Changing map screens

You can copy this code to get a function that will copy over a map into the empty space of the $(0, 0)$ map

```
-- takes the map found at (xI,yI)
-- and draws it on the screen
-- this is different than map in
-- that it doesn't just draw it it
-- sets it
-- for the purposes of this program
-- we're going to be using map(0,0)
-- as our blank map
function setMap(xI,yI)
 for i=0,30 do
         for j=0,17 do
           mset(i,j,mget(30*xI+i,17*yI+j))
         end
 end
end
```

Links

- The main TIC-80 website `https://tic.computer`
- TIC-80 wiki `https://github.com/nesbox/TIC-80/wiki`
- A small but fully featured platformer `http://clarissalittler.github.io/platformer.tic`
- A small racing game with a level select `http://clarissalittler.github.io/driversers.tic`

Making games in TIC-80

Contents

- The width of the other square
- The height of the other square

This function works by testing whether they *can't* overlap and then returns the opposite.

This is calculated as

- Is the right edge of the first rectangle to the left of the left edge of the other?
- Is the left edge of the first rectangle to the right of the right edge of the other?
- Is the bottom of the first rectangle above the top of the other?
- Is the top of the first rectangle beneath the bottom of the other?

If any of these are true then they *can't* intersect, but if none of them are true then they must.

```
function collision(x1,y1,w1,h1,
                   x2,y2,w2,h2)
 local notCollided = x1 + w1 < x2 or
                     x1 > x2 + w2 or
                     y1 + h1 < y2 or
                     y1 > y2 + h2
 return not notCollided
end
```

Copy and use this function as you need!

```
-- how many squares to the right to include in sprite
-- (optional) height,
-- how many squares down to include in the sprite
spr(10,50,50)


-- animating a sprite
-- this means that every 30 frames (half second)
-- alternating between sprite 1 and sprite 2
spr(1+(t%60)//30,50,50)
t = t + 1
```

Moving a sprite

```
player = {x = 50, y = 50}
function TIC()
 cls(0)
 if btn(0) then player.y = player.y - 1 end
 if btn(1) then player.y = player.y + 1 end
 if btn(2) then player.x = player.x - 1 end
 if btn(3) then player.x = player.x + 1 end
 spr(1,player.x,player.y)
end
```

Hit boxes

Here we include a function to calculate whether two boxes actually are intersecting. This functions takes

- The upper left-hand corner coordinates of one square
- The width of one square
- The height of one square
- The upper left-hand corner coordinates of the other square

Introduction to TIC-80

TIC-80 is a *fantasy console*, an all-in-one system for making 8-bit looking retro games. In TIC-80 you can draw your sprites, design maps, create sound effects, and score music in addition to writing the code that makes your game run.

This guide will help you make a small game in TIC-80 and learn a little bit more about Lua, the programming language TIC-80 uses.

TIC-80 mini-ref

`TIC()` function

Every TIC-80 program needs a function called `TIC` defined.

```
function TIC()
 cls(15)
 print("A tiny game",50,50,0)
 if btnp() ~= 0 then exit() end
end
```

Drawing text to the screen

```
-- arguments to print()
-- string to print
-- (optional) x position
-- (optional) y position
-- (optional) color
print("This is my message to the world",20,20)
```

Drawing a sprite to the screen

```
-- arguments to spr()
-- id of sprite to print
-- (optional) x position
-- (optional) y position
-- (optional) colorkey,
-- the color that should be transparent as the sprite moves
-- (optional) scale, makes the sprite bigger or smaller
-- (optional) flip
-- (optional) rotate
-- (optional) width,
```

Using tables like objects

```
-- make an object with properties
player = {hp = 5, x = 80, y = 30}
-- get those properties
player.hp = player.hp - 1
```

Repeating yourself

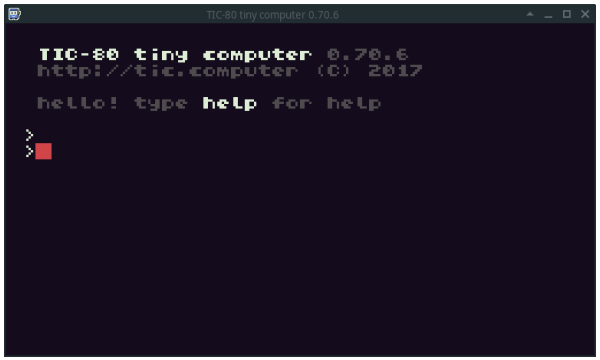
```
-- print the numbers 0 through 10
-- down the screen
for i=0,10 do
 print(i,10*i,10*i)
end
-- count backwards
for i=10,0,-1 do
 print(i,10*(10-i),10*(10-i))
end
```

Looping over a list

```
ourList = {10,20,30,40}
-- i is the place in the list
-- n is the item in the list at the i'th place
for i,n in ipairs(ourList) do
 print(i.."th element: "..n,10*i,10*i)
end
```


First steps

When you're loading up TIC-80 for the first time you should see a blinking cursor on a blank line. This is the command line!



