

Making a Top-Down Adventure Game in PICO-8

Complete Walkthrough

by Matthew DiMatteo

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

as seen in GAM-120: Intro to Game Logic



Contents

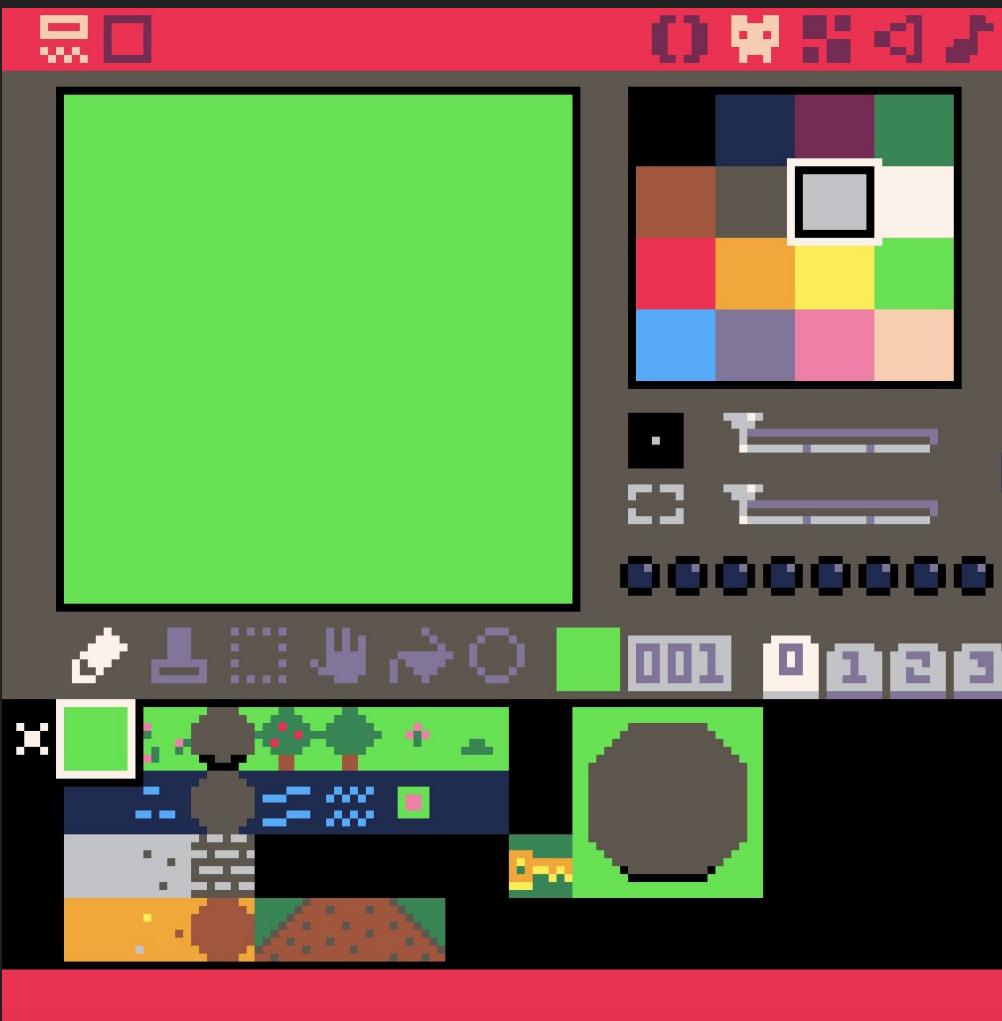
- 4) Drawing Player & Map Sprites
- 8) Using the Map Editor
- 17) Types of Movement
- 21) Tile-Based Movement
- 32) Collision with Map Tiles
- 44) Using Sprite Flags for Collision
- 52) Collecting Items
- 79) Lock & Key Systems

Contents

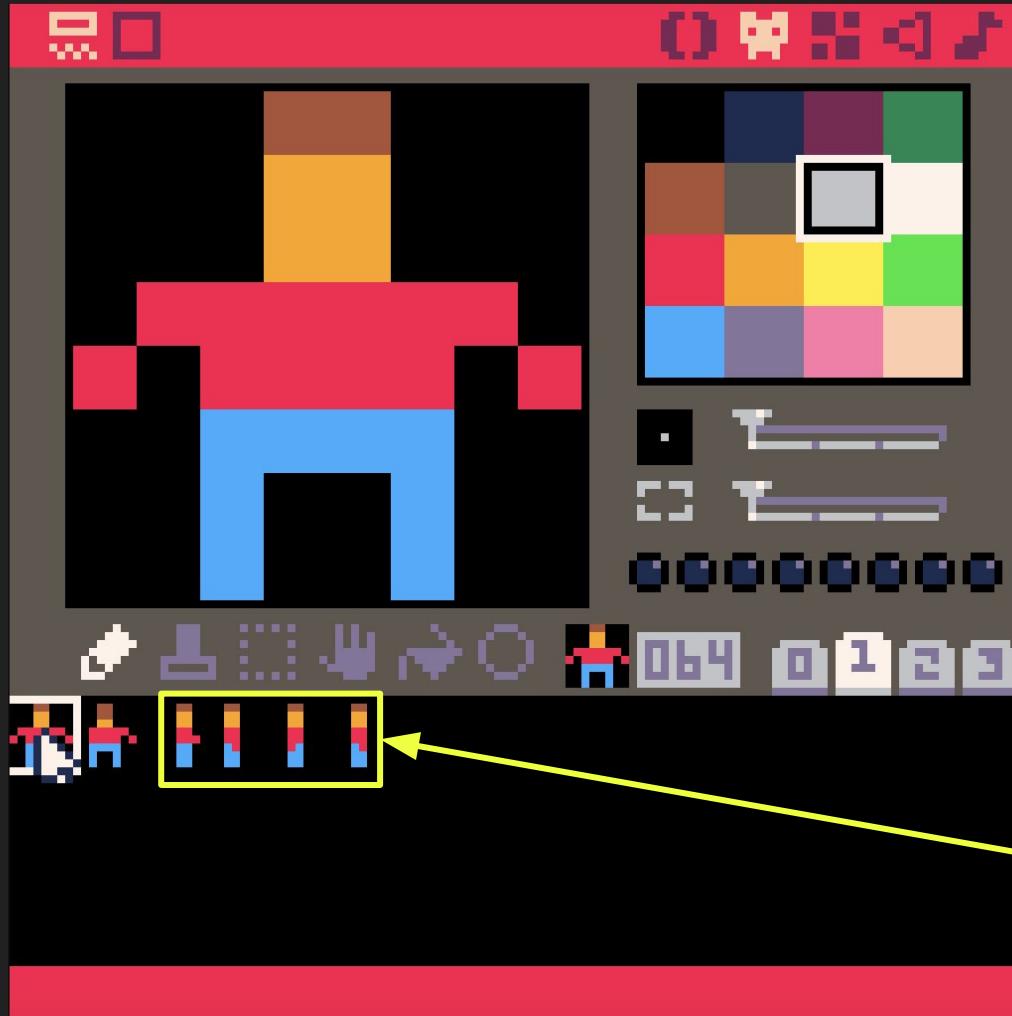
- 86) Positioning the Camera (Center on Player)
- 95) Positioning the Camera (Screen-Based)
- 100) Warping
- 108) Pixel-Based Movement
- 127) Pixel-Based Map Collision
- 155) Learning Resources

Drawing Player & Map Sprites

Download Assets File: [topdown_00_assets_only](#)

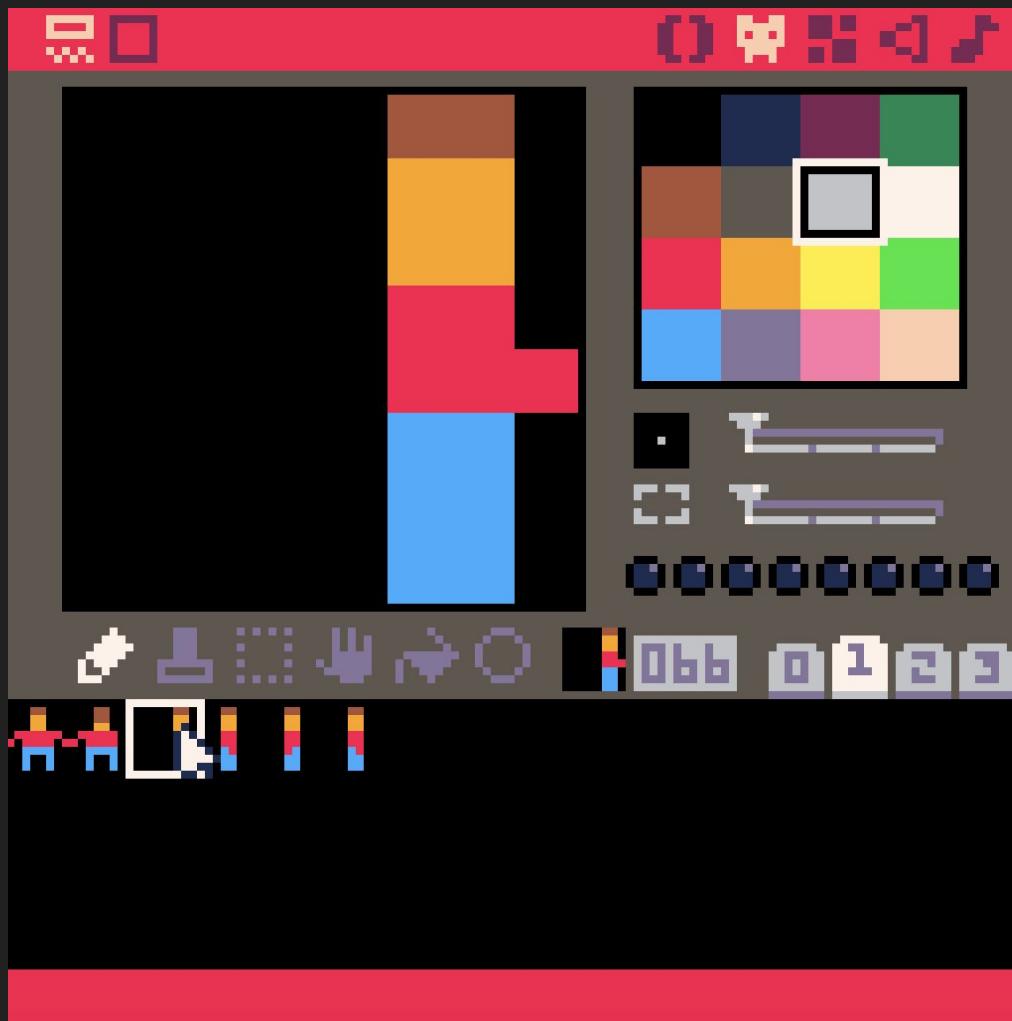


I've drawn a
bunch of sprites
for my map tiles



I've drawn **player sprites** on a separate tab

It's a good idea to *draw frames for an animation on consecutive sprite sheet slots*

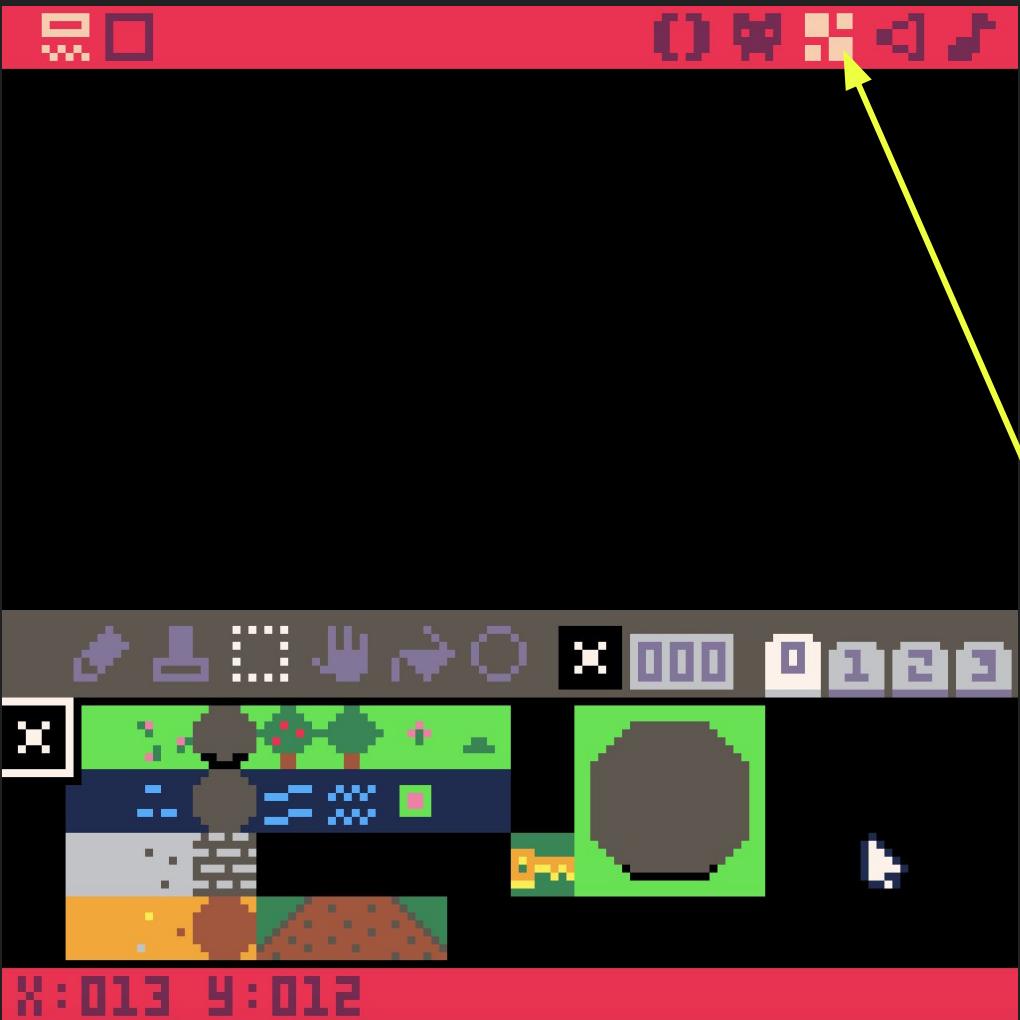


If a sprite will face both left and right (and is symmetrical), *you only need to draw it facing in one direction*

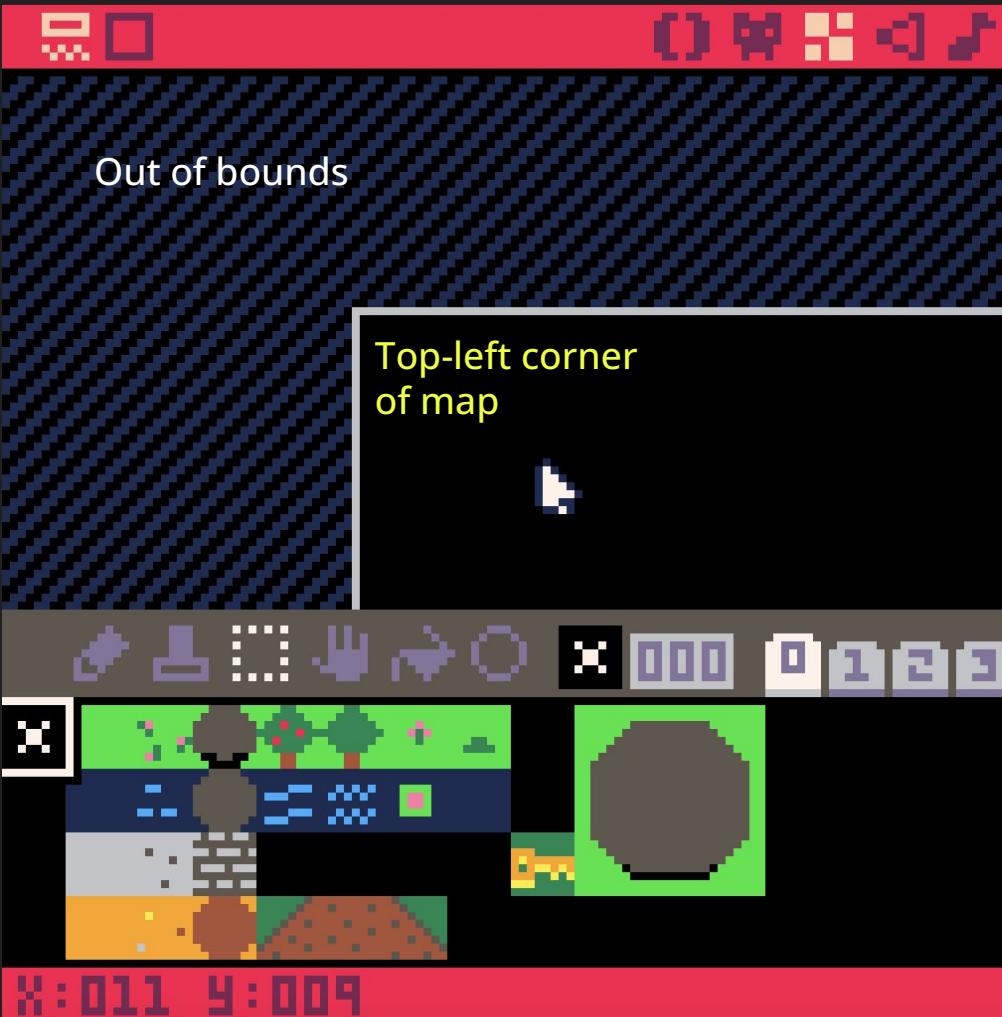
You can use code to flip the sprite on either axis

Using the Map Editor

Download Assets File: [topdown_00_assets_only](#)



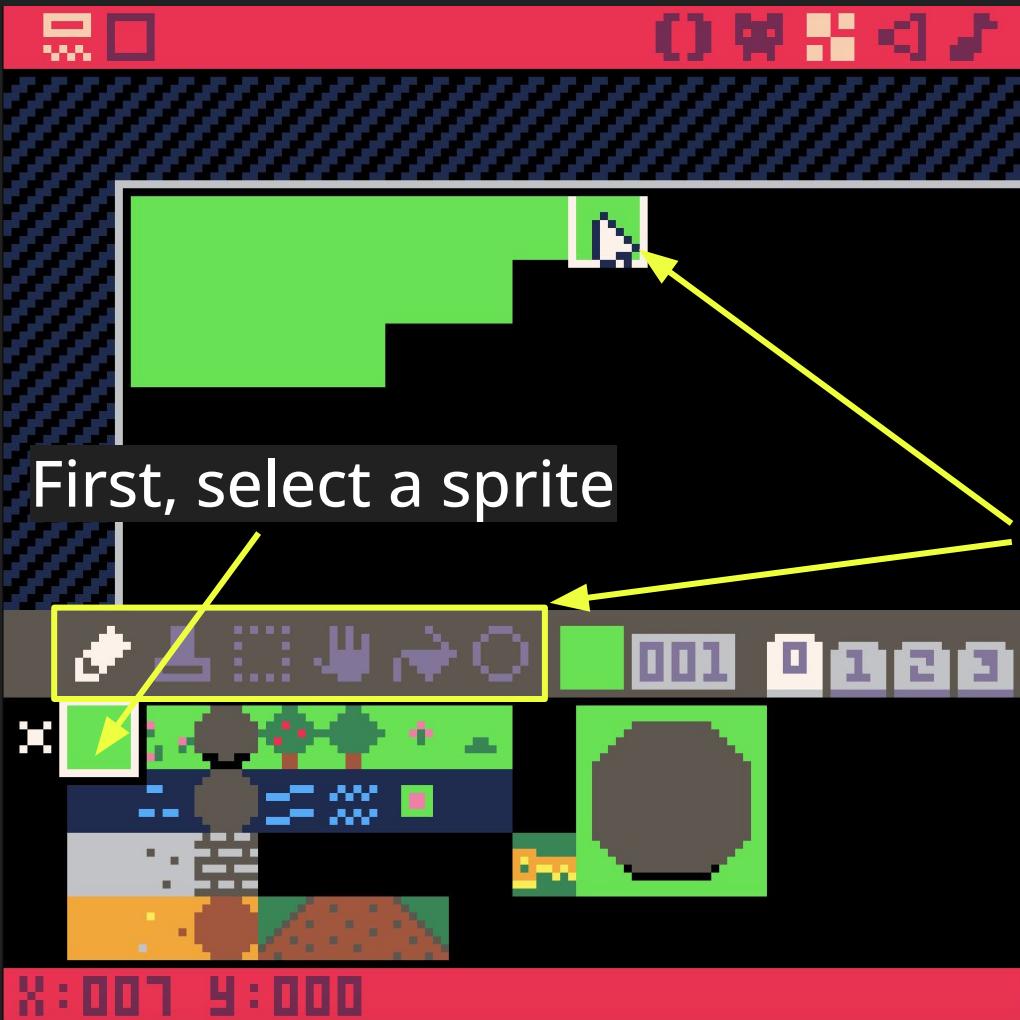
You can use your sprites to draw map tiles in the map editor – *navigate to the map editor by selecting the third option in the top right navigation*



