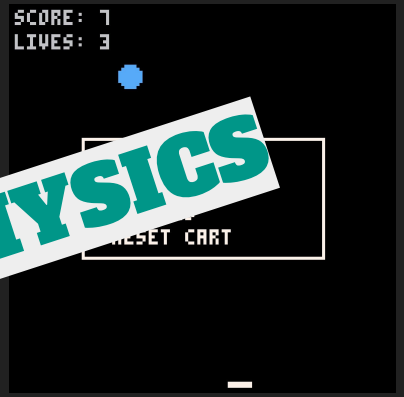
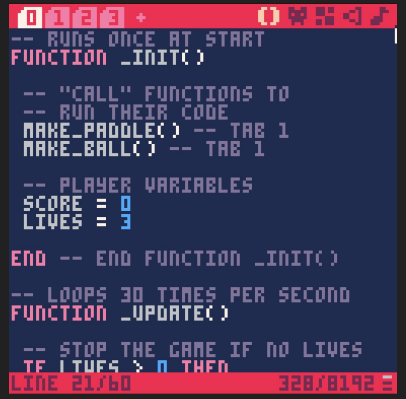
Making a Paddleball Game in PICO-8





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

as seen in GAM-120: Intro to Game Logic

Before You Start

Before Your Start

- This guide begins midway through the process of creating a paddleball game in PICO-8
- *It assumes you already understand:*
  - *How to work in the PICO-8 editor*
  - *Variables, functions, and conditional statements*
  - *How the PICO-8 game loop works*
  - *How to check for and respond to input*
- If you haven't yet, check out the basic version, which covers those concepts in detail

Contents

Game Objects as Tables

Download Example File: paddleball_physics_01_variables.p8

Game Objects

- We’ve been using our variable names to differentiate between paddle variables and ball variables

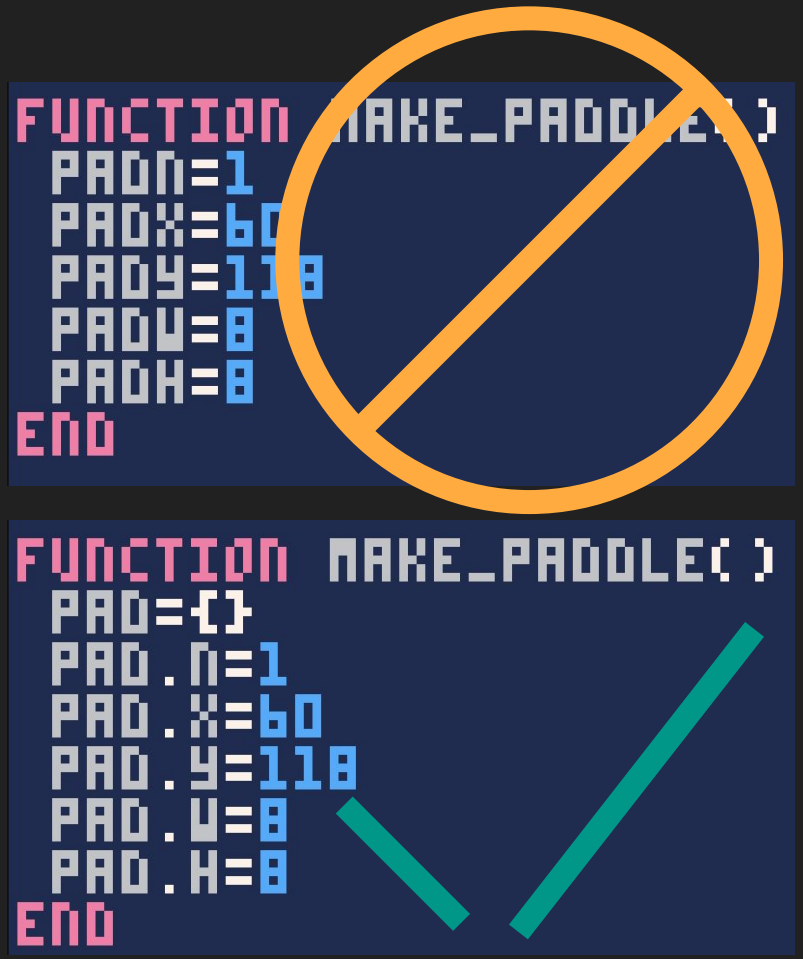
```
function make_paddle()
 padn=1
 padx=60
 pady=118
 padv=8
 padh=8
end

function make_ball()
 baln=2
 balx=60
 baly=2
 padv=8
 padh=8
end
```

Game Objects

- We've been using our variable names to differentiate between paddle variables and ball variables – but *this approach will limit us when we need to determine whether the ball and paddle are touching*
- We need another pair of variables for the paddle and ball themselves

Game Objects

- We can have more control over our data by grouping related variables within “tables”
- Each table represents a game object



Game Objects

- A table is a special kind of variable that can contain other variables
- *Other programming languages may call this an "array" or "list"*

Think of how data is organized in a **spreadsheet** – it's much the same with tables

```
FUNCTION MAKE_PADDLE()
 PAD={}
  PAD.N=1
  PAD.X=60
  PAD.Y=118
  PAD.W=8
  PAD.H=8
END
```

	PADDLE	BALL
2	x=60	x=60
3	y=118	y=2
4	n=1	n=2

Game Objects

- Assign a variable a value of {} to make it a table
- PAD={} creates a table named PAD

```
FUNCTION MAKE_PADDLE()
 PAD={}
 PAD.N=1
 PAD.X=60
 PAD.Y=118
 PAD.W=8
 PAD.H=8
END
```

Game Objects

- You can then add variables that belong to the object's table by typing the object/table name, followed by a period and the name of the variable

```
FUNCTION MAKE_PADDLE()
  -- GAME OBJECT IS SPECIAL TYPE
  -- OF VARIABLE THAT CAN CONTAIN
  -- OTHER VARIABLES)
  PADDLE = {}

  -- OBJECT PROPERTIES (MUST BE
  -- WRITTEN WITH OBJECT NAME,
  -- FOLLOWED BY A PERIOD AND THE
  -- NAME OF THE PROPERTY
  PADDLE.N = 1 -- SPRITE NUMBER
  PADDLE.X = 60
  PADDLE.Y = 118
  PADDLE.SPEED = 3
END
```

Game Objects

- You can also assign variables to a table *within* the curly brackets, separated by commas (without using the dot syntax)

```
function make_paddle()
  pad={}
  pad.n=1
  pad.x=60
  pad.y=118
  pad.w=8
  pad.h=8
end
```

Both approaches will work the same

```
function make_ball()
  bal={
  n=2,
  x=60,
  y=2,
  w=8,
  h=8
  }
end
```

Both approaches will work the same

Game Objects

- I prefer to write tables as in the above method
  - Less chance of forgetting a comma
  - You'll need to refer to objects using the dot syntax anyway
- But it's totally up to you! Both ways work the same

```
function make_paddle()
 pad={}
 pad.n=1
 pad.x=60
 pad.y=118
 pad.w=8
 pad.h=8
end
```

Both approaches will work the same

```
function make_ball()
 bal={
 n=2,
 x=60,
 y=2,
 w=8,
 h=8
 }
end
```

Both approaches will work the same

If you've renamed any of your variables (such as changing pad_x to pad.x), *make sure you're referring to them consistently* throughout the program!

Gravity & Physics, Delta Time

Download Example File: paddleball_physics_05_ball.p8

Gravity and Physics

- I'll create a variable called gravity (or grav for short) in _init()
- If I add any other objects (like powerups), I'd like them to fall at the same speed as the ball
- In real life, gravity affects everything equally

```
function _init()
 gravity = 3
 bal = {}
 bal.n = 2
 bal.x = 60
 bal.y = 2
end
```

*Conversely, if you decide you want to have different objects fall at different speeds (because f*ck realism), you might choose to have variables belong to objects like bal.grav and pwrup.grav*

Gravity and Physics

- We can move the ball lower on the screen by simply incrementing its **y** value
- But this motion is kind of boring and unrealistic (flat rate)

```
function _init()
 gravity = 3
 bal = {}
 bal.n = 2
 bal.x = 60
 bal.y = 2
end

function _update()
 bal.y += gravity
end

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

Gravity and Physics

- To mimic real-life physics, I'll create another variable called **dy** to determine how much the ball should move, and *then increment y* by the value of dy

```
function _init()
 gravity = 3
 bal = {}
 bal.n = 2
 bal.x = 60
 bal.y = 2
 bal.dy= 0
end

function _update()
 bal.dy += gravity
 bal.y  += bal.dy
end

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

Gravity and Physics

- It's customary to name this variable **dx** or **dy**
- The d stands for delta, or "*change in*"
- ***dy** is then the change in **y***

```
function _init()
 gravity = 3
 bal = {}
 bal.n = 2
 bal.x = 60
 bal.y = 2
 bal.dy= 0
end

function _update()
 bal.dy += gravity
 bal.y  += bal.dy
end

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

Gravity and Physics

- Think of what happens when you press the accelerator pedal in your car to go from 0 to 75 MPH

A	B
SPEED	**ACCELERATION**
0	0
5	5
15	10
30	15
50	20
75	25

Not to say you should actually floor it like this . . . but for argument's sake ;)

Gravity and Physics

- The acceleration is the *change in speed from one unit of time to the next*
- **dy** is effectively velocity (change in distance over time)
- But because of the looping nature of the program, when we increment **dy** by **gravity**, we achieve **acceleration**

A	B
SPEED	**ACCELERATION**
0	0
5	5
15	10
30	15
50	20
75	25

Not to say you should actually floor it like this . . . but for argument's sake ;)

Gravity and Physics

```
function _update()
 ball.dy += gravity
 ball.y  += ball.dy
end
```

Notice how the ball's y position is moving by a larger amount each frame, because this calculation causes a compounding effect

GRAVITY	DY	Y	FRAME	TIME
3	0	0	0	0
3	3	3	1	1/30 sec
3	6	9	2	2/30 sec
3	9	18	3	3/30 sec
3	12	30	4	4/30 sec
3	15	45	5	5/30 sec
3	18	63	6	6/30 sec
3	21	84	7	7/30 sec
3	24	108	8	8/30 sec
3	27	135	9	9/30 sec
3	30	165	10	10/30 sec

Gravity and Physics

