRUE  
  
-- BECAUSE OF VERTICAL SPEED.  
-- THE PLAYER CAN FALL A FEW  
-- PX INTO THE FLOOR. THIS  
-- CALCULATES HOW MANY PX AND  
-- RE-ADJUSTS Y (CREDIT TO  
-- NERDYTEACHERS.COM FOR THIS  
-- FORMULA)  
  
PLYR.Y-=  
((PLYR.Y+PLYR.H+1)X8)-1  
END -- END IF MCOLLIDE DOWN
```

LINE 1/67

378/8192 E

Because the solution uses a bit more complicated math (*the % operator calculates the remainder of an equation after division*), I'll allow you to **copy and paste** this into your code, as I've done with my in-progress game here

Add it to the end of your if statement handling collision below the player



```
plyr.y-=((plyr.y+plyr.h+1)%8)-1
```

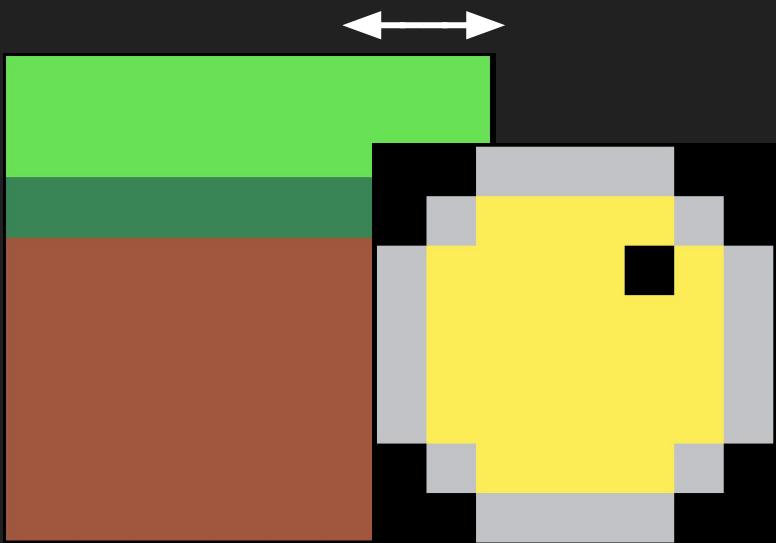
The important thing is understanding generally what this math accomplishes:

This equation:

- Takes the y coordinate one pixel below the player ( $y+h+1$ )
- Divides it by 8
- And calculates the **remainder** by using the **%** operator (called **modulo** or **modulus**)
- Then subtracts that value from the player's y to move it up however many pixels it's off by

We can apply this logic to  
the **x-axis** as well, to prevent  
the player from getting  
stuck in walls

*This code calculates how far away the player is (on both the left and right) from the nearest x coordinate divisible by 8 (which would indicate a tile edge)*



```
0123 + 098<10
-- CALCULATE HOW FAR THE PLAYER
-- IS FROM A TILE WHOSE X
-- COORDINATE IS DIVISIBLE BY 8
-- (INDICATING THE TILE'S EDGE)
FIXL=1-((PLYR.X+1)X8)
FIXR=((PLYR.X+PLYR.W+1)X8)-1

-- PREVENT OVERCORRECTION IF
-- MORE THAN HALFWAY PAST THE
-- ADJACENT TILE
IF ABS(FIXL) > 4 THEN
  FIXL = 8-ABS(FIXL)
ENDIF -- END IF ABS(FIXL) > 4

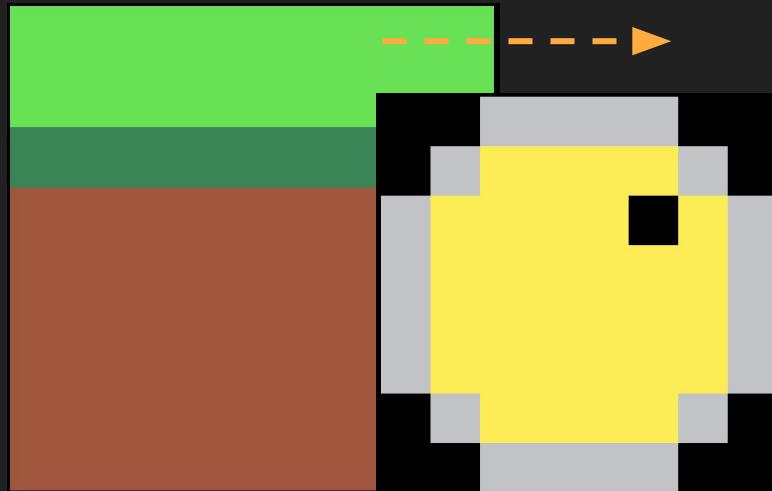
-- PREVENT MOVEMENT THROUGH
-- WALLS TO THE LEFT
IF MCOLLIDE(PLYR,"LEFT",0)
  AND PLYR.DX < 0 THEN
LINE 65/89          427/8192 E
```

In your player movement function, after checking collision below, but before checking collision to the left and right, add the code that calculates variables called **fixl** (fix left) and **fixr** (fix right)

```
0123 + 0123+  
-- CALCULATE HOW FAR THE PLAYER  
-- IS FROM A TILE WHOSE X  
-- COORDINATE IS DIVISIBLE BY 8  
-- (INDICATING THE TILE'S EDGE)  
FIXL=1-((PLYR.X+1)%8)  
FIXR=((PLYR.X+PLYR.W+1)%8)-1  
  
-- PREVENT OVERCORRECTION IF  
-- MORE THAN HALFWAY PAST THE  
-- ADJACENT TILE  
IF ABS(FIXL) > 4 THEN  
    FIXL1=FIXL --PRECORRECTION  
    FIXL = 8-ABS(FIXL)  
    FIXL2=FIXL -- POSTCORRECTION  
END -- END IF ABS(FIXL) > 4
```

Then, in your if statement checking collision with wall tiles on the left:

Add **fixl** to the player's x to move the player the necessary number of pixels to the right, out of the wall to the left



```
-- PREVENT MOVEMENT THROUGH  
-- WALLS TO THE LEFT  
IF MCOLLIDE(PLYR,"LEFT",0)  
AND PLYR.DX < 0 THEN  
    PLYR.DX = 0
```

```
-- APPLY CORRECTION TO AVOID  
-- GETTING STUCK IN THE WALL  
    PLYR.X+=FIXL  
END -- END IF MCOLLIDE LEFT
```

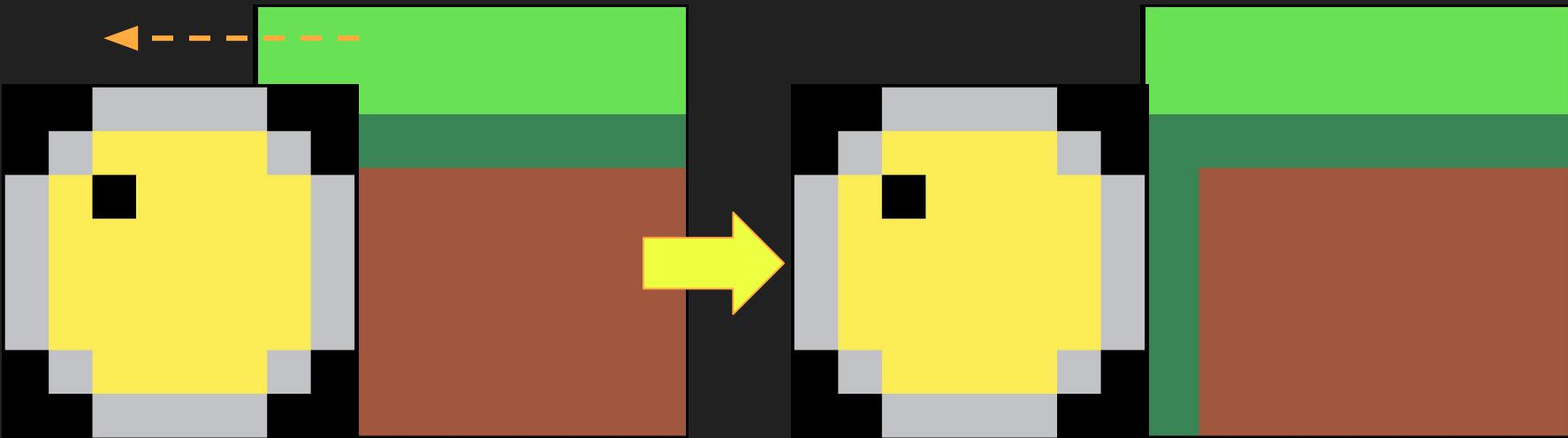


Then, in your if statement checking collision with wall tiles on the right:

**Subtract fixr from the player's x** to move the player the necessary number of pixels to the left, out of the wall to the right

```
-- PREVENT MOVEMENT THROUGH  
-- WALLS TO THE RIGHT  
IF MCOLLIDE(PLYR,"RIGHT",0)  
AND PLYR.DX > 0 THEN  
    PLYR.DX = 0
```

```
-- APPLY CORRECTION TO AVOID  
-- GETTING STUCK IN THE WALL  
    PLYR.X-=FIXR  
END -- END IF MCOLLIDE RIGHT
```



# Bonus: map collision correction illustrated

Download Example File: [platformer\\_10a\\_correction\\_illustrated.p8](#)

Our player movement is now pretty clean!

Next, we'll center the **camera** on the player so we can explore the rest of the map

# Step 11: Camera

Download Example File: [platformer\\_11\\_camera.p8](#)

# Positioning the Camera

- There is a built-in PICO-8 function for setting the camera, called camera()
- But it places the camera at **0,0** (the top-left corner of the screen)
- ***We need our camera to follow the player***, so we'll write a function to adjust its position based on the player's position

There is a built-in PICO-8 function for setting the camera, called [camera\(\)](#)

But it places the camera at **0,0**  
(the top-left corner of the screen)

***We need our camera to follow the player,*** so we'll write a function to adjust its position based on the player's position

```
01234 + 098765  
-- CAMERA  
FUNCTION CAM_FOLLOW()  
END -- END FUNCTION CAM_FOLLOW()
```

I'll start this function on the next tab

```
01234 + 09876543210  
-- TAB 2: MOVE PLAYER  
-- TAB 3: MAP COLLISION FUNCTION  
-- TAB 4: CAMERA FUNCTION  
  
FUNCTION _INIT()  
    FRIC = 0.85 -- FRICTION  
    GRAV = 0.3 -- GRAVITY  
    MAKE_PLAYER() -- TAB 1  
END -- END FUNCTION _INIT()  
  
FUNCTION _UPDATE()  
    MOVE_PLAYER() -- TAB 2  
    CAM_FOLLOW() -- TAB 4  
END -- END FUNCTION _UPDATE()
```

LINE 18/30

444/8192 ⏴

```
01234 + 09876543210  
-- CAMERA  
FUNCTION CAM_FOLLOW()  
END -- END FUNCTION CAM_FOLLOW()
```

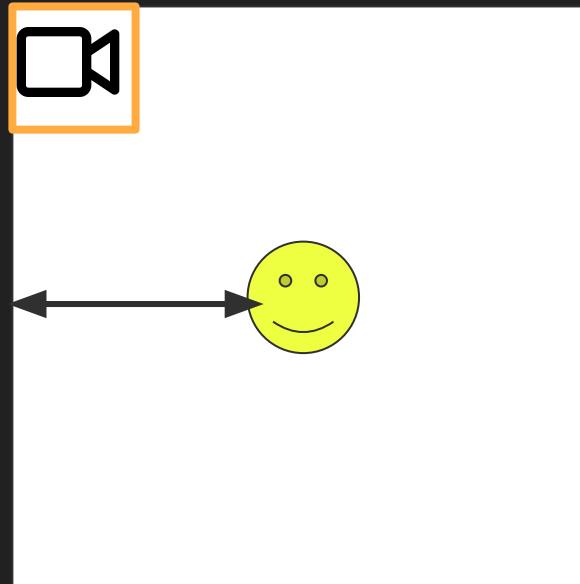


I'll start this function on the next tab

And I'll **call the function** in my  
`_update()` loop

# Positioning the Camera

If I want the player to always be in the center, I need to set the **camera half a screen to the player's left**

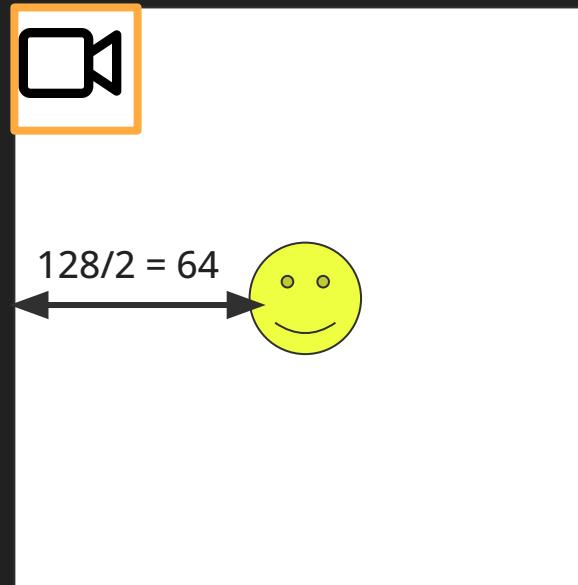


# Positioning the Camera

If I want the player to always be in the center, I need to set the **camera half a screen to the player's left**

The screen is **128** pixels wide, so half of that is **64**

That means that we must **subtract 64** from the player's x coordinate to get the camera's x coordinate

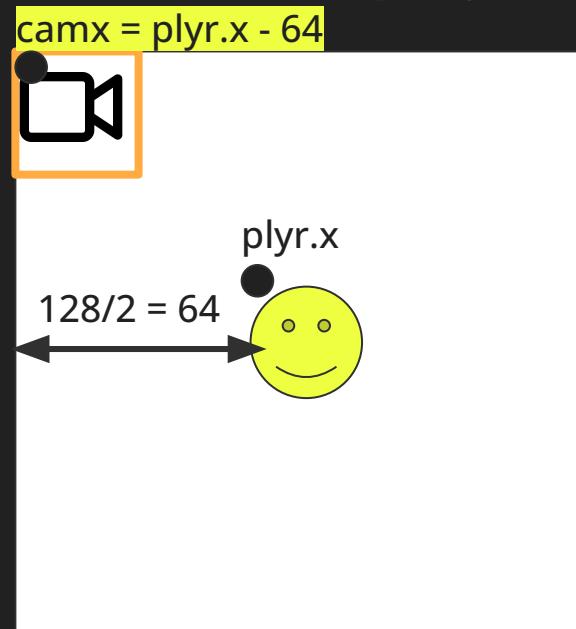


# Positioning the Camera

If I want the player to always be in the center, I need to set the **camera half a screen to the player's left**

The screen is **128** pixels wide, so half of that is **64**

That means that we must **subtract 64** from the player's x coordinate to get the camera's x coordinate



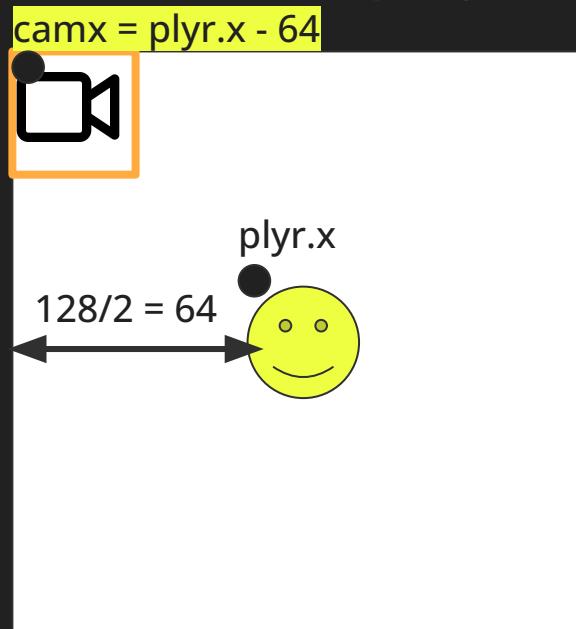
We can express this as:  
`camx = plyr.x - 64`

# Positioning the Camera

If I want the player to always be in the center, I need to set the **camera half a screen to the player's left**

The screen is **128** pixels wide, so half of that is **64**

That means that we must **subtract 64** from the player's x coordinate to get the camera's x coordinate



We can express this as:  
**camx = plyr.x - 64**

I can set camx to this value in my camera function

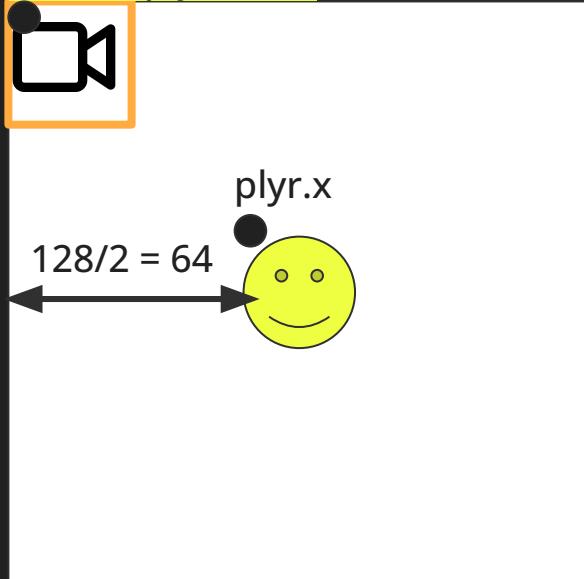
```
01234+ [ ] --CAMERA  
FUNCTION CAM_FOLLOW()  
CAMX = PLYR.X - 64  
CAMY = 0  
CAMERA(CAMX,CAMY)  
END -- END FUNCTION CAM_FOLLOW()
```

Let's zoom in on the code

Here I'm setting **camx** equal to 64 less than the player's x position

