PICO-8 ZINE #3

CODE-PLAY-SHARE-BE-THINK-DESTROY-LEARN-BREAK-LOVE-MAKE

DECEMBER 2015

CONTENTS

HACK-RUN-LEARN
SHARE-LOVE-PLAY
CODE-CREATE-DRAW
MAKE-DESIGN-BE
THINK-WRITE-BREAK
PARTICIPATE-RETRY

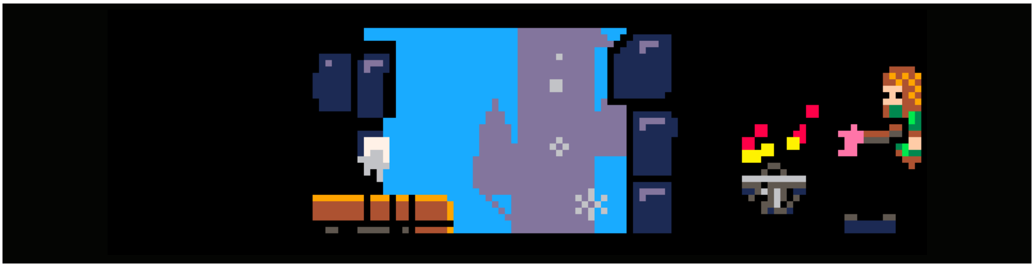
PICO-8 is a fanzine made by and for PICO-8 users.
The title is used with permission from Lexaloffle Games LLP.
For more information: www.pico-8.com
Contact: @arnaud_debock
Cover illustration by @JUSTIN_CYR
Special thanks to @dan_sanderson and @lexaloffle

MINIMALISM PAYS DOUBLE

What do you love most about PICO-8?

For me, it's the constraints. A blank canvas can be paralyzing. It's easy to feel the love with which PICO-8 limits you. When we "play" this wonderful little console, we hope to pass some of that love over to our audience. It's counter-intuitive, but it seems to me that enabling creativity is as much about setting smart constraints as it is about breaking down barriers.

When we started my local multiplayer project TowerFall, I chose a lot of constraints to help focus our work. For example, I decided we would work with Game Boy Advance inputs (A, B, L, R, and d-pad) and resolution (320x240). In those early days I didn't know what "the point" of TowerFall was. I was asking myself a lot of big questions, like "Why do I love local multiplayer, and how do I explore those concepts with my game's design?" Digging like this through your prototype and yourself is tough work. Constraints force you to confront tough decisions about what deserves to live in the limited design space you have.



Some amount of enforced simplicity can help us create more effectively, but more importantly this approach bleeds into the player's experience. In a local multiplayer game you often want people to feel comfortable jumping in with very little ceremony or introduction - worse yet, with spectators! - so minimalism pays double. Our goal with TowerFall became facilitating creative play, giving players the confidence to experiment with very little knowledge.

If you're using PICO-8, you probably know this. You must already love the colorful fence that circles its playground. I just want to confirm that the minimal design philosophy of the console extends beautifully to local multiplayer design, perhaps more than most genres.

Have fun (:

-Matt
www.mattmakesgames.com
@MattThorson

@JctWood

NOTES ON CREATING SUCCER

Succer (http://www.lexaloffle.com/bbs/?tid=2614) is an old-school soccer game inspired by Kick-off and Sensible Soccer.

Making a local multiplayer sport game

How to decide which player is human controlled? Simple : take the one closer to the ball. A better method may be to take the ball’s velocity into account.
sidenote: measuring distance in a fixed float 16 bits format can be quite painful... My solution : use Manhattan distance to avoid the power and sqrt when precision is not necessary... It was suggested to me that I could use smaller units when doing such computations, but I’m not that smart :D

```
FUNCTION MANHATTAN(A,B)
  RETURN ABS(B.X-A.X)+ABS(B.Y-A.Y)
END
```

It may be a good idea to be able to locate the controlled player when he is outside the screen. In an earlier version of the game when there was no AI to move uncontrolled players, that was still possible. At that time, I used an indicator drawn on the border of the screen in the direction of the controlled player. The size of the indicator was relative to the distance to the border of the screen: the closer the bigger.