```
function _update()
 bal.dy += gravity
 bal.y  += bal.dy
end
```

This is actually *way too fast* – the ball will be off screen (y > 128) after less than ⅓ of a second

A value of 0.3 for gravity works better

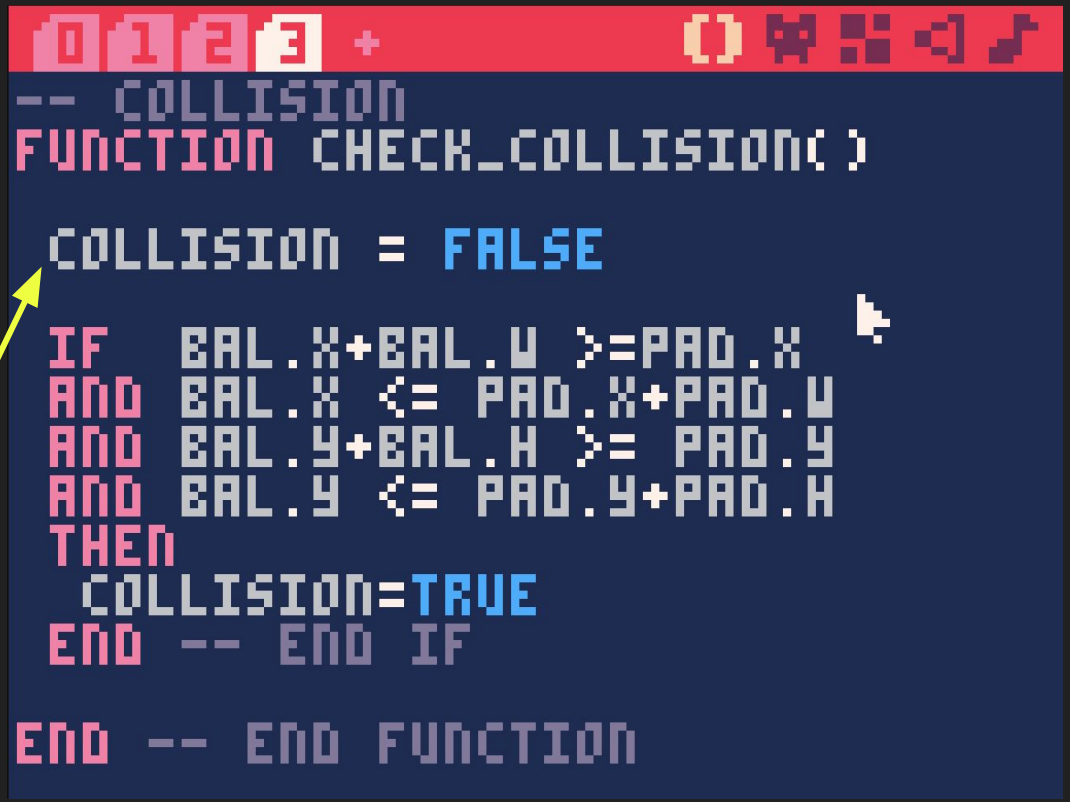
GRAVITY	DY	Y	FRAME	TIME
3	0	0	0	0
3	3	3	1	1/30 sec
3	6	9	2	2/30 sec
3	9	18	3	3/30 sec
3	12	30	4	4/30 sec
3	15	45	5	5/30 sec
3	18	63	6	6/30 sec
3	21	84	7	7/30 sec
3	24	108	8	8/30 sec
3	27	135	9	9/30 sec
3	30	165	10	10/30 sec

Collision Detection Between Objects

Download Example File: paddleball_physics_06_object_collision.p8

Collision Detection Between Objects

This function creates a boolean variable (*can be either true or false*) called **collision** and sets it to false initially

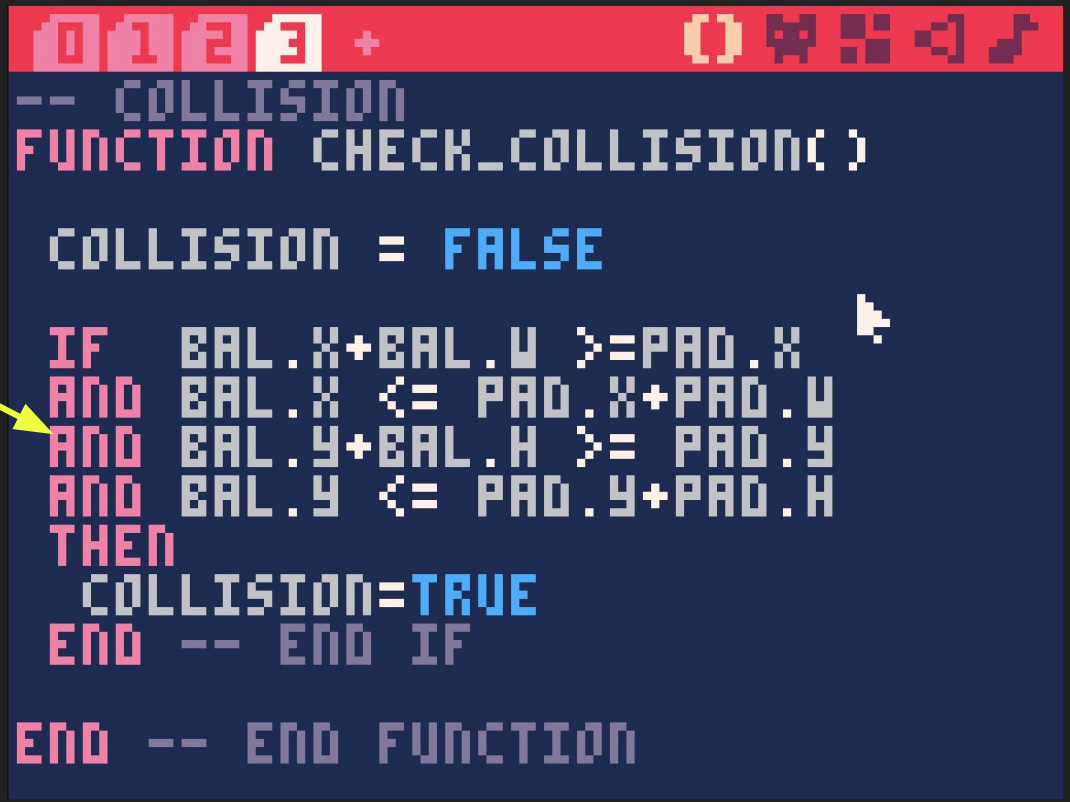


Collision Detection Between Objects

We then need four conditions to be met for collision to be set to true

We can use the keyword **AND** to stack conditions

AND is an example of a "logical operator" – others are OR, NOT



Collision Detection Between Objects

In our code that moves the ball, we need to **call the check_collision function** after gravity is applied, but before Y is updated

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 CHECK_COLLISION()
 IF COLLISION == TRUE THEN
  BAL.DY = 0
 END

 BAL.Y  += BAL.DY
END
```

If collision is true, we can set the ball's DY to 0 to make it stop moving

Collision Detection Between Objects

We can check whether a variable is equal to a particular value using **two equals signs ==**

```
CHECK_COLLISION()
IF COLLISION == TRUE THEN
 BAL.DY = 0
END
```

Using == for comparison is customary in many programming languages; this is called a "comparison operator"

Reversing the Ball's Direction

Download Example File: paddleball_physics_07_reverse_ball_direction.p8

Collision Detection Between Objects

To make the ball bounce, we can instead **multiply its DY by -1** to reverse its vertical direction

```
FUNCTION MOVE_BAL()
  BAL.DY += GRAV

  CHECK_COLLISION()
  IF COLLISION == TRUE THEN
    BAL.DY *= -1
  END

  BAL.Y  += BAL.DY
END
```

If the ball's dy was 7 as it struck the paddle, we want it to be bounce and move equally fast in the opposite direction, which would be a value of -7

This works, but what if we want to detect collision between additional objects? Our function is not flexible enough to do that yet

Collision Detection Between Objects

We can modify our function to require a pair of values fed into it

Let's call these A and B

The values go inside the parentheses

```
FUNCTION COLLISION(A,B)

 IF   B.X+B.W >=A.X
 AND  B.X  <= A.X+A.W
 AND  B.Y+B.H >= A.Y
 AND  B.Y  <= A.Y+A.H
 THEN
  RETURN TRUE
 ELSE
  RETURN FALSE
 END -- END IF/ELSE

END -- END FUNCTION
```

Collision Detection Between Objects

Remember how built-in functions like SPR() require values in a particular order?

We can design our custom functions the same way

```
FUNCTION COLLISION(A,B)
```

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

A and B are temporary variables, like a nickname or an alias

The same way you can feed any value into SPR() – it doesn't need to be called n,x,y, etc. like in the documentation – you can feed any variable into this collision function, and those variables will temporarily be treated as A and B

Collision Detection Between Objects

There are some requirements for whichever variables we are treating as A and B:

They must be objects with properties named X,Y,W,H (or however you name those in your collision function)

```
FUNCTION COLLISION(A,B)

 IF   B.X+B.W >=A.X
 AND  B.X <= A.X+A.W
 AND  B.Y+B.H >= A.Y
 AND  B.Y <= A.Y+A.H
 THEN
  RETURN TRUE
 ELSE
  RETURN FALSE
 END -- END IF/ELSE

END -- END FUNCTION
```

Collision Detection Between Objects

Finally, we can use the keyword RETURN to provide “output” for the function

If the four conditions are met, we return TRUE

Otherwise, we return FALSE

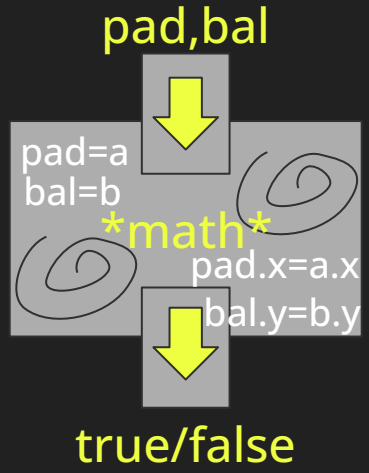
This is another convention that is common in other programming languages

```
FUNCTION COLLISION(A,B)

  IF   B.X+B.W >=A.X
  AND  B.X <= A.X+A.W
  AND  B.Y+B.H >= A.Y
  AND  B.Y <= A.Y+A.H
  THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  END -- END IF/ELSE

END -- END FUNCTION
```

Collision Detection Between Objects

Functions are kind of like a meat grinder

You provide input (variables) that are used to perform a calculation, which results in output (the returned true or false value)




```
FUNCTION COLLISION(A,B)

 IF   B.X+B.W >=A.X
 AND  B.X <= A.X+A.W
 AND  B.Y+B.H >= A.Y
 AND  B.Y <= A.Y+A.H
 THEN
  RETURN TRUE
 ELSE
  RETURN FALSE
 END -- END IF/ELSE

END -- END FUNCTION
```

Collision Detection Between Objects

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 CHECK_COLLISION()
 IF COLLISION == TRUE THEN
  BAL.DY *= -1
 END

 BAL.Y  += BAL.DY
END
```

Old, inflexible way

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 IF COLLISION(BAL,PAD) == TRUE
 THEN
  BAL.DY *= -1
 END

 BAL.Y  += BAL.DY
END
```

New, more flexible way

To use this function, you would plug BAL and PAD in as values between the parentheses; if the result is true, then you change the ball's dy

Collision Detection Between Objects

With this approach, you can plug in ***any*** objects, as long as they have properties named W and H

For example, PWRUP.W and PWRUP.H

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 IF COLLISION(BAL,PAD) == TRUE
 THEN
  BAL.DY *= -1
 END

 BAL.Y  += BAL.DY
END
```

```
IF COLLISION(BALL,PWRUP)==TRUE
 THEN
  SCORE += 1000
 END
```

Collision Detection Between Objects

```
FUNCTION COLLISION(A,B)

 IF   B.X+B.W >=A.X
 AND B.X <= A.X+A.W
 AND B.Y+B.H >= A.Y
 AND B.Y <= A.Y+A.H
 THEN
  RETURN TRUE
 ELSE
  RETURN FALSE
 END -- END IF/ELSE

END -- END FUNCTION
```

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 IF COLLISION(BAL,PAD) == TRUE
 THEN
  BAL.DY *= -1
 END

 BAL.Y  += BAL.DY
END
```

When you “plug in” BAL and PAD into the function, BAL is treated as A, and PAD is treated as B – so A.X becomes BAL.X, and A.Y becomes BAL.Y

Collision Detection Between Objects

```
FUNCTION COLLISION(A,B)

 IF   B.X+B.W >=A.X
 AND B.X <= A.X+A.W
 AND B.Y+B.H >= A.Y
 AND B.Y <= A.Y+A.H
 THEN
  RETURN TRUE
 ELSE
  RETURN FALSE
 END -- END IF/ELSE

END -- END FUNCTION
```