The map may not be apparent at first because it is blank and zoomed in

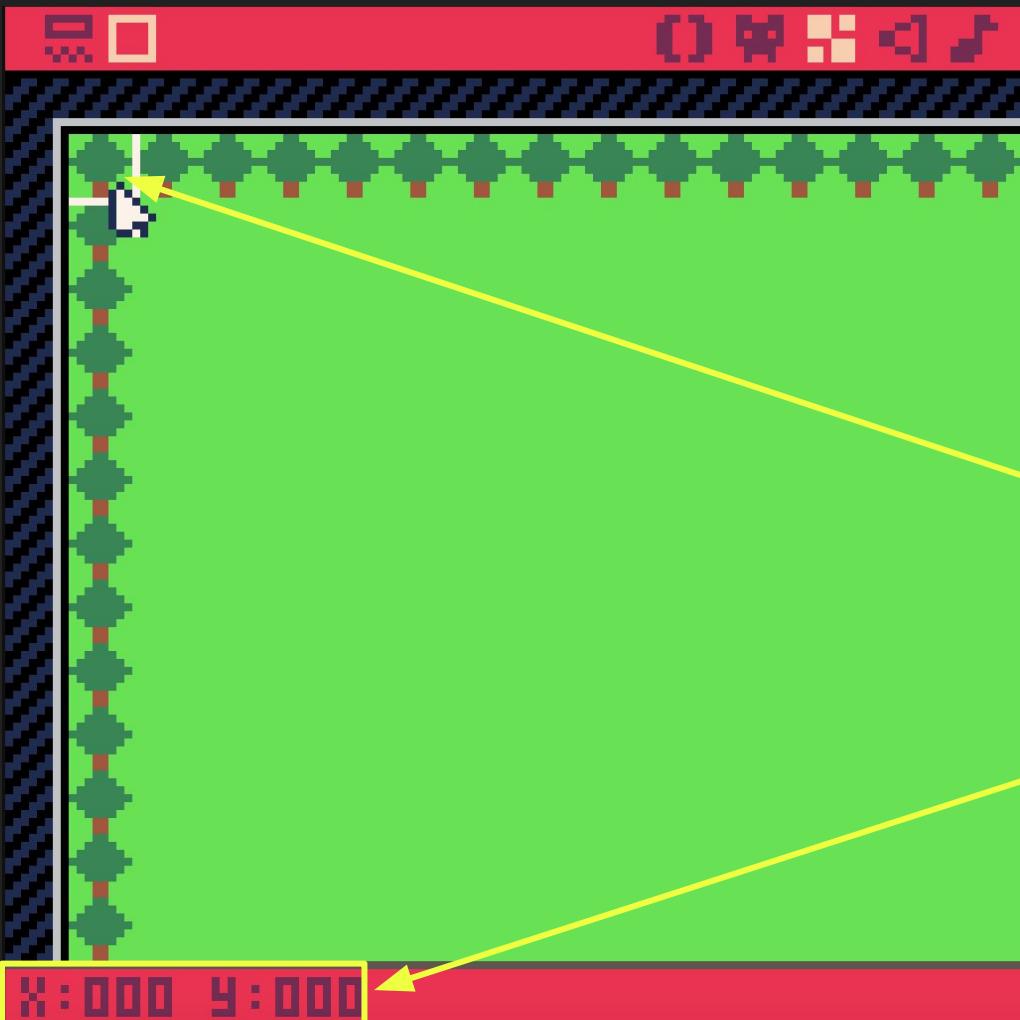
Scroll down to zoom out

(And scroll up to zoom back in)



First, select a sprite

Then use the drawing tools (similar to those in the sprite editor) to paint with sprites in the map editor



Each map screen (as it will display in the game) is **16x16** tiles

But the top-left origin starts at **0,0**

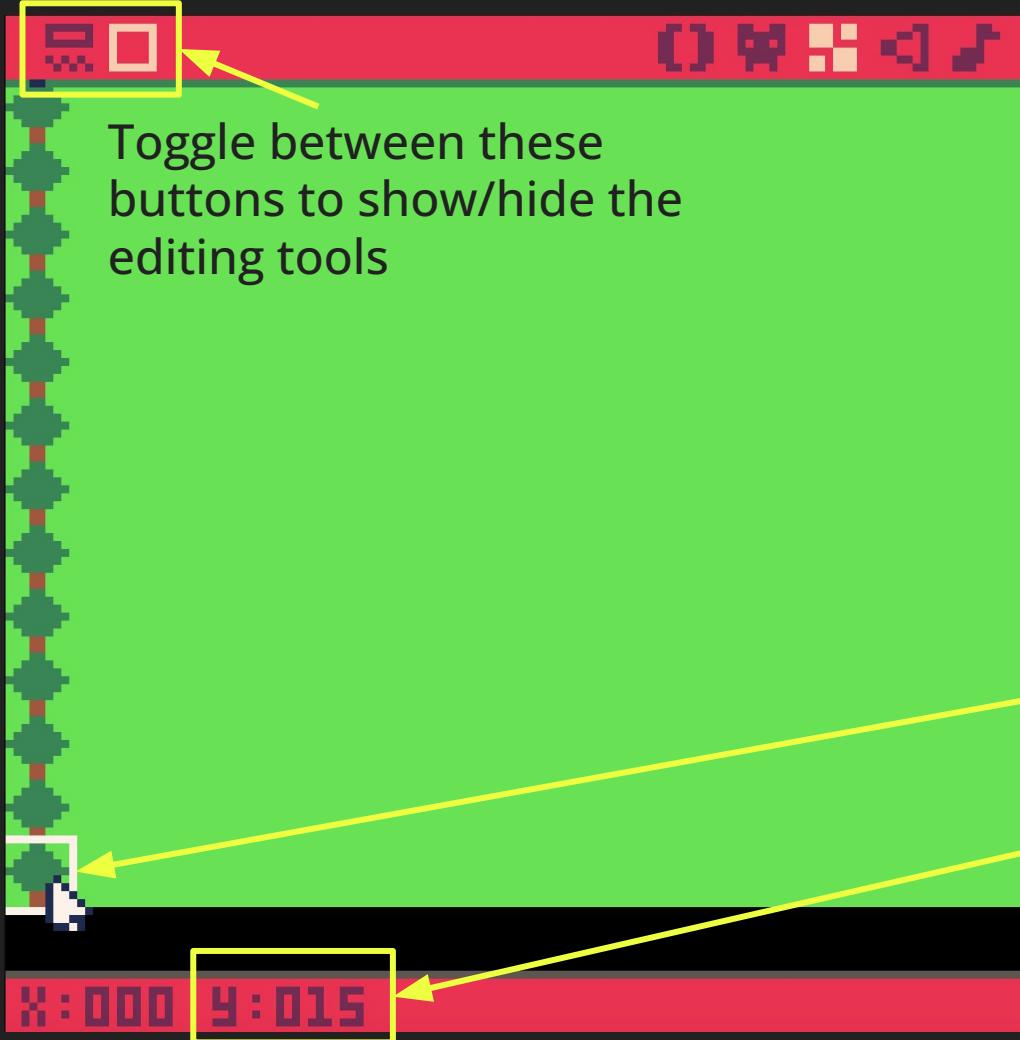
You can see the x,y coordinates of a selected tile in the bottom left of the map editor



Each map screen (as it will display in the game) is **16x16** tiles

But the top-left origin starts at 0,0

So the last tile on the first screen will have a value of 15



Toggle between these buttons to show/hide the editing tools

Each map screen (as it will display in the game) is **16x16** tiles

But the top-left origin starts at 0,0

So the last tile on the first screen will have a value of 15



I've added a variety of objects to give context to the player's movement



Mouse over a map tile to see its x,y position in the bottom left

This can be helpful for determining the player's starting position

Types of Movement

Types of Movement

- For a top-down game, there are two types of movement you can choose between:
 - **Tile-based movement** (8 pixels at a time)
 - **Pixel-based movement** (a smaller number of pixels at a time)

Tile-Based vs. Pixel-Based Movement

- **Tile-based movement**
 - 8 pixels at a time
 - Pros: Easier to implement
 - Cons: Less nuance in movement
 - Best for: games without an emphasis on real-time reaction (**exploration and collection**)
- **Pixel-based movement**
 - A smaller number of pixels at a time
 - Pros: More nuance in movement
 - Cons: More work for collision detection
 - Best for: games requiring real-time reaction (**action, shooting**)

Types of Movement

- Whichever type of movement you choose, it's a good idea to **make your character and map sprites the same size**
- For tile-based movement, it's a good idea to **make your sprites move the same number of tiles as their size**

Tile-Based Movement

Download Example File: [topdown_01_tilebased_movement_and_collision.p8](#)

```
0 +          0 9 8 < > F
-- RUNS ONCE AT START
FUNCTION _INIT()
END -- END FUNCTION _INIT()

-- LOOPS 30X/SEC
FUNCTION _UPDATE()
END -- END FUNCTION _UPDATE()

-- LOOPS 30X/SEC
FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
MAP() -- DRAW MAP AT 0,0
END -- END FUNCTION _DRAW()
```

Create an empty game loop (the `_init`, `_update`, and `_draw` functions)

Use the [map\(\)](#) function in `_draw()` to show the map

0

+

0 0 0 < >

```
-- RUNS ONCE AT START
FUNCTION _INIT()
END -- END FUNCTION _INIT()

-- LOOPS 30X/SEC
FUNCTION _UPDATE()
END -- END FUNCTION _UPDATE()

-- LOOPS 30X/SEC
FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
MAP() -- DRAW MAP AT 0,0
END -- END FUNCTION _DRAW()
```



Use the [map\(\)](#) function in
[_draw\(\)](#) to show the map



In the map editor, you can see the coordinates for a selected tile in the bottom left

7,3 looks like a good place to start my player

The image shows a Scratch script on the left and its corresponding visual representation on the right. The script starts with a **Player** hat icon, followed by a **Set [PLYR v] to [64]** block. This is followed by a **repeat (10)** control loop. Inside the loop, there is a **Set [X] to [7]** block and a **Set [Y] to [3]** block. The loop ends with an **end** block. A yellow arrow points from the **Set [PLYR v] to [64]** block to the text "The player sprite number was 64". Another yellow arrow points from the **Set [X] to [7]** and **Set [Y] to [3]** blocks to the text "And its x,y coordinates on the map were 7,3". The visual representation on the right shows a green map with a red diamond at position 7,3. A white square highlights the diamond at position 7,3. The bottom of the screen shows the Scratch stage with a character and some blocks.

```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR = {}
PLYR.v=64
-- TILE COORDINATES
PLYR.X=7
PLYR.Y=3
END
```

I'll create a function to make my player on a separate tab

The player **sprite number** was **64**

And its **x,y** coordinates on the map were **7,3**

```
-- TILE COORDINATES  
PLYR.X=7  
PLYR.Y=3
```

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



However, when I plug these x,y coordinates into the spr() function, *my player does not appear at the expected position*

This is because the map editor uses **tile** values, but the `spr()` function uses **pixel** values

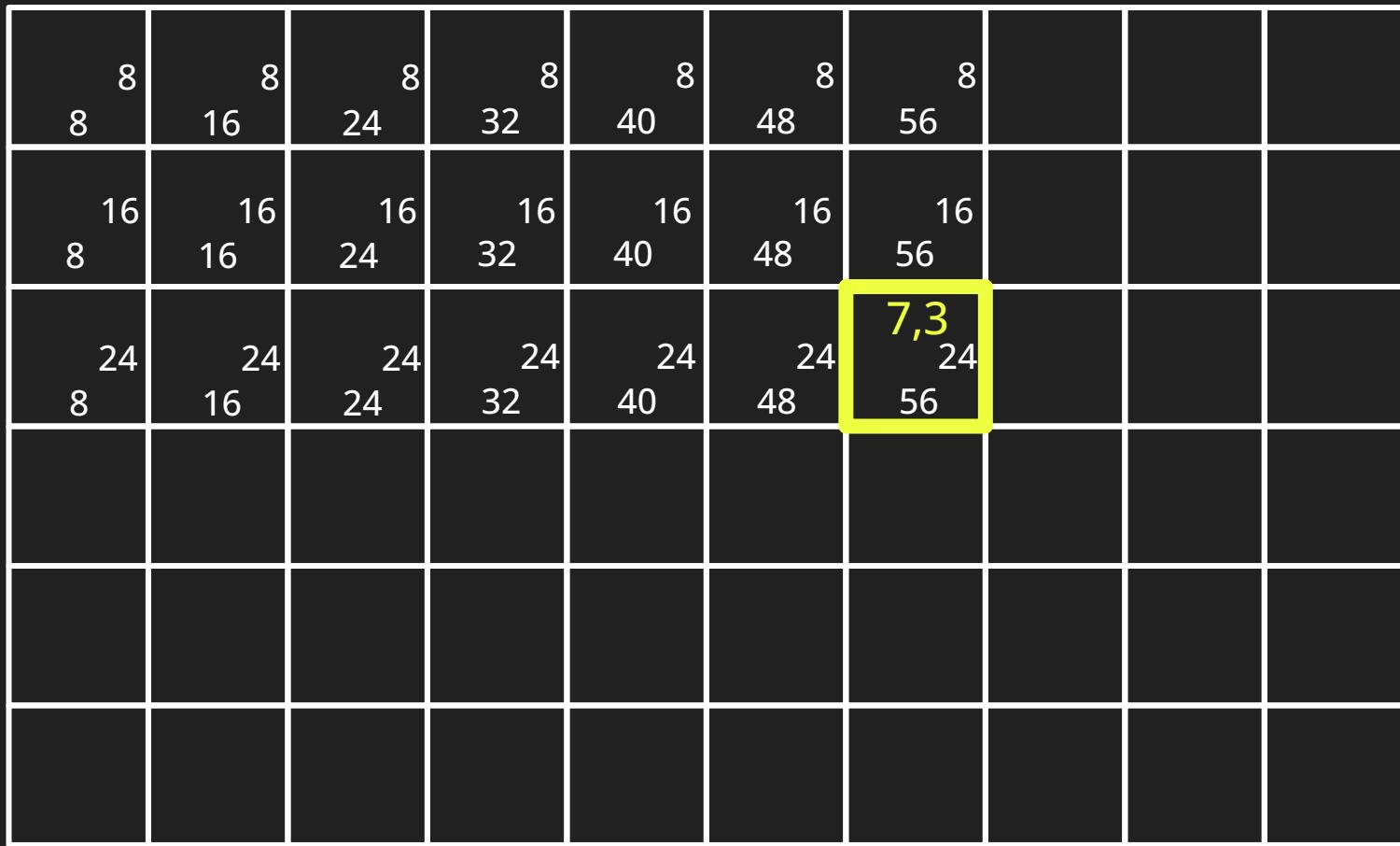
Each tile is 8x8 pixels, so *we need to convert the tile value into a pixel value*

Tile Values vs. Pixel Values

- The map uses tile values
- But the spr() function uses pixel values
- Each map tile is 8x8 pixels square
- So we need to *convert from tiles to pixels*
- We can multiply tile values by 8 to get the pixel value

7 tiles * 8 pixels each = 56 pixels

3 tiles * 8 pixels each = 24 pixels



0 1 2 +



```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR = {}
PLYR.n=64

-- TILE COORDINATES
PLYR.x=1
PLYR.y=3
END
```

For *tile based movement*, leave the player's x,y coordinates as tile values in your make_plyr() function, and *do the conversion directly in the spr() function*, multiplying x and y by 8 to provide a pixel value

```
FUNCTION _DRAW()
CLS()
MAP()
SPR(PLYR.n, PLYR.x*8, PLYR.y*8)
END
```

Moving the player by 1 tile will move them by 8 pixels

002 +

FUNCTION MOVE_PLYR()

IF BTNP(0) THEN
PLYR.X -= 1

END

IF BTNP(1) THEN
PLYR.X += 1

END

IF BTNP(2) THEN
PLYR.Y -= 1

END

IF BTNP(3) THEN
PLYR.Y += 1

END

END

LINE 19/20

003 +

-- TILE COORDINATES

PLYR.X=7

PLYR.Y=3

FUNCTION _DRAW()

CLS()

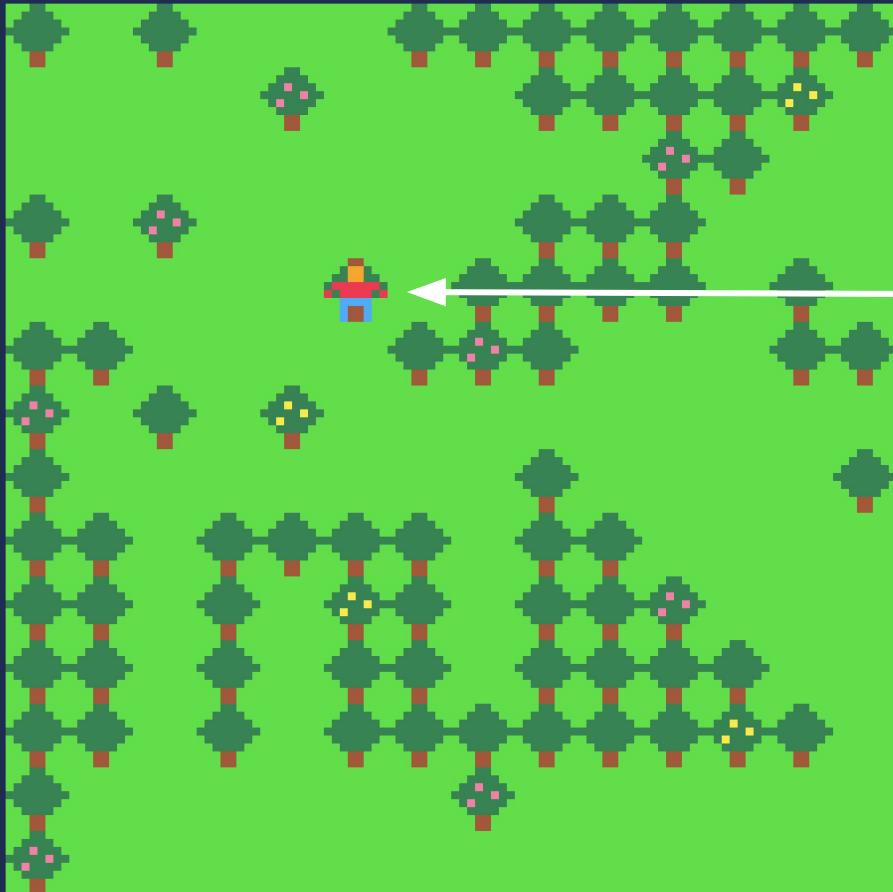
MAP()

SPR(PLYR.n, PLYR.X*8, PLYR.Y*8)

END

Moving the player by 1 tile will move them by 8 pixels

86/8192



Of course, we haven't implemented collision yet, so our player will still go through tiles that look like they should be solid

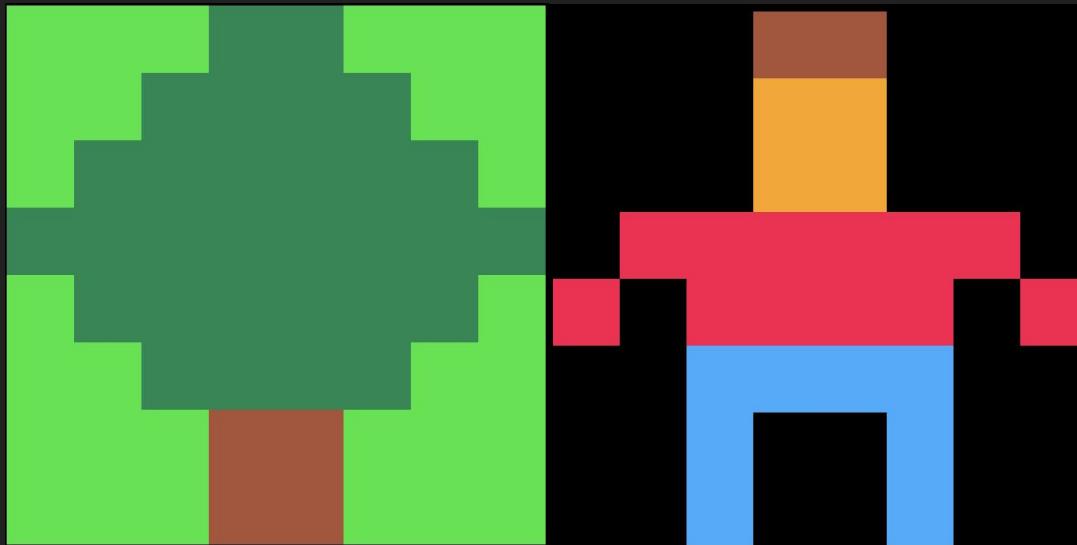
Implementing Collision with Map Tiles

Download Example File: [topdown_01_tilebased_movement_and_collision.p8](#)