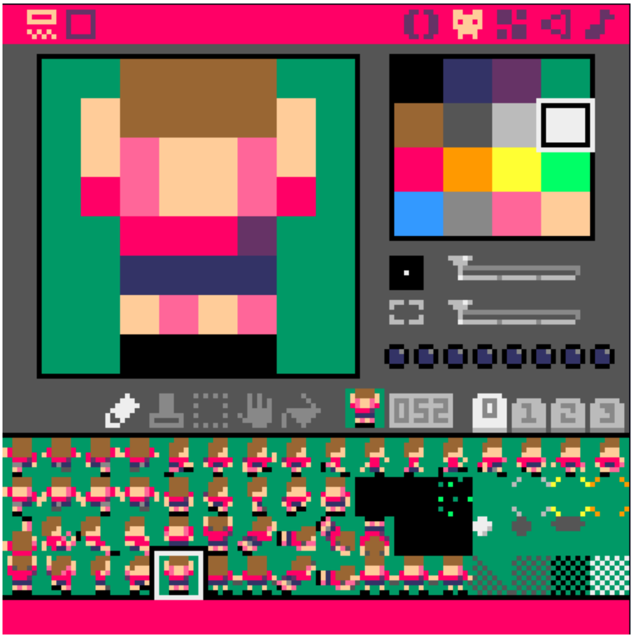
To avoid having to draw too many sprites, use pal to change the player's jersey colors.

Controls

I really wanted the game to feel like Kick-Off and Sensible Soccer. That’s the reason the game only uses one button for all actions.
I have fond memories of me and my friends running around the ball trying to control it without much success. So at first, I made the ball not sticky at all. The only way to control it was pushing it (or kicking it.)

But as I let people play early versions of the game, the feedback was clear: Control was a nightmare.

So, I spent a little time playing the original Sensible Soccer to reflect on the control scheme, and surprisingly as I rediscovered the game after so many years, controls were much more friendly than I remembered!
So, I added ball stickiness by lerping the ball position to be in front of the player that controls it by a small amount (20%). After that change, the game was much more enjoyable. Even the AI became less stupid as it was able to recover a ball running to the touch when it would have just pushed it to the exit before.



AI

Even in two player mode, a soccer game needs AI to control all the other little footballers around!

The AI is made of simple rules:
The closest player to the ball is the main player.
The main player AI is:
1. Try to get the ball.
2. If he has the ball, try to shoot to the goal.

3. If he is too far away, try to pass the ball to a teammate.
4. If he can't pass, just dribble to the goal.

This high level behavior is implemented using simpler functions like RUN_TO which makes a player run to a specified position.
This function acts like a gamepad input adding to the velocity of the player in the wanted direction until it reaches the maximum speed value.
It is called each frame until it returns TRUE when the player is within a minimum distance to the desired location.

The other ones are just supporting, trying to get to a helpful position. There are two formations in the game that define the wanted location of each player in one team. The first formation contains the positions to use during the kick-off. These are offset for the team with the ball in order to align the players with the middle of the field.
The other formation contains the positions the players try to reach relative to the ball. Those positions are clamped to avoid players wandering outside the field.

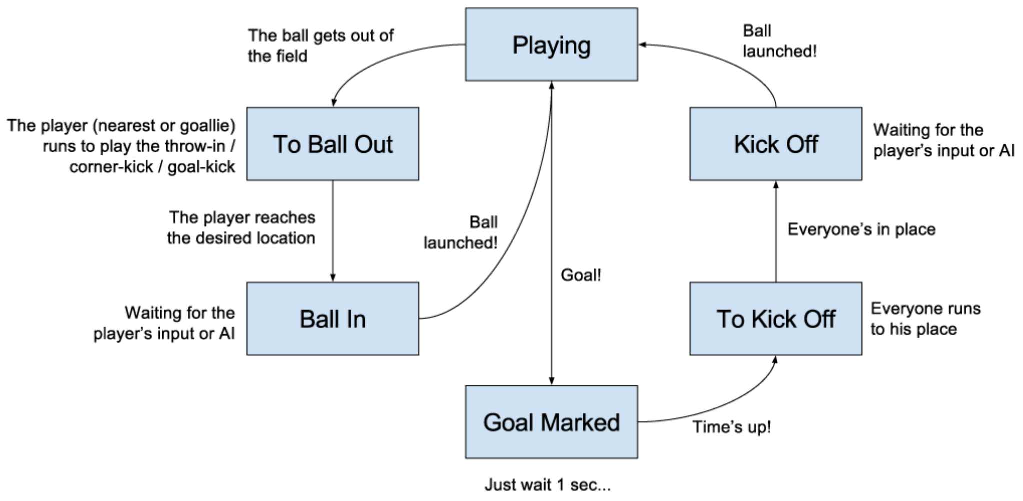
There are special cases for the other phases of the game such as playing a throw-ins, goal-kicks or corners. The positions are offset and the point of interest moved from the ball to ensure a better placement (in the center of the field for a goal-kick, near the goal for a corner and halfway between the position of a throw-in and the median vertical line.