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 IF COLLISION(BAL,PAD) == TRUE
 THEN
  BAL.DY *= -1
 END

 BAL.Y  += BAL.DY
END
```

BAL and PAD need to have a W and H because A.W will be BAL.W, and B.H will be PAD.H, for example

Collision Detection Between Objects

```
FUNCTION COLLISION(A,B)

 IF   B.X+B.W >=A.X
 AND B.X <= A.X+A.W
 AND B.Y+B.H >= A.Y
 AND B.Y <= A.Y+A.H
 THEN
  RETURN TRUE
 ELSE
  RETURN FALSE
 END -- END IF/ELSE

END -- END FUNCTION
```

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 IF COLLISION(BAL,PAD) == TRUE
 THEN
  BAL.DY *= -1
 END

 BAL.Y  += BAL.DY
END
```

We could **reverse the order** and feed in PAD and BAL instead of BAL and PAD, and the **result would be the same** because our collision check function is **symmetrical**

Collision Detection Between Objects

```
FUNCTION COLLISION(A,B)

 IF   B.X+B.W >=A.X
 AND B.X <= A.X+A.W
 AND B.Y+B.H >= A.Y
 AND B.Y <= A.Y+A.H
 THEN
  RETURN TRUE
 ELSE
  RETURN FALSE
 END -- END IF/ELSE

END -- END FUNCTION
```

```
FUNCTION MOVE_BAL()
 BAL.DY += GRAV

 IF COLLISION(BAL,PAD) == TRUE
 THEN
  BAL.DY *= -1
 END

 BAL.Y  += BAL.DY
END
```

But that's just this function; others you write (or built-in functions you use) may require particular orders, like SPR(n,x,y)

Making the Ball Move Horizontally

Download Example File: paddleball_physics_08_horizontal_ball_movement.p8

Moving the Ball Horizontally

If the paddle is moving sideways when it hits the ball, we want the ball to move in the same direction

We'll need to add a new variable to our ball for its horizontal speed, called dx

```
FUNCTION MAKE_BAL()
 BAL = {}
 BAL.N = 2
 BAL.X = 60
 BAL.Y = 2
 BAL.V = 8
 BAL.H = 8
 BAL.DX= 0 -- HORIZONTAL SPD
 BAL.DY= 0 -- VERTICAL SPD
END
```

Moving the Ball Horizontally

If the paddle is moving sideways when it hits the ball, we want the ball to move in the same direction

We can assign bal.dx the value of the paddle's speed, which is pad.spd

We can use negative pad.spd (-pad.spd) for moving left

```
IF COLLISION(BAL,PAD) == TRUE
THEN
 BAL.DY *= -1

 IF BTN(⬅) THEN
  BAL.DX = -PAD.SPD
 END -- END IF BTN(⬅)

 IF BTN(➡) THEN
  BAL.DX = PAD.SPD
 END -- END IF BTN(➡)

END -- END IF COLLISION

BAL.X += BAL.DX -- UPDATE X
BAL.Y += BAL.DY -- UPDATE Y
```

Moving the Ball Horizontally

Make sure that the ***if/then*** *statements checking for a key press are wrapped within the if/then block handling collision*

If your ball is always moving in the same direction as the paddle, you likely placed these outside the collision block

```
IF COLLISION(BAL,PAD) == TRUE
THEN
 BAL.DY *= -1

 IF BTN(⬅) THEN
  BAL.DX = -PAD.SPD
 END -- END IF BTN(⬅)

 IF BTN(➡) THEN
  BAL.DX = PAD.SPD
 END -- END IF BTN(➡)

END -- END IF COLLISION

BAL.X += BAL.DX -- UPDATE X
BAL.Y += BAL.DY -- UPDATE Y
```

Moving the Ball Horizontally

```
IF COLLISION(BAL,PAD) == TRUE
THEN
 BAL.DY *= -1

 IF BTN(⬅️) THEN
  BAL.DX = -PAD.SPD
 END -- END IF BTN(⬅️)

 IF BTN(➡️) THEN
  BAL.DX = PAD.SPD
 END -- END IF BTN(➡️)

END -- END IF COLLISION

BAL.X += BAL.DX -- UPDATE X
BAL.Y += BAL.DY -- UPDATE Y
```

Finally, we need to apply the calculated change in x (bal.dx) to the value of x (bal.x)

Moving the Ball Horizontally

If you write the if/then statements handling key presses *outside* the if/then statement for collision, then *the ball will move whenever the player moves the paddle*

Could be a cool effect in a different game, but probably not desirable here

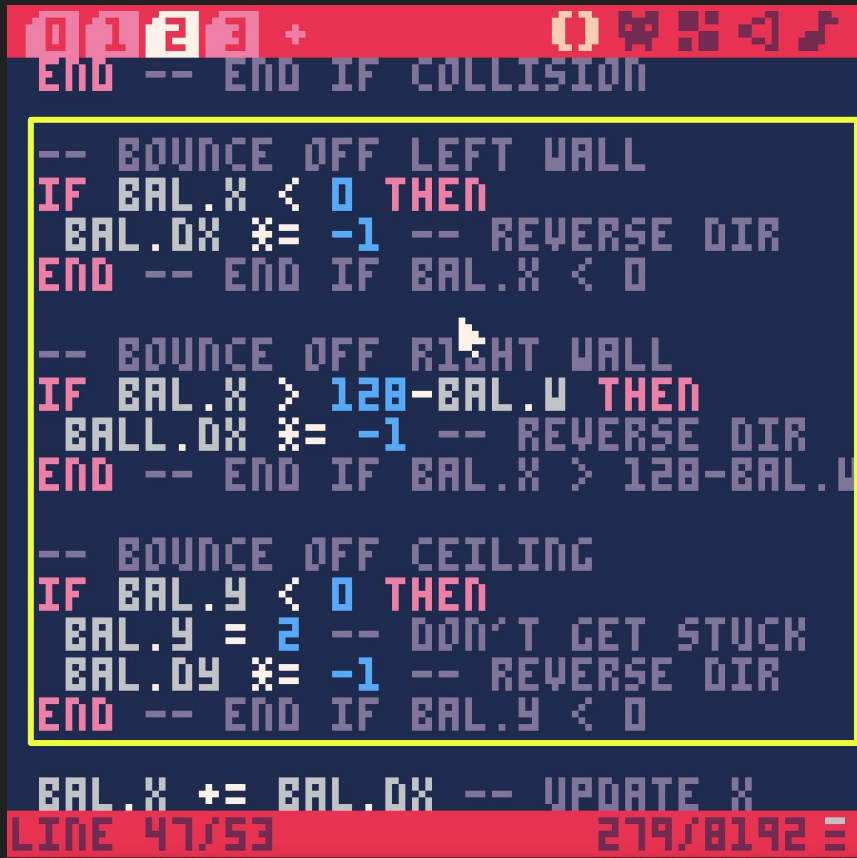
```
IF COLLISION(BAL,PAD) == TRUE
THEN
  BAL.DY *= -1
END -- END IF COLLISION

IF BTN(⬅) THEN
  BAL.DX = -PAD.SPD
END -- END IF BTN(⬅)

IF BTN(➡) THEN
  BAL.DX = PAD.SPD
END -- END IF BTN(➡)

BAL.X += BAL.DX -- UPDATE X
BAL.Y += BAL.DY -- UPDATE Y
```

Bouncing off the Walls & Ceiling

Download Example File: paddleball_physics_09_constrain_and_reset_ball.p8

Bouncing off the Walls and Ceiling

After the collision check, but before we update x and y, we can make the ball bounce off the walls

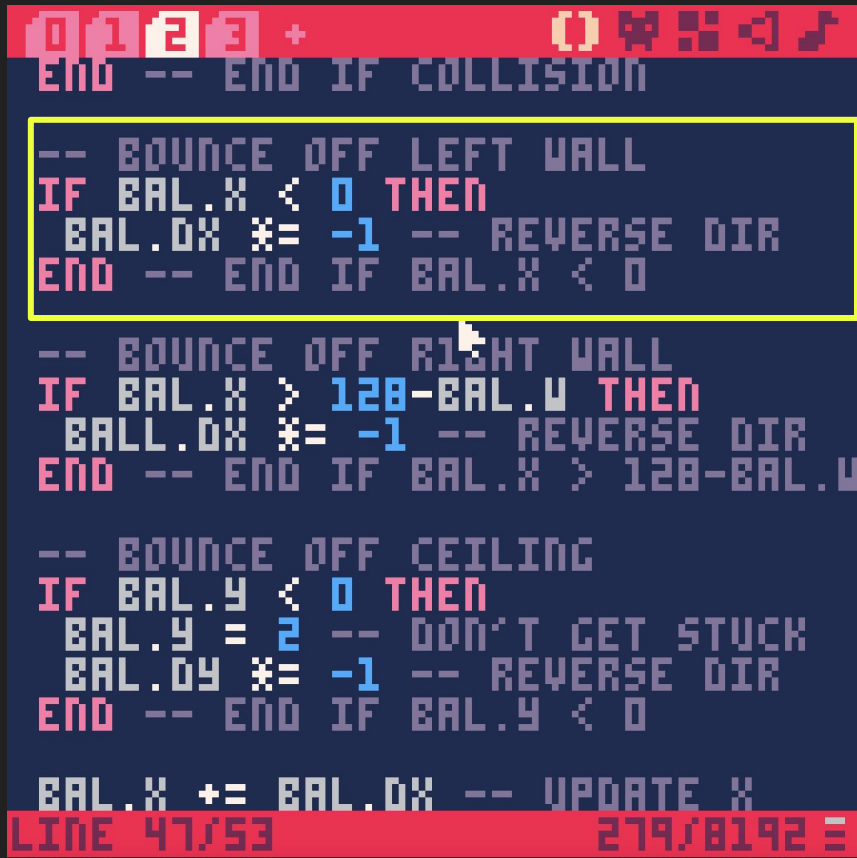


Can anyone spot the typo in my code? :P

Bouncing off the Walls and Ceiling

Similarly to keeping the paddle on screen, we want to check for if the ball has gone off screen to the left by checking if its x value is less than 0