```
0 1 2 3 4 + 0 1 2 3 4 < > <= >= <>
-- CAMERA
FUNCTION CAM_FOLLOW()
    CAMX = PLYR.X - 64
    CAMY = 0
    CAMERA(CAMX,CAMY)
END -- END FUNCTION CAM_FOLLOW()
```

camx = plyr.x - 64



Let's zoom in on the code

Here I'm setting **camx** equal to 64 less than the player's x position

```
-- CAMERA
FUNCTION CAM_FOLLOW()
    CAMX = PLR.X - 64
    CAMY = 0
    CAMERA(CAMX,CAMY)
END -- END FUNCTION CAM_FOLLOW()
```

The Scratch script shows a function named "CAM\_FOLLOW()". Inside the function, the variable "CAMX" is set to the value of "PLR.X" minus 64. The variable "CAMY" is set to 0. The "CAMERA" block is then used with the values of "CAMX" and "CAMY". Finally, the word "END" is used to end the function definition.

*The camera's y position (**camy**) doesn't need to change, as long as we've drawn our level exactly 16 tiles tall (one screen height)*

So I'll leave **camy** as 0 for now

*But I'll keep it a variable in case I want to change it later – which I will if the player must be transported to a level that's lower on the map*

Let's zoom in on the code

Here I'm setting **camx** equal to 64 less than the player's x position



```
0 1 2 3 4 + () × □ < > 
-- CAMERA
FUNCTION CAM_FOLLOW()
    CAMX = PLYR.X - 64
    CAMY = 0
    CAMERA(CAMX,CAMY)
END -- END FUNCTION CAM_FOLLOW()
```

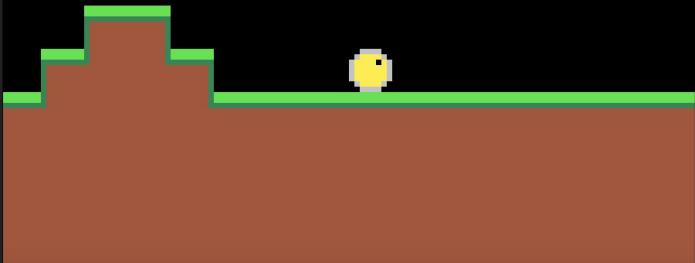
A Scratch script titled "CAMERA". It contains a function named "CAM\_FOLLOW()". Inside the function, the variable "CAMX" is set to the value of "PLYR.X" minus 64. The variable "CAMY" is set to 0. Finally, the "CAMERA" block is used with arguments "CAMX" and "CAMY". An arrow points from the text "Finally, I must plug camx and camy into the camera() function to actually move it to those coordinates" to the "CAMERA" block.

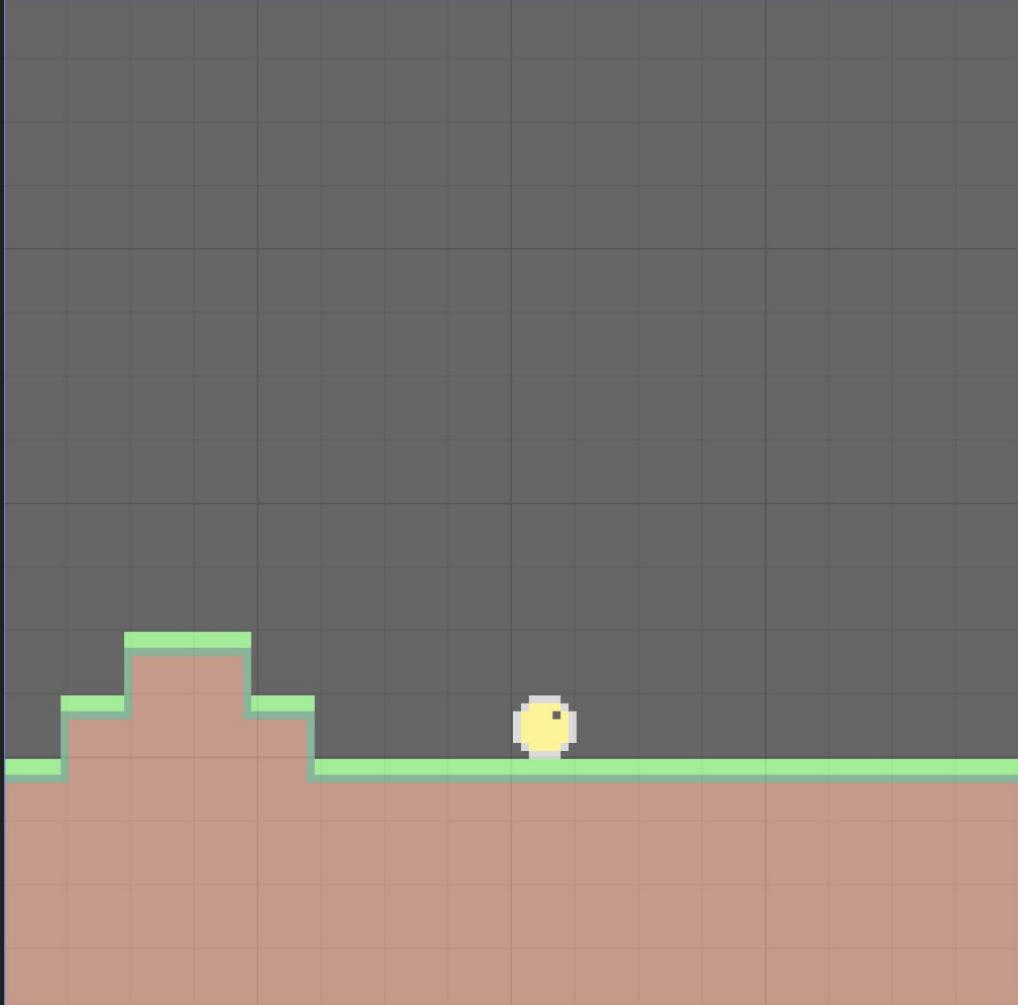
Finally, I must plug camx and camy into the camera() function to actually move it to those coordinates

```
0 1 2 3 4 + 0 1 2 3 4 +  
-- CAMERA  
FUNCTION CAM_FOLLOW()  
CAMX = PLR.X - 64  
CAMY = 0  
CAMERA(CAMX,CAMY)  
END -- END FUNCTION CAM_FOLLOW()
```

The result is pretty good – I can now move to the right, and the camera follows the player, revealing more of the map at the same time

But it's not 100% perfect



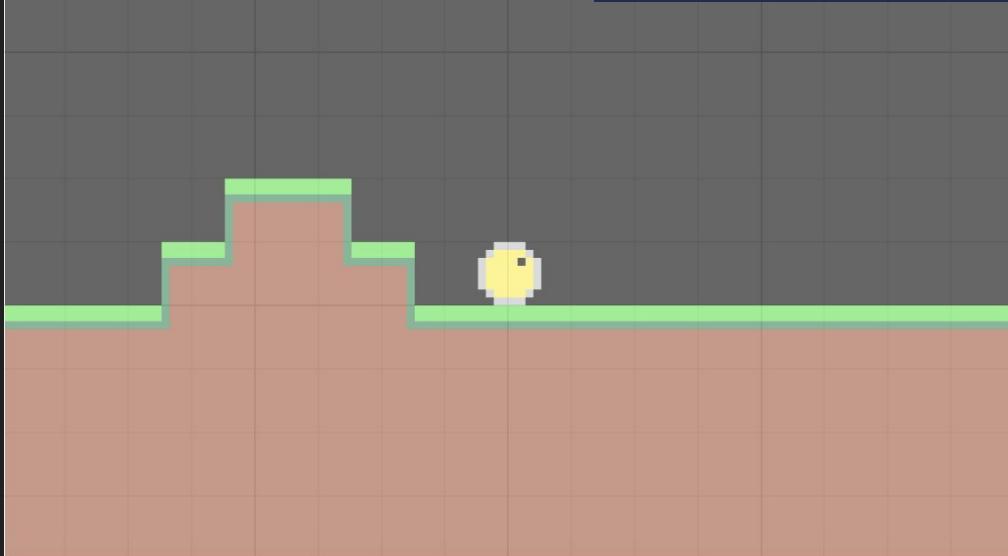


If I were to draw a grid over my screen, I can see that *the player sprite is just to the right of the center*

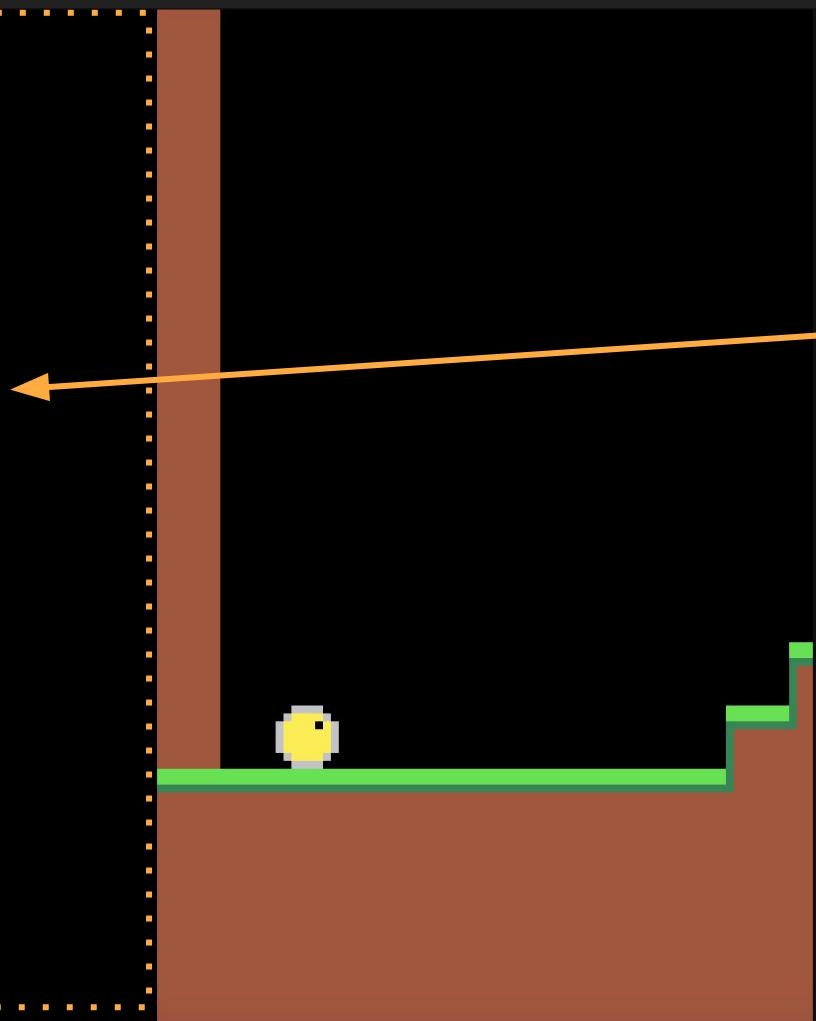
This is because the player is being drawn from the left

So to fine-tune our camera-positioning equation, we need to *add half the player's width* back after subtracting half the screen width

```
01234 + 089<: 
-- CAMERA
FUNCTION CAN_FOLLOW()
    CANX = PLYR.X - 64 + PLYR.W/2
    CANY = 0
    CAMERA(CANX, CANY)
END -- END FUNCTION CAN_FOLLOW()
```



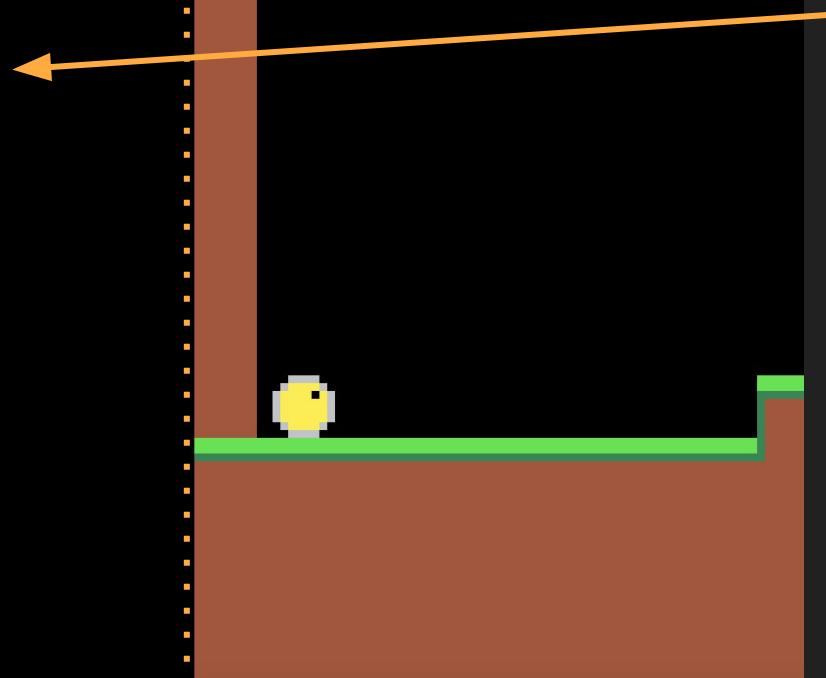
You can see that by adding half the player's width, we perfect the equation and place the player directly in the center of the screen



There is an issue, however, if the player moves close enough to the left or right edge of the map:

*We can see the blank space to the left of the map* here because the camera is effectively 60 pixels to the left of the player, but the player is less than 60 pixels from the edge of the screen

CAMX: -50  
PLYR.X: 10



There is an issue, however, if the player moves close enough to the left or right edge of the map:

*We can see the blank space to the left of the map* here because the camera is effectively 60 pixels to the left of the player, but the player is less than 60 pixels from the edge of the screen

This becomes obvious when I print both the camera's and player's x coordinates

I'll add an **if** statement to override the player-centering camera behavior

If the camera's x coordinate is less than 0 (to the left of the map, which is why blank space is showing in that case), simply reset it to 0

Effectively creating a minimum value for camera x

```
01234 + () [] < >
-- CAMERA
FUNCTION CAM_FOLLOW()
CAMX = PLYR.X - 64 + PLYR.W/2
CAMY = 0
-- PIN CAMERA TO LEFT EDGE
IF CAMX < 0 THEN
CAMX = 0
END
CAMERA(CAMX, CAMY)
END -- END FUNCTION CAM_FOLLOW()
```

CARX: 0  
PLHR,X: B