Implementing Collision with Map Tiles

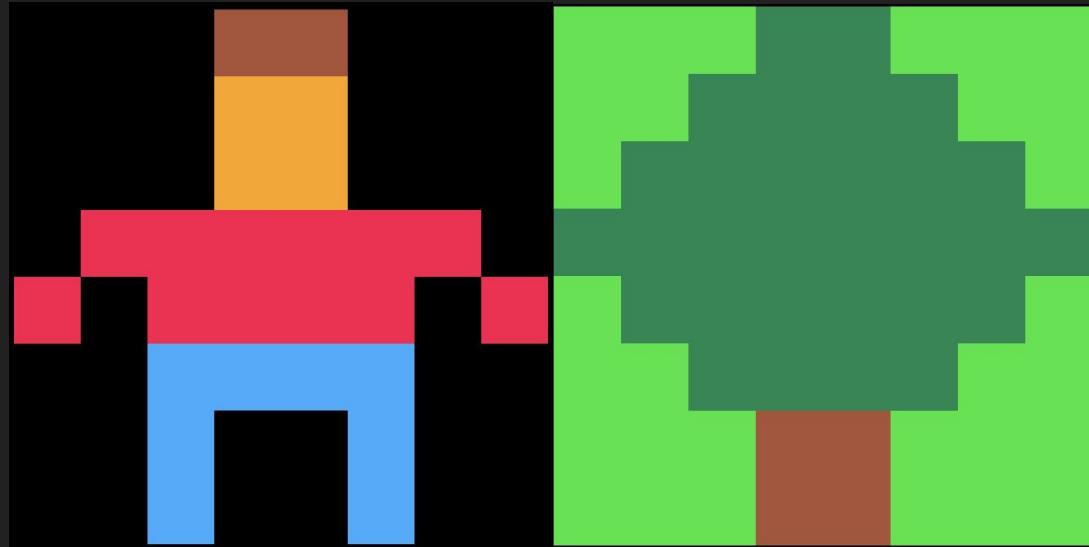
- PICO-8 has a function called mget() that takes an **x,y location** and finds the **sprite number** at that location
- So if we know that sprite number **5** is a tree, and trees should not be walked through, all we need to do is calculate the x,y position of the tree tile

plyr.x - 1



- To calculate the location of a tree tile to the left of the player, we can express the tile's x coordinate as **plyr.x - 1**
- The tree has the same y coordinate as the player: **plyr.y**

`plyr.x + 1`



- To calculate the location of a tree tile to the right of the player, we can express the tile's x coordinate as `plyr.x + 1`
- The tree has the same y coordinate as the player: `plyr.y`

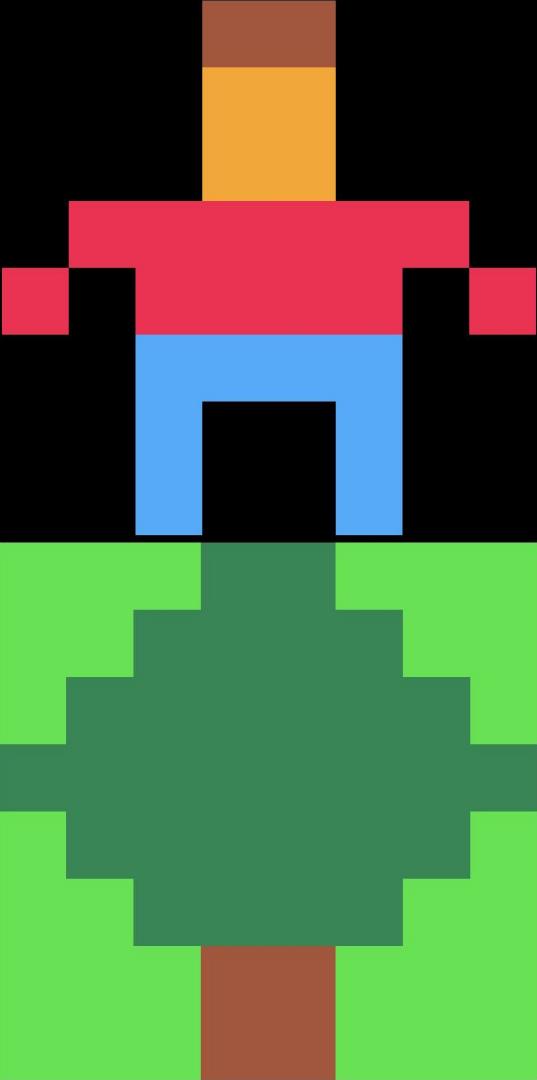
`plyr.y - 1`



To calculate the location of a tree tile above the player, we can express the tile's y coordinate as `plyr.y - 1`

The tree has the same x coordinate as the player: `plyr.x`

`plyr.y + 1`



To calculate the location of a tree tile below the player, we can express the tile's y coordinate as `plyr.y + 1`

The tree has the same x coordinate as the player: `plyr.x`

```
IF BTNP(0) THEN  
--PLYR.X -= 1  
TX = PLYR.X - 1  
TY = PLYR.Y  
END
```

```
IF BTNP(0) THEN  
--PLYR.X += 1  
TX = PLYR.X + 1  
TY = PLYR.Y  
END
```

We can use the variables **tx** and **ty** to track the player's target position based on which arrow key is being pressed, offsetting by 1 tile

Don't forget to declare tx and ty in `_init()` - I just set them equal to the player's x and y position initially

```
IF BTnR(0) THEN  
    --PLYR.Y -= 1  
    TX = PLYR.X  
    TY = PLYR.Y - 1  
END
```

```
IF BTnR(0) THEN  
    --PLYR.Y += 1  
    TX = PLYR.X  
    TY = PLYR.Y + 1  
END
```

We can use the variables **tx** and **ty** to track the player's target position based on which arrow key is being pressed, offsetting by 1 tile

Don't forget to declare tx and ty in `_init()` - I just set them equal to the player's x and y position initially

```
IF BTNP(0) THEN  
--PLYR.Y -= 1  
TX = PLYR.X  
TY = PLYR.Y - 1  
END
```

```
IF BTNP(0) THEN  
--PLYR.Y += 1  
TX = PLYR.X  
TY = PLYR.Y + 1  
END
```

Once we've calculated **tx** and **ty**, we can plug them into the mget() function to *get the sprite number of the tile the player is trying to move to* (trees are 5 in my example)

TILE_SPRITE_NUMBER = **MGET(TX, TY)**

```
IF BTNP(0) THEN
--PLYR.Y += 1
TX = PLYR.X
TY = PLYR.Y + 1
END

TILE_SPRITE_NUMBER=GET(TX, TY)
TREE=5

-- MOVE TO TARGET LOCATION IF
-- TILE IS NOT A TREE
IF TILE_SPRITE_NUMBER != TREE
THEN
    PLYR.X = TX
    PLYR.Y = TY
END -- END IF
```

Then once we know the sprite number, we can *move the player to the target location (tx, ty) only if the sprite number does not belong to a tree*

```
TILE_SPRITE_NUMBER=GET(TX, TY)
```

```
TREE=5  
TREE2=6  
TREE3=7  
ROCK=3  
ROCK2=51  
WALL=35
```

But what about all our other solid tiles?
This would require a lot of **OR** statements

... fortunately, there's a better way!

```
-- MOVE TO TARGET LOCATION IF  
-- TILE IS NOT A TREE  
IF TILE_SPRITE_NUMBER != TREE  
THEN  
    PLYR.X = TX  
    PLYR.Y = TY  
END -- END IF
```

PICO-8 has a feature in the Sprite Editor called “**flags**” which you can use to **label** or **categorize** tiles

*Each flag has a **corresponding number** which you can track using code*



Using Sprite Flags for Collision

Download Example File: [topdown_01_tilebased_movement_and_collision.p8](#)



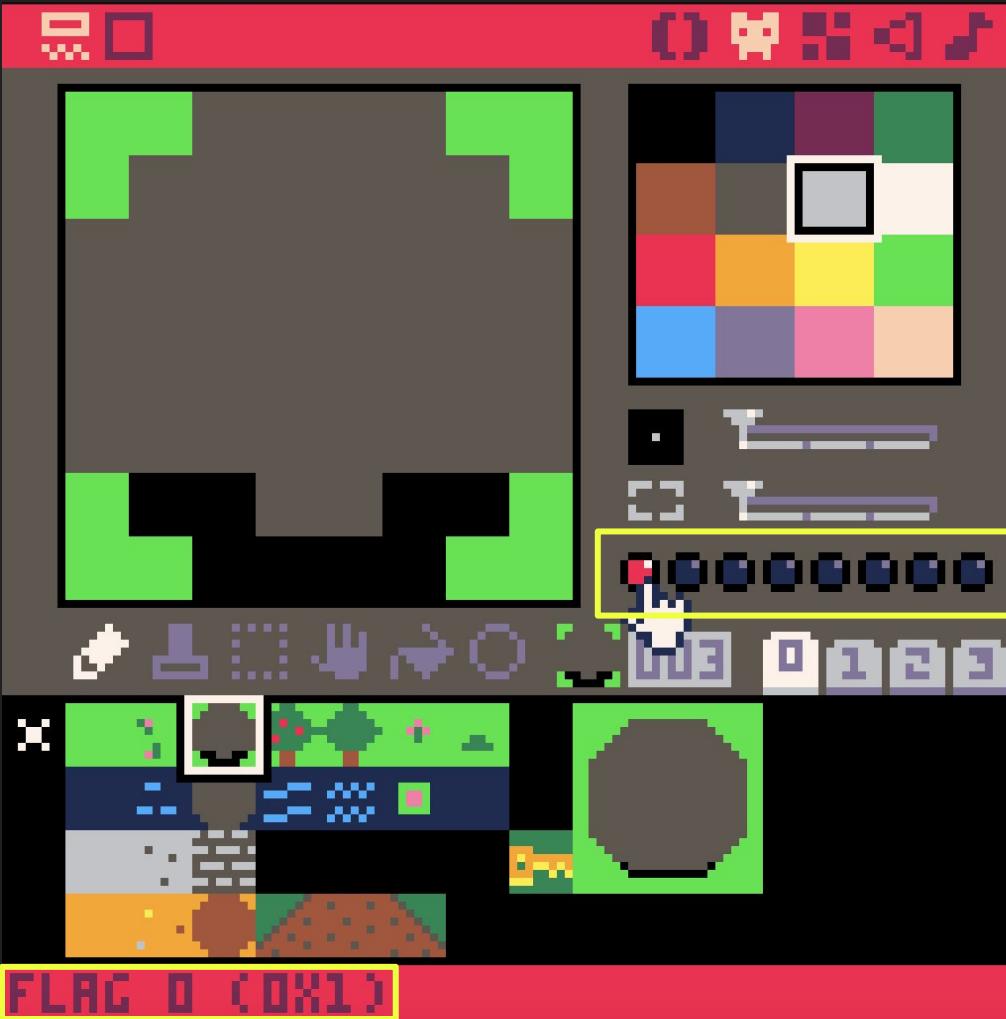
Navigate to the **sprite editor** (not the map editor)

Click on a sprite - you'll see eight dots here - these are called sprite "**flags**" which are like labels

Click on a dot to turn that flag on



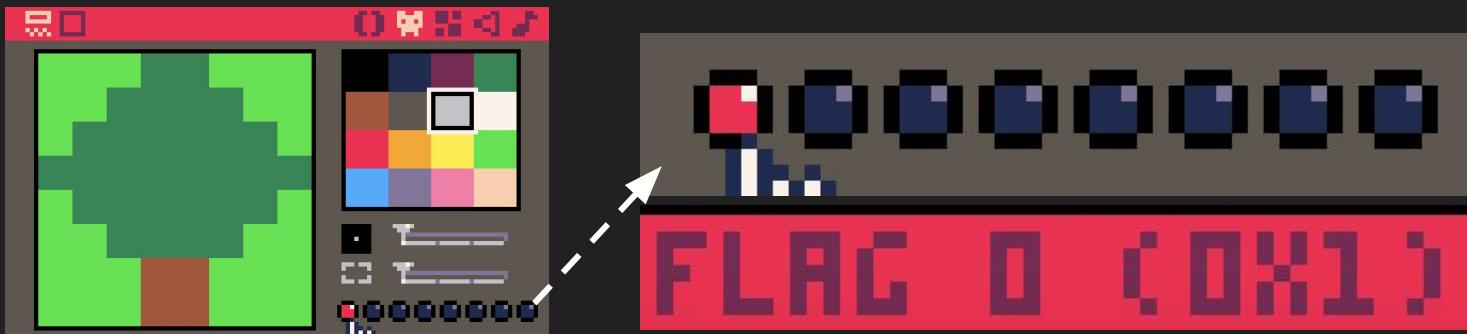
Each **flag** has a corresponding **number**, starting with **0**, which you can see in the **bottom left corner**



Turn on the **same flag** for all map tiles that you want to be **solid** (*that the player cannot move through*)

Using Sprite Flags for Collision

- Sprite flags are like labels
- Use the *same flag* to denote *solid tiles*
- You can then use code to check whether the player is trying to move to a solid tile



Implementing Collision with Map Tiles

- PICO-8 has another function called fget() that takes a **sprite number** and **finds whether a flag is turned on**
- We can use this in conjunction with mget() to implement our map collision

Implementing Collision with Map Tiles

Calculate tx,ty

mget(tx,ty) = sprite_number

fget(sprite_number,flag_number) = true or false

012 +

0 0 0 0 0 0

```
-- SPRITE NUMBER OF TILE THE  
-- PLAYER IS TRYING TO GO TO  
TILE_SPRITE_NUMBER=MGET(TX,TY)
```

```
-- FOR SHORTER VARIABLE NAME  
TSN=TILE_SPRITE_NUMBER
```

```
-- FLAG NUMBER FOR SOLID TILES  
SOLID = 0
```

```
-- RETURNS TRUE OR FALSE  
IS_SOLID=FGET(TSN,SOLID)
```

```
-- MOVE TO TARGET LOCATION IF  
-- TILE IS NOT A TREE  
IF IS_SOLID == FALSE THEN  
    PLYR.X = TX  
    PLYR.Y = TY  
END -- END IF
```

LINE 42/45

149/8192 E

We plug the calculated tx and ty values into the mget() function, which **returns the sprite number** of the tile

Then we can **plug the tile's sprite number** into the fget() function – *along with the flag number* we used, which was 0 – to return **true or false**



Then we can *only move the player to the target position if is_solid is false*

Collecting Items

Download Example File: [topdown_02_item_collection.p8](#)

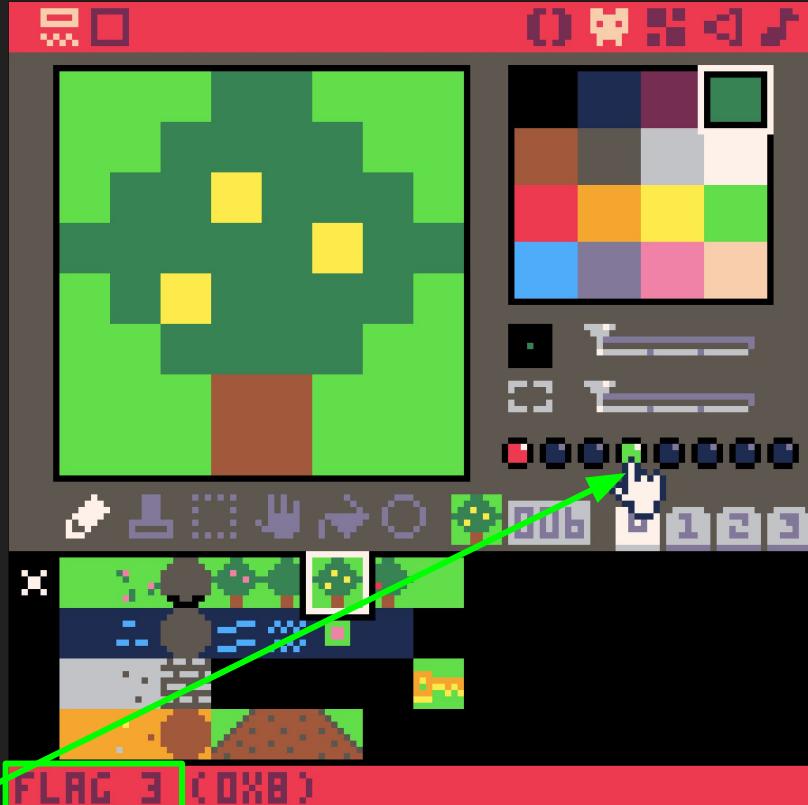
Collecting Items

- We can use sprite flags along with the mget() and fget() functions to **implement item collection as well**



A sprite can have multiple flags turned on at once

I can turn on flag 0 to treat this tree as a solid tile, but **also** turn on flag 1 to indicate that it has a **collectible** property



I'll turn on flags **2** and **3** to indicate that the player will *collect greater amounts of fruit* from these trees

012 +

0 0 0 < >

-- COLLECTIBLE FRUIT FLAGS

FRUIT1 = 1
FRUIT2 = 2
FRUIT3 = 3

IS_FRUIT1 = FGET(TSN, FRUIT1)
IS_FRUIT2 = FGET(TSN, FRUIT2)
IS_FRUIT3 = FGET(TSN, FRUIT3)

-- PRESS X NEAR TREE TO
-- COLLECT FRUIT

IF BTNP(8) THEN
 IF IS_FRUIT1 THEN
 FRUIT += 1
 ELSEIF IS_FRUIT2 THEN
 FRUIT += 2
 ELSEIF IS_FRUIT3 THEN
 FRUIT += 3
 END -- END IF IS_FRUIT1/2/3

LINE 62/66

220/8192 E

012 +

0 0 0 < >

-- SPRITE NUMBER OF TILE THE
-- PLAYER IS TRYING TO GO TO
TILE_SPRITE_NUMBER=FGET(TX, TY)

-- FOR SHORTER VARIABLE NAME
TSN=TILE_SPRITE_NUMBER

-- FLAG NUMBER FOR SOLID TILES
SOLID = 0

-- RETURNS TRUE OR FALSE
IS_SOLID=FGET(TSN, SOLID)

-- MOVE TO TARGET LOCATION IF
-- TILE IS NOT A TREE

IF IS_SOLID == FALSE THEN
 PLYR.X = TX
 PLYR.Y = TY
END -- END IF

LINE 42/45

149/8192 E



Then, similarly to
checking solid tiles, I can
check whether a tile is a
collectible tile

012 +



-- COLLECTIBLE FRUIT FLAGS

```
FRUIT1 = 1  
FRUIT2 = 2  
FRUIT3 = 3
```

```
IS_FRUIT1 = FGET(TSN, FRUIT1)  
IS_FRUIT2 = FGET(TSN, FRUIT2)  
IS_FRUIT3 = FGET(TSN, FRUIT3)
```

-- PRESS X NEAR TREE TO
-- COLLECT FRUIT

```
IF BTNP(8) THEN  
  IF IS_FRUIT1 THEN  
    FRUIT += 1  
  ELSEIF IS_FRUIT2 THEN  
    FRUIT += 2  
  ELSEIF IS_FRUIT3 THEN  
    FRUIT += 3  
END -- END IF IS_FRUIT1/2/3
```

LINE 62/66 220/8192

Set variables for each fruit tree's **flag number**



Plug in the **sprite number** of **the tile** the player's trying to collect from (stored as TSN here) as well as the **flag** for the fruit sprite

Then let the player interact with a tile by pressing X

If the tile is a fruit tree, increase the fruit count

I added a variable called **fruit** in `_init()` and set **fruit = 0**

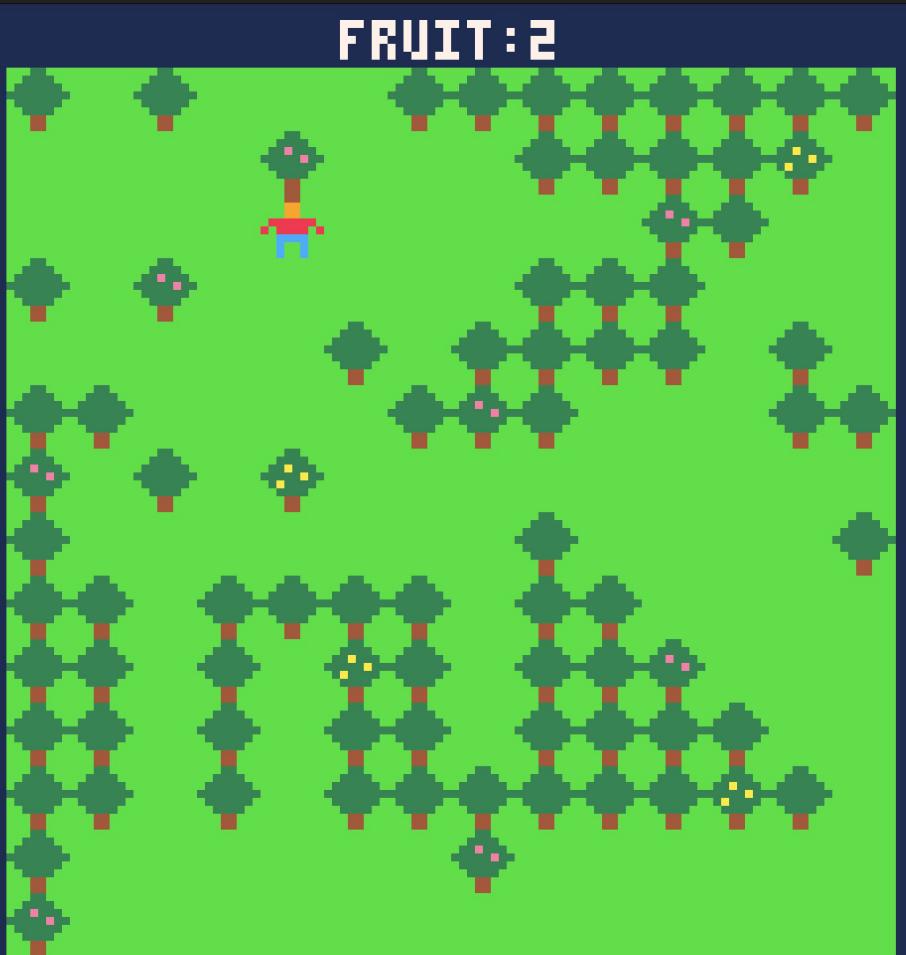
```
FUNCTION _INIT()
MAKE_PLYR() -- TAB 1
TX = PLYR.X
TY = PLYR.Y

-- INVENTORY
FRUIT = 0
END
```

```
PRINT("FRUIT:"..FRUIT, 50, 2, 7)
```



I declared **fruit** in `_init()` and added a `print()` statement in `_draw()` to **output the value** through the HUD



I can press X next to a fruit tree to increase my fruit count, *but it doesn't always work* ...