If so, we can multiply the ball's dx by -1 to reverse its horizontal direction

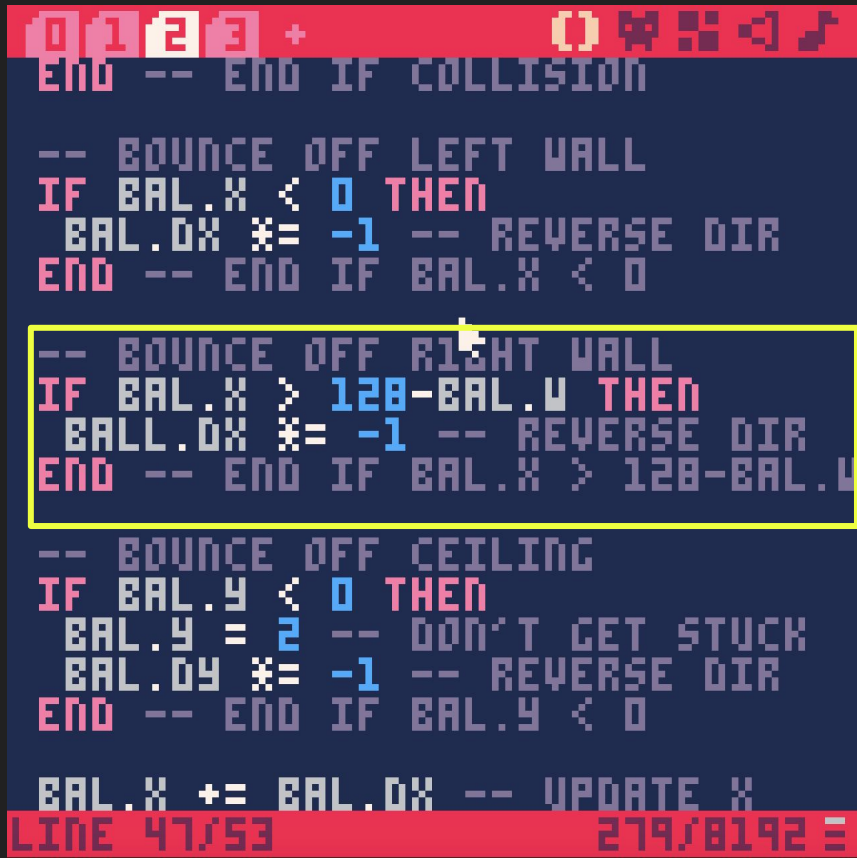


Can anyone spot the typo in my code? :P

Bouncing off the Walls and Ceiling

Similarly to keeping the paddle on screen, we want to check for if the ball has gone off screen to the right by checking if its x value is greater than 128 minus the ball's width

If so, we can multiply the ball's dx by -1 to reverse its horizontal direction



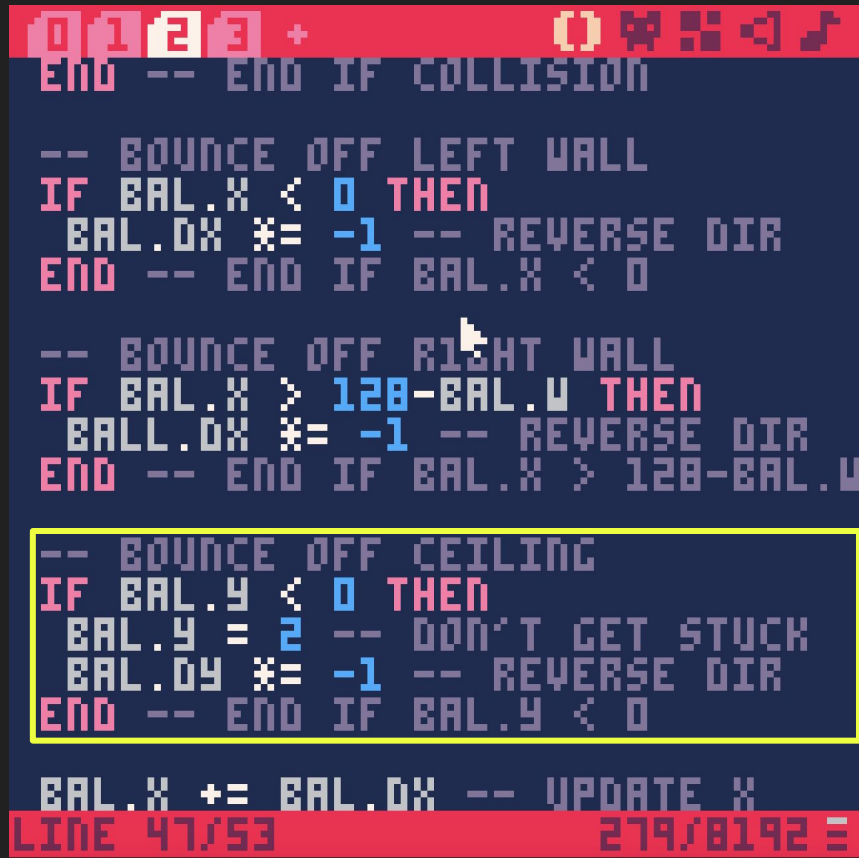
Can anyone spot the typo in my code? :P

Bouncing off the Walls and Ceiling

And we can check if the ball has gone off the top of the screen by checking if its y is less than 0

If so, we can multiply the ball's dy by -1 to reverse its horizontal direction

I also reset the y position to 2 so that the ball doesn't get stuck



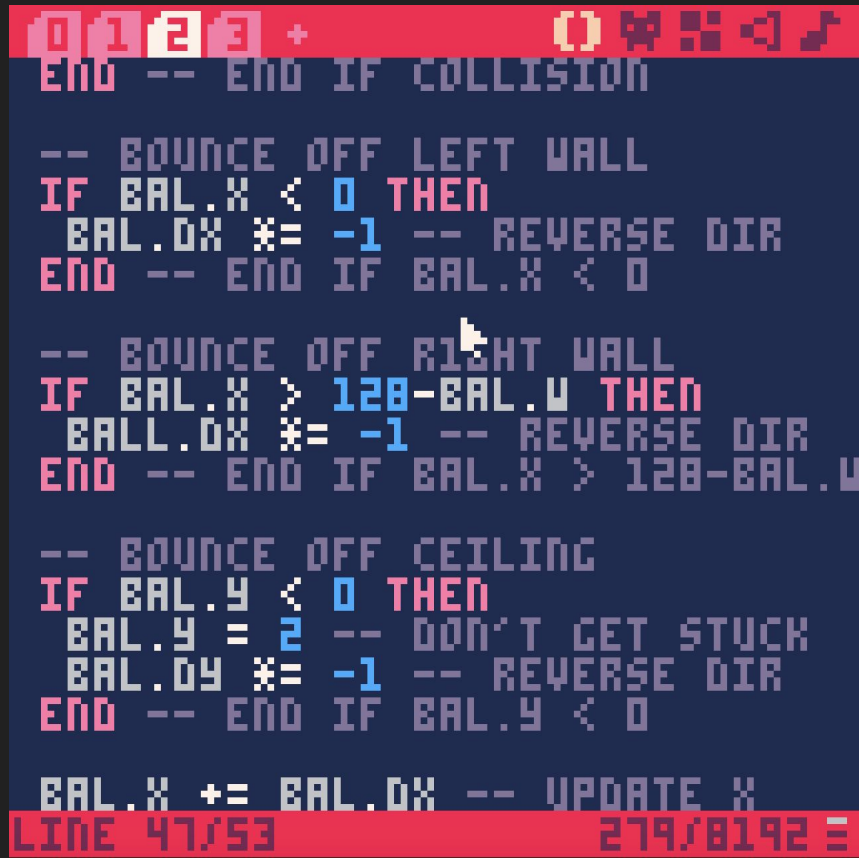
Can anyone spot the typo in my code? :P
ball.dx should be bal.dx

Bouncing off the Walls and Ceiling

I realize our move_bal() function is getting pretty long

Here's the entire function in VS Code on the left

```
function move_bal()

    bal.dy += grav -- apply gravity

    if collision(bal,pad) == true
    then
        bal.dy *= -1

        if btn(⬅️) then
            bal.dx = -pad.spd
        end -- end if btn(⬅️)

        if btn(➡️) then
            bal.dx = pad.spd
        end -- end if btn(➡️)

    end -- end if collision

    -- bounce off left wall
    if bal.x < 0 then
        bal.dx *= -1 -- reverse dir
    end -- end if bal.x < 0

    -- bounce off right wall
    if bal.x > 128-bal.w then
        ball.dx *= -1 -- reverse dir
    end -- end if bal.x > 128-bal.w

    -- bounce off ceiling
    if bal.y < 0 then
        bal.y = 2 -- don't get stuck
        bal.dy *= -1 -- reverse dir
    end -- end if bal.y < 0

    bal.x += bal.dx -- update x
    bal.y += bal.dy -- update y

end -- end function move_bal()
```

