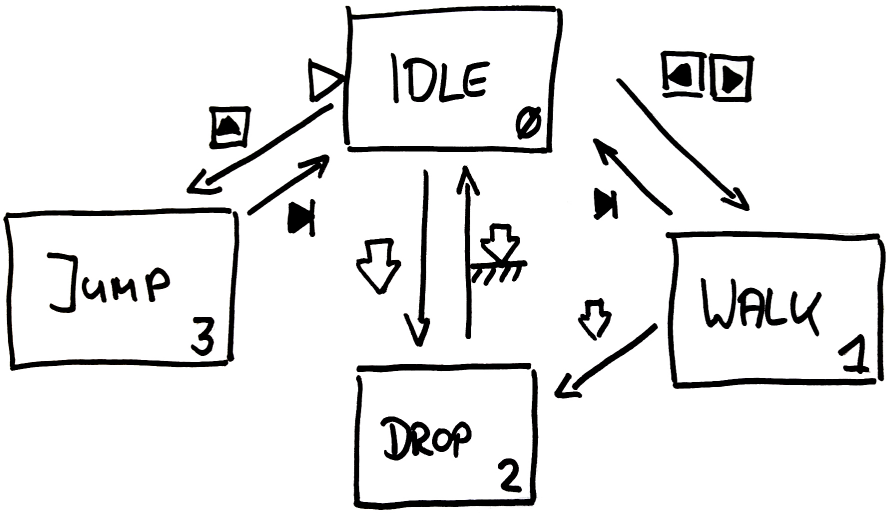
PicoJump

Platformers have appeal. They are quite literally the pixel-manifestation of the Hero's Journey and have shaped the gaming landscape over for decades. I grew up with an Amiga 500 and games like Turrican, Prince of Persia and The Shadow of the Beast! Well, and let's not forget about Sonic and Super Mario, right?

This tutorial assumes you know the basics of PICO-8 programming, how to draw sprites, the `_init()`, `_update()` and `_draw()` functions and how to write simple programs with them.

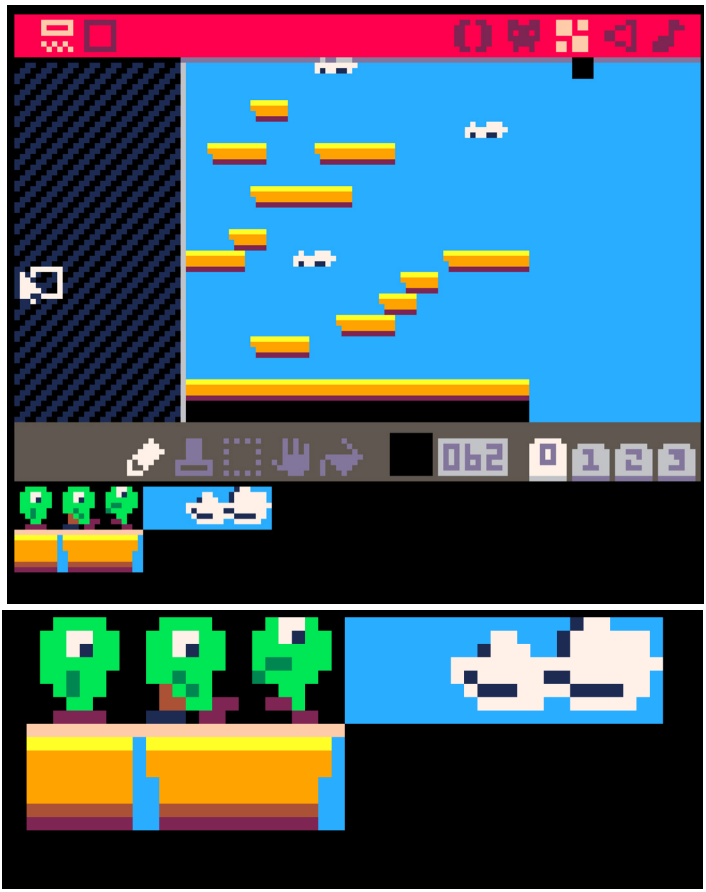
At first it is always good to think about what we want the character (I think I will name him Tutorial-Bob) in our platformer game will be able to do. For the time being, let's start with walking, jumping, falling and being idle. To model this behaviour, we will make use of a mechanism generally used in computing and engineering. We will build a finite state machine or in short FSM. And in our case it looks like this: the boxes represent the states out player can be in. The arrows between them are the transitions, which - when a certain condition is met - will point to the next state Bob will be in.



For example, if Bob walks along and suddenly there is no more ground, he will start falling. Once he hits ground again, he will stop falling and be idle. Or if the user presses the left or right arrow, Bob will start walking. When we have made up our state machine, it is very straightforward to start writing code.

Before we begin, here some sprites and a little world in preparation for the game. Any map sprite that we want to use as a ground Bob can stand on, is tagged with the first sprite flag being set to True. This way we can easily determine with what to collide using `FGET()` on the map tile.