1. Type `demo` and hit "Enter"
2. Type `ls` and hit "Enter" to see all the games loaded
3. Type `load fire.tic` to load the demo
4. Type `run` to run it
5. Move the fire with the arrow keys and watch the smoke!
6. Hit `Esc` twice in order to enter the game editor
7. Click around on the tabs to look around the different parts of the game editor

Starting a game

If you want to start a new game you can type `new` and hit "Enter".

If you run the game you'll see something like

You can now start editing this game!

Saving a game

Saving at command line Type `save` or `save GAMENAME` to save the game with a new name

Saving while in game editor Type `Ctrl-s`

Saving the cartridge file Type `folder` and then transfer the `.tic` file to the storage of your choice

Learning Lua

Variables

```lua
-- make variables by setting them
highScore = 10
-- use variables with their name
print(highScore)
```

Functions

```lua
-- use functions by giving their arguments in parentheses
spr(0,50,50)
-- define functions like this
function myFun(arg1,arg2)
  print("This function is pointless")
  return arg1 + arg2
end
```

Using tables like lists

```lua
-- make an empty table
myTable = {}
-- make a table with stuff in it
myTable = {2,3,5,7,11,"stuff",{1}}
-- get things out of the list
myTable[1] -- returns 2
#myTable -- length of list
myTable[#myTable] -- returns last item
table.insert(myTable,"heck") -- adds something to the end
myTable[#myTable] = nil -- removes the last item
```

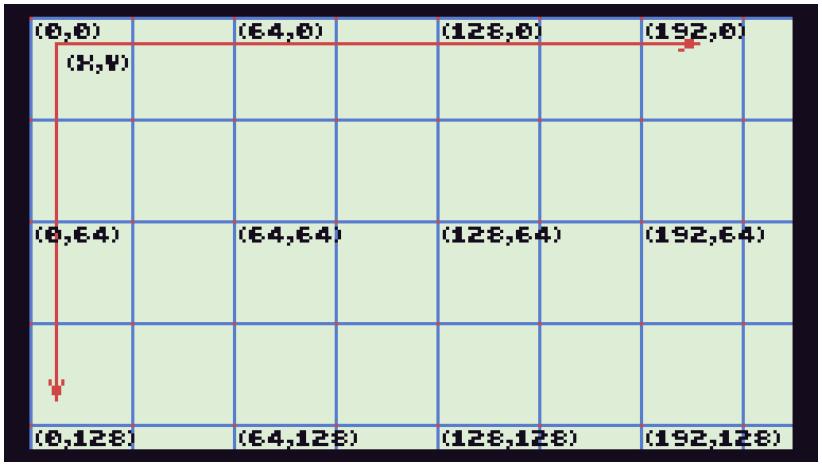
Screenshots and concept art

Now that you've brainstormed your concept a bit, it can be helpful to try and draw examples of what different characters and levels might look like. Take the space below to sketch that out!

TIC-80 technical details

Screen layout

The screen is 240 pixels by 136 pixels. The upper-left corner is $(0, 0)$ and the bottom-right is $(239, 135)$



Framerate

The `TIC` function runs at a *constant* 60fps.

Button layout

Action	Button ID	Keyboard
Up	0	Up arrow
Down	1	Down arrow
Left	2	Left arrow
Right	3	Right arrow
A	4	Z key
B	5	X key
X	6	A key
Y	7	S key

Designing a game

Brainstorming

The first thing you need to do when designing a game is pick a genre or concept for the game. TIC-80 is well-suited to making 2d retro games. Things like

- Shoot-em-'ups
- Endless runners
- Racing games
- Puzzle games
- Platformers
- ...

What kind of game do you want to make? Take the space below to try and write out some ideas for a game!

☐ **Hitting the ground**

☐ Draw sprites for the ground

☐ Use the map editor to draw in the ground

☐ Add in a call to `map(0,0)` just below your call to `cls`

☐ Check if your sprite is touching the ground

☐ Add an if-then-else around your code that increases `vy` HINT: it'll look something like

```
if ...  then
 player.vy = player.vy + 0.1
else
 ...
end
```

☐ After the "if" but before the "then" add in a check of

```
mget(x//8,y//8) ~= GROUNDSPRITEID
```

☐ In the "else" clause set `vy` to 0

Memory layout

TIC-80 works by pretending to be an old console from the 80s. It even has its own memory that you can manipulate in code to modify the game. Here's the memory map straight from the TIC-80 wiki.