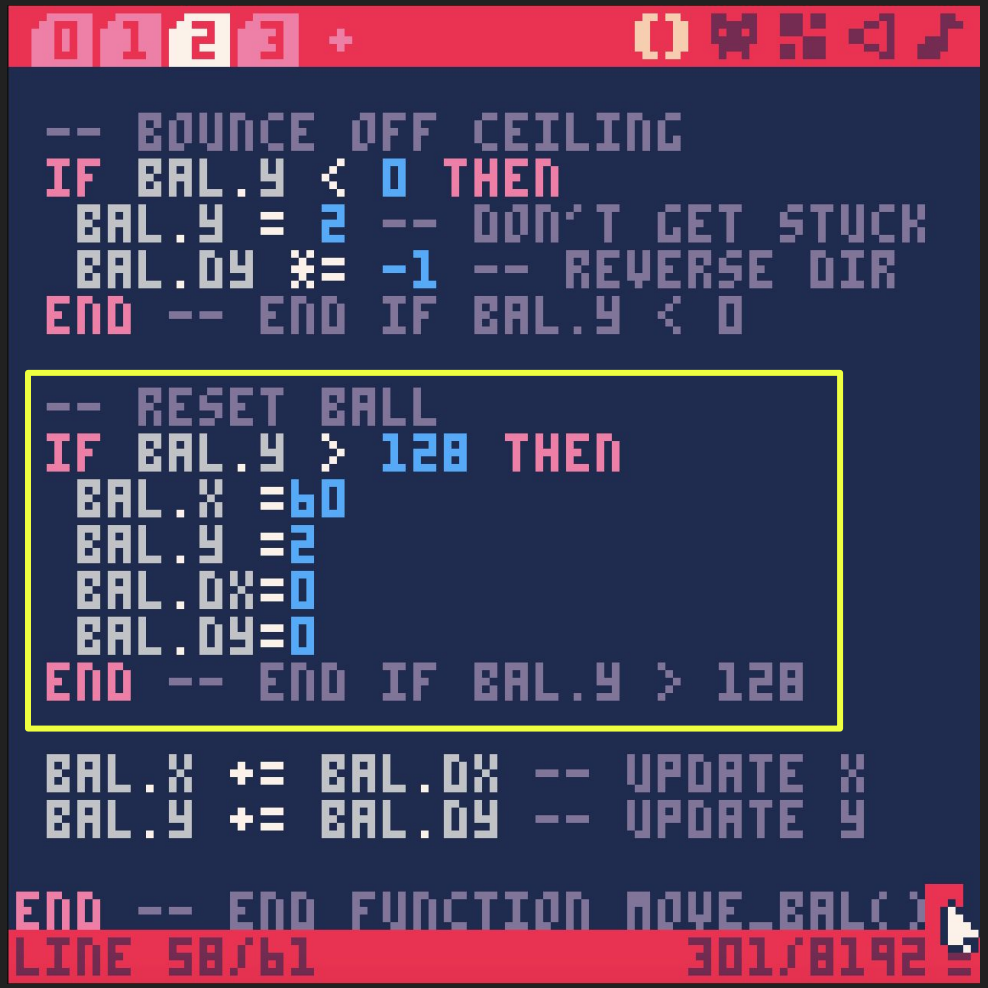


Resetting the Ball After a Miss

Download Example File: paddleball_physics_09_constrain_and_reset_ball.p8

Resetting the Ball

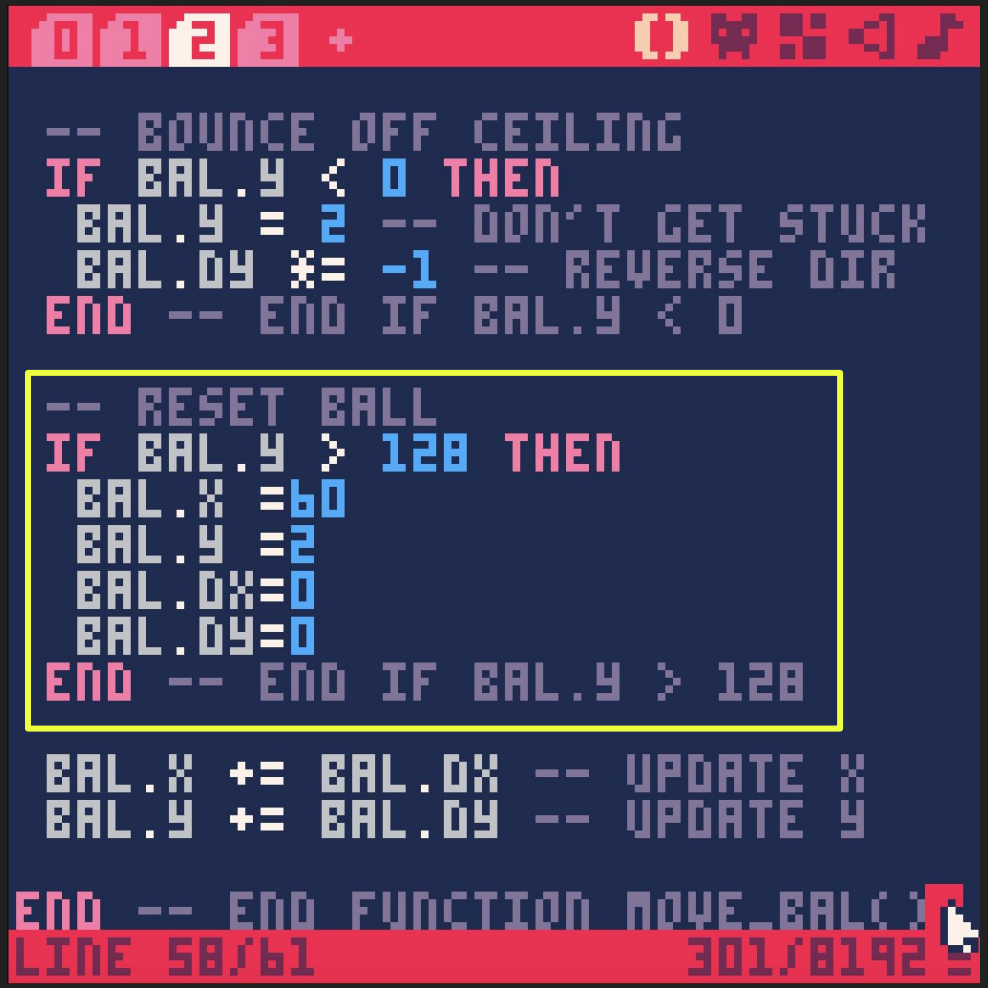
If the player misses the ball and it falls off the bottom of the screen, we need to restore several of its properties to their initial values



Resetting the Ball

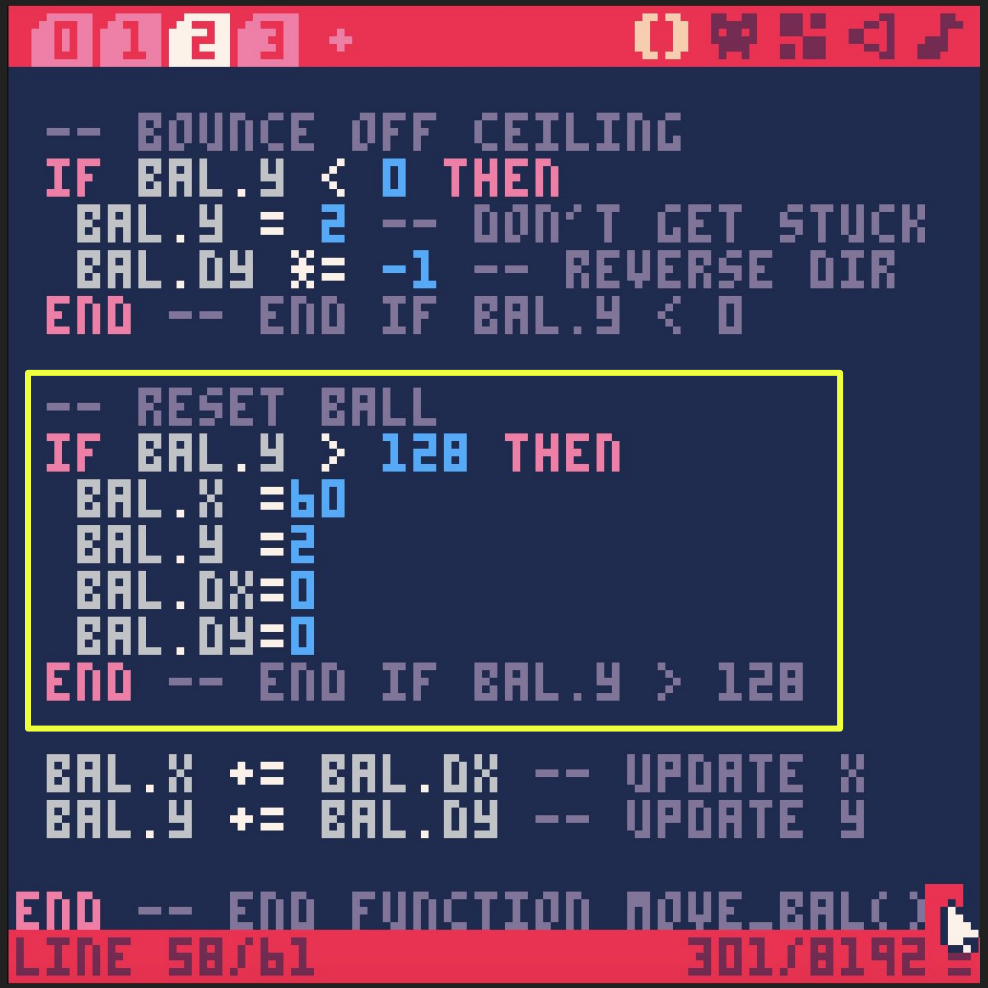
We reset both the ball's dx and its dy to 0 to stop it from moving (*at first*)

Gravity will then make it start falling again on the next frame when _update() loops 30x/second)



Resetting the Ball

We reset both the X and Y to 60 and 2, respectively (or whatever you initially set yours at)