Why could this be?

```
FUNCTION _DRAW()
CLS()
RAP()
SPR(PLYR.n, PLYR.X#8, PLYR.Y#8)

-- DRAW A BOX OVER THE
-- TARGET POSITION
RECT(TX#8, TY#8, 8+TX#8, 8+TY#8)

PRINT("FRUIT:"..FRUIT, 50, 2, 7)
END
```

I multiplied tx and ty by 8 to convert from tiles to pixels, and added 8 to the second pair of coordinates to draw the other sides of the box

To better understand where our code is checking for a fruit tile, let's **draw a box around the target position**



If I approach from below by pressing up until I reach the tree, I will fail to collect fruit because my target position is one tile short



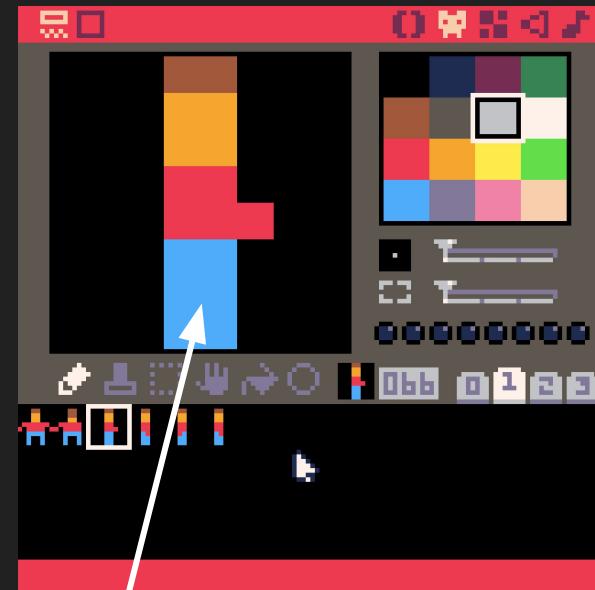
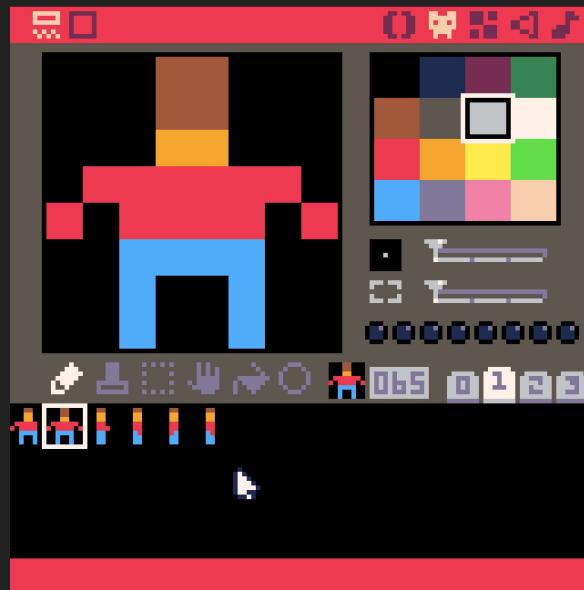
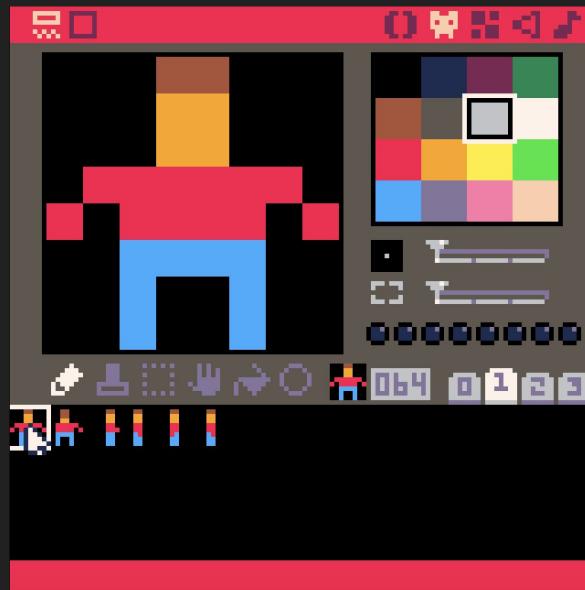
But if I press up once more, I can collect it

Collecting Items

- The problem is that **tx** and **ty** (which I am using to find the location of fruit tiles) are *only being updated when the player is moving*
- If the player **stops**, we **still need to know the location of the next tile in the direction they are facing**, in order to collect fruit



It would also help us solve this problem if we could see which direction the player is facing



I've drawn player sprites for facing
different directions

*(I did not draw one facing left,
because I can **flip** a sprite using code)*

0 1 2 +

0 4 5 < >

```
-- MAKE PLAYER
FUNCTION MAKE_PLAYER()
PLYR = {}
PLYR.N=64

-- TILE COORDINATES
PLYR.X=7 -- = 56 PX
PLYR.Y=3 -- = 24 PX

-- DIRECTION
PLYR.DIR = "DOWN"

-- FLIP SPRITE?
PLYR.FLIP = FALSE

END
```



I've added a variable to keep track of which direction the player is facing, which is initially set to "DOWN"

You could also use faces of the clock (3,6,9,12) if that's easier to keep track of

0 1 2 +

0 1 2 3 4 5

```
-- MAKE PLAYER
FUNCTION MAKE_PLAYER()
PLYR = {}
PLYR.N=64

-- TILE COORDINATES
PLYR.X=7 -- = 56 PX
PLYR.Y=3 -- = 24 PX

-- DIRECTION
PLYR.DIR = "DOWN"

-- FLIP SPRITE?
PLYR.FLIP = FALSE
```

END



LINE 16/16

358/8192 E

And the **plyr.flip** variable will be a boolean (either true or false) for whether to flip the sprite (horizontally) or not

012 +

0 9 8 < >

```
FUNCTION _UPDATE()
MOVE_PLAYER() -- TAB 2
END
```

```
FUNCTION _DRAW()
CLS()
MAP()
```

```
-- DRAW PLAYER SPRITE
SPR(PLYR, 0, PLYR.X*8, PLYR.Y*8,
1, 1, PLYR.FLIP)
```

```
-- DRAW A BOX OVER THE <
-- TARGET POSITION
RECT(TX*8, TY*8, B+TX*8, B+TY*8)

PRINT("FRUIT:"..FRUIT, 50, 2, 1)
END
```

LINE 29/30

358/8192 E

You can plug `plyr.flip` (a true or false value) into the `spr()` function to achieve the flipping

You'll have to tell the program when to flip the player sprite in your movement function

012 +

0 9 8 < >

```
FUNCTION _UPDATE()
MOVE_PLAYER() -- TAB 2
END
```

```
FUNCTION _DRAW()
CLS()
MAP()
```

```
-- DRAW PLAYER SPRITE
SPR(PLAYER, 0, PLAYER.X*8, PLAYER.Y*8,
1,1,PLAYER.FLIP)
```

```
-- DRAW A BOX OVER THE
-- TARGET POSITION
```

```
RECT(TX*8, TY*8, 8+TX*8, 8+TY*8)
```

```
PRINT("FRUIT:"..FRUIT, 50, 2, 1)
```

END

LINE 29/30

358/8192 E

Please note that I must specify all spr() function values leading up to the flip

I used the default 1,1 for the number of tiles wide and tall the player sprite is

0 1 2 +

0 1 2 3 4 5

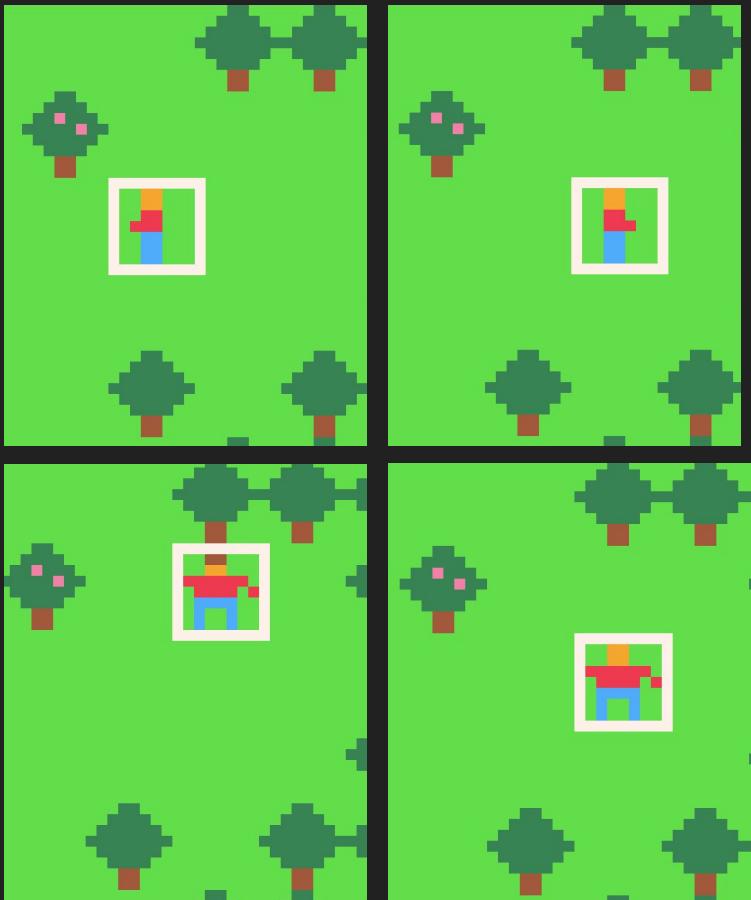
```
IF BTNP(0) THEN
    TX = PLYR.X - 1
    TY = PLYR.Y

    PLYR.DIR = "LEFT"
    PLYR.N = bb
    PLYR.FLIP = TRUE
END

IF BTNP(1) THEN
    TX = PLYR.X + 1
    TY = PLYR.Y

    PLYR.DIR = "RIGHT"
    PLYR.N = bb
    PLYR.FLIP = FALSE
END
```

In my player movement function, I set the **direction**, **sprite number**, and **flip** depending on which arrow key is pressed



```
-- DRAW PLAYER SPRITE
SPR(PLYR.N, PLYR.X#8, PLYR.Y#8,
1,1, PLYR.FLIP)
```

```
IF BTNP(0) THEN
    TX = PLYR.X
    TY = PLYR.Y - 1

    PLYR.DIR = "UP"
    PLYR.N = 65
    PLYR.FLIP = FALSE
END

IF BTNP(0) THEN
    TX = PLYR.X
    TY = PLYR.Y + 1

    PLYR.DIR = "DOWN"
    PLYR.N = 64
    PLYR.FLIP = FALSE
END
```

```
IF BTNP(0) THEN
    TX = PLYR.X
    TY = PLYR.Y - 1

    PLYR.DIR = "UP"
    PLYR.N = 65
    PLYR.FLIP = FALSE
END

IF BTNP(0) THEN
    TX = PLYR.X
    TY = PLYR.Y + 1

    PLYR.DIR = "DOWN"
    PLYR.N = 64
    PLYR.FLIP = FALSE
END
```

Here's the result as it all comes together

012 +

() [] { }

```
-- TILE IN THE DIRECTION THE
-- PLAYER IS FACING
IF PLYR.DIR == "LEFT" THEN
    OBJX = PLYR.X - 1
    OBJY = PLYR.Y
ELSEIF PLYR.DIR == "RIGHT" THEN
    OBJX = PLYR.X + 1
    OBJY = PLYR.Y
ELSEIF PLYR.DIR == "UP" THEN
    OBJX = PLYR.X
    OBJY = PLYR.Y - 1
ELSEIF PLYR.DIR == "DOWN" THEN
    OBJX = PLYR.X
    OBJY = PLYR.Y + 1
END -- END IF PLYR.DIR
```

To determine the location of the fruit I'm trying to collect, I'll add new variables **objx** and **objy** (similar to tx and ty, but tracking a different thing)

The math is identical, but this is done after the key press

```
-- DETERMINE SPRITE NUMBER OF  
-- NEXT TILE IN THE DIRECTION  
-- THE PLAYER IS FACING  
OBJN = MGET(OBJX, OBJY)  
  
-- COLLECTIBLE FRUIT FLAGS  
FRUIT1 = 1  
FRUIT2 = 2  
FRUIT3 = 3  
  
IS_FRUIT1 = FGET(OBJN, FRUIT1)  
IS_FRUIT2 = FGET(OBJN, FRUIT2)  
IS_FRUIT3 = FGET(OBJN, FRUIT3)
```

I can then follow a very similar process to the movement calculation, plugging **objx** and **objy** into the mget() function, then plugging the sprite number (**objn**) into fget()

012 +

0 9 8 < >

-- TOPDOWN TILE-BASED EXAMPLE

FUNCTION _INIT()

MAKE_PLAYER() -- TAB 1

-- PLAYER TARGET LOCATION

TX = PLYR.X

TY = PLYR.Y

-- FRUIT TARGET LOCATION

OBJX = PLYR.X

OBJY = PLYR.Y

-- INVENTORY

FRUIT = 0

END

FUNCTION _UPDATE()

MOVE_PLAYER() -- TAB 2

LINE 37/41

385/8192 E

To illustrate the location being calculated, I'll draw another box to represent where we're checking for fruit tiles

I declare **objx** and **objy** in `_init()`

012 +

0 4 5 < >

FUNCTION _DRAW()

CLS()

MAP()

-- DRAW PLAYER SPRITE

SPR(PLYR.N, PLYR.X*8, PLYR.Y*8,
1.1, PLYR.FLIP)

-- DRAW A BOX OVER THE MOVEMENT

-- TARGET POSITION

RECT(TX*8, TY*8, B+TX*8, B+TY*8)

-- DRAW A BOX OVER THE FRUIT

-- TARGET POSITION

RECT(OBJX*8, OBJY*8,
B+OBJX*8, B+OBJY*8, 14)

PRINT("FRUIT:". .FRUIT, 50, 2, 7)

END

LINE 37/41

385/8192 E

FRUIT:2



This code draws a pink
box at **objx** and **objy**

Collecting Items

- The detection now works perfectly, but you'll notice another flaw: *the player can collect fruit from the same tile infinitely*
- PICO-8 has a function mset() that lets us **change a tile to a different sprite**
- We can use this to **change a fruit tree back to a regular tree** once collected from

```

-- PRESS X NEAR TREE TO
-- COLLECT FRUIT
IF BTNP(8) THEN
  IF IS_FRUIT1 THEN
    FRUIT += 1
    MSET(OBJX,OBJY,5)
  ELSEIF IS_FRUIT2 THEN
    FRUIT += 2
    MSET(OBJX,OBJY,5)
  ELSEIF IS_FRUIT3 THEN
    FRUIT += 3
    MSET(OBJX,OBJY,5)
  END -- END IF IS_FRUIT1/2/3
END -- END IF BTNP(8)

```

Make sure to do this for each type of fruit tree

Plug into mset():

- the x,y coordinates of the sprite you want to change (objx,objy)
- followed by the sprite number of the sprite you want to change it to (5 for regular tree in my example)

FRUIT:0



FRUIT:2



SET(OBJX,OBJY,5)

We can use flags and
map functions to
make **lock** and **key**
systems, too

Lock & Key Systems

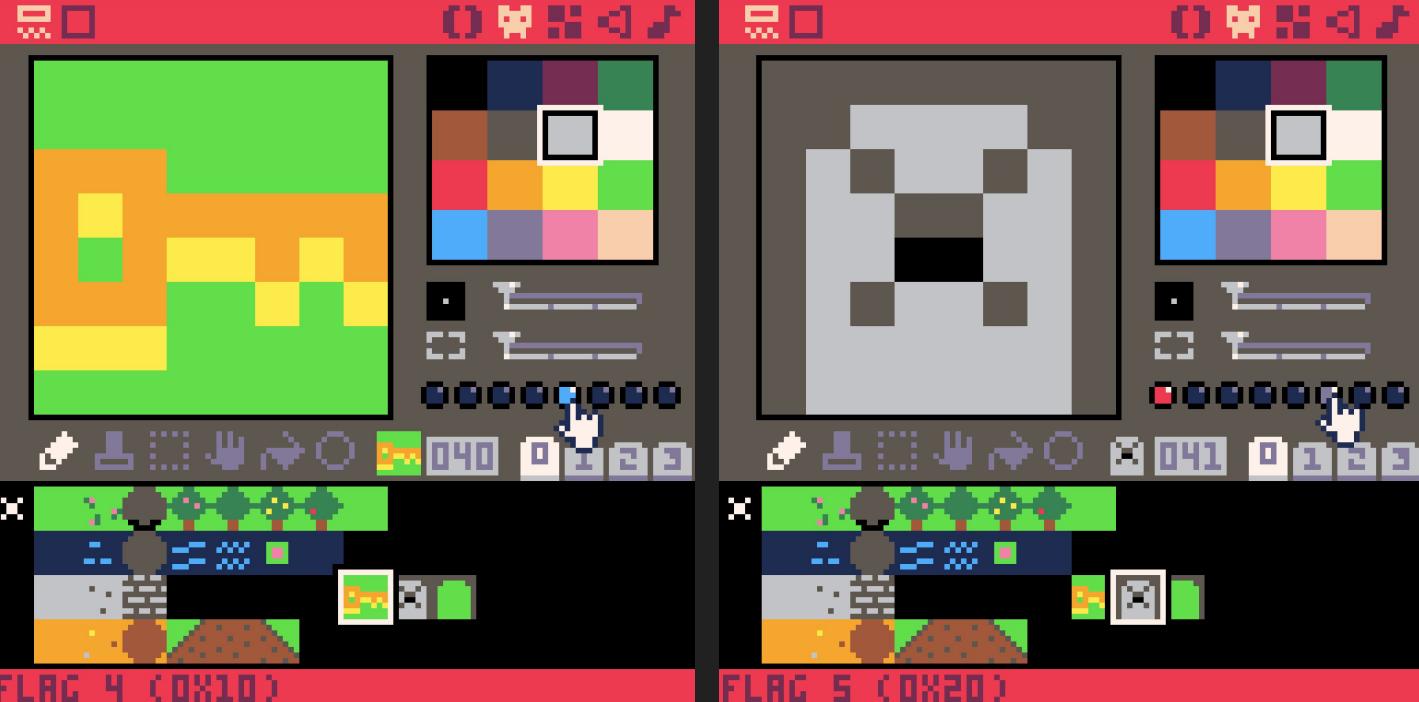
Download Example File: [topdown_03_lock_and_key.p8](#)

```
function _init()
    make_plyr() -- tab 1

    -- target tile coordinates
    -- (where the player is trying
    -- to move to)
    tx = plyr.x
    ty = plyr.y
    |
    -- target object coordinates
    -- (what the player is trying
    -- to collect)
    objx = tx
    objy = ty

    -- show target box
    show=true
    text="▢ = hide target box"

    -- sprite flags
    solid = 0
    fruit1= 1
    fruit2= 2
    fruit3= 3
    key    = 4
    door   = 5
    |
    -- inventory
    fruit=0
    keys =0
    |
end -- end function _init()
```



In my `_init()` function, I've added variables for **key** and **door flags**, as well as **key inventory**

```
function _update()
    move_plyr() -- tab 2
    inspect() -- tab 3
    pickup() -- tab 4
    toggle() -- tab 5
end
```

I've extracted my code for collecting fruit into a new function called **inspect()**, and called the **inspect()** function in **_update()**

```
-- check tiles for collectibles
function inspect()
    -- set objx,objy based on
    -- direction; must be 1 tile
    -- farther in that direction
    -- than player's position
    if plyr.dir=="left" then
        objx=plyr.x-1
        objy=plyr.y
    elseif plyr.dir=="right" then
        objx=plyr.x+1
        objy=plyr.y
    elseif plyr.dir=="up" then
        objx=plyr.x
        objy=plyr.y-1
    elseif plyr.dir=="down" then
        objx=plyr.x
        objy=plyr.y+1
    end -- end if/elseif dir

    -- sprite number of object tile
    objn=mget(objx,objy)

    -- true/false is flag on
    -- for that sprite
    is_fruit1 = fget(objn,fruit1)
    is_fruit2 = fget(objn,fruit2)
    is_fruit3 = fget(objn,fruit3)
    is_door   = fget(objn,door)
```

```
key    = 4  
door   = 5
```

```
-- sprite number of object tile  
objn=mget(objx,objy)  
  
-- true/false is flag on  
-- for that sprite  
is_fruit1 = fget(objn,fruit1)  
is_fruit2 = fget(objn,fruit2)  
is_fruit3 = fget(objn,fruit3)  
is_door   = fget(objn,door)
```

In the **inspect()** function, I'm now checking for **doors** in addition to fruit



And when I press X, I add an **elseif** case to handle what happens if the tile is a door

```
-- press x to inspect tile  
if btpn(x) then  
  
  -- collect fruit  
  if is_fruit1 == true then  
    sfx(1)  
    mset(objx,objy,5)  
    fruit += 1  
  elseif is_fruit2 == true then  
    sfx(1)  
    mset(objx,objy,5)  
    fruit += 2  
  elseif is_fruit3 == true then  
    sfx(2)  
    mset(objx,objy,5)  
    fruit += 3
```

```
-- open door  
  elseif is_door == true then  
    if keys > 0 then  
      mset(objx,objy,42)  
      sfx(3)  
      keys -= 1  
    else  
      sfx(4) -- play error sound  
    end -- end if keys > 0  
  
  end -- end if/else fruit/key  
  
end -- end if btpn(x)
```