This is here as an advanced reference and not something to try and understand on your first try through

ADDR	INFO	SIZE
00000	SCREEN	16320
03FC0	PALETTE	48
03FF0	PALETTE MAP	8
03FF8	BORDER COLOR	1
03FF9	SCREEN OFFSET	2
03FFB	MOUSE CURSOR	1
03FFC	...	4
04000	BG SPRITES	8192
06000	FG SPRITES	8192
08000	MAP	32640
0FF80	GAMEPADS	4
0FF84	MOUSE	4
0FF8C	...	16
0FF9C	SOUND REGISTERS	72
0FFE4	WAVEFORMS	256
100E4	SFX	4224
11164	MUSIC PATTERNS	11520
13E64	MUSIC TRACKS	408
13FFC	MUSIC POS	4
14000	...	0

Walkthrough: "Hello World"

Now let's learn the tools a bit by slowly going through how to modify the default "Hello World" program that every game starts with. We'll start by explaining, line by line, what this program does, how it works, and then we'll get to making changes in the next section.

The "Hello World" program looks like

```
-- title:  game title
-- author: game developer
-- desc:   short description
-- script: lua

t=0
x=96
y=24

function TIC()

        if btn(0) then y=y-1 end
        if btn(1) then y=y+1 end
        if btn(2) then x=x-1 end
        if btn(3) then x=x+1 end

        cls(13)
        spr(1+t%60//30*2,x,y,14,3,0,0,2,2)
        print("HELLO WORLD!",84,84)
        t=t+1
end
```

First we have these four lines at the top:

```
-- title:  game title
-- author: game developer
-- desc:   short description
-- script: lua
```


☐ **Sprites with gravity**

- ☐ Add a `vy` property to the player table to keep track of gravity
  - ☐ Make sure it starts at 0
- ☐ Change your up-arrow code so it sets `vy` to a negative number
- ☐ Add a line of code so that every frame the y-property is changed by `vy`
- ☐ Add a line of code so that every frame `vy` increases by a small number (like 0.1)

Exercise: Starting out with sprites

☐ **Controlling a sprite**

- ☐ Draw a sprite in the sprite-making tools
- ☐ Add code to your `TIC` function to draw the sprite to the screen
- ☐ Add a new variable for the state of the player
- ☐ Set this variable to a table that has properties for
  - ☐ x-position
  - ☐ y-position
- ☐ Change your drawing code to use the x and y from the player table
- ☐ Add code to change the player table's position when you press the arrow keys
  - ☐ Left arrow – `btn(2)`
  - ☐ Right arrow – `btn(3)`
  - ☐ Up arrow – `btn(0)`
  - ☐ Down arrow – `btn(1)`

These are *comments* and don't affect the program when it runs. They're mostly for communicating with other people reading your code! Comments always start with a `--` and then the rest of the line of code is going to be ignored when the program is run.

The next three lines

```
t=0
x=96
y=24
```

are *setting variables*. Variables in Lua, like most programming languages, are for giving data a name and a place to live. In this case, these three variables going to be used for

- the time elapsed since the beginning of the program, measured in frames
- the horizontal position on the screen, measured in pixels
- the vertical position on the screen, also measured in pixels

Next we're defining a *function*, which is a chunk of code that can be run over and over again easily. This is a very important function! The `TIC` function is your main game loop and it runs at a set 60 times per second.

```
function TIC()
 ...
end
```

Now let's move onto the inside of the function.

```
if btn(0) then y=y-1 end
if btn(1) then y=y+1 end
if btn(2) then x=x-1 end
if btn(3) then x=x+1 end
```

These lines are how we're controlling the sprite, allowing it to move when we press the arrow keys on our keyboard. You can read the first line as

> If button 0 is pressed, then subtract 1 from the y variable.

Since "button 0" is the up key and the vertical position gets *smaller* as you approach the top of the screen this really means

> If the up key is pressed, then move up one pixel

Similarly, the 1 button is the down key and the 2 and 3 keys are left and right respectively.

The next line

```
cls(13)
```

is *clearing the screen*. You have to do this every single frame! To understand why, it's a good exercise to try *commenting* the line out with two dashes and then running the game. Try moving around.

You should see that your little robot is leaving a smear on the screen as it moves because the result of every previous frame of the game is still on the screen.

The `13` in

```
cls(13)
```

is the color we're filling each pixel with.