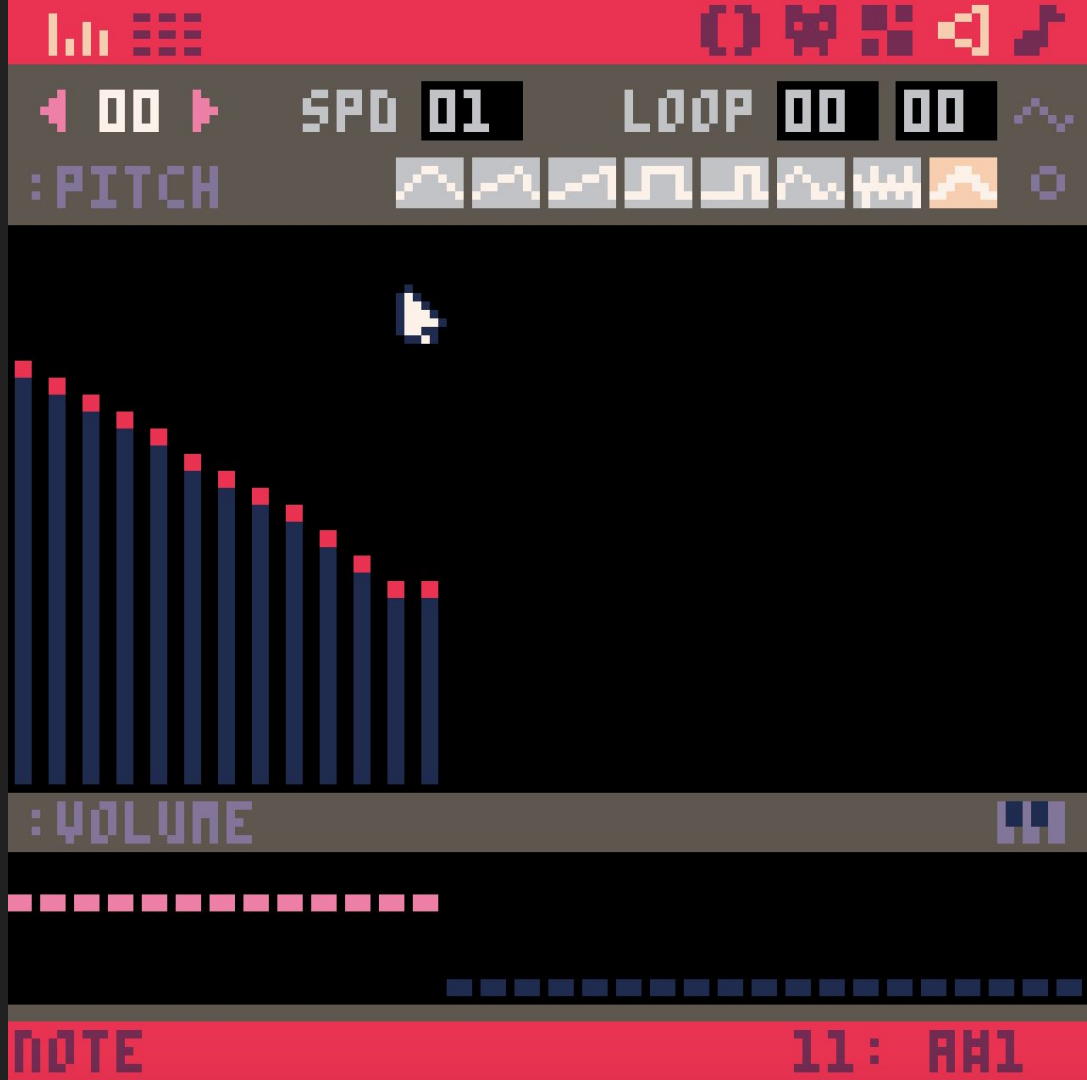


Adding Sounds

Download Example File: paddleball_physics_10_finetuning_and_troubleshooting.p8

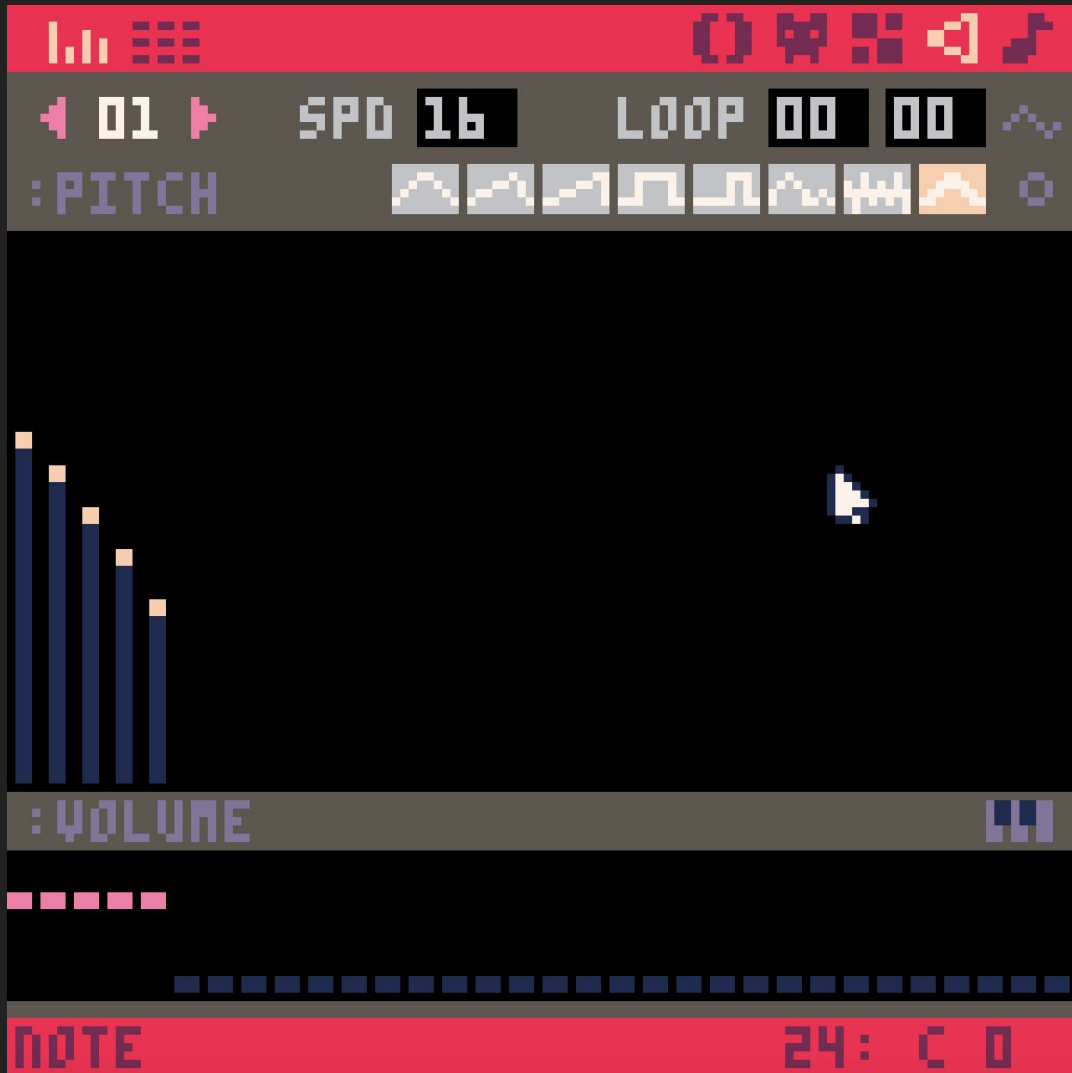
Adding Sounds

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



Adding Sounds

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

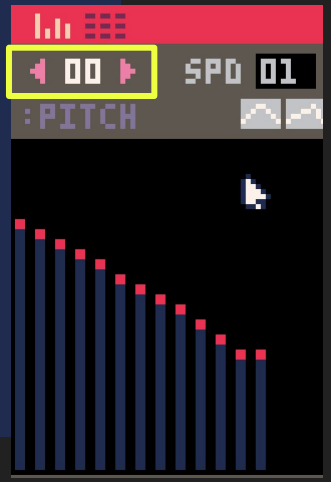


Adding Sounds

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

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

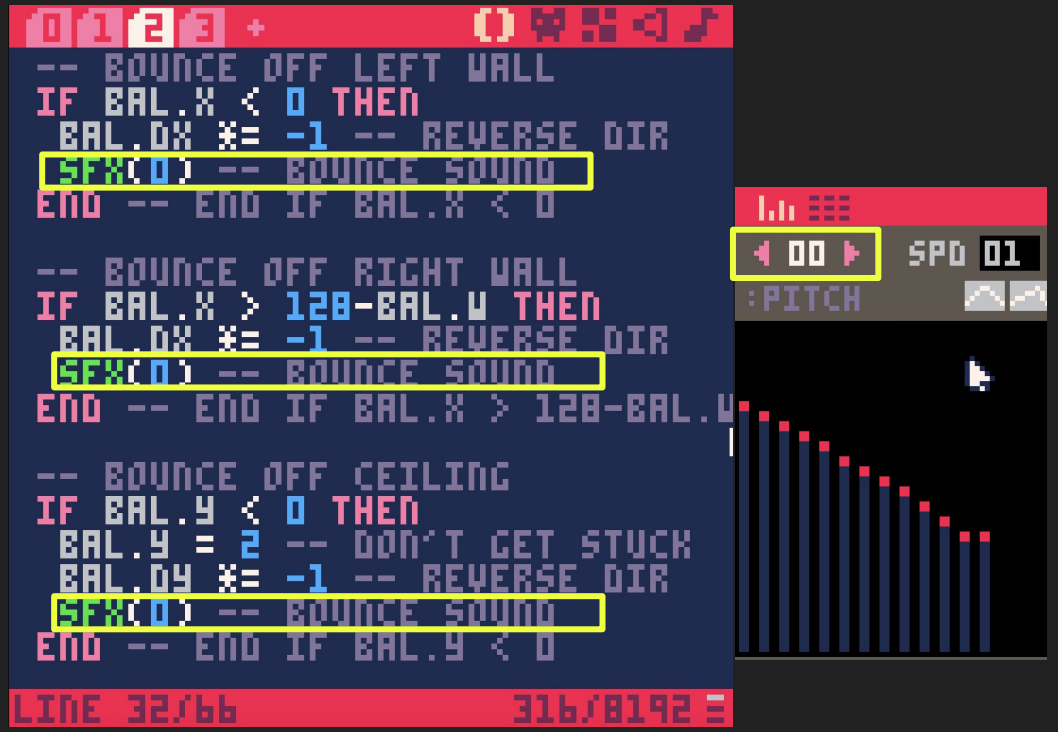
```
IF COLLISION(BAL,PAD) == TRUE
THEN
 BAL.DY *= -1
 SFX(0) -- BOUNCE SOUND

 IF BTN(⬅) THEN
  BAL.DX = -PAD.SPD
 END -- END IF BTN(⬅)

 IF BTN(➡) THEN
  BAL.DX = PAD.SPD
 END -- END IF BTN(➡)

END -- END IF COLLISION
```



Adding Sounds

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

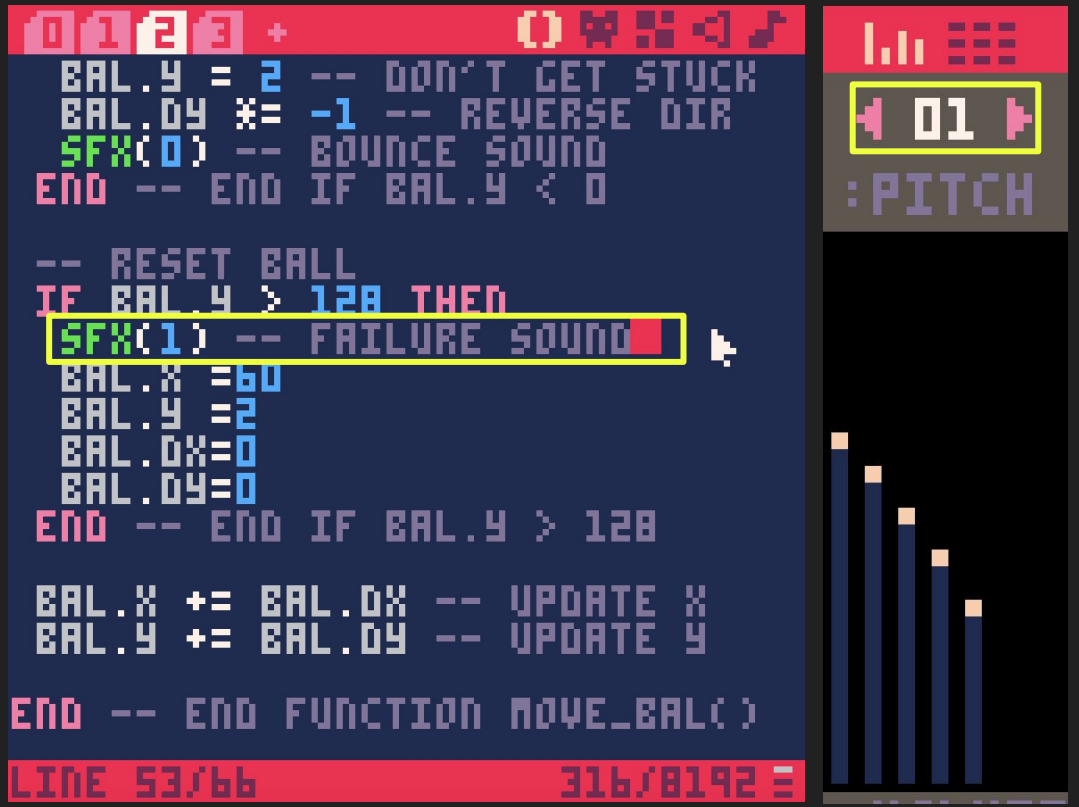
I'll add SFX(0) anywhere the ball bounces