Finite-State Machine (FSM)

(see picozine 2 for an introduction or just google it!)
FSM are everywhere! The game state management uses them. They're put to use to control the match phases.
The little men running all over the place use them.

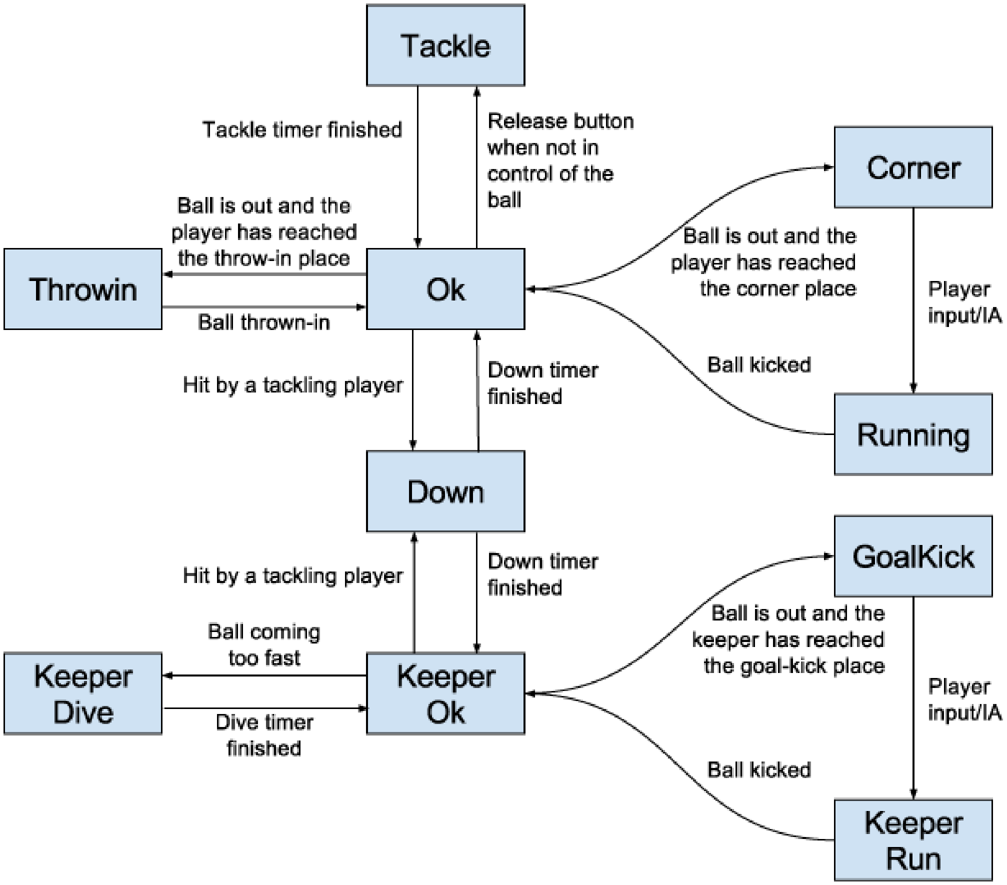
Here is the game FSM :
Boxes are the states, arrows are the transitions between states.
Each of this states defines an Init and Update function. The Init function is called whenever the game changes state to prepare the state execution.

The update function is called each frame to apply the state's behavior. For example in the "Goal Marked" state, the Init function resets a timer and the Update function handles the timer increment and checks if its limit has been reached in which case it transitions to the "To Kick Off" state.


Playing
The ball gets out of the field
Ball launched!
To Ball Out
The player (nearest or goallie) runs to play the throw-in / corner-kick / goal-kick
Kick Off
Waiting for the player's input or AI
The player reaches the desired location
Ball launched!
Everyone's in place
Goal!
Ball In
Waiting for the player's input or AI
To Kick Off
Everyone runs to his place
Goal Marked
Time's up!
Just wait 1 sec...