```
01234 + 0WWW  
-- CAMERA  
FUNCTION CAM_FOLLOW()  
CAMX = PLYR.X - 64 + PLYR.W/2  
CAMY = 0  
  
-- PIN CAMERA TO LEFT EDGE  
IF CAMX < 0 THEN  
CAMX = 0  
END  
  
CAMERA(CAMX, CAMY)  
END -- END FUNCTION CAM_FOLLOW()
```

This solution prevents the camera from moving any farther to the left than the left edge of the map

Now that the player can explore  
the rest of the level, you should  
design the rest of it



Remember to make sure your  
level is precisely 16 tiles tall

This will make controlling  
the camera's y coordinate  
much simpler

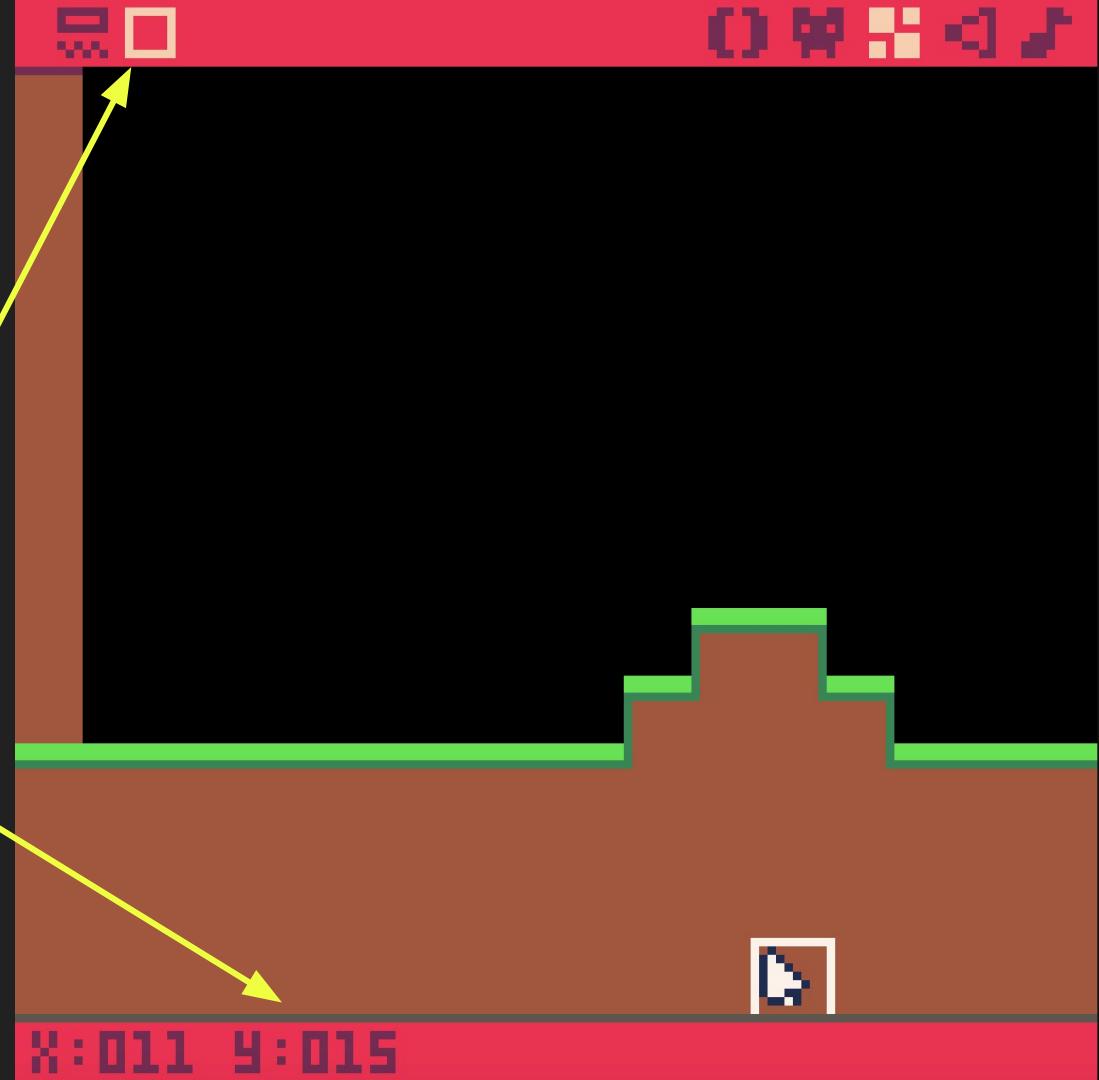
X:011 Y:015

Make sure your level is precisely 16 tiles tall

Click the square icon in the map editor to hide the lower tools and show the whole map screen

Hold space to see a grid

Mouse over the lowest cell on that grid - you should see **Y: 015** in the lower left of your screen, indicating the lowest tile on the first screen (*15, not 16, since it starts at 0*)



One thing you'll likely want to add  
is some **pits** to jump over

Let's cover how you can **respawn**  
the player after they fall down a pit

# Step 12: Pits & Respawning

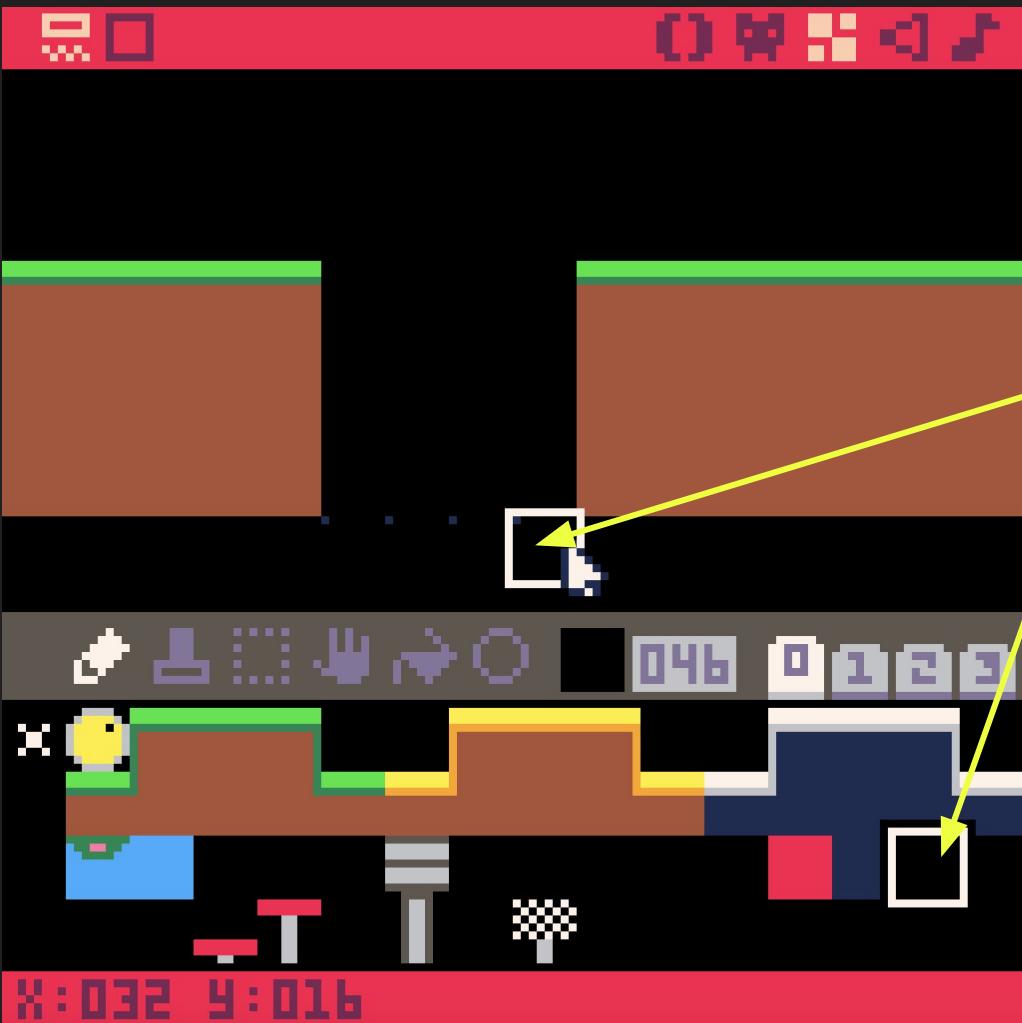
Download Example File: [platformer\\_12\\_pits\\_and\\_respawn.p8](#)



To detect when a player has fallen into a pit, we'll need map tiles that are designated as pits

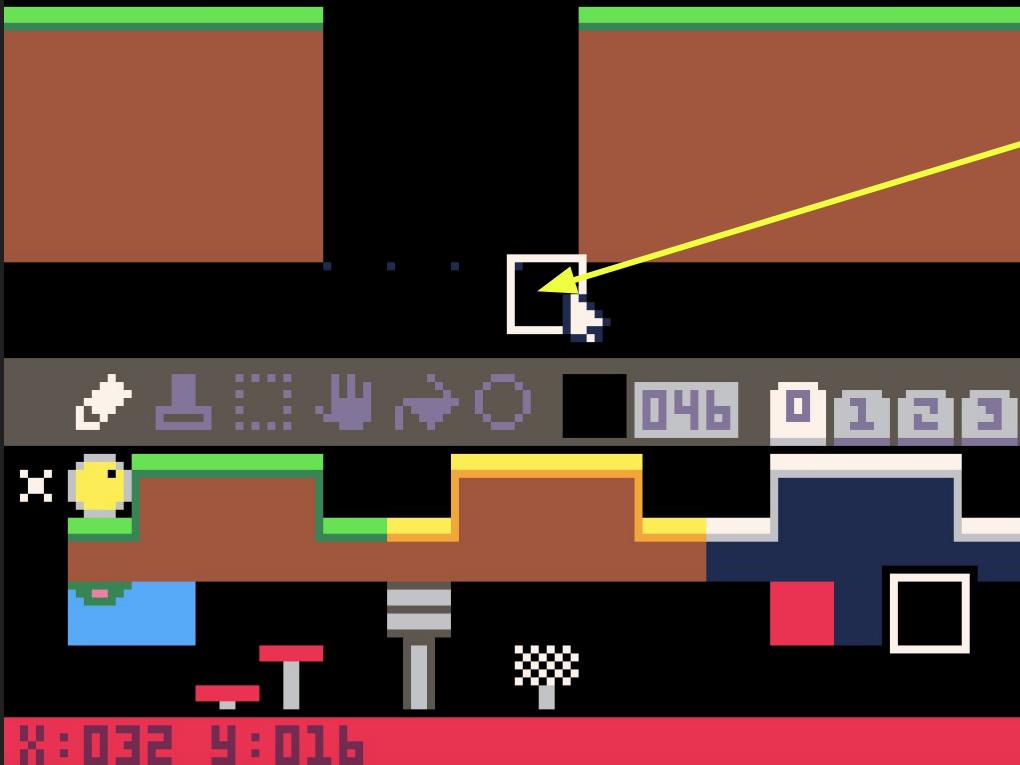
That means using a **sprite flag** to mark sprites as pit tiles

I chose a blank tile and turned **flag 7** on to use for pits



Next, I can paint with this specially flagged pit tile in my map editor

A convenient glitch in the map editor is that if you paint with blank tiles, they leave a blue dot in the top left corner of each tile, so you can actually see where you've placed your pits (*only in editing mode*)



I strategically placed my flagged tiles at the **bottom** of the pit

This will allow the player to fall into the pit for a few frames before they reach the flagged tiles

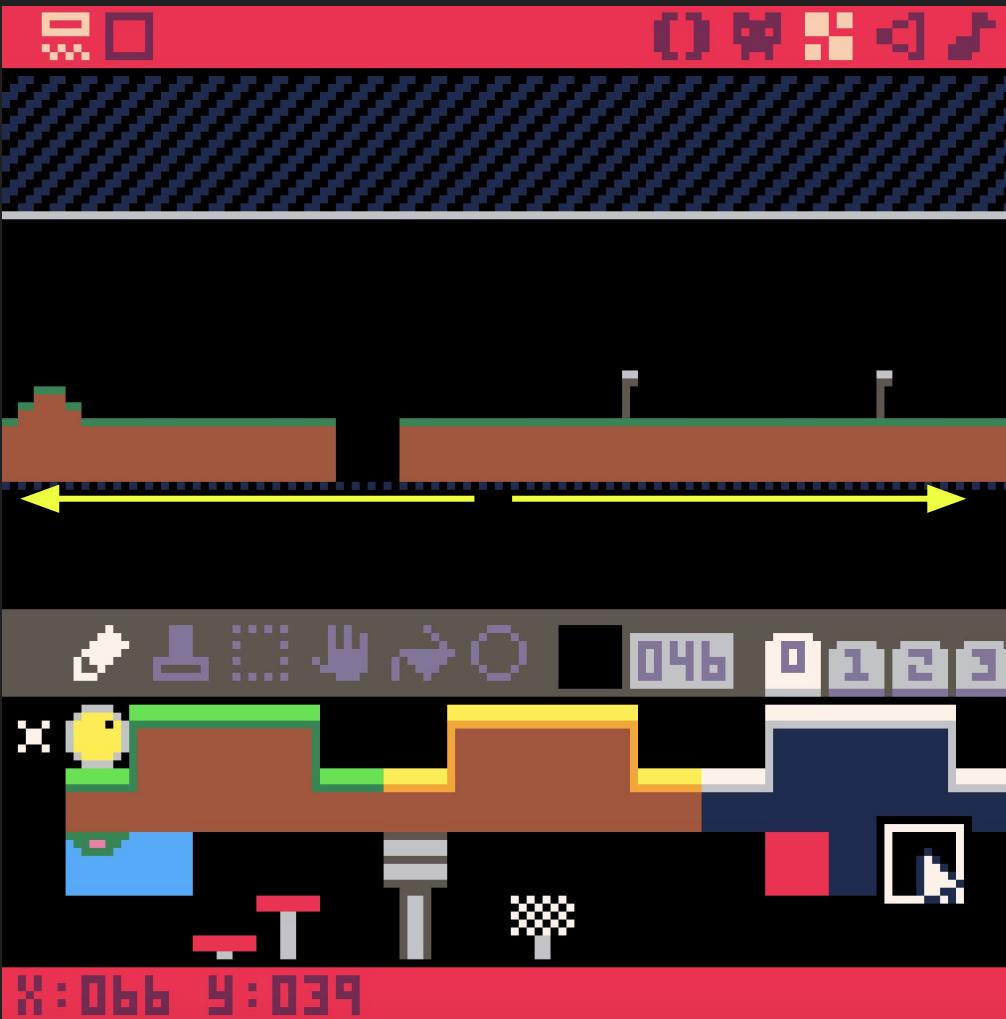
*Otherwise, they would "die" very quickly, and might not see what happened*



If I want to be very thorough, I could place these flagged tiles all across the bottom of my level, even where there aren't any pits

This would allow me to respawn the player if they ever somehow get out of bounds because of an unforeseen glitch

I call this a "**death plane**"



If I want to be very thorough, I could place these flagged tiles all across the bottom of my level, even where there aren't any pits

This would allow me to respawn the player if they ever somehow get out of bounds because of an unforeseen glitch

I call this a “**death plane**”

You can see the blue dots indicating blank tiles when you zoom out, too

0 1 2 3 4 5 +

0 1 2 3 4 5 < >

```
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER  
IF MCOLLIDE(PLYR,"DOWN",0)  
AND PLYR.DY > 0 THEN  
    PLYR.DY = 0  
    PLYR.LANDED=TRUE  
  
-- BECAUSE OF VERTICAL SPEED.  
-- THE PLAYER CAN FALL A FEW  
-- PX INTO THE FLOOR. THIS  
-- CALCULATES HOW MANY PX AND  
-- RE-ADJUSTS Y (CREDIT TO  
-- NERDYTEACHERS.COM FOR THIS  
-- FORMULA)  
    PLYR.Y-=  
    ((PLYR.Y+PLYR.H+1)%8)-1  
END -- END IF MCOLLIDE DOWN
```

To actually implement death and respawn, I can use our map collision function, **mcollide()**

Our condition (if statement) to check whether the player is touching a pit / the death plane (a tile with flag 7 turned on) will be very **similar to checking for solid tiles below the player** to stop falling

I'll copy this entire chunk of code from my `move_plyr()` function and paste it into a new function I'll write to handle respawning

```
0 1 2 3 4 5 + () [] < >
```

```
-- PITS AND RESPAWNING
FUNCTION RESPAWN()
END -- END FUNCTION RESPAWN()
```

I've created a new function called **respawn()** on tab 5

It'll detect when the player touches a pit tile, then subtract a life and return the player to the start of the level (*or last checkpoint once those are implemented*)

I'll *call* the respawn() function in my **\_update()** loop

```
0 1 2 3 4 5 + () [] < >
```

```
-- STARTING POSITION FOR CAMERA
CRANX = 0
CRANY = 0
END -- END FUNCTION _INIT()
```

```
FUNCTION _UPDATE()
MOVE_PLYR() -- TAB 2
CAM_FOLLOW() -- TAB 4
RESPAWN() -- TAB 5
END -- END FUNCTION _UPDATE()
```

```
FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
MAP() -- DRAW MAP
SPR(PLYR.N, PLYR.X, PLYR.Y)
```

```
0 1 2 3 4 5 + ()  
-- PITS AND RESPawning  
FUNCTION RESPAWN()  
END -- END FUNCTION RESPAWN()
```

In the new **respawn()** function, I've pasted the chunk of code that checks for collision

We'll need to modify this, but it's a good start because the process will be similar

```
0 1 2 3 4 5 + ()  
-- PITS AND RESPawning  
FUNCTION RESPAWN()  
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER  
IF MCOLLIDE(PLYR,"DOWN",0)  
AND PLYR.DY > 0 THEN  
PLYR.DY = 0  
PLYR.LANDED=TRUE  
  
-- BECAUSE OF VERTICAL SPEED,  
-- THE PLAYER CAN FALL A FEW  
-- PX INTO THE FLOOR. THIS  
-- CALCULATES HOW MANY PX AND  
-- RE-ADJUSTS Y (CREDIT TO  
-- NERDYTEACHERS.COM FOR THIS  
-- FORMULA)  
PLYR.Y-=  
((PLYR.Y+PLYR.H+1)/28)-1  
END -- END IF MCOLLIDE DOWN
```

LINE 19/20

536/8192 E

```
012345 + 012345  
-- PITS AND RESPAWNING  
FUNCTION RESPAWN()  
  -- STOP FALLING WHEN A SOLID  
  -- TILE IS BELOW THE PLAYER  
IF MCOLLIDE(PLYR,"DOWN",0)  
AND PLYR.DY > 0 THEN  
  PLYR.DY = 0  
  PLYR.LANDED=TRUE  
  
  -- BECAUSE OF VERTICAL SPEED,  
  -- THE PLAYER CAN FALL A FEW  
  -- PX INTO THE FLOOR. THIS  
  -- CALCULATES HOW MANY PX AND  
  -- RE-ADJUSTS Y (CREDIT TO  
  -- NERDYTEACHERS.COM FOR THIS  
  -- FORMULA)  
  PLYR.Y-=  
  ((PLYR.Y+PLYR.H+1)%8)-1  
END -- END IF MCOLLIDE down  
LINE 19/20      536/8192 E
```

```
012345 + 012345  
-- PITS AND RESPAWNING  
FUNCTION RESPAWN()  
  -- STOP FALLING WHEN A SOLID  
  -- TILE IS BELOW THE PLAYER  
IF MCOLLIDE(PLYR,"DOWN",0)  
AND PLYR.DY > 0 THEN  
  END -- END IF MCOLLIDE down  
END -- END FUNCTION RESPAWN()
```

I can remove the part of the copied and pasted code that runs if the condition is true, since we're doing something different than stopping the player from falling

It's the *condition* that's similar

```
0 1 2 3 4 5 + 0 1 2 3 4 5 + 0 1 2 3 4 5 + 0 1 2 3 4 5 +  
-- PITS AND RESPAWNING  
FUNCTION RESPAWN()  
  -- STOP FALLING WHEN A SOLID  
  -- TILE IS BELOW THE PLAYER  
  IF MCOLLIDE(PLYR, "DOWN", 0)  
    AND PLYR.DY > 0 THEN  
      END -- END IF MCOLLIDE down  
END -- END FUNCTION RESPAWN()
```