Adding Sounds

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

I'll add SFX(1) in the if/then block for resetting the ball



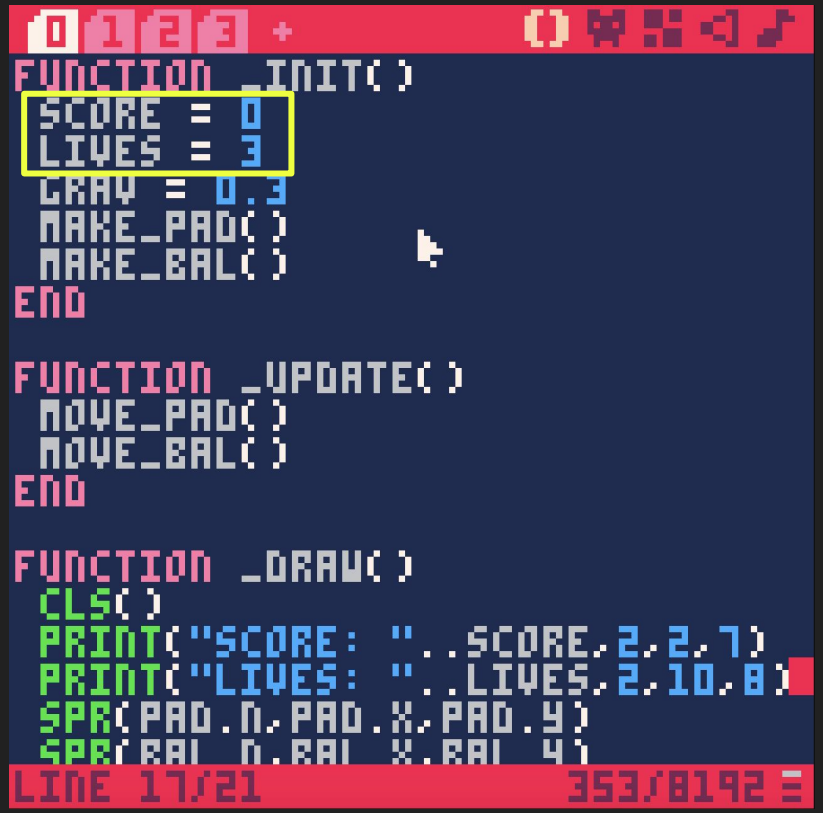
Adding Score, Lives, Game Over

Download Example File: paddleball_physics_11_score_and_lives.p8

Adding Score and Lives

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

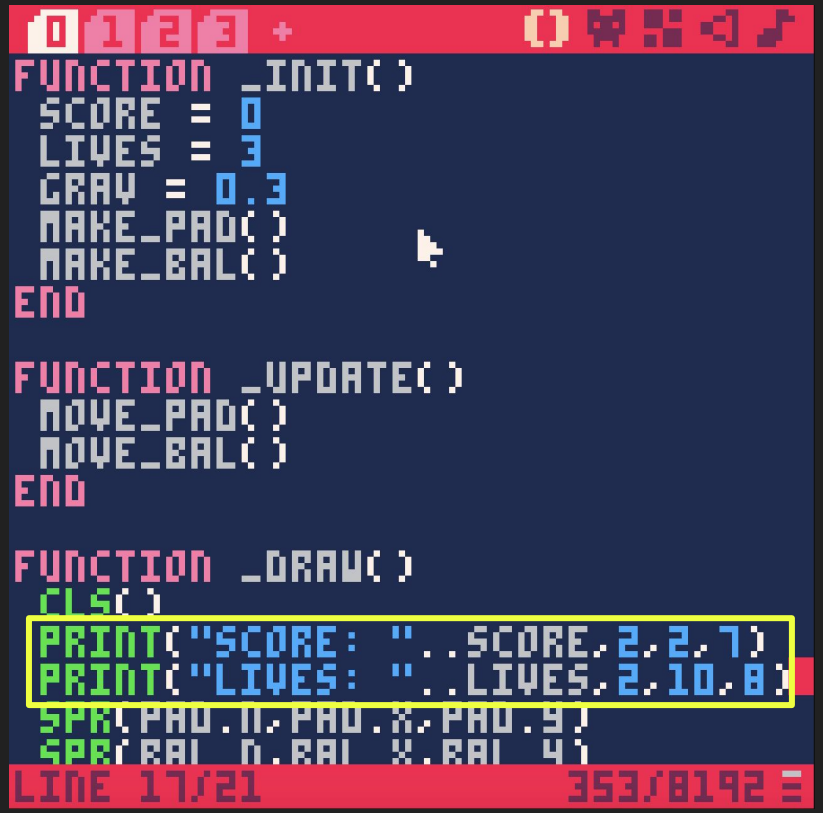
I'll do this in _init()



Adding Score and Lives

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

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

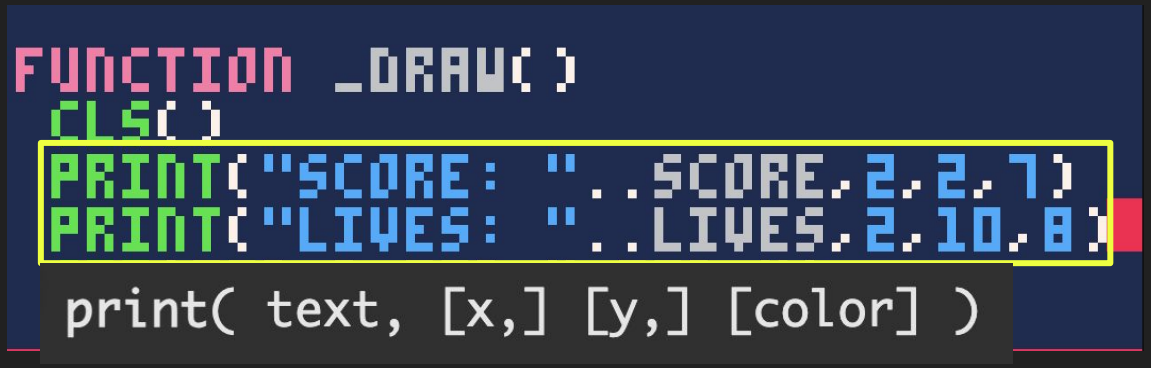


Adding Score and Lives

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

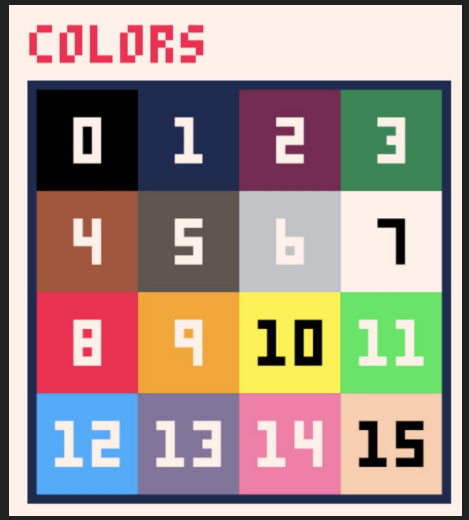
Text values are called "strings" in programming

Combining two "strings" together is called "concatenation"



Adding Score and Lives

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

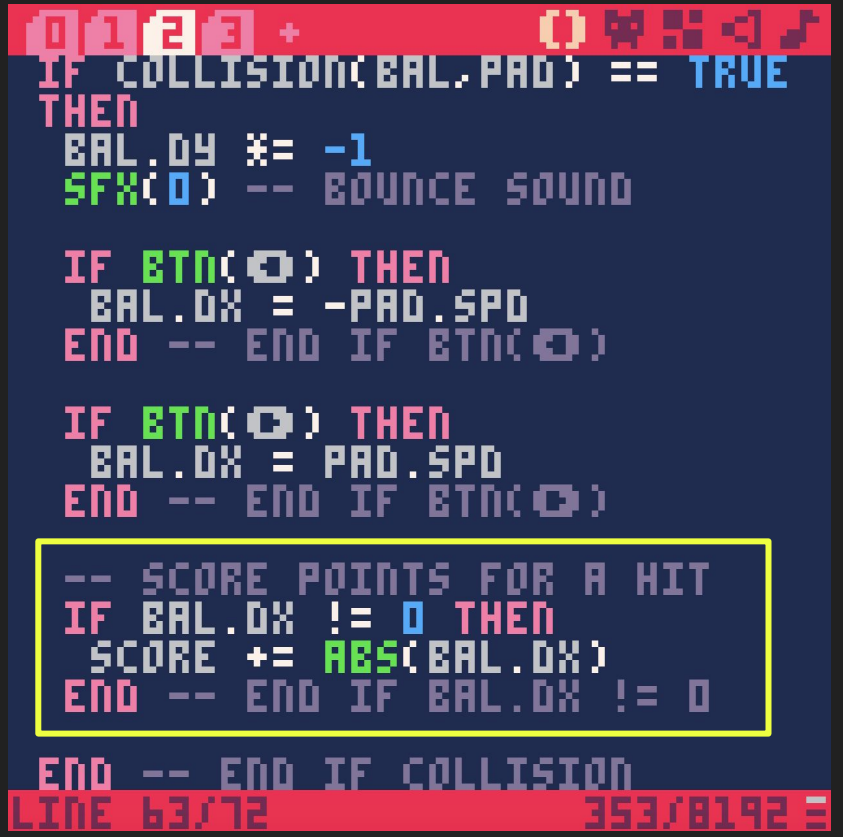



```
function _draw()
 cls()
 print("score: "..score,2,2,7)
 print("lives: "..lives,2,10,8)
```

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

Adding Score

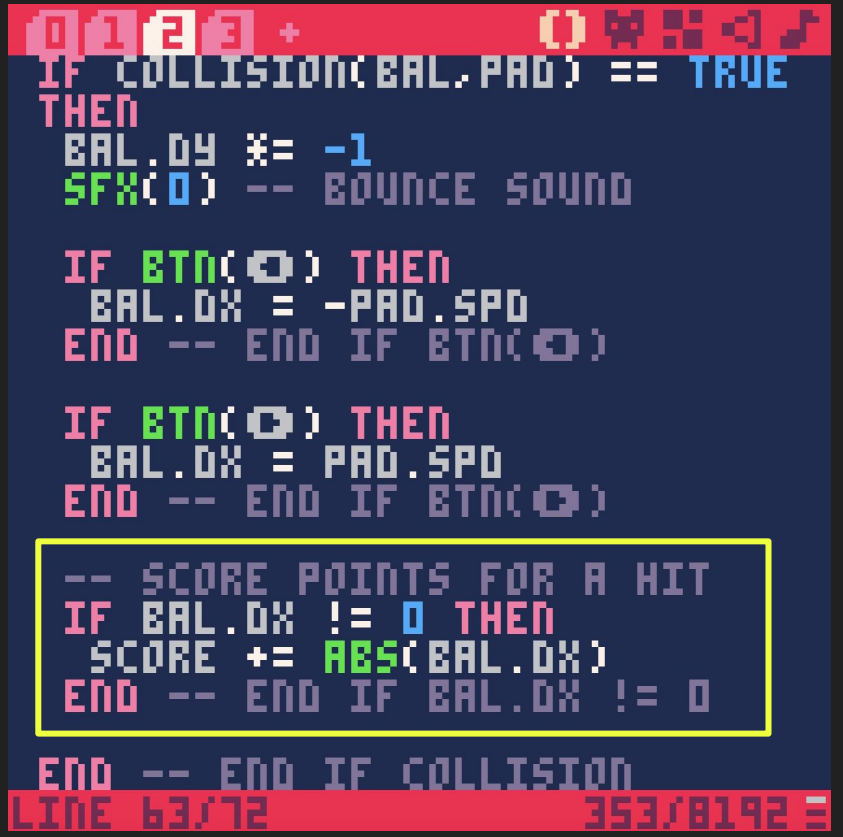
To change the score when the player hits the ball, we'll add to the score variable's value when collision between the ball and paddle is true, in the move_bal() function



Adding Score

But we *don't* want the player to score points *every* time this happens – otherwise they'd score without moving the paddle at the start of the game!

So I've wrapped the line that updates the score inside another if/then block



Adding Score

We know *the ball is moving if its dx value is not 0* (either positive or negative) – we can check if a value is NOT equal to something using **!=**

```
-- SCORE POINTS FOR A HIT
IF BAL.DX != 0 THEN
 SCORE += ABS(BAL.DX)
END -- END IF BAL.DX != 0
```

<, >, <=, >=, ==, != are called “comparison operators”

Adding Score

We could simply add 1, 10, or 100 points when the player hits the ball while it's moving – but I thought it would be cool to make the score related to the speed

```
-- SCORE POINTS FOR A HIT
IF BAL.DX != 0 THEN
 SCORE += ABS(BAL.DX)
END -- END IF BAL.DX != 0
```

Adding Score

If the ball is moving at 3 pixels per frame (bal.dx=3), then we score 3 points – but we *also want to score 3 points if the ball is moving left with a dx value of negative 3*

```
-- SCORE POINTS FOR A HIT
IF BAL.DX != 0 THEN
 SCORE += ABS(BAL.DX)
END -- END IF BAL.DX != 0
```

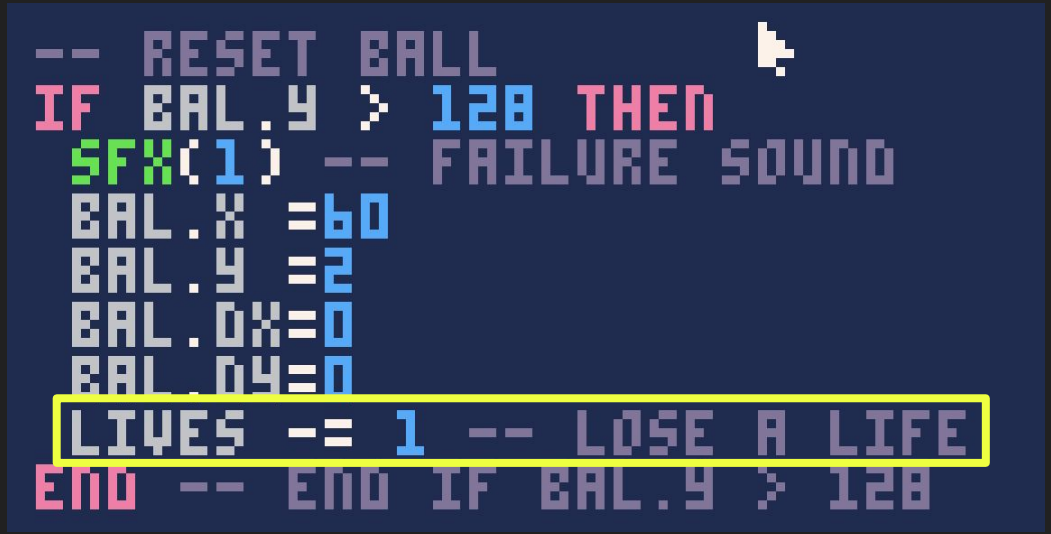
Adding Score

The ABS() function calculates the absolute value of a number (how far it is from zero; turns a negative number positive) – just provide the number inside the parentheses

```
-- SCORE POINTS FOR A HIT
IF BAL.DX != 0 THEN
  SCORE += ABS(BAL.DX)
END -- END IF BAL.DX != 0
```