Lets start writing the program. Here the player initialization and drawing routines for the game.



```
function _init()
  px=20  -- x-position
  py=64  -- y-position
  pstate=0 -- current player state
  pspr=0  -- current sprite
  pdir=0  -- current direction
  pat=0 -- player state timer
end

function _draw()
  -- draw the world
  map(0,0,0,0,16,16)
  -- draw the player, we use dir to mirror sprites
  spr(pspr,px,py,1,1,pdir==-1)
end
```

In the `_init()` method we are setting a whole lot of variables to keep track of Bob.
We have `px` and `py` for the world position in pixel, `pdir` is for the direction we are looking at, we have `pspr` to store the current sprite and a `pstate` for the active state itself.
Look at the state machine graph again. See the numbers in the corner of the boxes? This is the identifier for each state which will be assigned to `pstate`.
We also hold a variable called `pat`, which is the state counter or sometimes called state clock. Every time we run `_update()` that counter will increase by one. On every change of state, the counter will be reset to zero. This way we always know how long bob has been in his current state and use that information for animation and movement calculations.

Before we write the state machine behaviour, we need one more helper function to tell us, if Bob is hovering in the air (`true`) or standing on the ground (`false`).
Be aware that `mget()` wants map coordinates, so we have to divide the player position by 8.

```
function canfall()
  -- get the map tile under the player
  v=mget(flr((px+4)/8),flr((py+8)/8))
  -- see if it's flagged as wall
  return not fget(v,0)
end
```

Now we get to the state machine part.
Each time we move from one state to another, we have to set the **pstate** variable to the new state id and also reset the **pat** counter. Let's do this in a dedicated function. Later on, when things get more complex, we could also implement state OnEnter behaviour in here. For now, this is beyond the scope of this tutorial and our function just looks like this.

```
function change_state(s)
  pstate=s
  pat=0
end
```

Our **_update()** function will deal with all the actual state behaviour and starts like this

```
function _update()
  b0=btn(0) -- button0 state
  b1=btn(1) -- button1 state
  b2=btn(2) -- button2 state

  px=(px+128)%128 -- no bounds left and right
  pat+=1 -- increment state clock
```

We capture what buttons are pressed by the user and also make sure that when Bob leaves the screen on one side, he will come back in on the other. Also notice, that we increment the **pat** on every call of **_update()**

Next, we gonna implement the four states, one by one.
For each state we will write what is happening to Bob while he is in the state, as well as define the conditions under which we are transitioning into another behaviour.

```
-- IDLE STATE
IF PSTATE==0 THEN
  PSPR=0
  IF (B0 OR B1) CHANGE_STATE(1)
  IF (B2) CHANGE_STATE(3)
  IF (CANFALL()) CHANGE_STATE(2)
END
```

If Bob is in the Idle state, we set the current sprite to idle. This is pretty much all that happens here.
The three if-statements check for user input and send Bob into the walking, jumping or falling state according to our diagram above.

```
-- WALK STATE
IF PSTATE==1 THEN
  IF (B0) PDIR=-1
  IF (B1) PDIR=1
  PX+=PDIR*MIN(PAT,2)
  PSPR=FLR(PAT/2)%2

  IF (NOT (B0 OR B1)) CHANGE_STATE(0)
  IF (B2) CHANGE_STATE(3)
  IF (CANFALL()) CHANGE_STATE(2)
END
```

The walk state is a little more elaborate. Based on what button the user pressed, we set the sprite direction. We also increment or decrement Bobs x position **PX**. Note that we are using **PAT** to make him move just one pixel in the first tick of the state and then two in any following, to create a sense of acceleration. We also set the current sprite alternating between 0 and 1 based on the **PAT** again. See that I also have divided the **PAT** by 2 two to slow down the sprite change to not get too flickery. This - again - is followed by the transitions into falling, jump and idle.

The state implementation for falling looks a bit more complicated, as it has to deal with collisions and intersections.

```
-- FALL STATE
IF PSTATE==2 THEN
  PSPR=2
  IF (CANFALL()) THEN
    IF (B0) PX-=1 -- STEER LEFT
    IF (B1) PX+=1 -- STEER RIGHT
    PY+=MIN(4,PAT) -- MOVE THE PLAYER
    IF (NOT CANFALL()) PY=FLR(PY/8)*8 -- CHECK GROUND CONTACT
  ELSE
    PY=FLR(PY/8)*8 -- FIX POSITION WHEN WE HIT GROUND
    CHANGE_STATE(0)
  END
END
```

Inside the fall state, we check for ground contact - which is the only transition out of here into the idle state.
If we are falling, we allow the player to move left and right to steer the fall.
Like in the real world, with every tick falling, we accelerate to fall a little bit faster. This is done by adding the **PAT** to the y-position of Bob, that speed is capped at a terminal velocity of 4.
In case Bob is hitting a ground tile, we need to make sure to fix his position back on top of a tile.
The mechanism used here is kept rather simple, but works. We just round the y-position back to the tile under Bob. That way we cannot get stuck halfway inside the ground

The last state is the JUMP state. It works similar to the FALL state.

```
-- JUMP STATE
IF PSTATE==3 THEN
  PSPR=2
  PY-=6-PAT
  IF (B0) PX-=2
  IF (B1) PX+=2
  IF (NOT B2 OR PAT>7) CHANGE_STATE(0)
END
```

And let's not forget about this one

```
END -- END OF THE UPDATE FUNCTION
```

Now you have a character driven by a simple state machine. The movement is still a bit jerky and we can't run, shoot, duck or slide.
But the biggest benefit of an FSM is, it is easy to add other states and transitions and also more complicated math to smoothen out the animation of the character later on.

Happy Coding!

Johannes Richter
http://www.lexaloffle.com/bbs/?tid=2520