If the player presses X while next to a door tile, I *check whether the player has any keys*

```
-- open door
elseif is_door == true then
    if keys > 0 then
        mset(objx,objy,42)
        sfx(3)
        keys == 1
    else
        sfx(4) -- play error sound
    end -- end if keys > 0
end -- end if btnp(x)
```

If so, then I use mset() to get rid of the door tile, turning it to an open gateway (sprite 42)

I also *subtract one from the key count, and play an error sound if the player has no keys*

```
function _update()
    move_plyr() -- tab 2
    inspect() -- tab 3
    pickup() -- tab 4
    toggle() -- tab 5
end
```

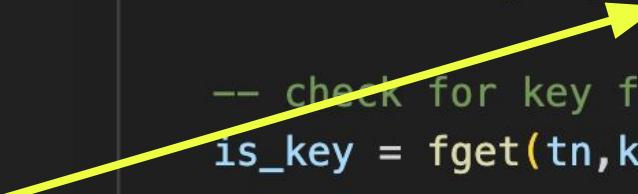
To actually pick up a key, I wrote a separate function called **pickup()** that is *called in _update()*



```
-- pick up items by walking over them
function pickup()
    -- get sprite number of tile
    -- at player's position
    local tn = mget(plyr.x,plyr.y)
    -- check for key flag
    is_key = fget(tn,key)
    -- collect key
    if is_key then
        mset(plyr.x,plyr.y,1)
        keys += 1
        sfx(3)
    end -- end if is_key
end -- end function pickup()
```

This works very similarly to the fruit collection, except I ***use the player's exact x,y position instead of target x,y***, because the key is directly beneath the player and not one tile away

```
-- pick up items by walking over them
function pickup()

    -- get sprite number of tile
    -- at player's position
    local tn = mget(plyr.x,plyr.y)

    -- check for key flag
    is_key = fget(tn,key)

    -- collect key
    if is_key then
        mset(plyr.x,plyr.y,1)
        keys += 1
        sfx(3)
    end -- end if is_key

end -- end function pickup()
```

Positioning the Camera

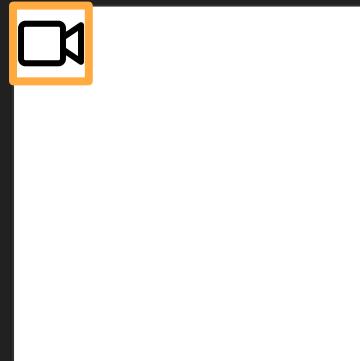
Download Example File: [topdown_04_camera_modes.p8](#)

Positioning the Camera

- There are two popular methods for positioning the camera in a top-down game:
 1. Centered on and following the player
 2. Fixed to one map screen at a time, scrolling all at once when the player moves to the next screen

Positioning the Camera

- There is a built-in PICO-8 function called [camera\(\)](#) that requires a pair of x,y coordinates
- By default, the camera is at (0,0), the top left of the screen

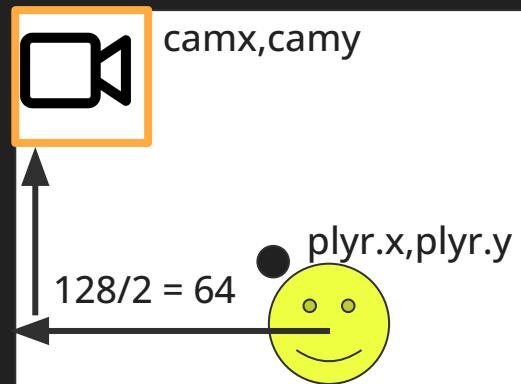


Positioning the Camera

To get the camera to center on the player, we can do a bit of math to calculate its position

Since the camera is above and to the left of the player, we subtract **64** from the player's x,y to get the camera's (rough) x,y

We then just need to add half the player's width and height to get the precise position



If the player is in the center, and the camera is in the top left, it must be roughly half a screen away from the player

```
0123 + 00000000  
-- CAMERA  
FUNCTION SET_CAMERA()  
CAMX = B*PLYR.X-64+PLYR.W/2  
CAMY = B*PLYR.Y-64+PLYR.H/2  
CAMERA(CAMX,CAMY)  
END -- END FUNCTION SET_CAMERA()
```

0123 +

() [] { } []

- CAMERA

```
FUNCTION SET_CAMERA()
CAMX = 8*PLYR.X-64+PLYR.U/2
CAMY = 8*PLYR.Y-64+PLYR.H/2
CAMERA(CAMX,CAMY)
END -- END FUNCTION SET_CAMERA()
```

- We can express the camera's x,y position relative to the player like so
- We need to **multiply by 8** to convert from the player's **tile** coordinates to the camera's **pixel** coordinates

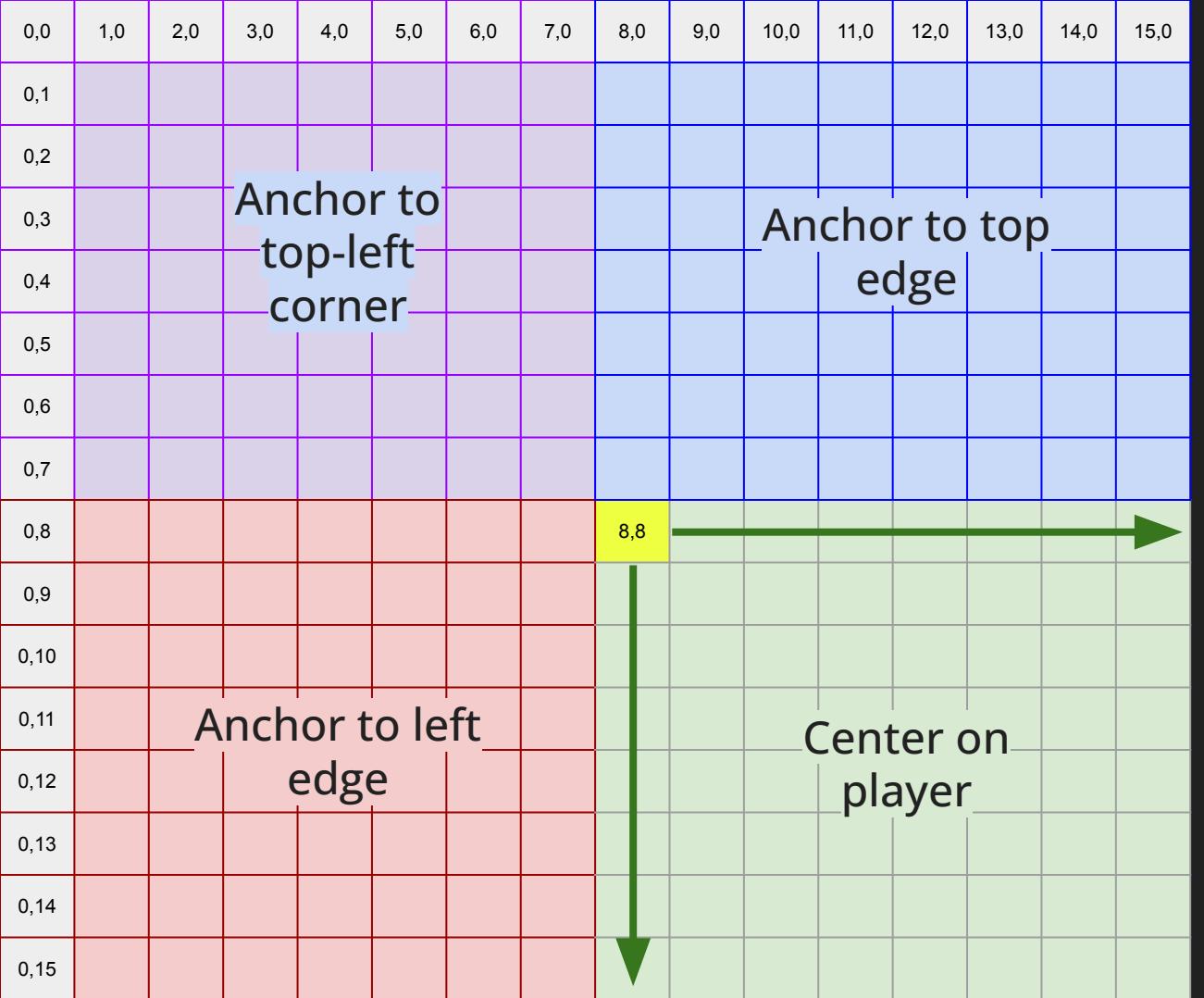


This blank space is *not just my slide background* – it's what you see when the game is running!



The previous calculation mostly works, but *we run into this issue when the player is at the edges of the map*

Because we made the camera **center the player on the screen**, this holds true even when that would mean showing an out-of-bounds area



We need to add overrides for when the player is close enough to the edge to anchor the camera to the edge

If the player is less than 8 tiles from the edge, we know part of the void will be showing while the camera is centered on the player – so if the player's x is less than 8, we know we need to *stop centering the camera on the player and anchor it to the edge of the map*





FRUIT

```
-- CENTER ON PLAYER  
CAMX=8*PLYR.X-64+PLYR.W/2  
CAMY=8*PLYR.Y-64+PLYR.H/2
```

```
-- ADJUST AT LEFT EDGE
```

```
IF PLYR.X < 8 THEN  
    CAMX += 8*(7-PLYR.X)+PLYR.W/2  
END
```

```
-- ADJUST AT TOP EDGE
```

```
IF PLYR.Y < 8 THEN  
    CAMY += 8*(7-PLYR.Y)+PLYR.H/2  
END
```

We can express how far the player is from the top edge as $7 - \text{plyr.y}$

We then multiply that value by 8 to convert from tiles to pixels

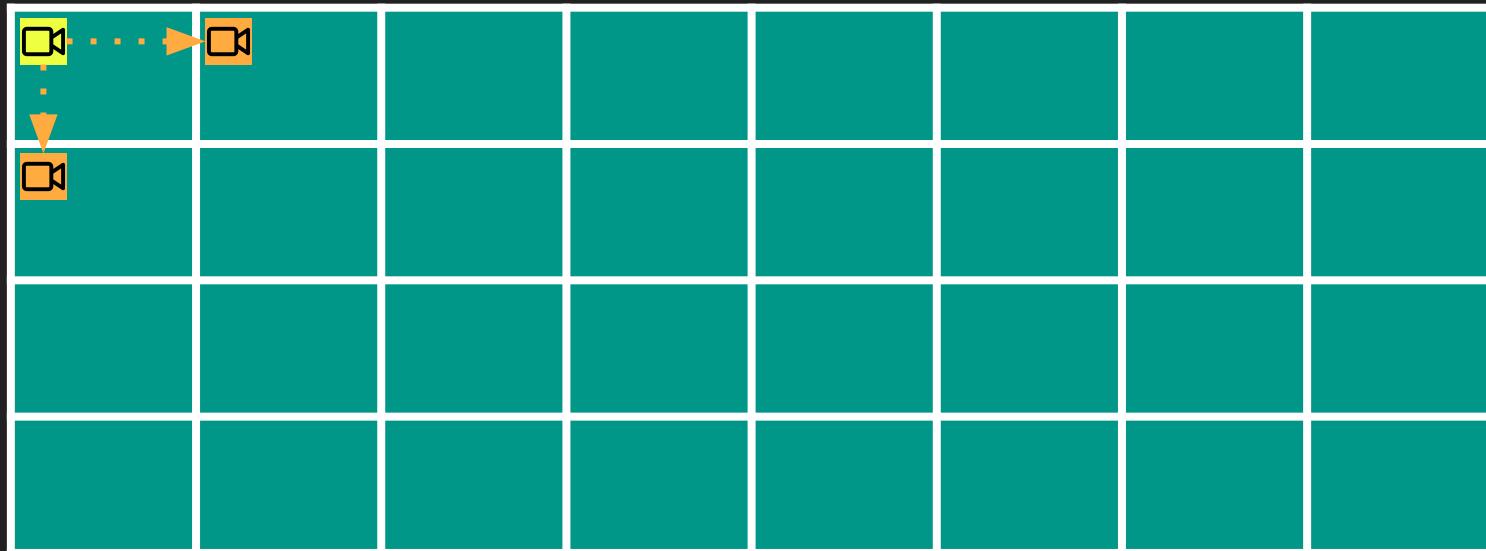
We add this amount, plus half the player's width, to camy to adjust

Screen-Based Camera

Download Example File: [topdown_04_camera_modes.p8](#)

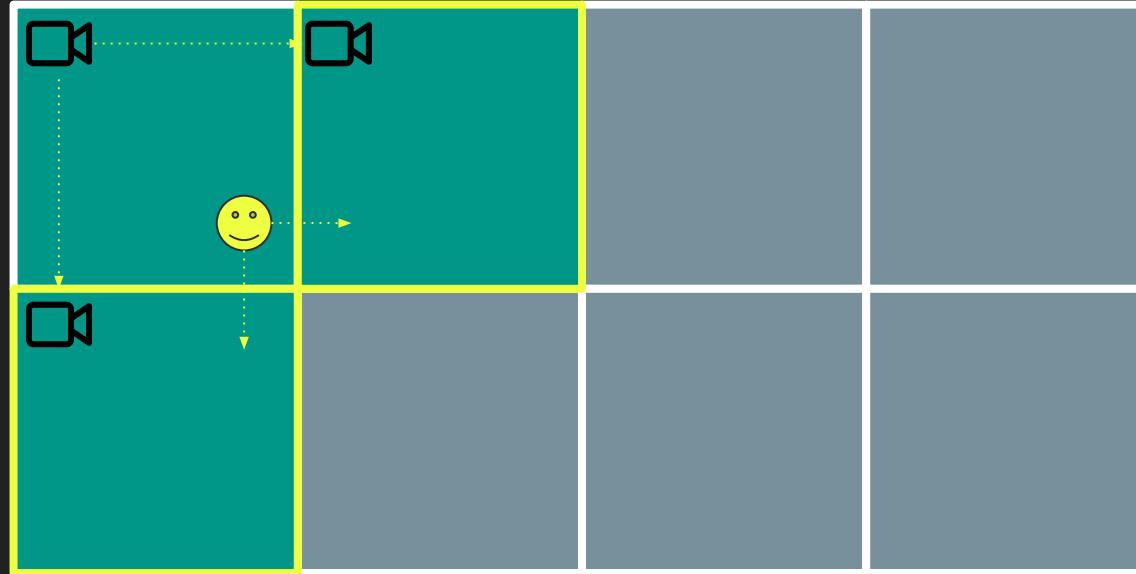
Screen-Based Camera

To show one screen at a time, we need to figure out what the **top-left tile coordinates of the current screen** are and position the camera there



Screen-Based Camera

If the player moves from one screen to another, the **camera** must be re-positioned in the top-right of the new screen



Screen-Based Camera

We can use this algorithm to place the camera in the top-left corner of the current screen

```
-- each tile is 8x8 pixels  
tilesize=8  
  
-- each screen is 16 tiles  
screensize=16  
  
-- calculate how many screens  
-- to the right of the origin  
-- (0,0) the player is  
screenx=floor(player.x/screensize)  
  
-- calculate how many screens  
-- below the origin (0,0) the  
-- player is  
screeny=floor(player.y/screensize)
```

```
-- multiply the camera coords  
-- by the number of pixels per  
-- tile to convert the player's  
-- tile coords to a pixel value  
camx = screenx * tilesize  
camy = screeny * tilesize  
  
-- multiply the camera coords  
-- by the number of tiles per  
-- screen to position it at the  
-- origin of the current screen  
-- (set of 16 tiles)  
camx*=screensize  
camy*=screensize
```

Screen-Based Camera

To help visualize the math, we can plug in 16 and 8 directly

16 tiles per screen

8 pixels per tile

16 tiles per screen

```
-- calculate how many screens  
-- to the right of the origin  
-- (0,0) the player is  
screenx=flr(plyr.x/16)
```

```
-- calculate how many screens  
-- below the origin (0,0) the  
-- player is  
screeny=flr(plyr.y/16)
```

```
-- multiply the camera coords  
-- by the number of pixels per  
-- tile to convert the player's  
-- tile coords to a pixel value  
camx = screenx * 8  
camy = screeny * 8
```

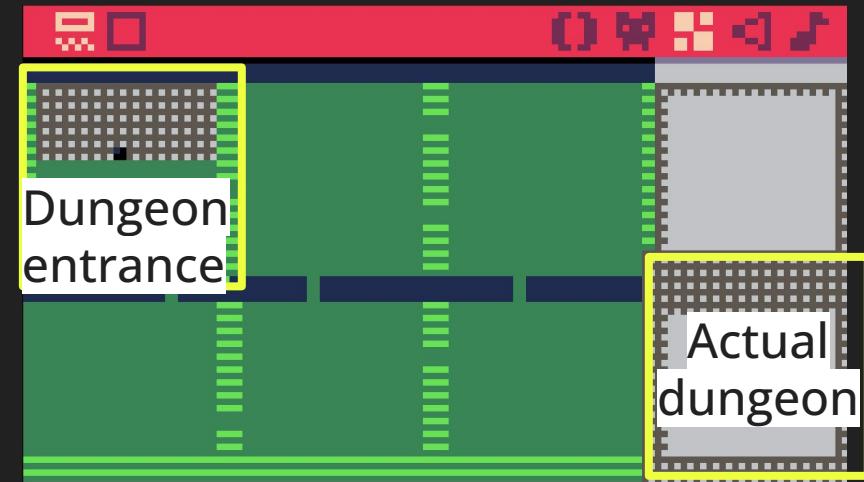
```
-- multiply the camera coords  
-- by the number of tiles per  
-- screen to position it at the  
-- origin of the current screen  
-- (set of 16 tiles)  
camx*=16  
camy*=16
```

Warping

Download Example File: [topdown_05_warping.p8](#)

Warping

- Let's say you want to divide your map into levels, or include houses or dungeons
- You can draw the interior portions anywhere on the map and then warp the player to that position when they touch the entrance



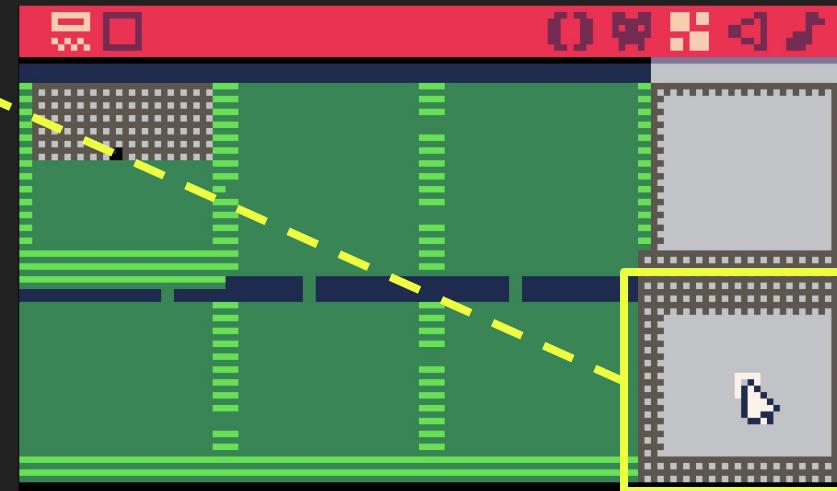


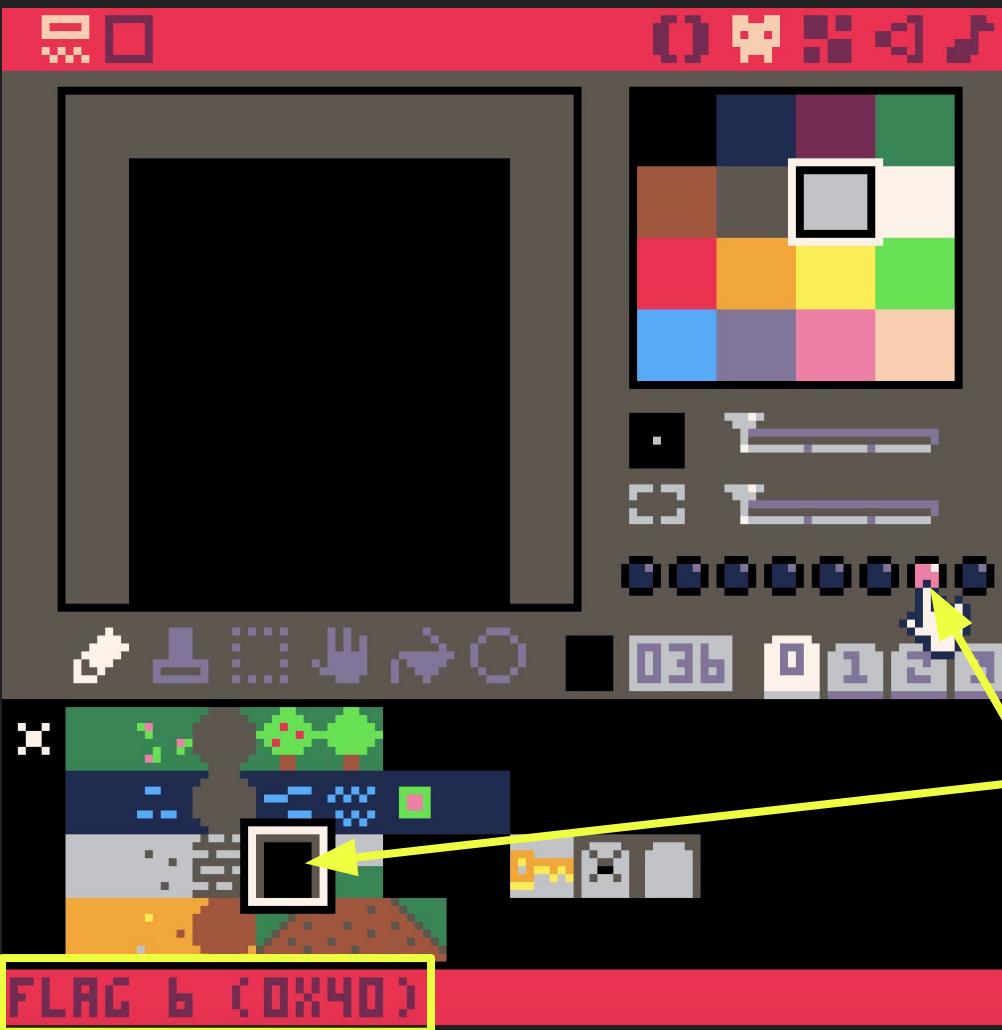
Here's the exterior
for my dungeon,
with its entrance at
23,7