```
0 1 2 3 4 5 + 0 1 2 3 4 5 + 0 1 2 3 4 5 + 0 1 2 3 4 5 +  
-- PITS AND RESPAWNING  
FUNCTION RESPAWN()  
  -- STOP FALLING WHEN A SOLID  
  -- TILE IS BELOW THE PLAYER  
  IF MCOLLIDE(PLYR, "DOWN", 7)  
    AND PLYR.DY > 0 THEN  
      END -- END IF MCOLLIDE down  
END -- END FUNCTION RESPAWN()
```

I'll need to **change the flag number** that we're checking for

We used flag 0 for solid tiles

But we're using flag 7 for pit / death tiles

```
0 1 2 3 4 5 + 0 0 0 0 0 0
-- PITS AND RESPAWNING
FUNCTION RESPAWN()
    -- STOP FALLING WHEN A SOLID
    -- TILE IS BELOW THE PLAYER
    IF MCOLLIDE(PLYR, "down", 1)
    AND PLYR.DY > 0 THEN
        LIVES = LIVES - 1
    END -- END IF MCOLLIDE down
END -- END FUNCTION RESPAWN()
```

If the condition is true (the player is touching a death tile), we should subtract a life

We should also change the player's x,y position and return them to where they began the level

```
0 1 2 3 4 5 + 0 1 2 3 4 5
-- PITS AND RESPAWNING
FUNCTION RESPAWN()
    -- STOP FALLING WHEN A SOLID
    -- TILE IS BELOW THE PLAYER
    IF MCOLLIDE(PLYR,"DOWN",1)
    AND PLYR.DY > 0 THEN
        LIVES = LIVES - 1
        PLYR.X = PLYR_START_X
        PLYR.Y = PLYR_START_Y
    END -- END IF MCOLLIDE DOWN
END -- END FUNCTION RESPAWN()
```

I'll use variables named **plyr\_start\_x** and **plyr\_start\_y** to refer to the starting position

*I just need to make sure I **declare** them first  
(and declare the variable **lives** as well)*

0 1 2 3 4 5 6 + ⌂ ⌃ ⌄ ⌅ ⌆ ⌇

```
FUNCTION _INIT()
    FRIC = 0.85 -- FRICTION
    GRAV = 0.3 -- GRAVITY
```

```
LIVES = 3
```

```
-- PLAYER STARTING COORDINATES
PLYR_START_X = 7*8
PLYR_START_Y = 11*8
```

```
MAKE_PLYR() -- TAB 1
```

```
-- STARTING POSITION FOR CAMERA
CAMX = 0
CAMY = 0
```

In my `_init()` function, I'll declare variables for `lives`, `plyr_start_x`, and `plyr_start_y`

I'll use the same starting coordinates as I defined in my `make_plyr()` function

*(We always need to declare variables before we refer to them elsewhere in the code)*

0 1 2 3 4 5 +

( ) [ ] < >

```
FUNCTION _UPDATE()
MOVE_PLAYER() -- TAB 2
CAM_FOLLOW() -- TAB 4
RESPAWN() -- TAB 5
END -- END FUNCTION _UPDATE()
```

```
FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
MAP() -- DRAW MAP
SPR(PLYR.n, PLYR.x, PLYR.y)
```

```
-- PRINT LIVES
PRINT("LIVES: "..LIVES,CAMX+2,0)
```

```
-- PRINT CAMERA AND PLAYER X
--PRINT("CAMX: "..FLR(CAMX),C
--PRINT("PLYR.X: "..FLR(PLYR.X)
END -- END FUNCTION _DRAW()
```

LINE 1/45                    525/8192 □

I'll also **print** the number of **lives** so we can keep track of that value as it changes and make sure our respawn code is working

The full line of code reads:

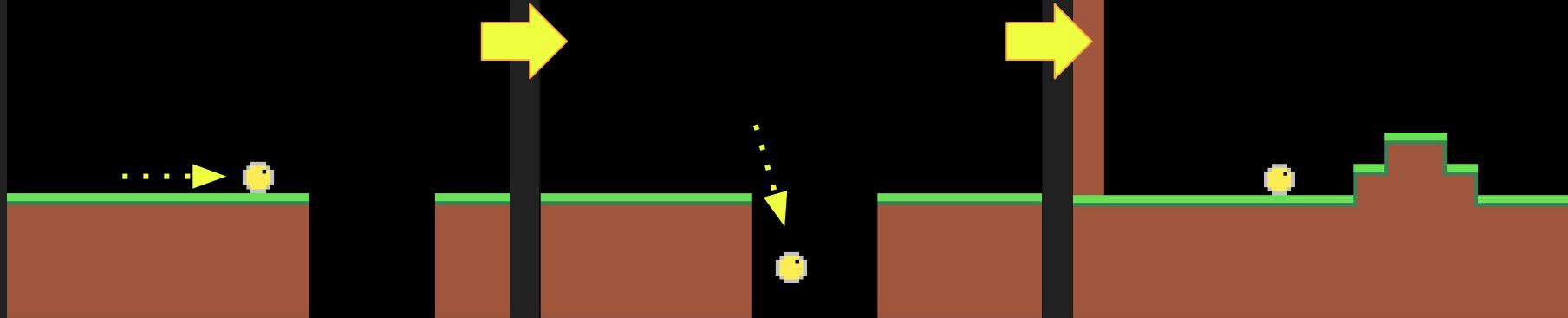
```
print("lives: "..lives,camx+2,camy+2,7)
```

We need to *add camx and camy to the coordinates for the text; otherwise, it would no longer be visible when the camera moves with the player*

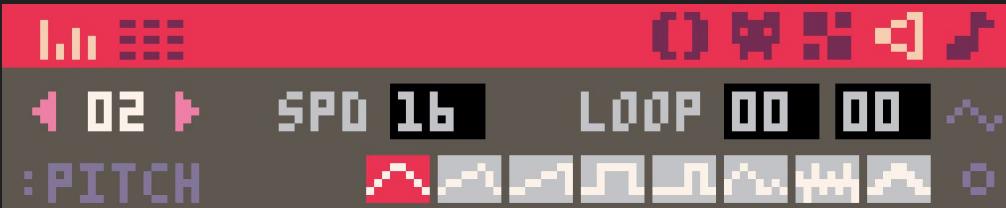
LIVES: 3

LIVES: 3

LIVES: 2



Test your game, and you should now see that the player returns to the starting position after falling down a pit – the number of lives is decreased by 1 as well



One last nice touch would be to add a **sound** when the player “dies”

I've made a failure chime on sound **02**

```
012345+ 09876543210
-- PITS AND RESPAWNING
FUNCTION RESPAWN()

-- WHEN FALLING INTO A PIT,
-- LOSE A LIFE AND RESPAWN THE
-- PLAYER AT THE START
IF MCOLLIDE(PLYR,"DOWN",1)
AND PLYR.DY > 0 THEN
LIVES = LIVES - 1
SFX(2) -- PLAY FAILURE SOUND
PLYR.X = PLYR_START_X
PLYR.Y = PLYR_START_Y
END -- END IF MCOLLIDE DOWN
END -- END FUNCTION RESPAWN()

LINE 10/16      528/8192 E
```

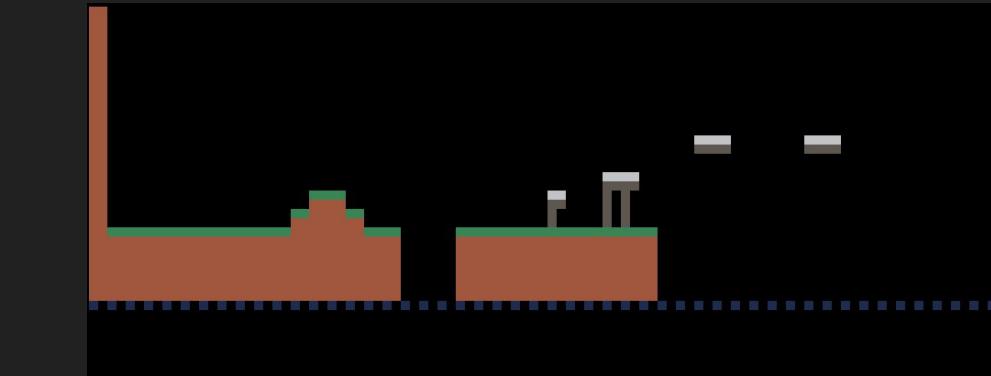
To actually play the sound, use the sfx() function, and include the sound number in the parentheses

Our platformer is coming along nicely . . . another nice touch would be an **animated player sprite**

# Step 13: Animated Sprite

Download Example File: [platformer\\_13\\_animated\\_sprite.p8](#)

Please note that, for the next few examples, I am using a reworked sprite sheet and map, and **code may be organized on different tabs than in the screenshots** (which were from an earlier version of the examples)



Please note that, for the next few examples, I am using a reworked sprite sheet and map, and **code may be organized on different tabs than in the screenshots** (which were from an earlier version of the examples)

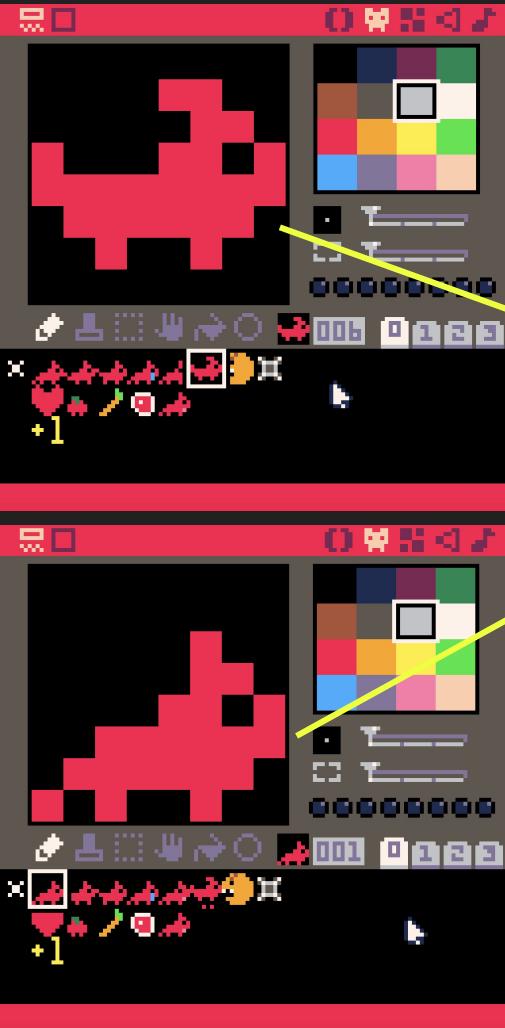
```
0123456 + 0123456  
-- ANIMATE PLAYER SPRITE  
FUNCTION ANIMATE()  
END -- END FUNCTION ANIMATE()
```

```
0 1 2 3 4 5 6 + 0 9 8 7 6 5 4 3 2 1 0  
FUNCTION _UPDATE()  
MOVE_PLYR() -- TAB 2  
CAN_FOLLOW() -- TAB 4  
RESPAWN() -- TAB 5  
ANIMATE() -- TAB 6  
END -- END FUNCTION UPDATE()  
  
FUNCTION _DRAW()  
CLS() -- CLEAR SCREEN  
MAP() -- DRAW MAP  
SPR(PLYR.N, PLYR.X, PLYR.Y)  
  
-- PRINT LIVES  
PRINT("LIVES: "..LIVES,CANX+2,CANY+1)
```

I'll begin by creating a new function called `animate()`

*(no, there's no pre-existing function for this like there was for camera)*

-And I'll *call* my function in `_update()`

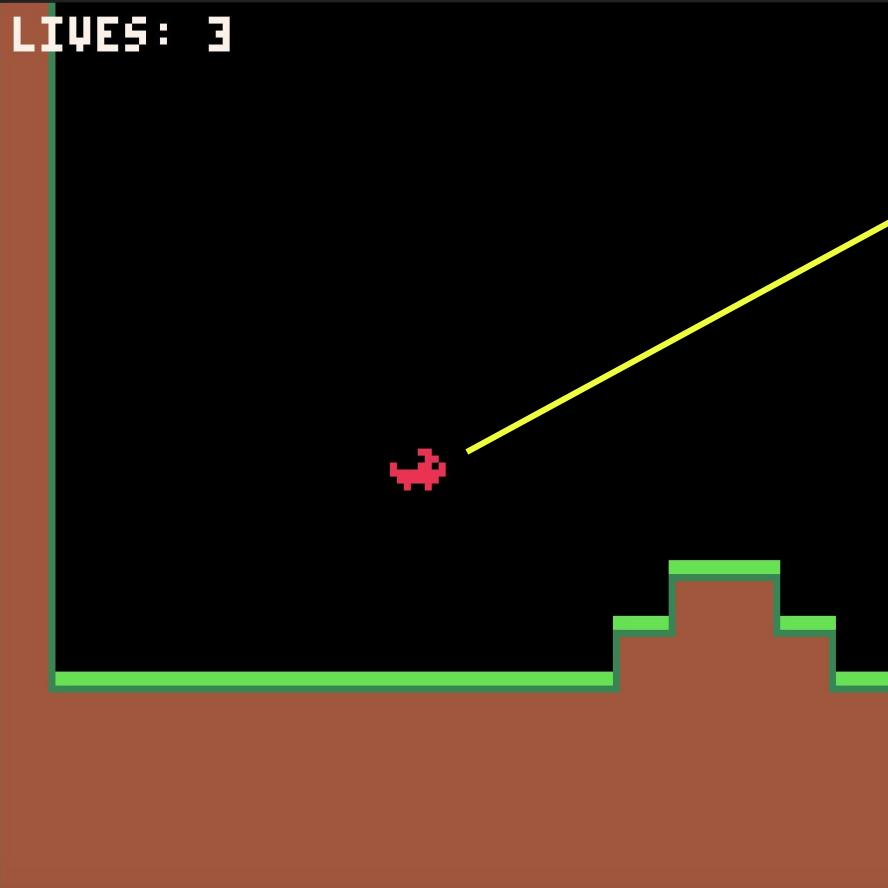


```
0123456 + ()<>[]  
-- ANIMATE PLAYER SPRITE  
FUNCTION ANIMATE()  
  
    -- CHANGE SPRITE FOR JUMPING  
    IF PLYR.LANDED == FALSE THEN  
        PLYR.n = 6 -- JUMPING SPRITE  
    ELSE  
        PLYR.n = 1 -- DEFAULT SPRITE  
    END -- END IF PLYR.LANDED FALSE  
  
END -- END FUNCTION ANIMATE()
```

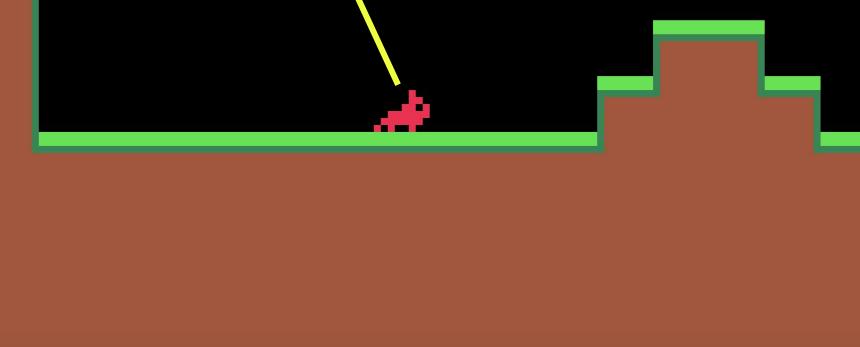
- One thing this function should do is display a different sprite when the player is jumping
- We know they are jumping when `plyr.landed` is not true
- So in that case, we can **change the value for the player's sprite number**
- And use **else** to change the sprite number back to the default in all other cases

Different sprites when the character is jumping (left) vs. landed (right)

LIVES: 3



```
0 1 2 3 4 5 6 + 0 4 5 < > 
-- ANIMATE PLAYER SPRITE
FUNCTION ANIMATE()
L: -- CHANGE SPRITE FOR JUMPING
IF PLYR.LANDED == FALSE THEN
    PLYR.n = 6 -- JUMPING SPRITE
ELSE
    PLYR.n = 1 -- DEFAULT SPRITE
END -- END IF PLYR.LANDED FALSE
END -- END FUNCTION ANIMATE()
```



Another thing we should do is flip the sprite when facing left

(we could have done this earlier, but I'll do it now while we're focusing on controlling how the sprite displays)

0 1 2 3 4 5 6 + ⚡

```
-- WIDTH AND HEIGHT IN PIXELS
-- NEEDED FOR MAP COLLISION
PLYR.W=8
PLYR.H=8

PLYR.XSPD=0.5 -- X SPEED
PLYR.YSPD=4 -- Y SPEED

PLYR.DX=0 -- CHANGE IN X
PLYR.DY=0 -- CHANGE IN Y

-- PLAYER STATE
PLYR.LANDED=False

-- FLIP SPRITE?
PLYR.FLIP=False

END -- END FUNCTION MAKE_PLAYER()
```

LINE 26/27 599/8192 ⚡

Step 1: Add a boolean (true or false) variable for whether to flip the player sprite (in the make\_player function)

0 1 2 3 4 5 6 + ⚡

```
PLYR.DY += GRAV

-- MOVE LEFT
IF BTn(0) THEN
    PLYR.DX -= PLYR.XSPD
    PLYR.FLIP = TRUE
END -- END IF BTn(0)

-- MOVE RIGHT
IF BTn(1) THEN
    PLYR.DX += PLYR.XSPD
    PLYR.FLIP = FALSE
END -- END IF BTn(1)