Note: Throw-in, corner kick and goal kick share the same state. It's not the cleanest way to handle those situations, but they are similar enough to re-use the same state and it saves a lot of tokens.
The player can be in these states depending if it is a goal keeper or field player : Only the down state is common to both.
A player state is made of 4 functions:
- A Start function called whenever a player enters this state to setup things like the timer for example.
- An AI function is used to specify what the behavior of the player will be when considered the main player.
- An Input function is used to specify which player should respond to the player's input (when considered the main player).
- All the other players use a more generic behavior which just tries to give them an interesting position and takes into account the game state.

The Corner state and the GoalKick state are exactly the same. But they're kept separate as they are used to define what kind



of ball exit has happened. I don't think I could have saved a lot of tokens by not duplicating these states as the logic involved surely would have added more tokens and the duplicated parts are already defined in functions, so there is not so much duplication after all.

General notes

8192 tokens isn't that much... Cramming in all soccer's rules and all that AI is very tricky...

Here are a few hints: Leveraging global/object variables : example: if an object variable needs to be referred to using the object, make it a global:

```
 MATCH.TIMER = 0
...
LOCAL FIRST_HALF = MATCH.TIMER<45
MATCH.TIMER+=1
LOCAL SECOND_HALF = MATCH.TIMER>=45
```

becomes :

```
  MATCHTIMER = 0
...
LOCAL FIRST_HALF = MATCHTIMER<45
MATCHTIMER+=1
LOCAL SECOND_HALF = MATCHTIMER>=45
```

and saves 8 tokens!

On the opposite, having variables embedded into object can also save tokens: For example, 2D coordinates are stocked into objects and these objects are passed around to some vector math functions. If those functions are called several times it becomes token saving efficient and worth the cost of the function declaration :

```
A = {X=0,Y=0,...}
DOT(A,B)
```

instead of

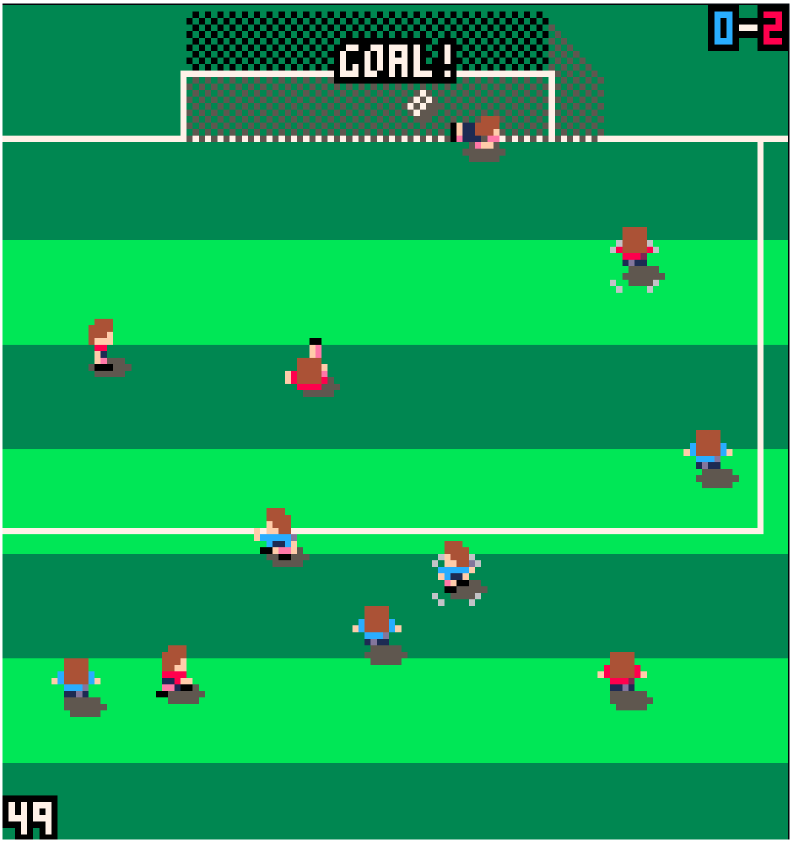
```
AX=0 AY=0 BX=0 BY=0
AX*BX+AY*BY
```

Finally, a good way to save tokens is to just cut what is not useful. One of the first things I did in the game was a particle system that spawned puffs of smoke whenever a player kicked the ball. It was fun and all, but as the camera system took shape, the particles became hardly visible. I could have made them more present by changing their size, colors, lifetime, whatever, but I think it would not have helped the action’s readability. So, I just squeezed them.

Conclusion

That's all! But, remember: There's no fault system implemented so don't hesitate to tackle other players! Have fun!