- ☐ Use the `%` operator and addition to reset the frame of animation to the starting frame if it reaches the end of the loop

☐ **Add variables for changing the motion of the text on the screen**

- ☐ Add a `vx` variable and set it to `math.rand(-1,1)`
- ☐ Add a `vy` variable and set it to `math.rand(-1,1)`
- ☐ Add a `textX` variable and set it to the starting x-position of the text
- ☐ Add a `textY` variable and set it to the starting y-position of the text

☐ **Change the code that writes the text to the screen and make it use the `textX` and `textY` variables**

☐ **Add in code that changes the x and y *position* by the `vx` and `vy` variables respectively**

☐ **Whoops it flies off the screen! Fix that by adding if statements to change `vx` and `vy` from positive to negative (or visa versa) if it goes off the screen**

Exercise: Changing "Hello World"

☐ **Erase the old hello world sprites**

☐ **Draw three frames of animation using the drawing tools**

  ☐ Extra credit: change the palette first!

☐ **Add a variable to the top of the program called `aniFrame`**

  ☐ Set `aniFrame` to the index of your first frame of animation

☐ **Change the call to `spr` to use `aniFrame` in its first slot**

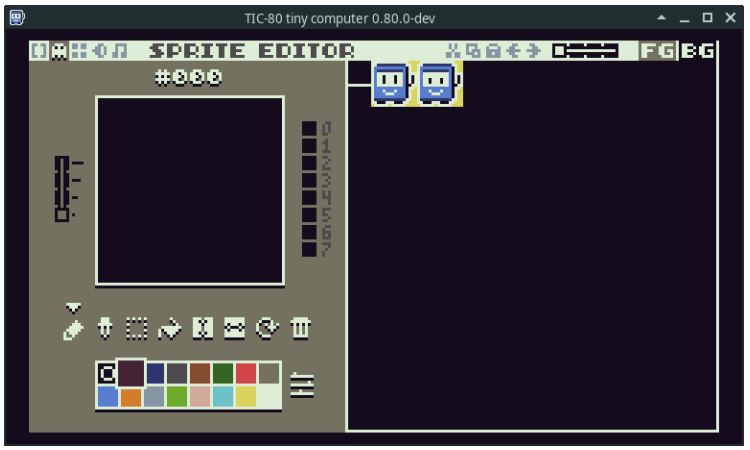
  ☐ You'll need to delete the last two arguments to the function so it doesn't grab too many tiles

☐ **Add an if-statement that will change the frame of animation accordingly**
hint: you'll want code just below the call to `spr` that looks like

```
if (t % 60 > 40) then

elseif (t % 60 > 20) then

else

end
```

  ☐ Add a line of code inside the if-statement that adds one to the frame

This is a good time to try clicking on the little ghost-like icon near the top of the screen, this takes you to the sprite and palette editing screen.



The colors at the bottom of the screen are the *palette*. Much like really old video game systems and computers, you can only have a small number of colors in use at a time. In TIC-80 you have a total of 16, which are indexed as 0–15.

Sure enough, if you count up from zero you'll find that thirteen is the light blue background.

> What do you think happens if you move the `cls(13)` to below the rest of the code in the `TIC` function? If you're not sure, then try it!

Now we're onto the most dense line of code in the "Hello World" program

```
spr(1+t%60//30*2,x,y,14,3,0,0,2,2)
```

`spr` is the function you call to draw a *sprite* to the screen. So this is the line of code that puts our little robot dude at the right position *and* animates them. This is our first function with multiple arguments passed into it. Specifically, there's *9* total. Let's list them out one by one.

Arguments to `spr`

1. The id of the sprite, or if it's a sprite made up of multiple tiles the id of the upper-left corner
2. The x-position at which to place the upper-left corner of the sprite
3. The y-position at which to place the upper-left corner of the sprite
4. The color-key, which lets you make parts of a sprite transparent like a green screen
5. The scaling factor of the sprite, in this case we're making it 3x larger
6. Whether to flip the sprite
7. How to rotate the sprite
8. How many tiles horizontally to include
9. How many tiles vertically to include

Now we just need to explain the expression

```
1+t%60//30*2
```


.

So I can tell you that this expression is either 1 or 3, switching everything half-second. This is how our animation works!

It's a bit hard to read if you've never programmed before, so instead let's rewrite this as

```
if t % 60 > 30 then
 frame = 3
else
 frame = 1
end
spr(frame,x,y,14,3,0,0,2,2)
```

using the if-statement we'd seen earlier. The only other piece to explain is that the % operator, pronounced "modulo", divides one number by another and gives back the *remainder*. This means that as we count up in time from 0 to 59 over and over again, switching every 30 frames—which means every half-second like I promised above!

The next line of code

```
print("HELLO WORLD!",84,84)
```

is what puts the text on the screen, at the x and y position in the second and third argument.

The final line

```
t=t+1
```

is what increments the `t` variable each frame so that we can keep track of our animation. That's the "Hello World" program line by line and you've actually seen a good number of programming concepts just here alone!