-- JUMP
IF BTnP(0) OR BTnP(1) THEN
    IF PLYR.LANDED THEN
        PLYR.DY -= PLYR.YSPD
    PLYR.LANDED=False
END -- END IF BTnP(0)
```

LINE 21/93 599/8192 ⚡

Step 2: Set flip to true when moving left, and false when moving right (in the move\_player function)

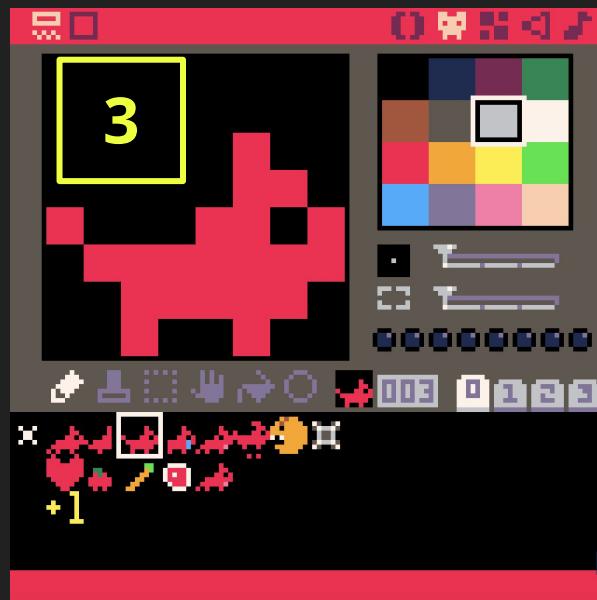
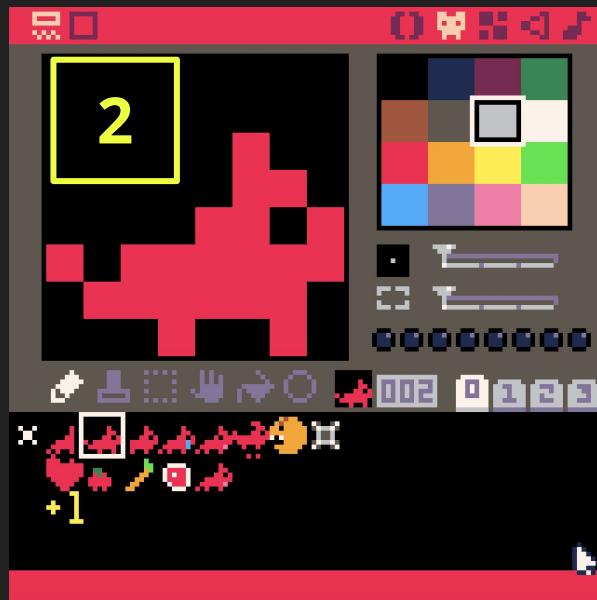
LIVES: 3



The player sprite now faces left when moving left (and right when facing right)

```
0 1 2 3 4 5 6 + ()  
PLYR.DY += GRAV  
  
-- MOVE LEFT  
IF BTn(0) THEN  
    PLYR.DX -= PLYR.XSPD  
    PLYR.FLIP = TRUE  
END -- END IF BTn(0)  
  
-- MOVE RIGHT  
IF BTn(1) THEN  
    PLYR.DX += PLYR.XSPD  
    PLYR.FLIP = FALSE  
END -- END IF BTn(1)  
  
-- JUMP  
IF BTnP(0) OR BTnP(1) THEN  
    IF PLYR.LANDED THEN  
        PLYR.DY -= PLYR.YSPD  
        PLYR.LANDED=FALSE  
    LINE 21/93  
    599/8192 E
```

Set flip to true when moving left, and false when moving right (*in the move player function*)



## Next, we'll make a run cycle for our player

It's a good idea to place the frames for an animation cycle consecutively

So you can just add 1 to the sprite number variable in your code  
( `plyr.n = plyr.n + 1` )

```
0 1 2 3 4 5 6 + () [] < > ⏪  
-- ANIMATE PLAYER SPRITE  
FUNCTION ANIMATE()  
  
-- CHANGE SPRITE FOR JUMPING  
IF PLYR.LANDED == FALSE THEN  
    PLYR.n = 6 -- JUMPING SPRITE  
ELSE  
    PLYR.n = 1 -- DEFAULT SPRITE  
END -- END IF PLYR.LANDED FALSE  
  
-- RUN CYCLE WHEN MOVING L/R  
IF BTn(0) OR BTn(1) THEN  
      
END -- END IF BTn(0/1)  
  
END -- END FUNCTION ANIMATE()
```

We know we only want to play the animation while the player is moving horizontally

So the condition in our **if** statement can be that ***either the left or right arrow key must be pressed*** in order to play the animation

```
0 1 2 3 4 5 6 + 0 0 0 0 0 0
```

```
-- ANIMATE PLAYER SPRITE  
FUNCTION ANIMATE()
```

```
-- CHANGE SPRITE FOR JUMPING  
IF PLYR.LANDED == FALSE THEN  
    PLYR.n = 6 -- JUMPING SPRITE  
ELSE  
    PLYR.n = 1 -- DEFAULT SPRITE  
END -- END IF PLYR.LANDED FALSE
```

```
-- RUN CYCLE WHEN MOVING LSR  
IF BTN(0) OR BTN(1) THEN
```

```
    PLYR.n = PLYR.n + 1
```

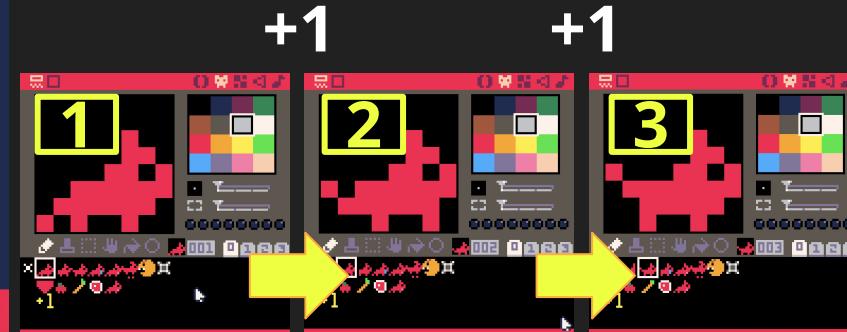
```
END -- END IF PLYR.DX
```

```
END -- END FUNCTION ANIMATE()
```

LINE 14/18

583/8192

Inside the if statement for the key press, we can add 1 to our player sprite variable



```
0 1 2 3 4 5 6 + () [] < >
PLYR.N = 6 -- JUMPING SPRITE
ELSE
  PLYR.N = 1 -- DEFAULT SPRITE
END -- END IF PLYR.LANDED FALSE
```

```
-- RUN CYCLE WHEN MOVING LSR
IF BTn(0) OR BTn(1) THEN
```

```
-- GO TO NEXT SPRITE
PLYR.N = PLYR.N + 1
```

```
-- LOOP BACK TO FIRST FRAME
```

```
IF PLYR.N > 3 THEN
```

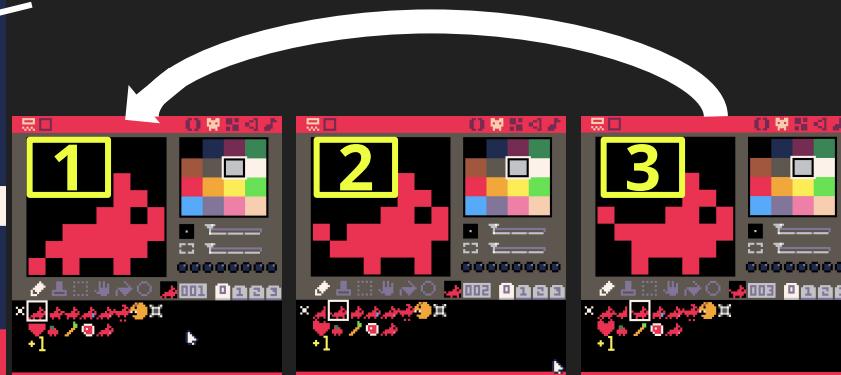
```
  PLYR.N = 1
```

```
END -- END IF PLYR.N > 3
```

```
END -- END IF PLYR.DX
```

```
END -- END FUNCTION ANIMATE()
LINE 20/24
593/8192 E
```

Then, when we get past the last frame in the animation (sprite 3), we change the value for the sprite number back to 1 (the first frame)



We want the sprite to change with a bit of **time between each frame**

So we need to add a **timer**

I'll add a new variable for the timer, called ***animtime***, in my **\_init()** function

Start the timer at **0**



```
0 1 2 3 4 5 6 + 0 999<>>
MOVE_PLAYER() -- TAB 1
-- STARTING POSITION FOR CAMERA
CAMX = 0
CAMY = 0
-- ANIMATION TIMER
ANIMTIME = 0
END -- END FUNCTION _INIT()

FUNCTION _UPDATE()
MOVE_PLAYER() -- TAB 2
CAM_FOLLOW() -- TAB 4
RESPAWN() -- TAB 5
ANIMATE() -- TAB 6
END -- END FUNCTION _UPDATE()

FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
LINE 1/51          605/8192 E
```

```

0 1 2 3 4 5 6 + 0 0 0 < >
-- RUN CYCLE WHEN MOVING L/R
IF BTn(0) OR BTn(1) THEN
    -- CHANGE SPRITE EVERY
    -- 0.15 SECONDS
    IF TIME() - ANIMTIME > 0.15 THEN
        ANIMTIME = TIME() -- RESET TIME
    -- GO TO NEXT SPRITE
    PLYR.N = PLYR.N + 1
    -- LOOP BACK TO FIRST FRAME
    IF PLYR.N > 3 THEN
        PLYR.N = 1
    END -- END IF PLYR.N
END -- END IF TIME
END -- END IF BTn(0) OR BTn(1)
LINE 29/31 605/8192 E

```

Then, back in our animation function, we can add a condition (if statement), *nested inside the condition for the button press*, that we only change the sprite if 0.15 seconds have elapsed

The function [time\(\)](#) tracks how much time has elapsed since the game has started running

If animtime starts at 0, then once 0.15 seconds have elapsed, time() will be equal to 0.15

$$\begin{aligned}
 \text{time()} - \text{animtime} \\
 = 0.15 - 0 \\
 = 0.15
 \end{aligned}$$

```

0 1 2 3 4 5 6 + 0 0 0 < >
-- RUN CYCLE WHEN MOVING L/R
IF BTn(0) OR BTn(1) THEN
    -- CHANGE SPRITE EVERY
    -- 0.15 SECONDS
    IF TIME() - ANIMTIME > 0.15 THEN
        ANIMTIME = TIME() -- RESET TIME
    -- GO TO NEXT SPRITE
    PLYR.N = PLYR.N + 1
    -- LOOP BACK TO FIRST FRAME
    IF PLYR.N > 3 THEN
        PLYR.N = 1
    END -- END IF PLYR.N
END -- END IF TIME
END -- END IF BTn(0) OR BTn(1)
LINE 29/31      605/8192 E

```

If animtime starts at 0, then once 0.15 seconds have elapsed, time() will be equal to 0.15

$$\begin{aligned}
 \text{time}() - \text{animtime} \\
 = 0.15 - 0 \\
 = 0.15
 \end{aligned}$$

This means the condition in the if statement will be met, and the code inside the if statement will run

```
0 1 2 3 4 5 6 + 0 0 0 < >
```

```
-- RUN CYCLE WHEN MOVING L/R  
IF BTn(0) OR BTn(1) THEN
```

```
-- CHANGE SPRITE EVERY  
-- 0.15 SECONDS  
IF TIME() - ANIMTIME > 0.15 THEN  
ANIMTIME = TIME() -- RESET TIME  
-- GO TO NEXT SPRITE  
PLYR.N = PLYR.N + 1  
-- LOOP BACK TO FIRST FRAME  
IF PLYR.N > 3 THEN  
PLYR.N = 1  
END -- END IF PLYR.N  
END -- END IF TIME
```