Adding Lives

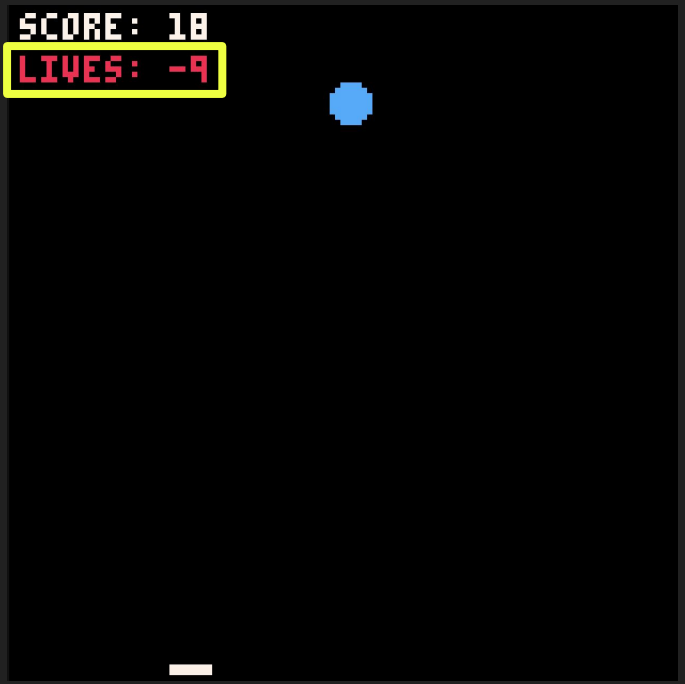
Inside the if/then block for resetting the ball, we can subtract a life when the ball goes off screen



Adding Lives

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

The value turns negative, but we can still keep playing



Adding Lives and Game Over

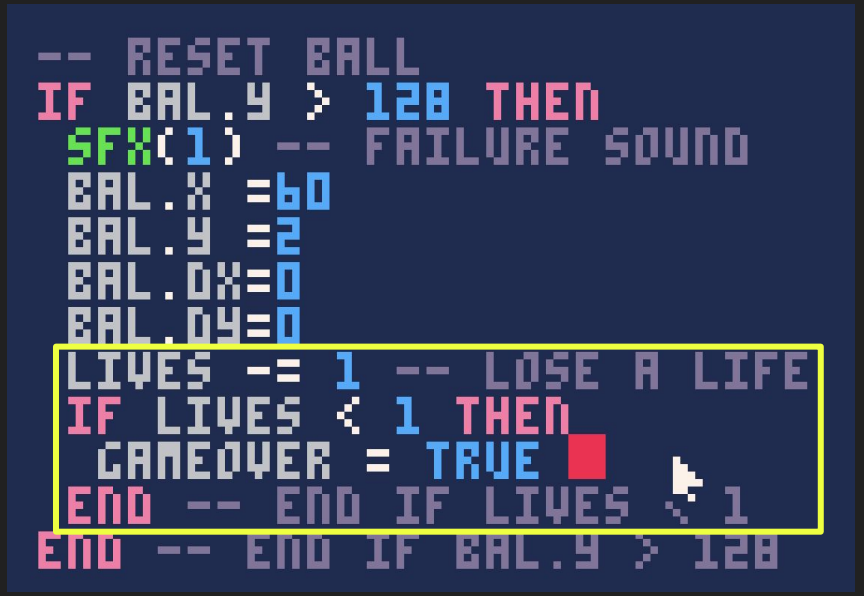
We'll fix this by adding one more variable called gameover

I'll do this in _init()

gameover must be a boolean variable (can be either true or false)

Adding Lives and Game Over

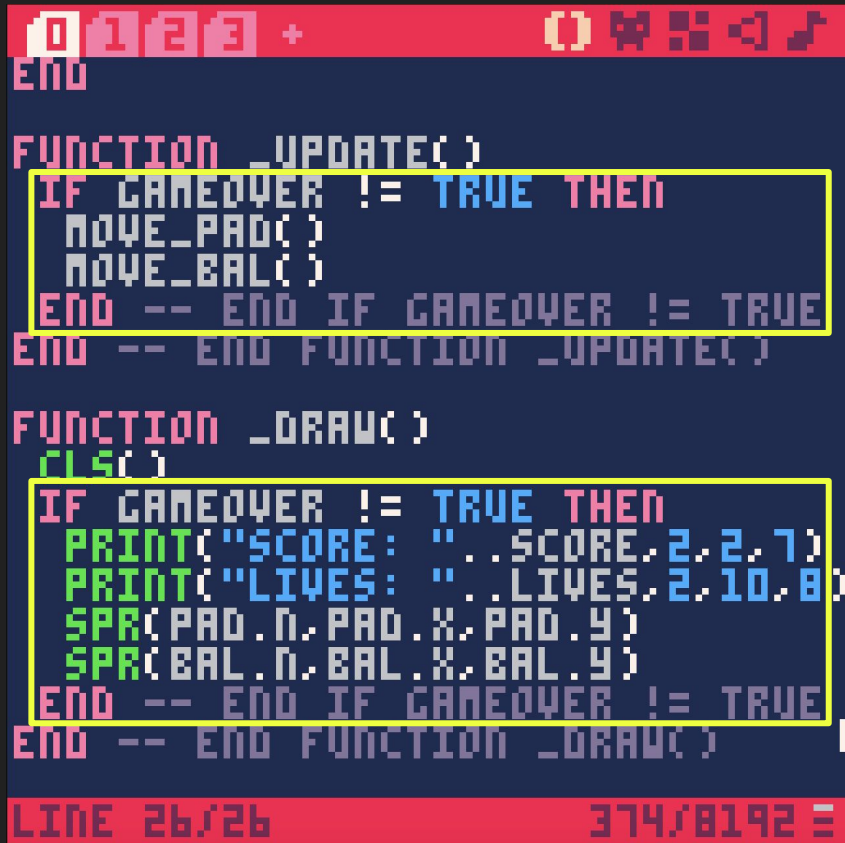
When we lose a life, we'll check if our lives have run out; if so, we'll turn gameover from false to true



Adding a Game Over Screen

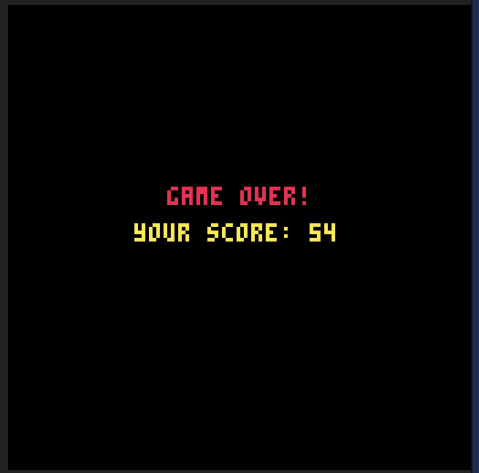
Next, we'll want to stop running the game if gameover is true

We can wrap our code in _update() and _draw() that runs and displays the game inside an if/then statement checking that gameover is NOT true (!=)



Adding a Game Over Screen

Finally, we'll instead display a game over screen if gameover is true




```
FUNCTION _DRAW()
 CLS()
 IF GAMEOVER != TRUE THEN
  PRINT("SCORE: "..SCORE,2,2,7)
  PRINT("LIVES: "..LIVES,2,10,8)
  SPR(PAD.N,PAD.X,PAD.Y)
  SPR(BAL.N,BAL.X,BAL.Y)
 ELSE
  -- DRAW GAMEOVER SCREEN
  PRINT("GAME OVER!",44,50,8)
  PRINT("YOUR SCORE: "..SCORE,
   35,60,10)
 END -- END IF GAMEOVER != TRUE
END -- END FUNCTION _DRAW()
```

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

Adding a Game Over Screen

If you really wanted to add detail, you could move the text one pixel over for each digit in the score, using several if/then statements

But I have a more fun idea if we're going to add a special condition for different score values

```
FUNCTION _DRAW()
 CLS()
 IF GAMEOVER != TRUE THEN
  PRINT("SCORE: "..SCORE,2,2,7)
  PRINT("LIVES: "..LIVES,2,10,8)
  SPR(PAD.N,PAD.X,PAD.Y)
  SPR(BAL.N,BAL.X,BAL.Y)
 ELSE
  -- DRAW GAMEOVER SCREEN
  PRINT("GAME OVER!",44,50,8)
  PRINT("YOUR SCORE: "..SCORE,
   35,60,10)
 END -- END IF GAMEOVER != TRUE
END -- END FUNCTION _DRAW()
```

Adding a Game Over Screen


```
IF GAMEOVER != TRUE THEN
 PRINT("SCORE: "..SCORE,2,2,7)
 PRINT("LIVES: "..LIVES,2,10,8)
 SPR(PAD.N,PAD.X,PAD.Y)
 SPR(BAL.N,BAL.X,BAL.Y)
ELSE

 -- DRAW GAMEOVER SCREEN
 PRINT("GAME OVER!",44,50,8)
 PRINT("YOUR SCORE: "..SCORE,
  35,60,10)

 IF SCORE == 0 THEN
  PRINT("YOU SUCK!",44,70,8)
 END -- END IF SCORE == 0

END -- END IF GAMEOVER != TRUE
```

Learning Resources

PICO-8 Resources

- PICO-8 Home
- Official Resources
- Cheat Sheet
- Forum
- Games

My GitHub Repository, helloworld

Code Examples

1. paddleball_physics_01_variables.p8
2. paddleball_physics_02_input.p8
3. paddleball_physics_03_constraining_movement.p8
4. paddleball_physics_04_custom_functions.p8
5. paddleball_physics_05_ball.p8
6. paddleball_physics_06_object_collision.p8
7. paddleball_physics_07_reverse_ball_direction.p8
8. paddleball_physics_08_horizontal_ball_movement.p8
9. paddleball_physics_09_constrain_and_reset_ball.p8
10. paddleball_physics_10_finetuning_and_troubleshooting.p8
11. paddleball_physics_11_score_and_lives.p8

Download all paddleball w/physics examples:_helloworld_01a_paddleball_physics.zip

PICO-8 Function Reference

PICO-8 Wiki – Functions Reference

- _init() Define starting conditions for your game
- _update() For input, calculations, and updating variable values
- _draw() For displaying text and graphics

- cls() Clears the screen
- spr() Draws a sprite to the screen
- print() Prints text to the screen
- btn() Checks whether a button is being held down
- btnp() Checks whether a button was pressed
- sfx() Plays a sound effect

</lesson>

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