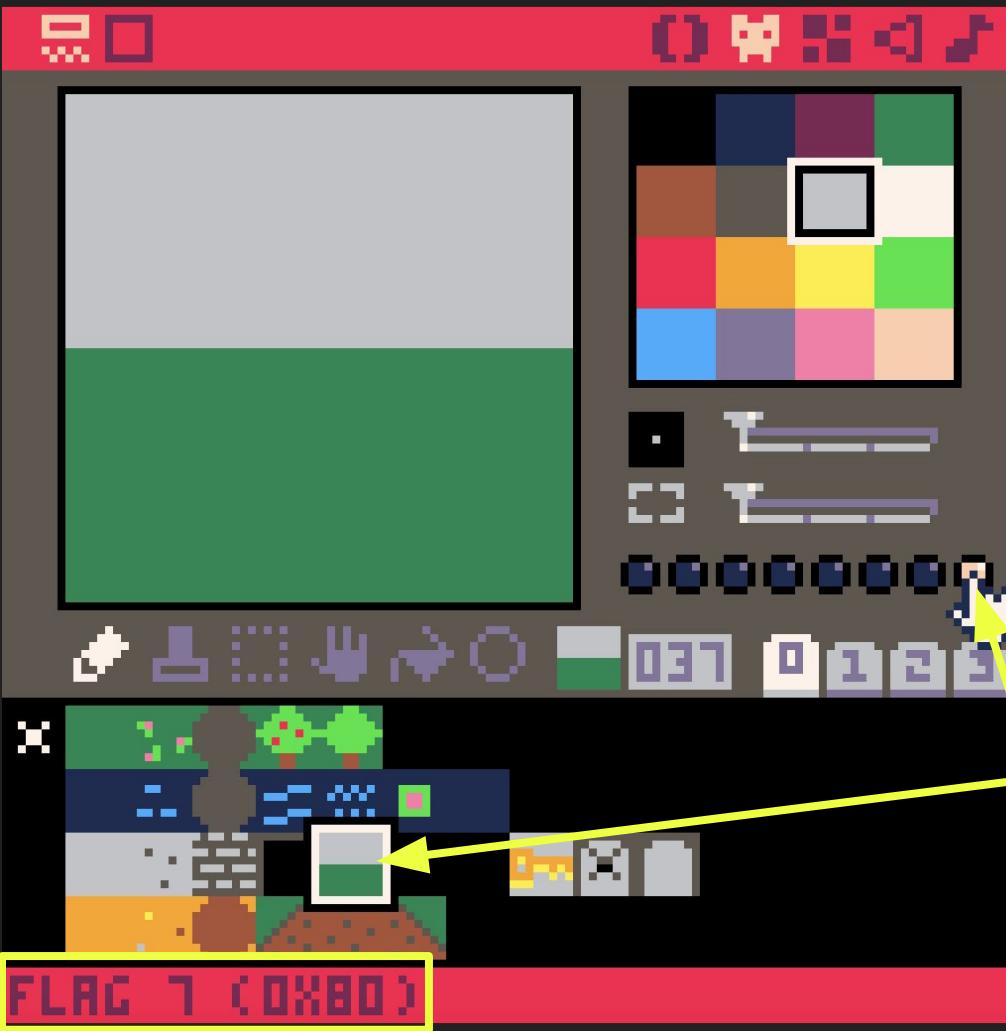


The dungeon itself is on the other side of the map, with the exit back outside at 88,31





I've turned on flag 6 for
the entrance tile



I've turned on flag 7 for
the exit tile

```
0123 + () × ◀ ▶
-- DUNGEON WARP
FUNCTION WARP( )
  -- SPRITE FLAGS
  ENTRANCE=6
  EXIT=7
  -- ENTRANCE TILE COORDS
  ENTX=88
  ENTY=29
  -- EXIT TILE COORDS
  EXITX=23
  EXITY=9
  -- GET SPRITE NUMBER OF TILE
  -- AT PLAYER'S POSITION
  LOCAL n=GET(PLYR.X, PLYR.Y)
  TS.ENTRANCE=FCFT(n, ENTRANCE)
LINE 1/36      378/8192 E
```

In my code, I set those flag numbers as the values for my variables:

entrance=6
exit=7

I've also set the tile coordinates for the entrance and exit as variables

0123 +

0 0 0 0

```
-- AT PLAYER'S POSITION  
LOCAL n=MGET(PLYR.X, PLYR.Y)  
IS_ENTRANCE=FGET(n, ENTRANCE)  
IS_EXIT=FGET(n, EXIT)
```

```
-- WARP INSIDE DUNGEON  
IF IS_ENTRANCE THEN  
PLYR.X=ENTX  
PLYR.Y=ENTY  
SET_CAMERA()  
END -- END IF IS_ENTRANCE  
  
-- WARP OUTSIDE DUNGEON  
IF IS_EXIT THEN  
PLYR.X=EXITX  
PLYR.Y=EXITY  
SET_CAMERA()  
END -- END IF IS_EXIT
```

Similarly to the pickup key function, I use mget() and fget() to check for the entrance and exit flags at the player's x,y position

Then, if either **flag** returns **true** (*meaning the player is touching the entrance or exit tile*), I set the player's x,y location equal to the corresponding destination's coordinates

Pixel-Based Movement

Download Example File: [topdown 06 pixelbased movement and collision.p8](#)

Tile-Based vs. Pixel-Based Movement

- **Tile-based movement**
 - 8 pixels at a time
 - Pros: Easier to implement
 - Cons: Less nuance in movement
 - Best for: games without an emphasis on real-time reaction (**exploration and collection**)
- **Pixel-based movement**
 - A smaller number of pixels at a time
 - Pros: More nuance in movement
 - Cons: More work for collision detection
 - Best for: games requiring real-time reaction (**action, shooting**)

```
0 +
-- RUNS ONCE AT START
FUNCTION _INIT()
END -- END FUNCTION _INIT()

-- LOOPS 30X/SEC
FUNCTION _UPDATE()
END -- END FUNCTION _UPDATE()

-- LOOPS 30X/SEC
FUNCTION _DRAW()
    CLS() -- CLEAR SCREEN
    MAP() -- DRAW MAP AT 0,0
END -- END FUNCTION _DRAW()
```

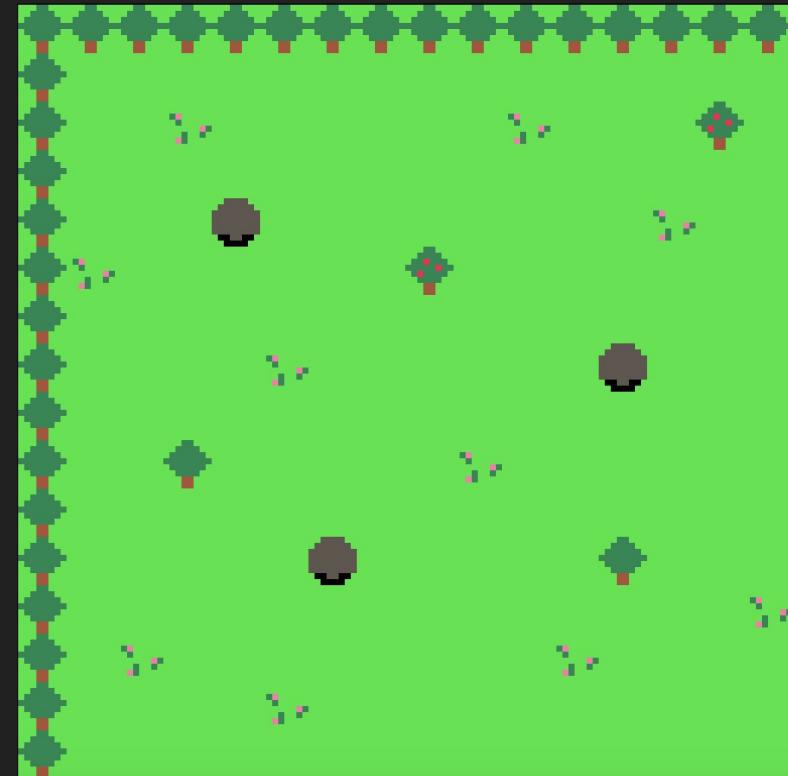
Create an empty game loop (the `_init`, `_update`, and `_draw` functions)

Use the [map\(\)](#) function in `_draw()` to draw the map

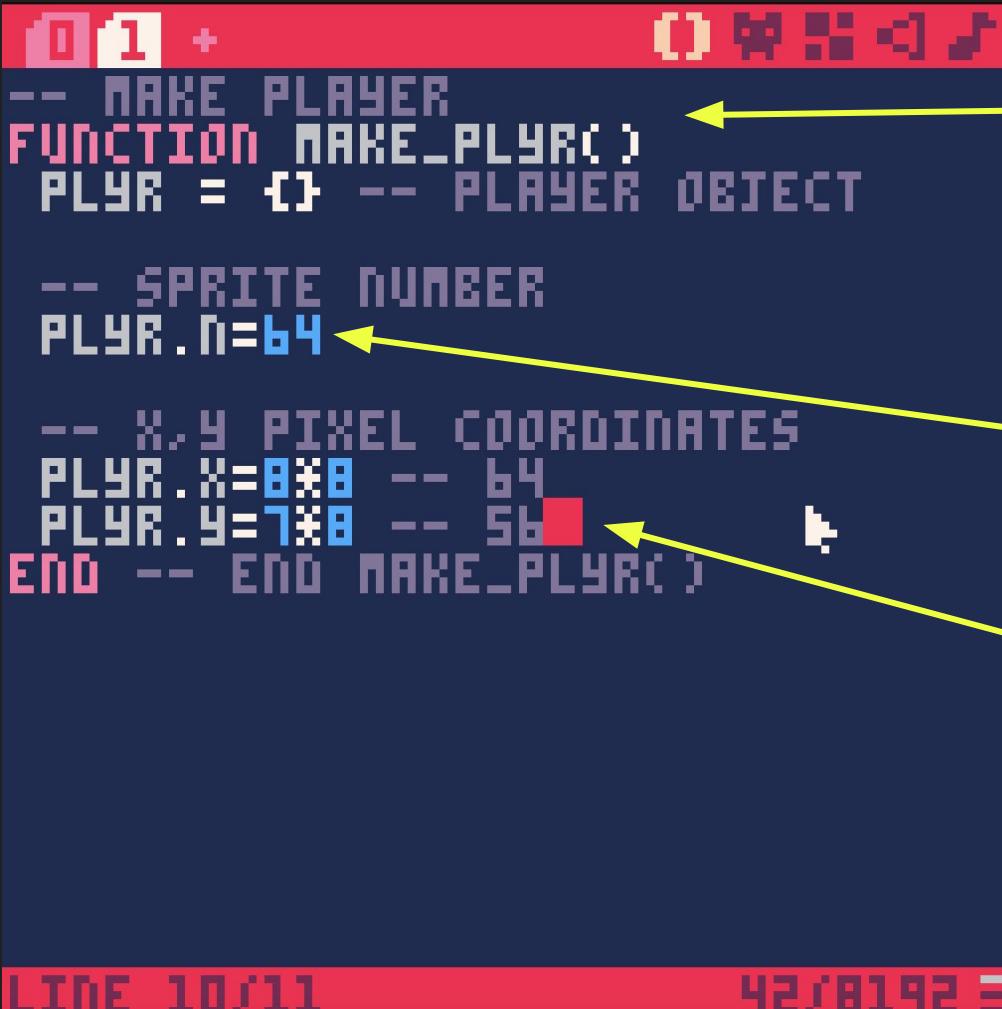
```
0 +          0 0 0 < > 
-- RUNS ONCE AT START
FUNCTION _INIT()
END -- END FUNCTION _INIT()

-- LOOPS 30X/SEC
FUNCTION _UPDATE()
END -- END FUNCTION _UPDATE()

-- LOOPS 30X/SEC
FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
MAP() -- DRAW MAP AT 0,0
END -- END FUNCTION _DRAW()
```



Use the [map\(\)](#) function in
[_draw\(\)](#) to draw the map



```
01 +
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR = {} -- PLAYER OBJECT
-- SPRITE NUMBER
PLYR.n=64
-- X,Y PIXEL COORDINATES
PLYR.X=8*8 -- 64
PLYR.Y=7*8 -- 56
END -- END MAKE_PLYR()
```

I'll create a function to make my player on a separate tab

The player **sprite number** was **64**

And its **x,y** coordinates on the map were **7,8**

```
01 +
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR = {} -- PLAYER OBJECT
-- SPRITE NUMBER
PLYR.n=64
-- X,Y PIXEL COORDINATES
PLYR.x=8*8 -- 64
PLYR.y=7*8 -- 56
END -- END MAKE_PLYR()
```



- The player's x,y coordinates on the map were 7,8
- *But the map editor displays tile values*
- Whereas the spr() function requires pixel values
- Each tile is 8x8 pixels, so we multiply 8 and 7 each by 8 to convert from tiles to pixels

```
-- MAKE PLAYER  
FUNCTION MAKE_PLYR()  
PLYR = {} -- PLAYER OBJECT  
  
-- SPRITE NUMBER  
PLYR.n=64  
  
-- X,Y PIXEL COORDINATES  
PLYR.X=8*8 -- 64  
PLYR.Y=7*8 -- 56  
END -- END MAKE_PLYR()
```

LINE 10/11

42/8192 E

```
-- RUNS ONCE AT START  
FUNCTION _INIT()  
MAKE_PLYR() -- TAB 1  
END -- END FUNCTION _INIT()  
  
-- LOOPS 30X/SEC  
FUNCTION _UPDATE()  
END -- END FUNCTION _UPDATE()  
  
-- LOOPS 30X/SEC  
FUNCTION _DRAW()  
CLS() -- CLEAR SCREEN  
MAP() -- DRAW MAP AT 0,0  
SPR(PLYR.n, PLYR.X, PLYR.Y)  
END -- END FUNCTION _DRAW()
```

LINE 15/16

57/8192 E

After writing our `make_plyr()` function on tab 1, we must *call make_plyr() in the `_init()` function* on tab 0

```
01 +
02 MMMM  
-- MAKE PLAYER  
FUNCTION MAKE_PLYR()  
PLYR = {} -- PLAYER OBJECT  
  
-- SPRITE NUMBER  
PLYR.n=64  
  
-- X,Y PIXEL COORDINATES  
PLYR.X=8*8 -- 64  
PLYR.Y=7*8 -- 56  
END -- END MAKE_PLYR()
```

```
01 +
02 MMMM  
-- RUNS ONCE AT START  
FUNCTION _INIT()  
MAKE_PLYR() -- TAB 1  
END -- END FUNCTION _INIT()  
  
-- LOOPS 30X/SEC  
FUNCTION _UPDATE()  
END -- END FUNCTION _UPDATE()  
  
-- LOOPS 30X/SEC  
FUNCTION _DRAW()  
CLS() -- CLEAR SCREEN  
DAP() -- DRAW DAP AT 0,0  
SPR(PLYR.n, PLYR.X, PLYR.Y)  
END -- END FUNCTION _DRAW()
```

LINE 10/11 42/8192 E

LINE 15/16 57/8192 E

We can then plug the player's **n**, **x**, and **y** variables into the spr() function in _draw() to draw the player at the desired location



Our player at the tile
coordinates 8,7

(Pixel coordinates 64,56)

0 1 2 +

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

Next, we can begin a  move_plyr() function in a separate tab ...

... and call move_plyr() inside the _update() function

0 1 2 +

```
-- RUNS ONCE AT START  
FUNCTION _INIT()  
MAKE_PLYR() -- TAB 1  
END -- END FUNCTION _INIT()
```

```
-- LOOPS 30X/SEC  
FUNCTION _UPDATE()  
MOVE_PLYR() -- TAB 2  
END -- END FUNCTION _UPDATE()
```

```
-- LOOPS 30X/SEC  
FUNCTION _DRAW()  
CLS() -- CLEAR SCREEN  
MAP() -- DRAW MAP AT 0,0  
SPR(PLYR.N, PLYR.X, PLYR.Y)  
END -- END FUNCTION _DRAW()
```

LINE 8/16

50/8192 ⏴

0/12 +

0 0 0 0 0 0

-- MOVE PLAYER

FUNCTION MOVE_PLYR()

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

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

END -- END FUNCTION MOVE_PLYR()

Let's begin our move_plyr() function by allowing the **left** and **right** arrow keys to decrease and increase our player's **x** value

Very similar to paddleball



012 +

() * < > ?

```
-- MOVE LEFT
IF BTn(0) THEN
  PLYR.X -= 1
END -- END IF BTn(0)

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

-- MOVE UP
IF BTn(2) THEN
  PLYR.Y -= 1
END -- END IF BTn(2)

-- MOVE DOWN
IF BTn(3) THEN
  PLYR.Y += 1
END -- END IF BTn(3)
```

LINE 22/24

86/8192 E

Likewise, we can use the **up** and **down** keys to decrease and increase our player's **y** value

012 +

()

```
-- MOVE PLAYER  
FUNCTION MOVE_PLAYER()  
  
-- MOVE LEFT  
IF BTn(0) THEN  
    PLYR.X -= PLYR.SPD  
ENDIF -- END IF BTn(0)  
  
-- MOVE RIGHT  
IF BTn(1) THEN  
    PLYR.X += 1  
ENDIF -- END IF BTn(1)  
  
-- MOVE UP  
IF BTn(2) THEN  
    PLYR.Y -= 1  
ENDIF -- END IF BTn(2)  
  
-- MOVE DOWN
```

1 pixel of movement is very slow, and something you'll likely want to tweak

It makes sense to use a variable so you only need to change the value in one place instead of four places

I used a variable called **plyr.spd** (for speed)

012 +

098<)

```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
PLYR = {} -- PLAYER OBJECT

-- SPRITE NUMBER
PLYR.N=64

-- X,Y PIXEL COORDINATES
PLYR.X=8*8 -- 64
PLYR.Y=7*8 -- 56

-- PLAYER SPEED
PLYR.SPD = 4

END -- END FUNCTION MAKE_PLYR()
```

Just make sure you **define** **the variable** in your make_plyr() function

0012 +

0 0 0 < >

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

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

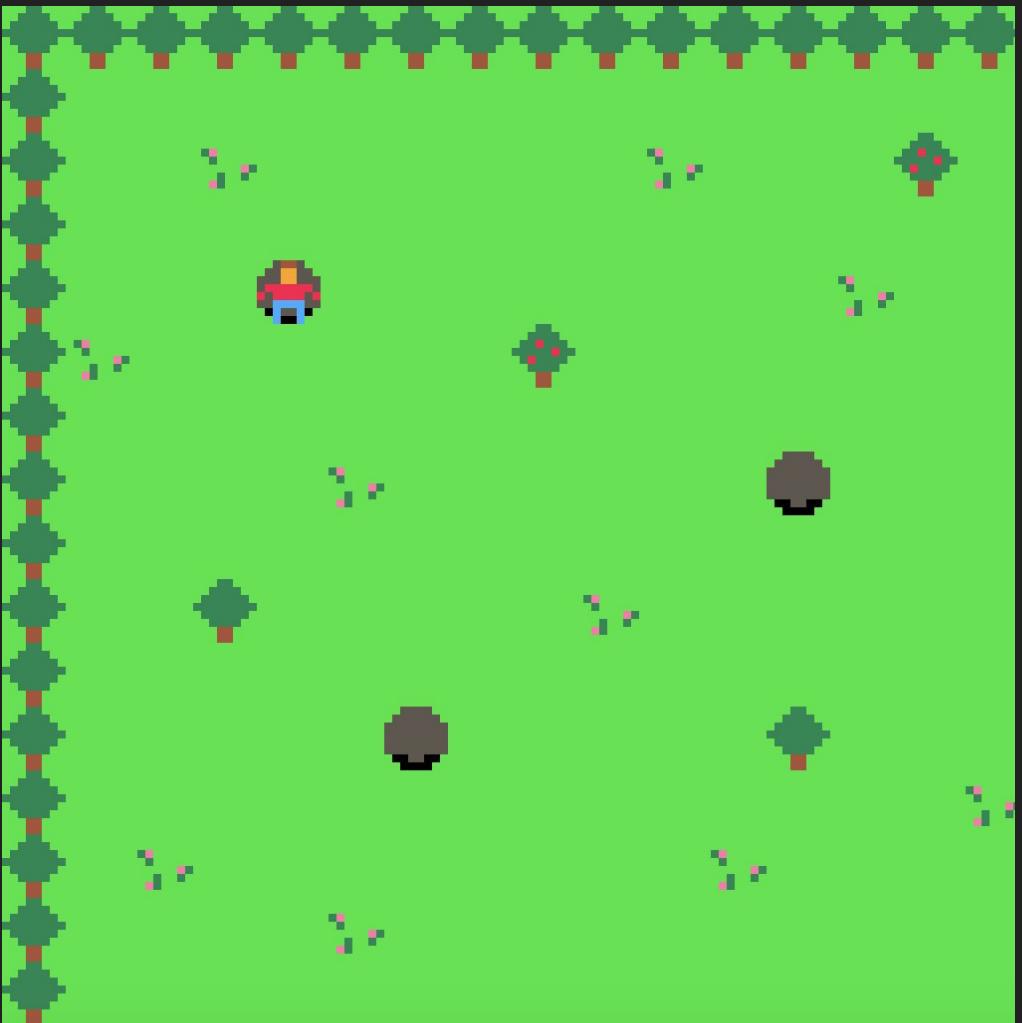
-- MOVE UP
IF BTn(2) THEN
    PLYR.Y -= PLYR.SPD
END -- END IF BTn(2)