```
END -- END IF BTn(0) OR BTn(1)  
LINE 29/31 605/8192 E
```

If animtime starts at 0, then once 0.15 seconds have elapsed, time() will be equal to 0.15

time() - animtime

$$= 0.15 - 0$$

$$= 0.15$$

We then will reset animtime to equal the current value of time()

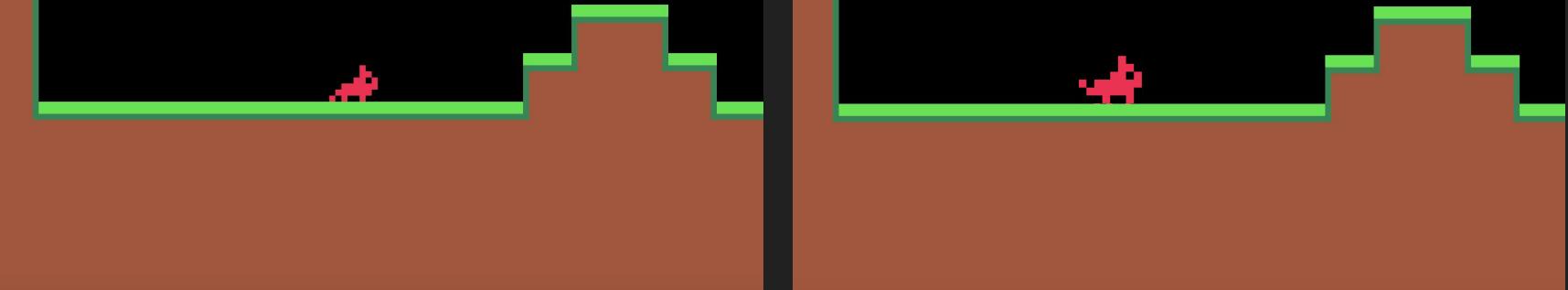
So when time() reaches 0.15, animtime will be set to 0.15

When time() reaches 0.30, the loop will run again

time()	animtime	> 0.15?
0.0-0.1499	0	false
0.15	0	true
0.15001-0.2999	0.15	false
0.30	0.15	true
0.31001-0.4499	0.30	false
0.45	0.30	true

LIVES: 3  
TIME() 1.8667  
ANIMTIME 0

LIVES: 3  
TIME() 5.4  
ANIMTIME 5.2667



Printing [time\(\)](#) and animtime helps illustrate how the timer works to implement the animation

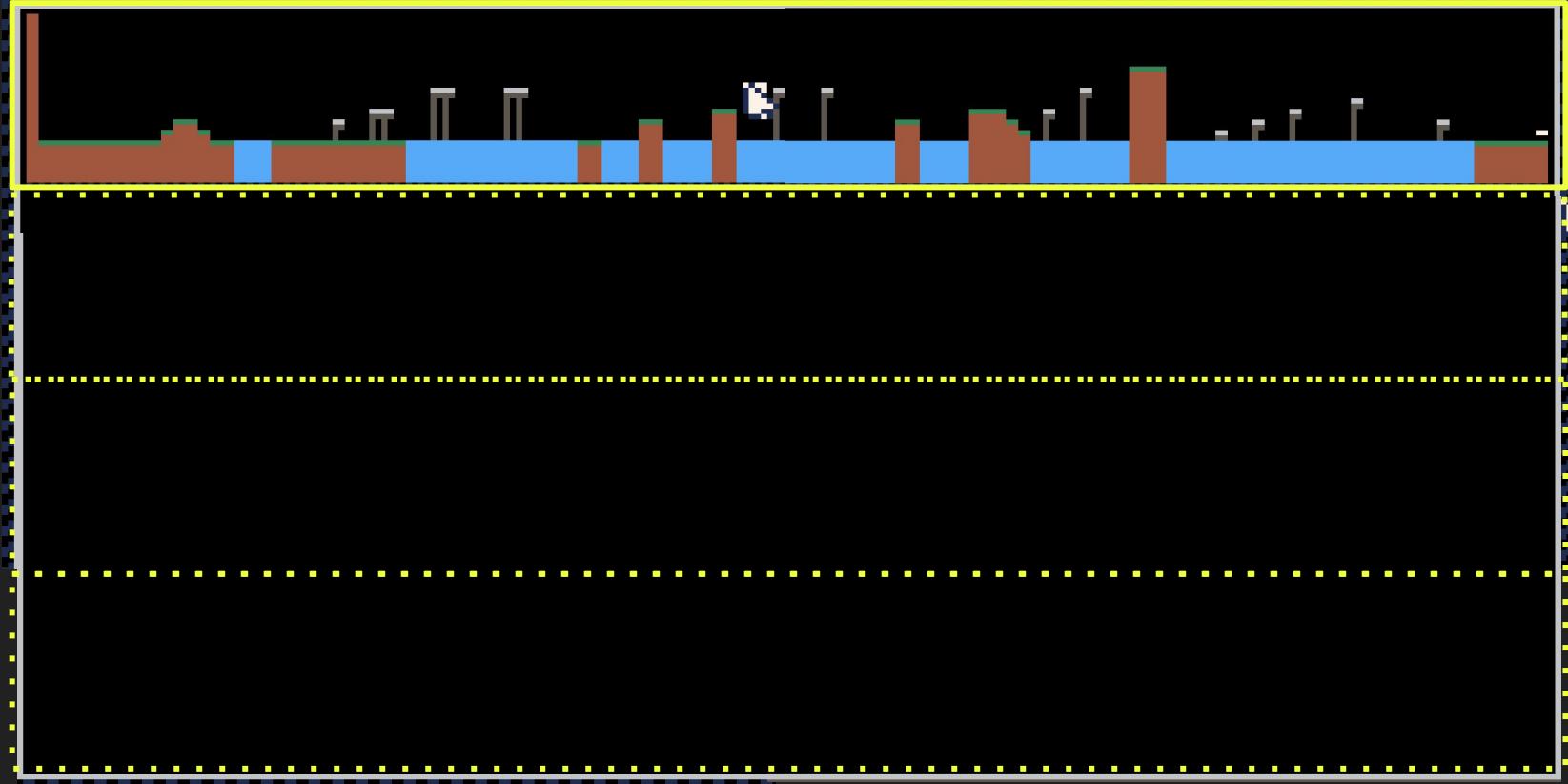
# Bonus: triggering an animation by keypress

Download Example File: [platformer\\_13a\\_toggle\\_animation.p8](#)

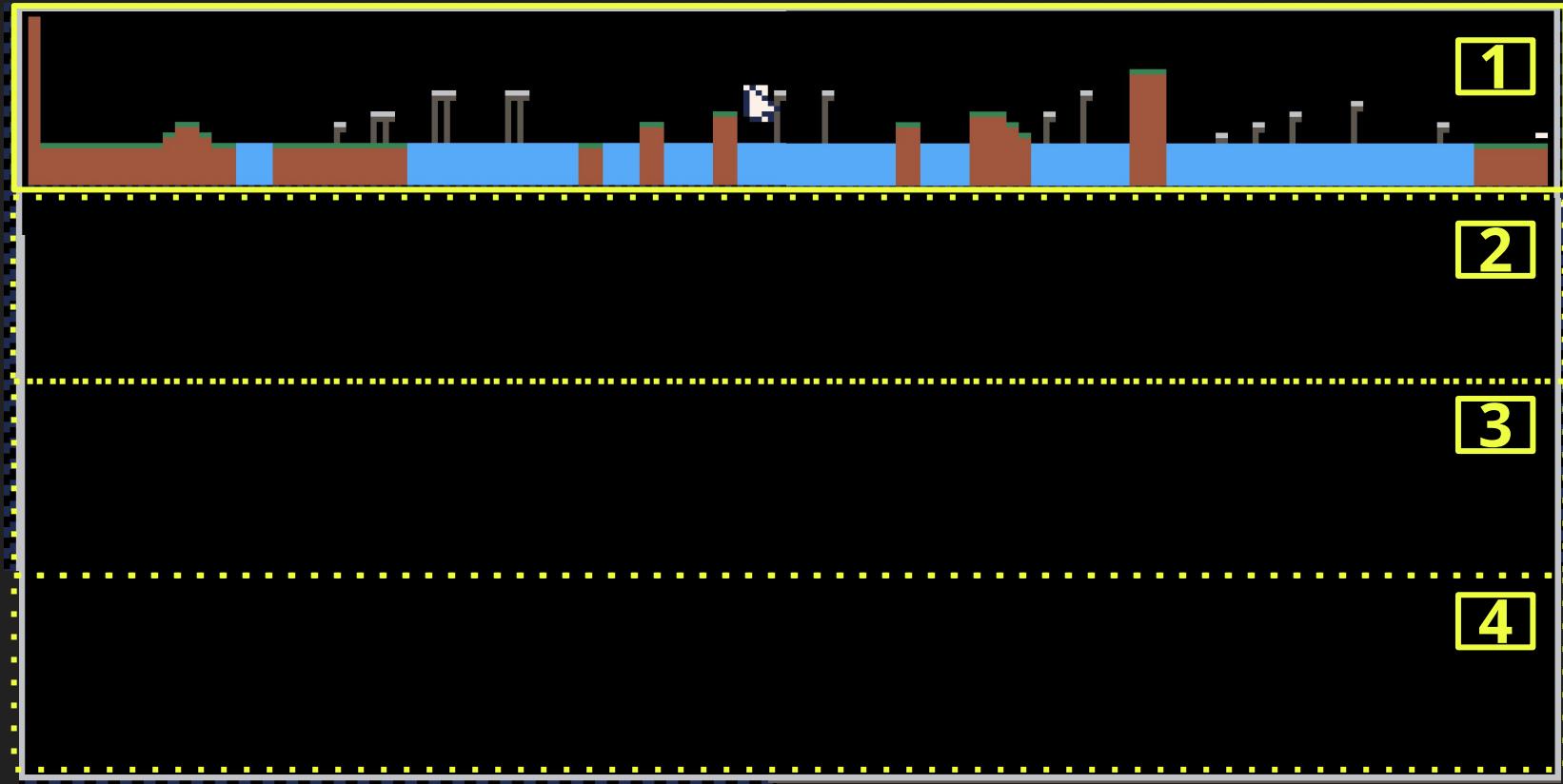
Next, we'll add additional levels

# Step 14: Multiple Levels

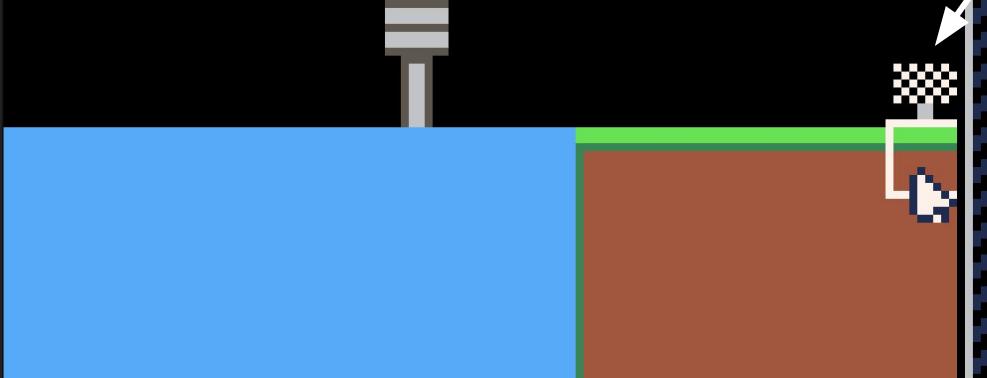
Download Example File: [platformer\\_14\\_multiple\\_levels.p8](#)



I've built my first "level" along the top row of the map  
*(this image is several screenshots spliced together; unfortunately, you can't view the entire map at once in the editor)*

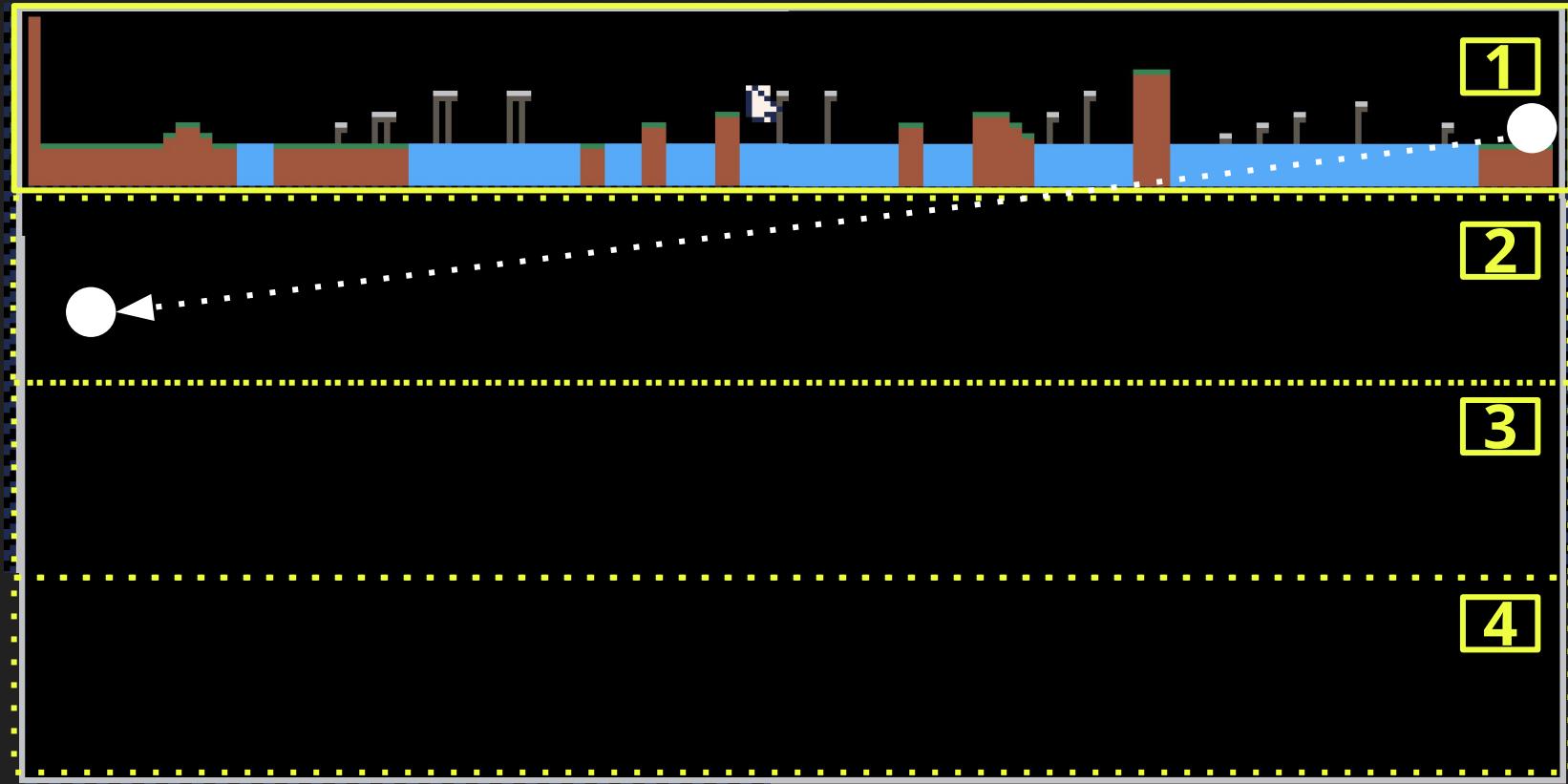


Notice how there is room for a total of **four screens** vertically  
Each of these screens could contain a separate level  
*(you could also design your game like Celeste and have vertical progression)*



At the far right end of my first level (the right edge of the map), I've placed a tile indicating the end of the level

I can draw my next level one “screen” below this one on the map, and set the player’s x,y coordinates to that location once they reach this point



I can draw my next level one “screen” below the first, and set the player’s x,y coordinates to that location once they reach the end of the first level



I started drawing my second level one screen below the first

If the **lowest tile on screen one is 15** ( $16 - 1$  because we start at 0), the **lowest tile on screen two will be 31** ( $16 * 2 - 1 = 32 - 1$ )

So I can start drawing there

And I can use these coordinates as the location to send my player once they've reached the end of the first level

```
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7  
-- STARTING POSITION FOR CAMERA  
CAMX = 0  
CAMY = 0  
  
-- ANIMATION TIMER  
ANIMTIME = 0  
END -- END FUNCTION _INIT()  
  
FUNCTION _UPDATE()  
MOVE_PLAYER() -- TAB 2  
CAM_FOLLOW() -- TAB 4  
RESPAWN() -- TAB 5  
ANIMATE() -- TAB 6  
WARP() -- TAB 7  
END -- END FUNCTION _UPDATE()  
  
FUNCTION _DRAW()  
CLS() -- CLEAR SCREEN  
LINE 38/49 645/8192 =
```

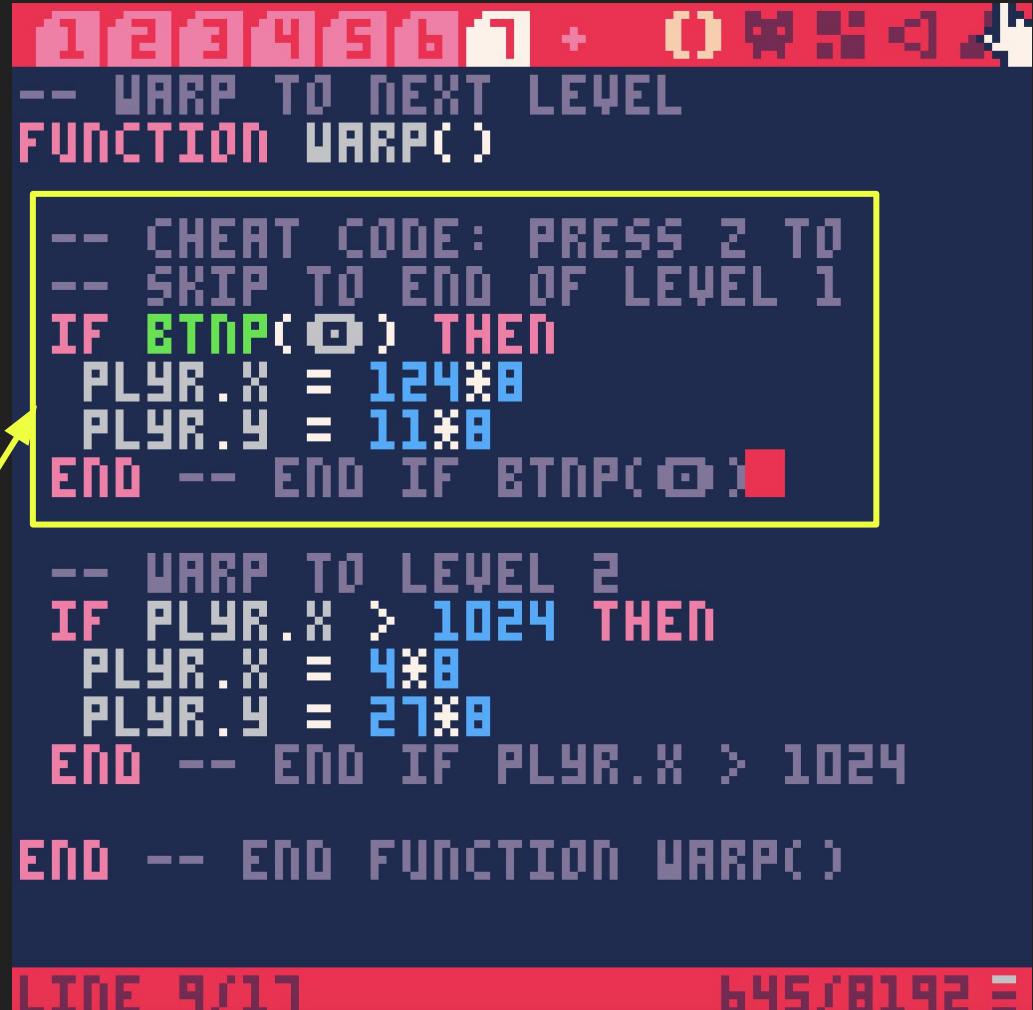
```
1234567 + 0WHLR  
-- WARP TO NEXT LEVEL  
FUNCTION WARP()  
  
-- WARP TO LEVEL 2  
IF PLR.X > 1024 THEN  
    PLR.X = 488  
    PLR.Y = 2788  
END -- END IF PLR.X > 1024  
  
END -- END FUNCTION WARP()
```

I've written a function called **warp()** that checks whether the player has reached the end of the map (*which is 1024px long*) and changes their x,y coordinates to where the second level begins

-I've *called* my warp() function in \_update()

While you should definitely playtest your game to make sure it's possible to make it through the first level, once you've done so, it would be convenient if you didn't have to play through the entire level just to test whether your warp function works

So I made a "cheat code" – the player can press Z to skip to the end of the first level – when the button is pressed, the player's x,y coordinates are changed



```
1 2 3 4 5 6 7 + () [] < > E
-- WARP TO NEXT LEVEL
FUNCTION WARP( )
-- CHEAT CODE: PRESS Z TO
-- SKIP TO END OF LEVEL 1
IF BTnP(0) THEN
  PLYR.X = 124*8
  PLYR.Y = 11*8
END -- END IF BTnP(0)

-- WARP TO LEVEL 2
IF PLYR.X > 1024 THEN
  PLYR.X = 4*8
  PLYR.Y = 27*8
END -- END IF PLYR.X > 1024

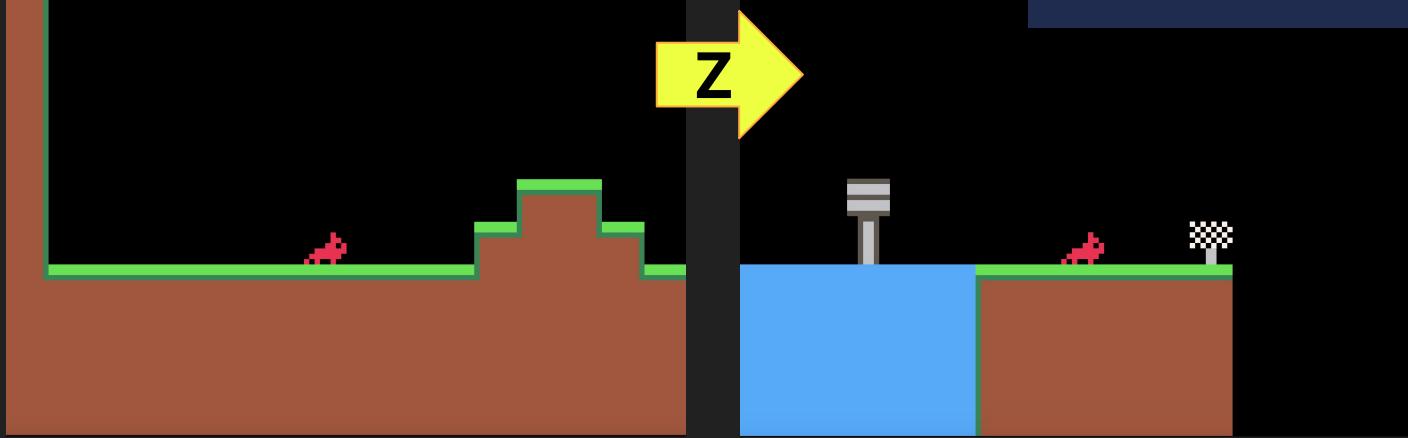
END -- END FUNCTION WARP()
```

LINE 9/17      645/8192 E

LIVES: 3

LIVES: 3