-- MOVE DOWN
IF BTn(3) THEN
    PLYR.Y += PLYR.SPD
END -- END IF BTn(3)
```

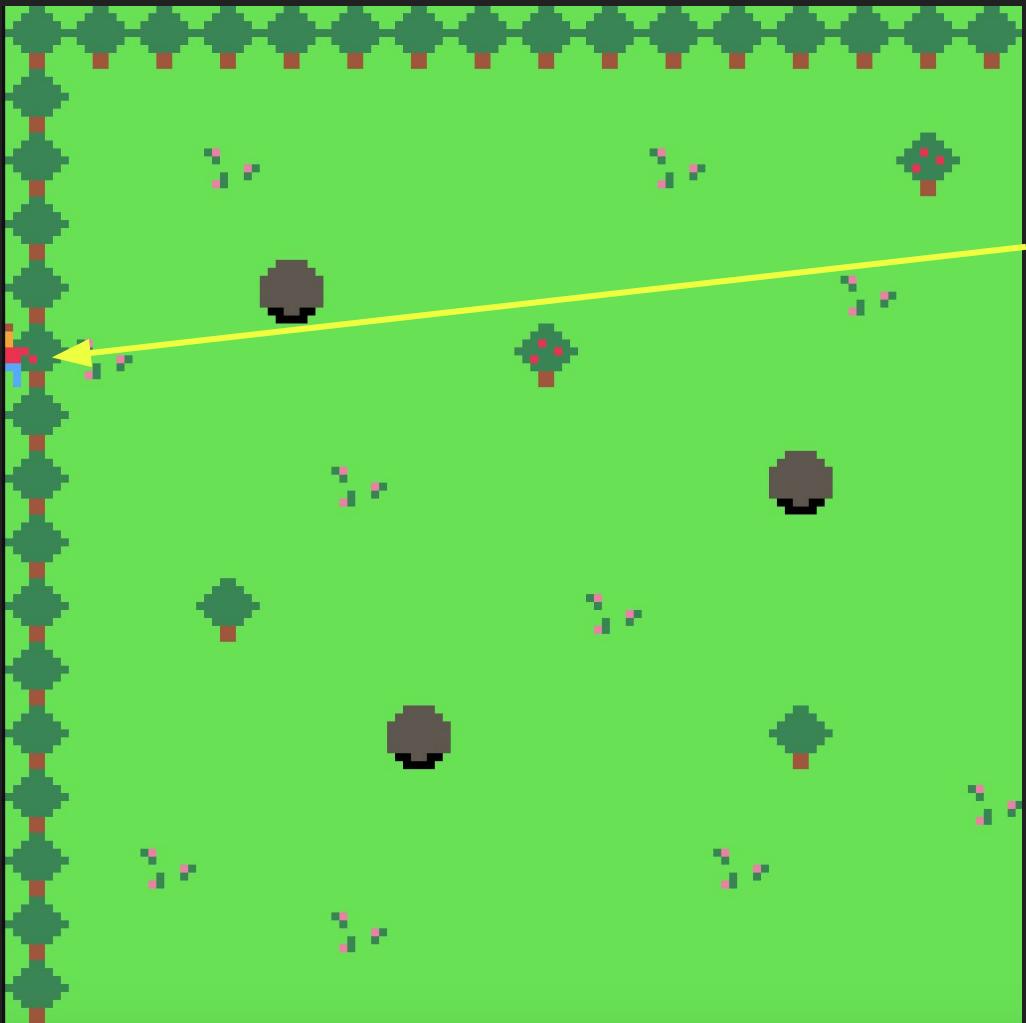
LINE 22/24

94/8192 ⏴

Here, I'm using `plyr.spd`, instead of a fixed number value, to increment and decrement my player's x or y position



Of course, we haven't implemented collision yet, so our player will still go through tiles that look like they should be solid



And we can even move off screen entirely

012 +

() <= > =

```
PLYR.Y -= PLYR.SPD  
END -- END IF BTn(0)
```

-- MOVE DOWN

```
IF BTn(1) THEN  
PLYR.Y += PLYR.SPD  
END -- END IF BTn(1)
```

-- KEEP ON SCREEN LEFT

```
IF PLYR.X < 0 THEN  
PLYR.X = 0  
END -- END IF PLYR.X < 0
```

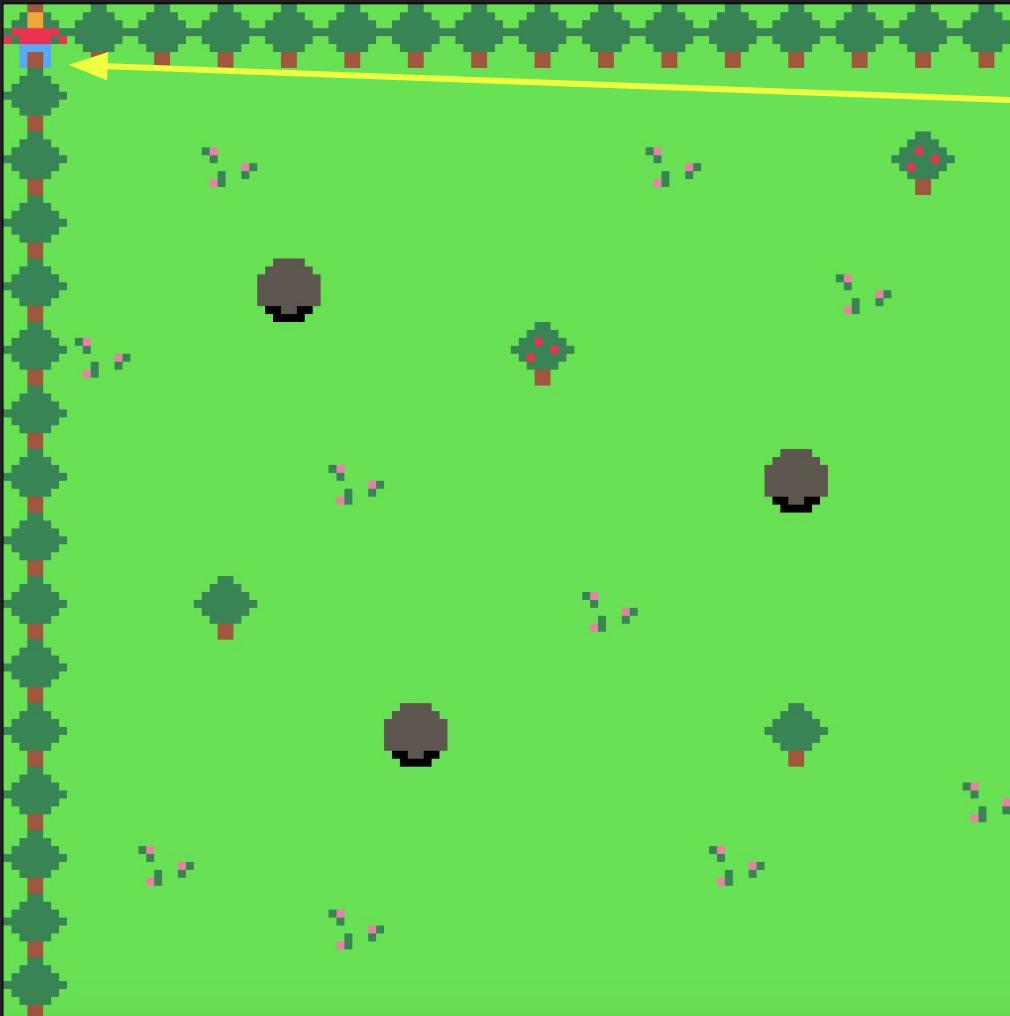
-- KEEP ON SCREEN TOP

```
IF PLYR.Y < 0 THEN  
PLYR.Y = 0  
END -- END IF PLYR.Y < 0
```

```
END -- END FUNCTION MOVE_PLYR()  
LINE 34/34
```

114/8192 E

We can keep the player on screen much like we kept the ball and paddle on screen



Now the player cannot move beyond the left or top edges of the map

```
-- KEEP ON SCREEN LEFT
IF PLYR.X < 0 THEN
    PLYR.X = 0
END -- END IF PLYR.X < 0

-- KEEP ON SCREEN TOP
IF PLYR.Y < 0 THEN
    PLYR.Y = 0
END -- END IF PLYR.Y < 0
```

Implementing Collision with Map Tiles

Download Example File: [topdown 06 pixelbased movement and collision.p8](#)

```
0 1 2 + 0 1 2 3 4 5
-- RUNS ONCE AT START
FUNCTION _INIT()
MAKE_PLAYER() -- TAB 1

-- TARGET X,Y (WHERE THE PLAYER
-- IS TRYING TO MOVE TO)
TX=PLAYER.X
TY=PLAYER.Y

END -- END FUNCTION _INIT()

-- LOOPS 30X/SEC
FUNCTION _UPDATE()
MOVE_PLAYER() -- TAB 2
END -- END FUNCTION _UPDATE()

-- LOOPS 30X/SEC
FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
LINE 9/22 120/8192 E
```

Like with tile-based movement, we'll use **variables** to track the x,y position the player is trying to move to - I'll call these **tx** and **ty** for target x and target y

I'll assign them a value in `_init()` for starters, but this value will change when the player presses an arrow key

0 1 2 +

()

```
-- MOVE LEFT  
IF BTn(0) THEN  
    TX=PLYR.X-1  
    TY=PLYR.Y  
END -- END IF BTn(0)
```

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

```
-- MOVE UP  
IF BTn(2) THEN  
    --PLYR.Y -= PLYR.SPD  
END -- END IF BTn(2)
```

```
-- MOVE DOWN  
IF BTn(3) THEN
```

LINE 7/35

We'll calculate the values for **tx** and **ty** in our **move_plyr** function

If we're trying to move **left**, the **target x location is one pixel less than the current x location**

This is expressed as:
tx=plyr.x-1

(The target y will just be the same as **plyr.y** because we are not moving up or down)

120/8192 E

0 1 2 +

() [] < > ↻

```
-- MOVE PLAYER
FUNCTION MOVE_PLYR()
    -- MOVE LEFT
    IF BTn(0) THEN
        TX=PLYR.X-1
        TY=PLYR.Y
    END -- END IF BTn(0)
```

```
-- MOVE RIGHT
IF BTn(1) THEN
    TX=PLYR.X+PLYR.W+1
    TY=PLYR.Y
END -- END IF BTn(1)
```

```
-- MOVE UP
IF BTn(2) THEN
    TX=PLYR.X
    TY=PLYR.Y-1
```

If we're trying to move right,
we *must account for the
player's width*

This is expressed as:
 $tx = \text{plyr.x} + \text{plyr.w}$

(Make sure to declare
plyr.w and *plyr.h*
variables in *make_plyr*)

```
-- MAKE PLAYER
FUNCTION MAKE_PLYR()
    PLYR = {} -- PLAYER OBJECT
    -- SPRITE NUMBER
    PLYR.N=64
    -- X,Y PIXEL COORDINATES
    PLYR.X=8*8 : b4
    PLYR.Y=7*8 -- 5b
    -- WIDTH AND HEIGHT (IN PIXELS)
    PLYR.W=8
    PLYR.H=8
    -- PLAYER SPEED
    PLYR.SPD = 2
END -- END FUNCTION MAKE_PLYR()
LINE 1/19
```

0/12 +

O □ □ □ □

```
-- MOVE RIGHT  
IF BTn(0) THEN  
  TX=PLYR.X+PLYR.W+1  
  TY=PLYR.Y  
END -- END IF BTn(0)
```

```
-- MOVE UP  
IF BTn(0) THEN  
  TX=PLYR.X  
  TY=PLYR.Y-1 ←  
END -- END IF BTn(0)
```

```
-- MOVE DOWN  
IF BTn(0) THEN  
  TX=PLYR.X  
  TY=PLYR.Y+PLYR.H+1 →  
END -- END IF BTn(0)
```

If we're trying to move **up** or **down**, we'll do the same thing, but changing **ty** while leaving **tx** equal to **plyr.x**

And account for the **height** moving **down** instead of the width

Implementing Collision with Map Tiles

- Now we've calculated the x,y location that the player is trying to move to (expressed in our code as **tx,ty**)
- Next, we can **use this target x,y location** to figure out whether the **sprite flag** for the **map tile** is turned **on**

Implementing Collision with Map Tiles

- PICO-8 offers two useful built-in functions for working with map tiles and sprite flags
- mget() - takes an **x,y location** and **finds the sprite number** at that location
- fget() - takes a **sprite number** and **finds whether a flag is turned on**

Implementing Collision with Map Tiles

Calculate tx,ty

mget(tx,ty) = sprite_number

fget(sprite_number,flag_number) = true or false

Here's an overview of
how that code would
look ...

```
-- convert from pixels to tiles  
-- divide by 8 and round down  
tx=floor(tx/8)  
ty=floor(ty/8)
```

Calculate tx,ty

```
-- mget finds the sprite number  
-- of a tile at an x,y location  
-- but it needs this value in  
-- terms of map tiles, not px  
tile=mget(tx,ty)
```

mget(tx,ty) = sprite_number

```
-- fget finds whether a flag  
-- is turned on for a sprite  
-- it needs the sprite number  
-- and flag number  
is_wall=fget(tile,0)
```

fget(sprite_number,flag_number)
= true or false

Let's do this step by step

```
012 +
END -- END IF BTn(0)
-- MOVE DOWN
IF BTn(0) THEN
  TX=PLYR.X
  TY=PLYR.Y+PLYR.H+1
END -- END IF BTn(0)
```

```
-- CONVERT FROM PIXELS TO TILES
-- DIVIDE BY 8 AND ROUND DOWN
```

```
TX=FLR(TX/8)
TY=FLR(TY/8)
```

```
-- MGET FINDS THE SPRITE NUMBER
-- OF A TILE AT AN X,Y LOCATION
-- BUT IT NEEDS THIS VALUE IN
-- TERMS OF MAP TILES, NOT PX
TILE=MGET(TX,TY)
```

LINE 37/49

183/8192 E

Above, we calculated tx and ty based on which direction the player is trying to move

Next, we need to **divide tx and ty by 8**, to *convert from pixels to tiles*, because mget() requires a tile value

We can use flr() to **round down**

Then we can plug the updated tx and ty values into the mget() function, which **returns the sprite number** of the tile

012 +

0 1 2 3 4 5

```
-- MGET FINDS THE SPRITE NUMBER  
-- OF A TILE AT AN X,Y LOCATION  
-- BUT IT NEEDS THIS VALUE IN  
-- TERMS OF MAP TILES, NOT PX  
TILE=MGET(TX,TY)
```

```
-- FGET FINDS WHETHER A FLAG  
-- IS TURNED ON FOR A SPRITE  
-- IT NEEDS THE SPRITE NUMBER  
-- AND FLAG NUMBER  
IS_WALL=FGET(TILE,0)
```

```
-- ONLY MOVE IF NOT A WALL AT  
-- TARGET LOCATION  
IF NOT IS_WALL THEN  
-- PLAYER MOVEMENT HERE  
END -- END IF NOT IS_WALL
```

LINE 48/61

193/8192 E

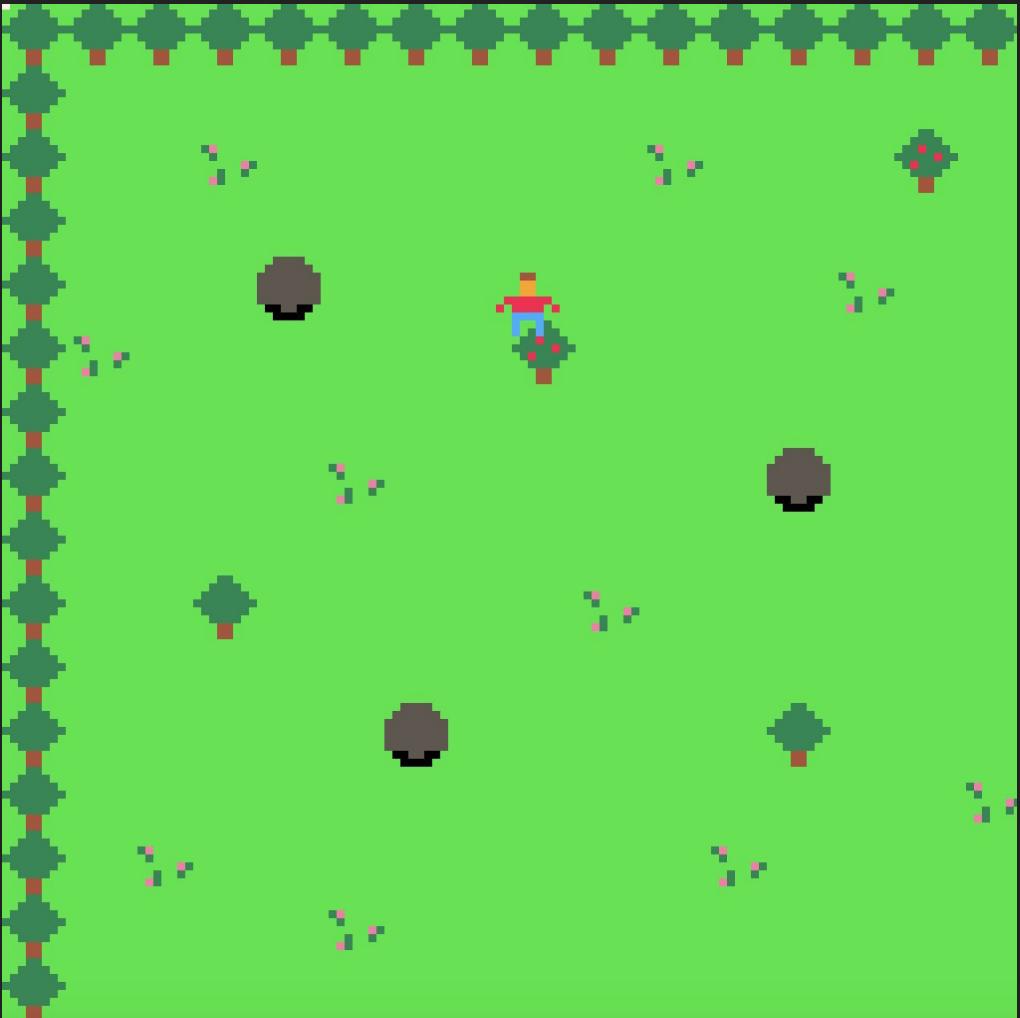
We plug the updated **tx** and **ty** values into the mget() function, which returns the **sprite number** of the tile

Then we can **plug the tile's sprite number** into the fget() function – *along with the flag number we used, which was 0 – to return true or false*



Then we can *only move the player if is_wall returned false*

We can use the keyword NOT to flip true to false or vice-versa



This mostly works, except
sometimes the player still goes through solid tiles

This is because we are **only checking one point** relative to the player

We'll need to **check a point on the opposite side of the player** to make our collision check comprehensive

0 1 2 +

() [] < > ↴ ↵

-- RUNS ONCE AT START

FUNCTION _INIT()

MAKE_PLYR() -- TAB 1

-- TARGET X,Y (WHERE THE PLYR
-- IS TRYING TO MOVE TO)

TX=PLYR.X

TY=PLYR.Y



-- CHECK OTHER SIDE OF SPRITE

TX2=TX

TY2=TY

END -- END FUNCTION _INIT()

-- LOOPS 30X/SEC

FUNCTION _UPDATE()

MOVE_PLYR() -- TAB 2

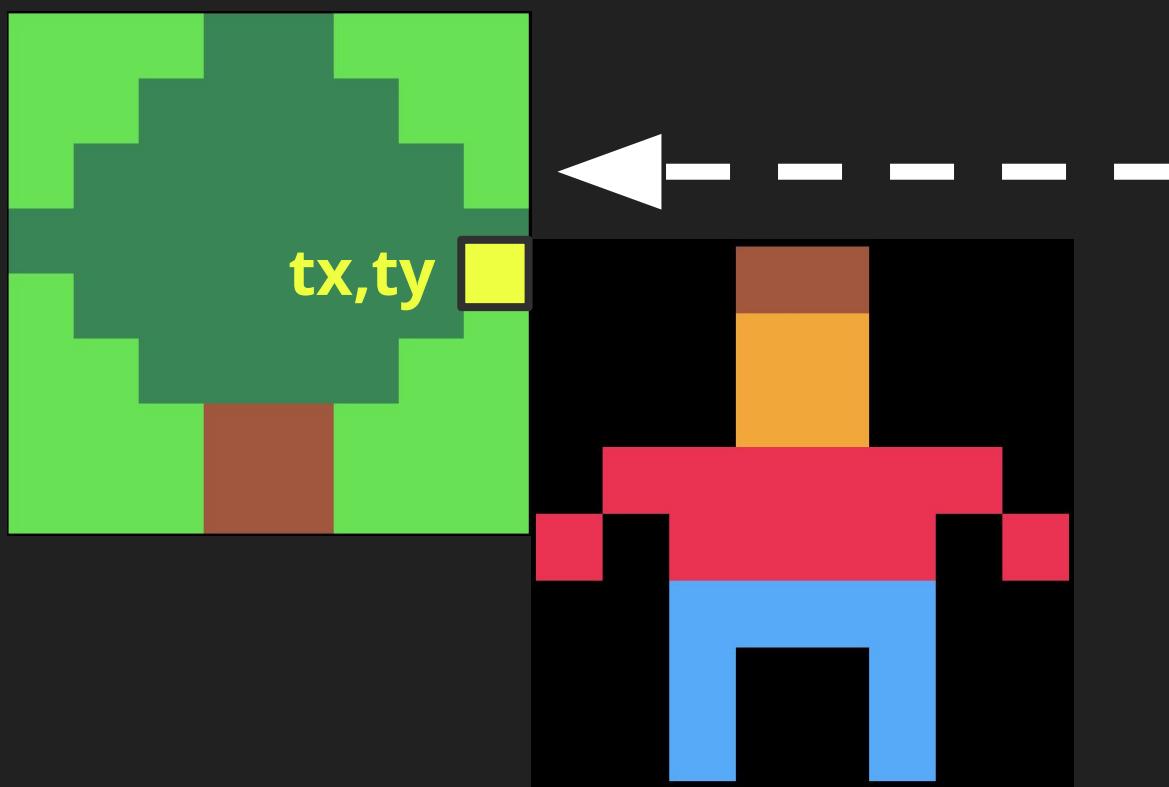
END -- END FUNCTION _UPDATE()

LINE 10/30

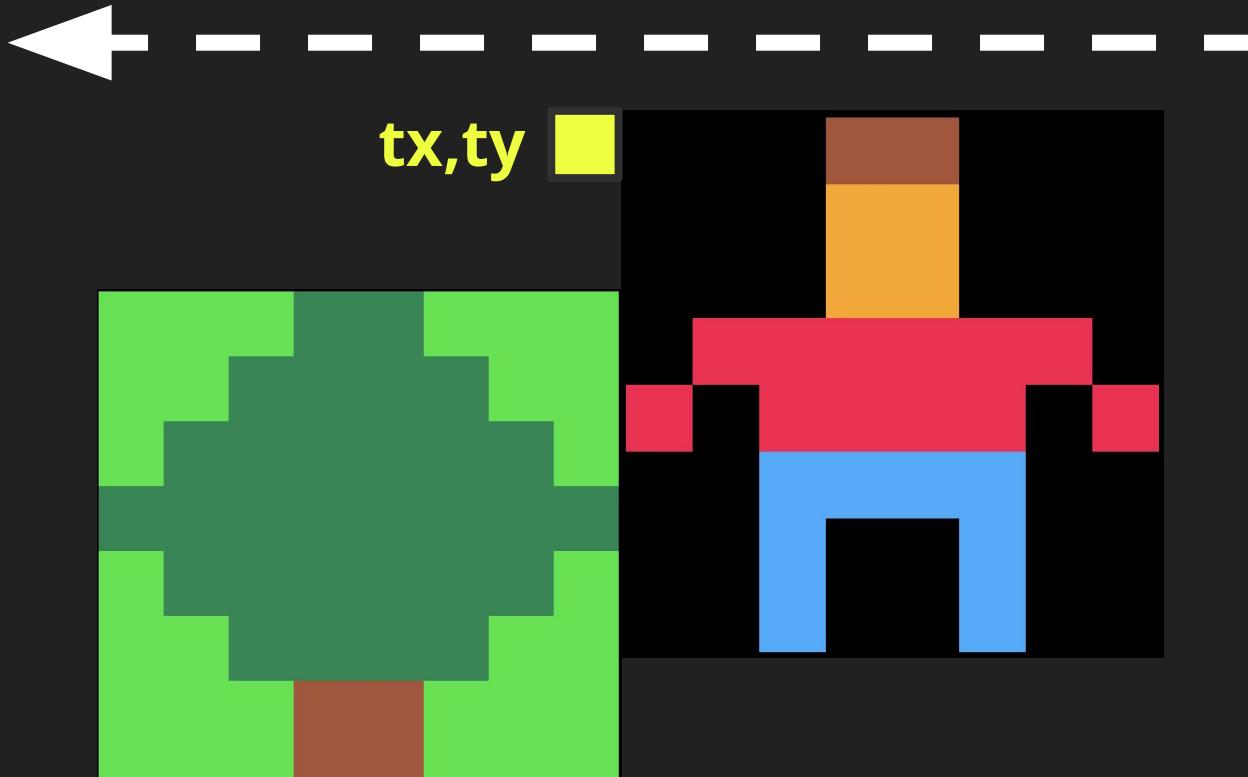
247/8192 E

We'll create a **second pair** of target x,y variables

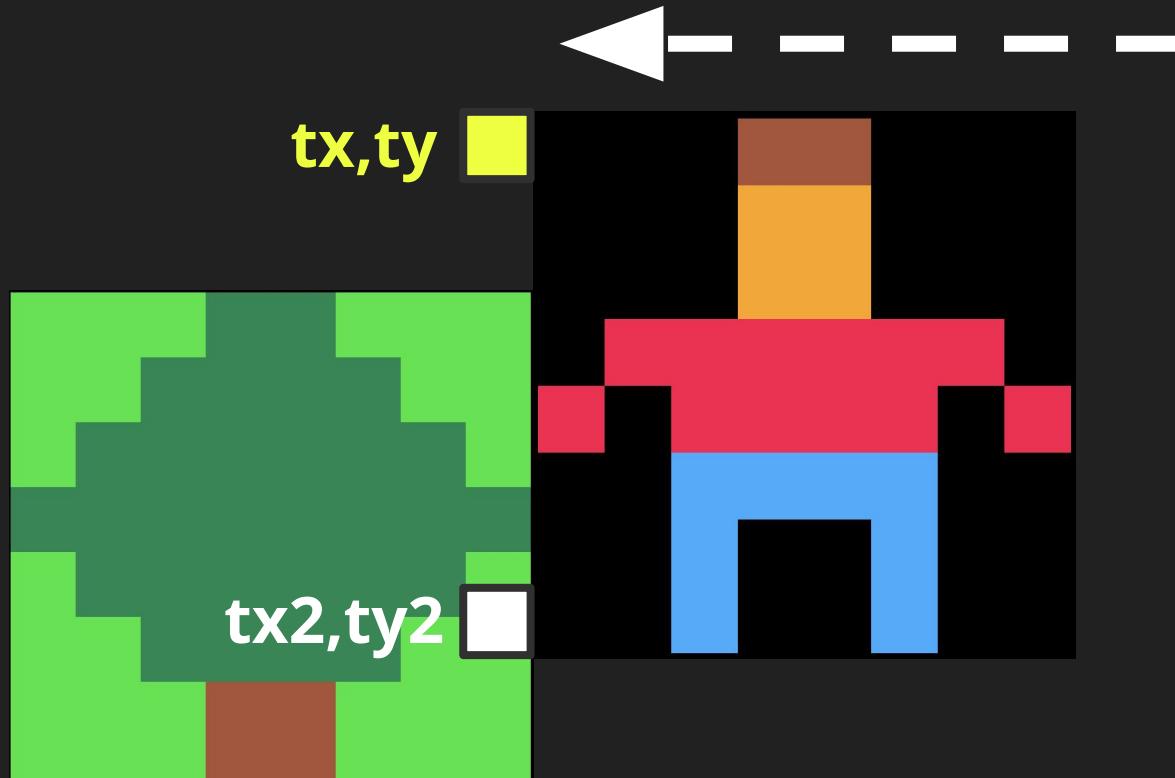
I named mine **tx2** and **ty2**



If the player is trying to move **left**, in this instance, the calculated position of **tx,ty** *intersects with the tree sprite, preventing movement*



But if the tree is below the top of the player, ***tx,ty is above the tree, so the player will move through it***



We'll create a second pair of x,y coordinates (**tx2,ty2**) to
check the bottom of the player when moving left/right



And we'll *check*
the right side of
the player when
moving up/down

0 1 2 +

0 1 2 3 4 5 6 7

FUNCTION MOVE_PLAYER()

```
-- MOVE LEFT  
IF BTn(0) THEN  
    TX=PLYR.X-1  
    TY=PLYR.Y
```

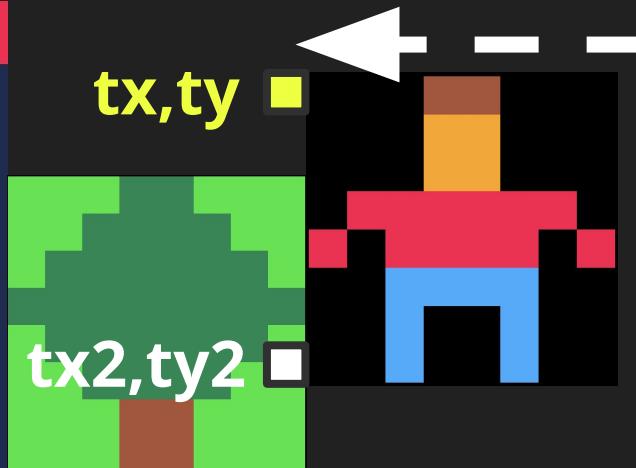
```
    TX2=TX  
    TY2=PLYR.Y+PLYR.H  
END -- END IF BTn(0)
```

```
-- MOVE RIGHT  
IF BTn(1) THEN  
    TX=PLYR.X+PLYR.W+1  
    TY=PLYR.Y
```

```
    TX2=TX  
    TY2=PLYR.Y+PLYR.H  
END -- END IF BTn(1)
```

LINE 10/98

316/8192



When moving **left/right**, we need to check the **lower** half of the player, so **ty2** will add the player's height

0 1 2 +



FUNCTION MOVE_PLAYER()

```
-- MOVE LEFT  
IF BTn(0) THEN  
    TX=PLYR.X-1  
    TY=PLYR.Y
```

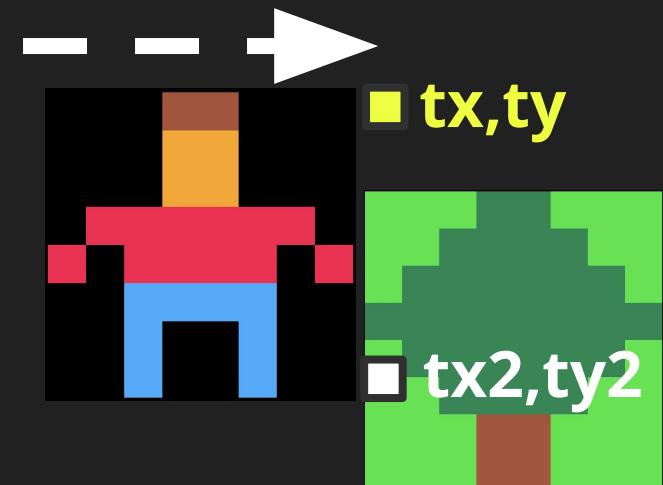
```
    TX2=TX  
    TY2=PLYR.Y+PLYR.H  
END -- END IF BTn(0)
```

```
-- MOVE RIGHT  
IF BTn(1) THEN  
    TX=PLYR.X+PLYR.W+1  
    TY=PLYR.Y
```

```
    TX2=TX  
    TY2=PLYR.Y+PLYR.H  
END -- END IF BTn(1)
```

LINE 10/98

316/8192 E



This **ty2** value will be the
same when moving **right**

0/12 +

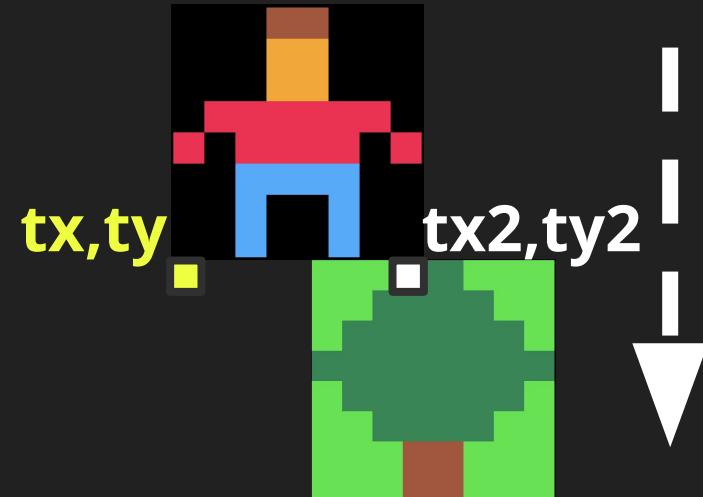


```
-- MOVE UP  
IF BTn(0) THEN  
TX=PLYR.X  
TY=PLYR.Y-1  
  
TX2=PLYR.X+PLYR.W  
TY2=TY  
END -- END IF BTn(0)
```

```
-- MOVE DOWN  
IF BTn(1) THEN  
TX=PLYR.X  
TY=PLYR.Y+PLYR.H+1  
  
TX2=PLYR.X+PLYR.W  
TY2=TY  
END -- END IF BTn(1)
```

LINE 39/98

316/8192 3



When moving **up/down**, we need to check the **right** side of the player, so **tx2** will add the player's **width**

0/12 +



```
-- CONVERT FROM PIXELS TO TILES  
-- DIVIDE BY 8 AND ROUND DOWN
```

```
TX=FLR(TX/8)  
TY=FLR(TY/8)
```

```
TX2=FLR(TX2/8)  
TY2=FLR(TY2/8)
```

```
-- MGET FINDS THE SPRITE NUMBER  
-- OF A TILE AT AN X,Y LOCATION  
-- BUT IT NEEDS THIS VALUE IN  
-- TERMS OF MAP TILES, NOT PX
```

```
TILE=MGET(TX, TY)
```

```
TILE2=MGET(TX2, TY2)
```

```
-- FGET FINDS WHETHER A FLAG  
-- IS TURNED ON FOR A SPRITE  
-- IT NEEDS THE SPRITE NUMBER
```

```
LIN 44/99
```

```
316/8192 E
```

We'll do the same conversion from tiles to pixels as we did for the first pair of coordinates

And then we'll need another variable for the sprite number of the second point we're checking

012 +

() [] {} < >

```
-- FGET FINDS WHETHER A FLAG  
-- IS TURNED ON FOR A SPRITE  
-- IT NEEDS THE SPRITE NUMBER  
-- AND FLAG NUMBER
```

```
IS_WALL=FGET(TILE,0)
```

```
IS_WALL2=FGET(TILE2,0)
```

```
-- ONLY MOVE IF NOT A WALL AT  
-- TARGET LOCATION
```

```
IF NOT IS_WALL
```

```
AND NOT IS_WALL2 THEN
```



```
-- MOVE LEFT
```

```
IF BTn(0) THEN
```

```
PLYR.X -= PLYR.SPD
```

```
END -- END IF BTn(0)
```

```
-- MOVE RIGHT
```

```
IF BTn(1) THEN
```

```
LINE 64/98
```

316/8192 E

We'll also need another **variable** for the true/false of the second point

And then we'll need to **add a condition** that the **second point is ALSO not on a wall tile** in our **if statement** allowing movement

0 1 2 +

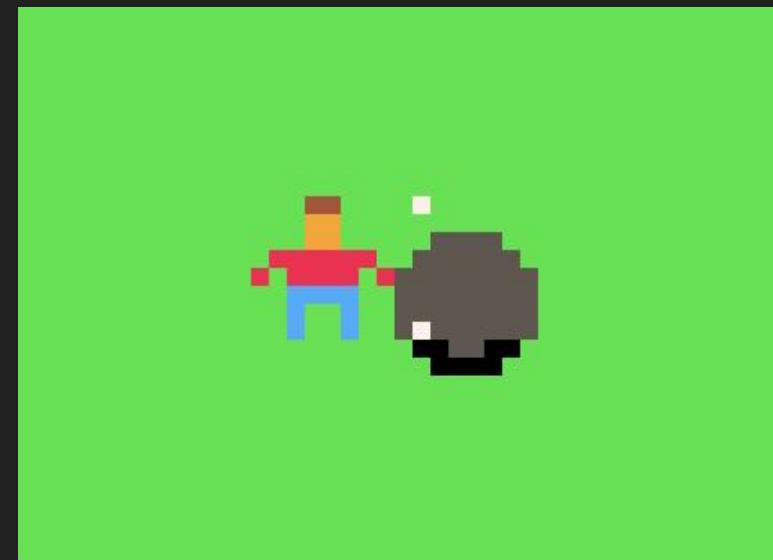
0 1 2 < >

```
-- LOOPS 30X/SEC
FUNCTION _UPDATE()
MOVE_PLYR() -- TAB 2
END -- END FUNCTION _UPDATE()
```

```
-- LOOPS 30X/SEC
FUNCTION _DRAW()
CLS() -- CLEAR SCREEN
MAP() -- DRAW MAP AT 0,0
SPR(PLYR.n, PLYR.x, PLYR.y)
```

```
-- DRAW TARGET X,Y
RECT(TX, TY, TX, TY, 1)
RECT(TX2, TY2, TX2, TY2, 1)
```

```
END -- END FUNCTION _DRAW()
```



To visualize this (optional), we can draw rectangles at the two points we're checking

0 1 2 +

() ← → ⏪ ⏹

```
-- CONVERT FROM PIXELS TO TILES  
-- DIVIDE BY 8 AND ROUND DOWN
```

```
x1=FLR(TX/8)  
y1=FLR(TY/8)  
x2=FLR(TX2/8)  
y2=FLR(TY2/8)
```



```
-- MGET FINDS THE SPRITE NUMBER  
-- OF A TILE AT AN X,Y LOCATION  
-- BUT IT NEEDS THIS VALUE IN  
-- TERMS OF MAP TILES, NOT PX
```

```
TILE=MGET(x1,y1)  
TILE2=MGET(x2,y2)
```

```
-- FGET FINDS WHETHER A FLAG  
-- IS TURNED ON FOR A SPRITE  
-- IT NEEDS THE SPRITE NUMBER  
-- AND FLAG NUMBER
```

LINE 57/98

323/8192 ⌂

Because the [rect\(\)](#) function uses **pixel coordinates**, and we *converted our tx,ty and tx2,ty2 to tile values*, I found it simpler to use separate variables for the converted values

I used **x1,y1** and **x2,y2**

```
-- only move if not a wall at  
-- target location
```

```
if not is_wall  
and not is_wall2 then
```

```
-- move left  
if btn(⬅️) then  
  plyr.x -= plyr.spd  
end -- end if btn(⬅️)
```

```
-- move right  
if btn(➡️) then  
  plyr.x += plyr.spd  
end -- end if btn(➡️)
```

```
-- move up  
if btn(⬆️) then  
  plyr.y -= plyr.spd  
end -- end if btn(⬆️)
```

```
-- move down  
if btn(⬇️) then  
  plyr.y += plyr.spd  
end -- end if btn(⬇️)
```

```
end -- end if not is_wall
```

Condition

Movement code

End Condition

Finally, to actually implement movement:

I placed the movement code inside the **if statement** that checks whether **is_wall** and **is_wall2** are *NOT* true

Implementing Collision with Map Tiles

- Again, you can make your life a lot easier by keeping character and map tile sprites the same size
- If you had a player sprite that was 24x24 pixels and a wall tile sprite that was 8x8 pixels, you would need to add a *third pair of target x,y values* to account for collision with the middle of the player

Learning Resources

Learning Resources

Top-Down Adventure Game Tutorial Playlist by Dylan Bennett on YouTube

1. Map Setup
2. Add Player
3. Player Movement
4. Camera
5. Keys
6. Inventory Screen

Learning Resources

[Top-Down Adventure Game Tutorial Playlist by Dylan Bennett on YouTube](#)

7. [Doors](#)
8. [Animated Tiles and Spike Traps](#)
9. [Game Over Screen](#)
10. [Reset and Package](#)
11. [Text](#)
12. [More Tile Types](#)

Function Reference

- [camera\(\)](#) Set the camera position
- [ceil\(\)](#) Round up to nearest integer
- [fget\(\)](#) Check for a flag on a sprite
- [flr\(\)](#) Round down to nearest integer
- [map\(\)](#) Draw the map
- [mget\(\)](#) Get the sprite number of a tile
- [mset\(\)](#) Set a tile to a different sprite

My Code Examples

From [_helloworld_02_topdown_adventure.zip](#)

1. [Tile-Based Movement and Map Collision](#)
2. [Item Collection](#)
3. [Lock & Key Systems](#)
4. [Positioning the Camera](#)
5. [Warping](#)
6. [Pixel-Based Movement and Map Collision](#)
7. [Pixel-Based Item Collection](#)



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

</lesson>

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