```
-- CHEAT CODE: PRESS Z TO  
-- SKIP TO END OF LEVEL 1  
IF BTNP(□) THEN  
    PLYR.X = 1248  
    PLYR.Y = 118  
END -- END IF BTNP(□)
```



it would be convenient if you didn't have to play through the entire level just to test whether your warp function works

So I made a “cheat code” – the player can press Z to skip to the end of the first level – when the button is pressed, the player's x,y coordinates are changed

LIVES: 3

```
-- WARP TO LEVEL 2
IF PLYR.X > 1024 THEN
    PLYR.X = 488
    PLYR.Y = 2788
END -- END IF PLYR.X > 1024
```

I might want to **tweak my condition** to trigger when the player reaches an x coordinate slightly less than 1024

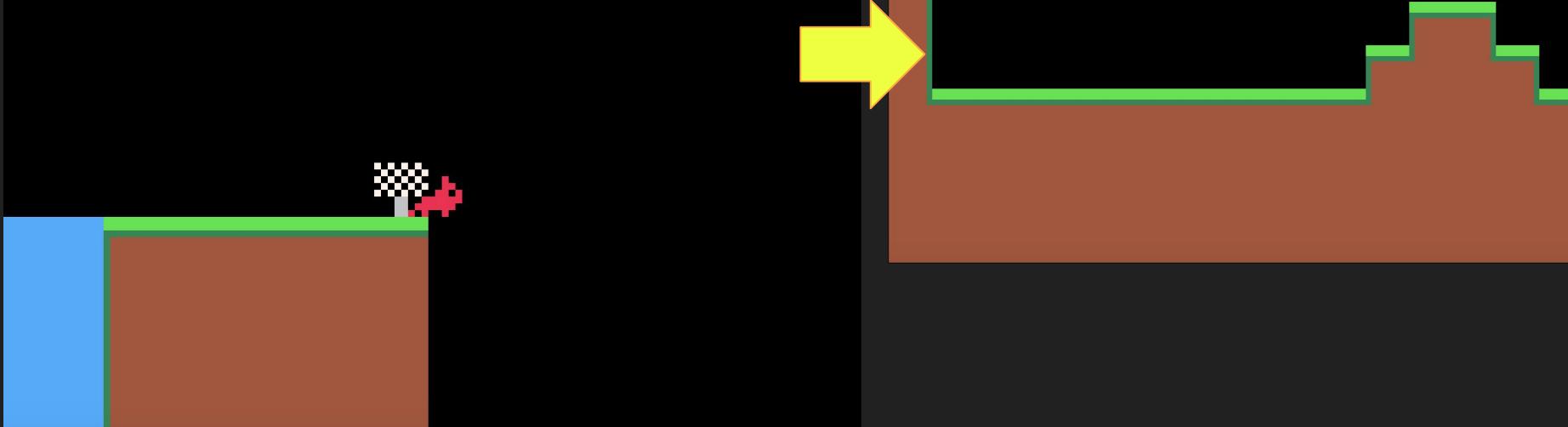


*Notice how they can walk off the edge of the map slightly (and also see out of bounds; we'll want to pin the camera on this end of the map)*

LIVES: 3

Something definitely happens when the player reaches an x coordinate of 1024

*However, it's not quite the result I wanted*



LIVES: 3

Why do I see the beginning of level 1?

Where did my player go?

LIVES: 3

I see the left side of the first level, because we've moved the player to the left side of the map

But I don't see my player, because the player's y position was moved lower on the map, but the *camera's y position was not updated* to follow it

So we'll need to make those changes in our warp function



```
1 2 3 4 5 6 7 + 0 ▶◀◀◀◀
```

-- SKIP TO END OF LEVEL 1  
**IF** BTNP(0) **THEN**  
 PLYR.X = 124\*8  
 PLYR.Y = 11\*8  
**END** -- END IF BTNP(0)

-- WARP TO LEVEL 2  
**IF** PLYR.X >= 1024-PLYR.W/2 **THEN**

-- MOVE THE PLAYER TO THE  
-- START OF LEVEL 2  
 PLYR.X = 4\*8  
 PLYR.Y = 27\*8

-- RESET THE CAMERA  
 CAMY = 16\*8

**END** -- END IF PLYR.X > 1024

**END** -- END FUNCTION WARP()

LINE 20/23 716/8192 ⏴

In our warp() function, we'll reset the camera position

*We can leave the camera's x the same, since it always follows the player's x*

But we must set the **camera's y** to 1 screen down (16 tiles down from the top of the map, multiplied by 8 pixels per map tile)

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7  
-- CAMERA  
FUNCTION CAM_FOLLOW()  
CAMX = PLYR.X - 64 + PLYR.W/2  
--CAMY = 0  
-- PIN CAMERA TO LEFT EDGE  
IF CAMX < 0 THEN  
    CAMX = 0  
END  
  
CAMERA(CAMX,CAMY)  
END -- END FUNCTION CAM_FOLLOW()
```

In order for our new camera y coordinate to take effect, we have to remove this line of code that was setting the camera's y to 0 (the default value) in our camera function

0 1 2 3 4 5 6 7 8 9 ⏪ ⏴ ⏵ ⏹ ⏺

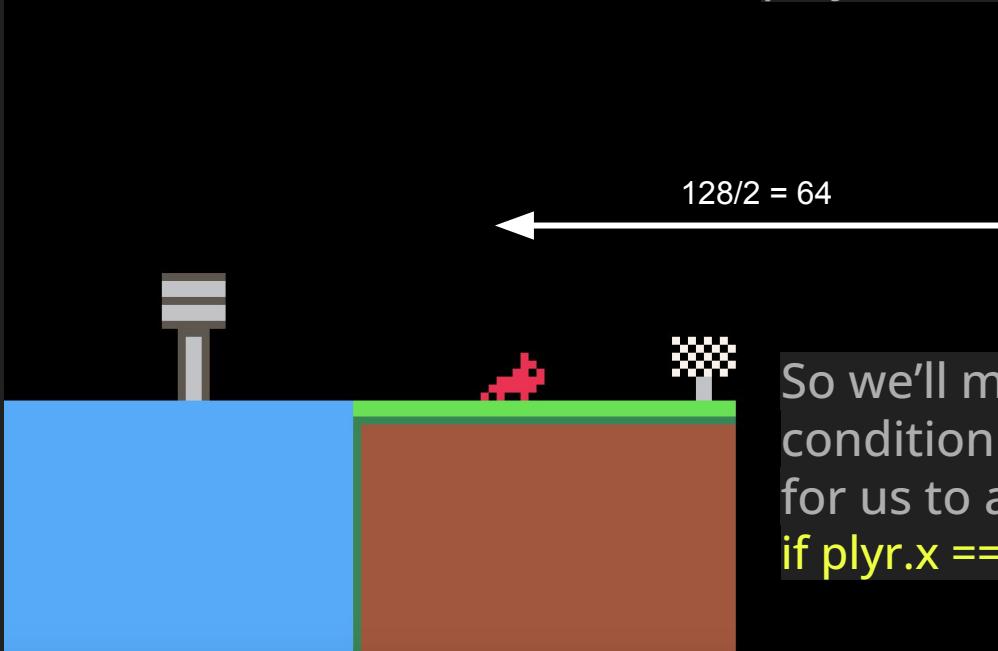
```
-- CAMERA  
FUNCTION CAM_FOLLOW()  
CAMX = PLYR.X - 64 + PLYR.W/2  
--CAMY = 0  
  
-- PIN CAMERA TO LEFT EDGE  
IF CAMX < 0 THEN  
    CAMX = 0  
END  
  
CAMERA(CAMX, CAMY)  
END -- END FUNCTION CAM_FOLLOW()
```

In order for our new camera y coordinate to take effect, we have to remove this line of code that was setting the camera's y to 0 (the default value) in our camera function

While we're in our camera function, we can also pin the camera to the right edge of the map when the player approaches, so that we don't see out of bounds



S: 3  
x: 992, 88  
cam: 932, 0



We know the out-of-bounds area will show when the player is within less than half a screen (64 pixels) of the right edge, minus half the player's width

So we'll make this the condition that must be met for us to adjust the camera:  
if `plyr.x == 1024 - 64 - plyr.w/2`

0 1 2 3 4 5 6 7 8 9 < >

```
-- CAMERA
FUNCTION CAM_FOLLOW()
CAMX = PLYR.X - 64 + PLYR.W/2
--CAMY = 0

-- PIN CAMERA TO LEFT EDGE
IF CAMX < 0 THEN
    CAMX = 0
END

-- PIN CAMERA TO RIGHT EDGE
IF PLYR.X >= 1024-64-PLYR.W/2
THEN
    CAMX = 1024-64-PLYR.W/2
END

CAMERA(CAMX, CAMY)
END -- END FUNCTION CAM_FOLLOW()
```

LINE 14/18

707/8192 E

We know the out-of-bounds area will show when the player is within less than half a screen (64 pixels) of the right edge, minus half the player's width

So we'll make this the condition that must be met for us to adjust the camera:  
if  $\text{plyr.x} == 1024 - 64 - \text{plyr.w}/2$



And we know we want the right edge of the map to align with the right edge of the screen

So if the camera is always in the top-left of the screen, we know we need the camera's x coordinate to be one screen to the left of the map edge (**128 pixels less than it**)

**So we can express the adjusted camera x as  $1024-128$**

0 1 2 3 4 5 6 7 ◻ ◻ ◻ ◻ ◻ ◻ ◻

-- CAMERA

**FUNCTION** CAM\_FOLLOW()  
CAMX = PLYR.X - 64 + PLYR.W/2  
--CAMY = 0

-- PIN CAMERA TO LEFT EDGE  
**IF** CAMX < 0 **THEN**  
CAMX = 0  
**END**

-- PIN CAMERA TO RIGHT EDGE  
**IF** PLYR.X >= 1024-64-PLYR.W/2  
**THEN**  
CAMX = 1024-128 ←  
**END**

CAMERA(CAMX,CAMY)  
**END** -- END FUNCTION CAM\_FOLLOW()

LINE 14/18

712/8192 E

We know we want the right edge of the map to align with the right edge of the screen

So if the camera is always in the top-left of the screen, we know we need the camera's x coordinate to be one screen to the left of the map edge (128 pixels less than it)

So we can express the adjusted camera x as 1024-128

0 1 2 3 4 5 6 7 ◎ □ ▢ ▣ ▤

LIVES: 3

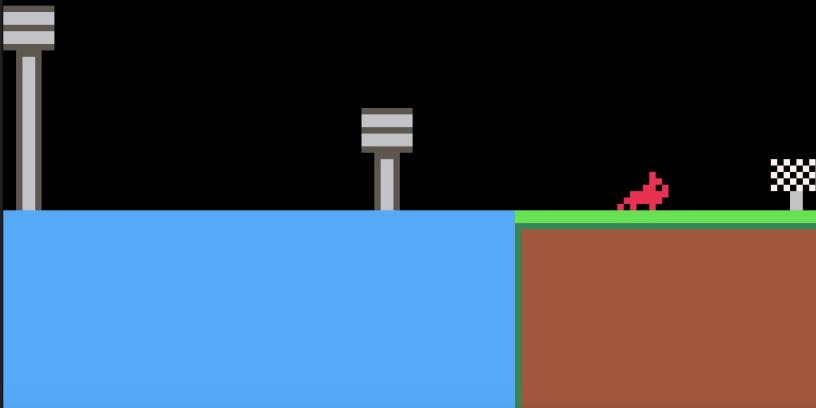
PLYR: 992.00

CAM: 896.0

-- CAMERA  
FUNCTION CAM\_FOLLOW()  
CAMX = PLYR.X - 64 + PLYR.W/2  
--CAMY = 0  
  
-- PIN CAMERA TO LEFT EDGE  
IF CAMX < 0 THEN  
CAMX = 0  
END  
  
-- PIN CAMERA TO RIGHT EDGE  
IF PLYR.X >= 1024-64-PLYR.W/2  
THEN  
CAMX = 1024-128  
END  
  
CAMERA(CAMX,CAMY)  
END -- END FUNCTION CAM\_FOLLOW()

LINE 14/18

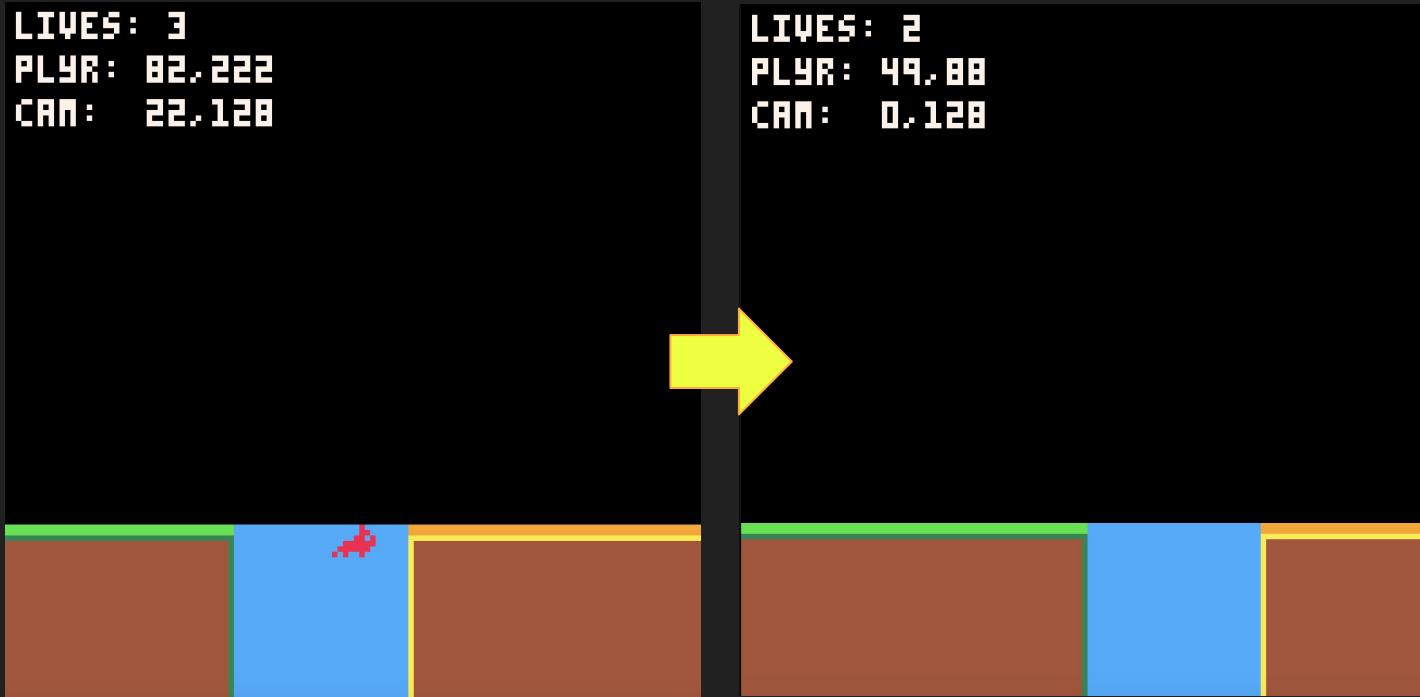
712/8192 E



This adjustment pins the camera so it never displays the out-of-bounds area

LIVES: 3  
PLYR: 82,222  
CRR: 22,128

LIVES: 2  
PLYR: 49,88  
CRR: 0,128



There's still some fine-tuning that needs to be done overall:  
*When we fall in a pit after starting level 2, the player sprite disappears*  
This is because we are still respawning the player at coordinates corresponding to the level 1 starting position – **these coordinates must be updated after advancing to the next level**

```
1 2 3 4 5 6 7 + () * % < > [ ]  
-- SKIP TO END OF LEVEL 1  
IF BTNP(0) THEN  
    PLYR.X = 124*8  
    PLYR.Y = 11*8  
END -- END IF BTNP(0)  
  
-- WARP TO LEVEL 2  
IF PLYR.X >= 1024-PLYR.W/2 THEN  
  
    -- MOVE THE PLAYER TO THE  
    -- START OF LEVEL 2  
    PLYR.X = 4*8  
    PLYR.Y = 27*8  
  
    -- RESET THE CAMERA  
    CAMY = 16*8  
END -- END IF PLYR.X > 1024  
  
END -- END FUNCTION WARP()  
LINE 20/23 716/8192 E
```

## Updated warp() function

```
1 2 3 4 5 6 7 + 0 * < > 
PLYR.Y = 11*8
END -- END IF BTNP(0)

-- WARP TO LEVEL 2
IF PLYR.X >= 1024-PLYR.W/2 THEN

    -- MOVE THE PLAYER TO THE
    -- START OF LEVEL 2
    PLYR.X = 4*8
    PLYR.Y = 27*8

    -- UPDATE THE RESPAWN COORDS
    PLYR_START_X = PLYR.X
    PLYR_START_Y = PLYR.Y

    -- RESET THE CAMERA
    CAMY = 16*8
END -- END IF PLYR.X > 1024
```

**There's still some fine-tuning that needs to be done overall:**

*When we fall in a pit after starting level 2, the player sprite disappears*

This is because we are still respawning the player at coordinates

corresponding to the level 1 starting position – ***these coordinates must be updated after advancing to the next level***

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

```
-- TAB 6: ANIMATE PLAYER SPRITE
```

```
FUNCTION _INIT()
```

```
    FRIIC = 0.85 -- FRICTION  
    GRAVY = 0.3 -- GRAVITY
```

```
    LIVES = 3
```

```
    LEVEL = 1
```



```
-- PLAYER STARTING COORDINATES
```

```
    PLYR_START_X = 7*7
```

```
    PLYR_START_Y = 11*8
```

```
MAKE_PLYR() -- TAB 1
```

```
-- STARTING POSITION FOR CAMERA
```

```
    CAMX = 0
```

```
    CAMY = 0
```

```
LINE 18/53
```

```
726/8192 E
```

I'll also add a variable called **level** in my `_init()` function

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

```
-- WARP TO NEXT LEVEL
```

```
FUNCTION WARP()
```

```
-- CHEAT CODE: PRESS Z TO
```

```
-- SKIP TO END OF LEVEL 1
```

```
IF BTNP(0) THEN
```

```
    PLYR.X = 124*8
```

```
    PLYR.Y = 11*8
```

```
END -- END IF BTNP(0)
```

```
-- WARP TO LEVEL 2
```

```
IF PLYR.X >= 1024-PLYR.W/2 THEN
```

```
    LEVEL = LEVEL + 1
```



```
-- MOVE THE PLAYER TO THE
```

```
-- START OF LEVEL 2
```

```
    PLYR.X = 4*8
```

```
    PLYR.Y = 27*8
```

```
LINE 14/29
```

```
740/8192 E
```

And I'll **add 1** to the value of **level** when we warp to it

1 2 3 4 5 6 7 + ⌂ ⌃ ⌄ ⌅ ⌆

```
-- WARP TO LEVEL 2
IF PLYR.X >= 1024-PLYR.W/2 THEN
    LEVEL = LEVEL + 1
    -- MOVE THE PLAYER TO THE
    -- START OF LEVEL 2
    PLYR.X = 488
    PLYR.Y = 2788
-- UPDATE THE RESPAWN COORDS
PLYR_START_X = PLYR.X
PLYR_START_Y = PLYR.Y
-- RESET THE CAMERA
CAMY = 1688
END -- END IF PLYR.X > 1024
```

TOKEN COUNT

140/8192 E

It's important to **keep track of the current level**, because where we warp the player to afterward will depend on that

We should add an **if statement** here that only sets these new coordinates if **level**'s new value is 2

And then when **level** becomes 3, we would add another if statement and set the player to the coordinates for level 3

Our cheat codes will also need to be adjusted

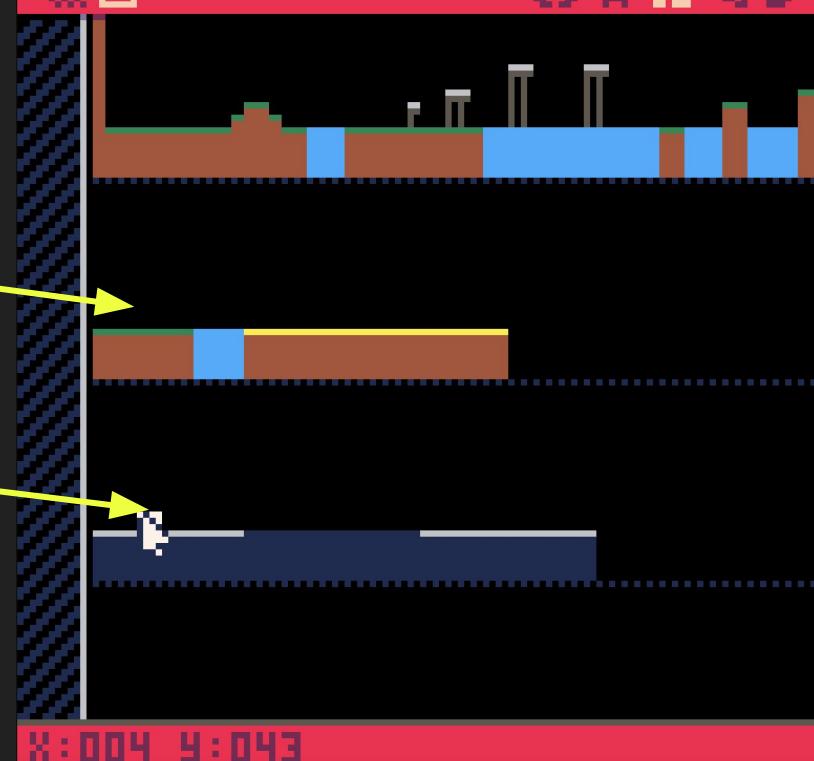
1 2 3 4 5 6 7 + ⏪ ⏴ ⏵ ⏹

```
-- START OF NEXT LEVEL AND
-- RESET THE CAMERA
IF LEVEL == 2 THEN
    PLYR.X = 4*8
    PLYR.Y = 27*8
    CARRY = 16*8
ELSEIF LEVEL == 3 THEN
    PLYR.X = 4*8
    PLYR.Y = 43*8
    CARRY = 32*8
ELSE
    -- RETURN TO START AFTER
    -- LAST LEVEL
    PLYR.X = 4*8
    PLYR.Y = 11*8
    CARRY = 0
    LEVEL = 1
END -- END IF LEVEL == 2/3
```

LINE 41/51 825/8192 ⏹

戻り

戻り



X:004 Y:043

Factoring the current level into  
the warp() function that sends  
the player to the next level

1 2 3 4 5 6 7 + ⌂ ⌃ ⌁ ⌂

-- WARP TO NEXT LEVEL

FUNCTION WARP()

-- CHEAT CODE: PRESS 2 TO

-- SKIP TO END OF LEVEL

IF BTNP(2) THEN

IF LEVEL == 1 THEN

PLYR.X = 124\*8

PLYR.Y = 11\*8

ELSEIF LEVEL == 2 THEN

PLYR.X = 123\*8

PLYR.Y = 27\*8

ELSEIF LEVEL == 3 THEN

PLYR.X = 121\*8

PLYR.Y = 43\*8

END -- END IF LEVEL 1/2/3

END -- END IF BTNP(2)

-- WARP TO LEVEL 2

LINE 41/51

825/8192 E



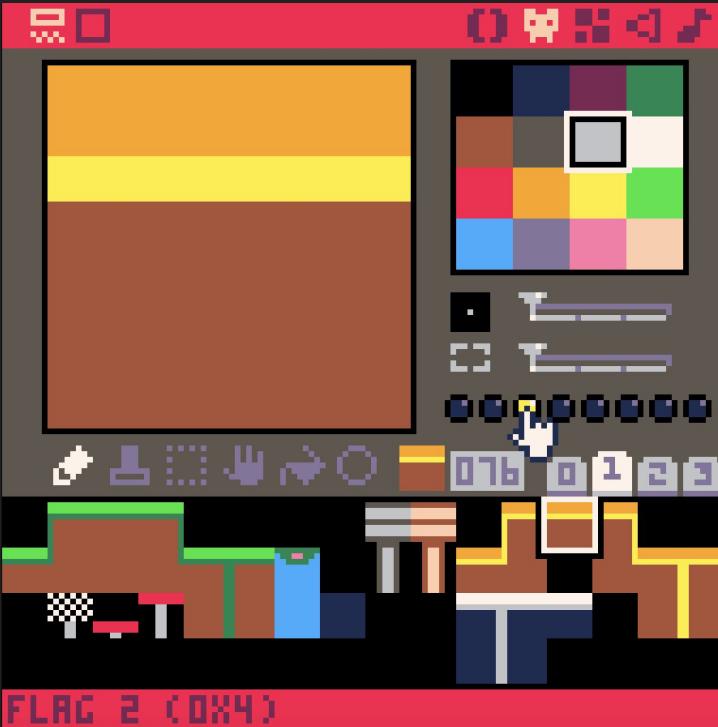
Factoring the current level into  
the cheat code that sends the  
player to the end of the level

Now that we have different levels,  
it would be cool if they featured  
different gameplay

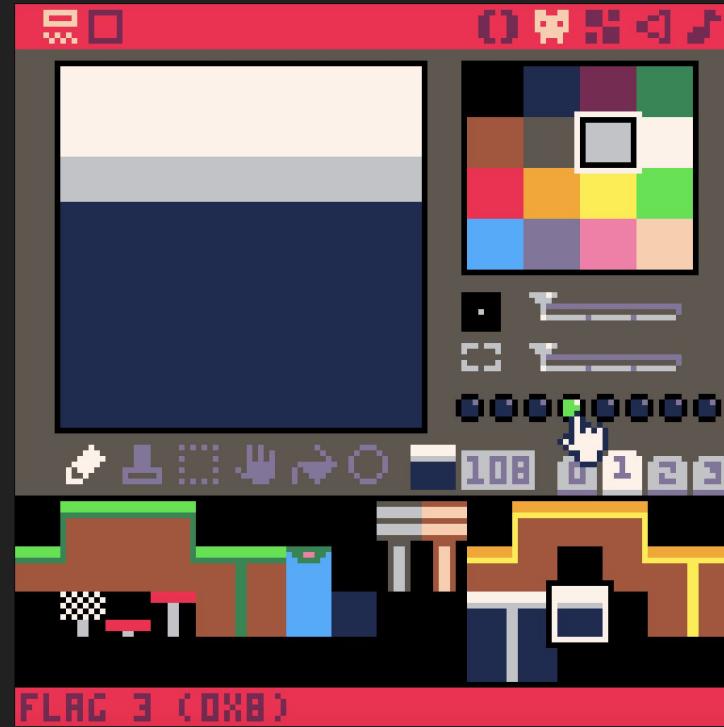
We could include different types of  
terrain (such as sand and ice) that  
affect the player's movement speed

# Step 15: Varied Terrain/Friction

Download Example File: [platformer\\_15\\_terrain\\_and\\_friction.p8](#)



I've turned on **flag 2** for **sand tiles**  
These will **slow** the player down



I've turned on **flag 3** for **ice tiles**  
These will **speed** the player up

*Please note that I've turned flag 0 OFF each of these because fget() becomes more complicated when there are multiple flags on for a sprite – we'll need to adjust our movement function to detect collision with these flags as well as regular solid tiles*

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9  
-- PLAYER STARTING COORDINATES  
PLYR_START_X = 7*7  
PLYR_START_Y = 11*8  
  
MAKE_PLYR() -- TAB 1  
  
-- STARTING POSITION FOR CAMERA  
CAMX = 0  
CAMY = 0  
  
-- ANIMATION TIMER  
ANIMTIME = 0  
  
-- SPRITE FLAGS FOR TERRAIN  
NORMAL = 0  
SAND = 2  
ICE = 3  
  
-- TRACK TYPE OF TERRAIN
```

In `_init()`, I'll declare variables for each sprite flag

0 1 2 3 4 5 6 7 0 9 8 < >

```
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER  
IF MCOLLIDE(PLYR, "DOWN", 0)  
AND PLYR.DY > 0 THEN  
    PLYR.DY = 0  
    PLYR.LANDED=TRUE  
  
-- BECAUSE OF VERTICAL SPEED,  
-- THE PLAYER CAN FALL A FEW  
-- PX INTO THE FLOOR. THIS  
-- CALCULATES HOW MANY PX AND  
-- RE-ADJUSTS Y (CREDIT TO  
-- NERDYTEACHERS.COM FOR THIS  
-- FORMULA)  
    PLYR.Y-=  
    ((PLYR.Y+PLYR.H+1)*8)-1  
END -- END IF MCOLLIDE DOWN
```

TOKEN COUNT 825/8192 E

Here's how I'll ensure collision  
with the new types of terrain

0 1 2 3 4 5 6 7 0 9 8 < >

```
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER  
IF  
(  
    MCOLLIDE(PLYR, "DOWN", 0)  
OR MCOLLIDE(PLYR, "DOWN", SAND)  
OR MCOLLIDE(PLYR, "DOWN", ICE)  
)  
AND PLYR.DY > 0 THEN  
    PLYR.DY = 0  
    PLYR.LANDED=TRUE
```

```
-- BECAUSE OF VERTICAL SPEED,  
-- THE PLAYER CAN FALL A FEW  
-- PX INTO THE FLOOR. THIS  
-- CALCULATES HOW MANY PX AND  
-- RE-ADJUSTS Y (CREDIT TO  
-- NERDYTEACHERS.COM FOR THIS
```

LINE 49/98

906/8192 E

0 1 2 3 4 5 6 7 8 9 A B C D E F

```
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER  
IF MCOLLIDE(PLYR, "DOWN", 0)  
AND PLYR.DY > 0 THEN  
    PLYR.DY = 0  
    PLYR.LANDED=TRUE  
  
-- BECAUSE OF VERTICAL SPEED,  
-- THE PLAYER CAN FALL A FEW  
-- PX INTO THE FLOOR. THIS  
-- CALCULATES HOW MANY PX AND  
-- RE-ADJUSTS Y (CREDIT TO  
-- NERDYTEACHERS.COM FOR THIS  
-- FORMULA)  
    PLYR.Y-=  
    ((PLYR.Y+PLYR.H+1)*8)-1  
END -- END IF MCOLLIDE DOWN
```

TOKEN COUNT 825/8192 E

Note the parentheses to group  
my stacking **OR**-conditions

0 1 2 3 4 5 6 7 8 9 A B C D E F

```
-- STOP FALLING WHEN A SOLID  
-- TILE IS BELOW THE PLAYER  
IF  
(  
    MCOLLIDE(PLYR, "DOWN", 0)  
    OR MCOLLIDE(PLYR, "DOWN", SAND)  
    OR MCOLLIDE(PLYR, "DOWN", ICE)  
)  
AND PLYR.DY > 0 THEN  
    PLYR.DY = 0  
    PLYR.LANDED=TRUE
```

```
-- BECAUSE OF VERTICAL SPEED,  
-- THE PLAYER CAN FALL A FEW  
-- PX INTO THE FLOOR. THIS  
-- CALCULATES HOW MANY PX AND  
-- RE-ADJUSTS Y (CREDIT TO  
-- NERDYTEACHERS.COM FOR THIS
```

LINE 49/98

906/8192 E

```
2 3 4 5 6 7 8 + () ← → ⌂
```

-- CHECK TERRAIN

**FUNCTION** CHECK\_TERRAIN()

-- CHANGE FRICTION DEPENDING  
-- ON THE TYPE OF TERRAIN THE  
-- PLAYER IS ON

**IF** MCOLLIDE(PLYR,"DOWN",SAND)  
**THEN**

FRIC = 0.50 -- SLOWER  
TERRAIN = "SAND"

**ELSEIF** MCOLLIDE(PLYR,"DOWN",ICE)  
**THEN**

FRIC = 0.93 -- FASTER  
TERRAIN = "ICE"

**ELSE**

FRIC = 0.85 -- DEFAULT  
TERRAIN = "NORMAL"

**END** -- END IF MCOLLIDE DOWN

LINE 6/20

887/8192 ↴

```
0 1 2 3 4 5 6 7 + () ← → ⌂
```

**END** -- END FUNCTION \_INIT()

**FUNCTION** \_UPDATE()

MOVE\_PLYR() -- TAB 2

CAN\_FOLLOW() -- TAB 4

RESPAWN() -- TAB 5

ANIMATE() -- TAB 6

WEAP() -- TAB 7

**CHECK\_TERRAIN()** -- TAB 8

**END** -- END FUNCTION \_UPDATE()

**FUNCTION** \_DRAW()

CLS() -- CLEAR SCREEN

MAP() -- DRAW MAP

SPR(PLYR.N, PLYR.X, PLYR.Y, 1, 1, PL)

-- PRINT LIVES

PRINT("LIVES: "...LIVES,CANX+2,C)

LINE 51/72 887/8192 ↴

This is my new function to check terrain and set friction accordingly, called in \_update()

2 3 4 5 6 7 8 + () [] < >

-- CHECK TERRAIN

FUNCTION CHECK\_TERRAIN()

```
-- CHANGE FRICTION DEPENDING
-- ON THE TYPE OF TERRAIN THE
-- PLAYER IS ON
IF MCOLLIDE(PLYR,"DOWN",SAND)
THEN
    FRIC = 0.50 -- SLOWER ←
    TERRAIN = "SAND"
ELSEIF MCOLLIDE(PLYR,"DOWN",ICE)
THEN
    FRIC = 0.93 -- FASTER ←
    TERRAIN = "ICE"
ELSE
    FRIC = 0.85 -- DEFAULT
    TERRAIN = "NORMAL"
END -- END IF MCOLLIDE DOWN
```

Because we're multiplying the player's dx (effectively their speed) by friction, a smaller value makes the player move more slowly (like on sand)

And a higher value (closer to 1.0) makes the player move faster (like on ice)



You'll want to *design your level to take advantage of the altered speed*

You could add difficulty to the sand level by requiring movement in tight spaces

And some long-distance jumps become possible in the ice level

# Learning Resources

# Step-by-Step Examples

1. platformer\_01\_starting\_position.p8
2. platformer\_02\_horizontal\_movement.p8
3. platformer\_03\_delta\_x.p8
4. platformer\_04\_friction.p8
5. platformer\_05\_jumping.p8
6. platformer\_06\_gravity.p8
7. platformer\_07\_map\_collision.p8
8. platformer\_08\_jumping\_2.p8
9. platformer\_09\_walls.p8
10. platformer\_10\_movement\_refined.p8

# Step-by-Step Examples

11. platformer\_11\_camera.p8
12. platformer\_12\_pits\_and\_respawn.p8
13. platformer\_13\_animated\_sprite.p8
14. platformer\_14\_multiple\_levels.p8
15. platformer\_15\_terrain\_and\_friction.p8

Extras:

- platformer\_10a\_correction\_illustrated.p8
- platformer\_13a\_toggle\_animation.p8

# More Learning Resources

- [Side-Scrolling Platformer Game Tutorial Playlist by Nerdy Teachers on YouTube](#)
- Function reference:
  - [map\(\)](#)
  - [mget\(\)](#)
  - [fget\(\)](#)
  - [mset\(\)](#)
  - [camera\(\)](#)
  - [time\(\)](#)



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](mailto:mdimatteo@rider.edu) anytime!