Laury MICHEL
@RylauChelmi



DOM8VERSE

In this tutorial, you will learn how to shoot bullets, those bullets will move and detect collision to be destroyed by a wall.

1 Define the bullet skeleton

First of all, we define a function that creates a bullet. We won't define a global variable at the start of the program for the bullets, because they will be created on the fly when the player shoots. To do that, let's define a function that creates an object that represents a newly created bullet with all the attributes that this bullet will need.

```
BULLETCONSTRUCT = FUNCTION(X, Y)

  LOCAL OBJ = {}
  --AN ARRAY CONTAINING X AND Y POSITION
  OBJ.POSITION = {X=X, Y=Y}
  --THE SPRITE NUMBER USED TO DRAW THE BULLET
  OBJ.SPRITE = 3

--DEFINE AN 'UPDATE' FUNCTION THAT WILL BE CALLED BY THE PROGRAM
  OBJ.UPDATE = FUNCTION(THIS)
    --MOVE THE BULLET TO THE RIGHT
    THIS.POSITION.X += 1
  END

  --RETURN THE BULLET
  RETURN OBJ
END
```

2 Shoot the bullets

This code is just a definition, you have to call it in order to create a bullet, let’s say that the player has to hit a button to create a new bullet.

```
--ARRAY WITH ALL OBJECTS PRESENT IN THE GAME
OBJECTS = {}
FUNCTION _UPDATE()
  --FIRST PLAYER PRESS THE SHOOT BUTTON
  IF BTNP(5, 0) THEN
  --CREATE A NEW BULLET AND ADD IT IN THE 'OBJECTS' OF THE GAME
  --WE PASS THE PLAYER POSITION AS A PARAMETER, SO THE BULLET WILL
  --APPEAR AT THE PLAYER POSITION
  ADD(OBJECTS, BULLETCONSTRUCT(PLAYER1.POSITION.X,PLAYER1.
POSITION.Y))
  END
END
```

But the program will also need to update this bullet. Add a loop into the `_UPDATE` function to update all the objects present in the game.

```
FUNCTION _UPDATE()
  --LAUNCH UPDATE() METHOD ON EACH OBJECT
  FOREACH(OBJECTS, FUNCTION(OBJ)
    OBJ.UPDATE(OBJ)
  END
END
```

This `FOREACH` special syntax allow to loop into the objects of an array. Here we define an anonymous function that executes code direction on the object received (here named `OBJ`). We will do exactly the same kind of thing to draw all the objects of the game.

```
FUNCTION _DRAW()
  FOREACH(OBJECTS, FUNCTION(OBJ)
    SPR(OBJ.SPRITE, OBJ.POSITION.X, OBJ.POSITION.Y)
  END
END
```

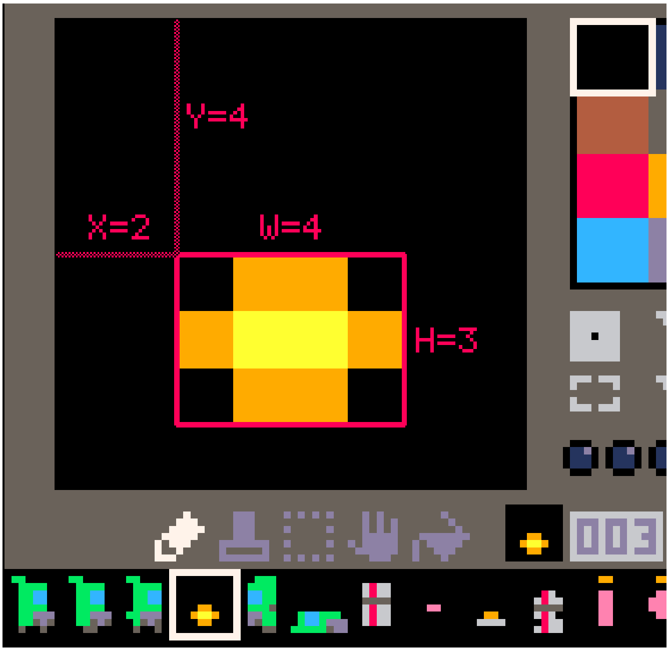
3 Adding a hitbox to the bullet

Now we can shoot bullets, and they will be automatically updated and drawn. But we need them to be destroyed on a wall or to kill

another player on hit. To do that, we will need to complete the BULLETCONSTRUCT by adding collision detection.
First of all, we do not want the whole sprite size (8x8) to collide. The bullet is much smaller, so let's define a hitbox for the bullet:

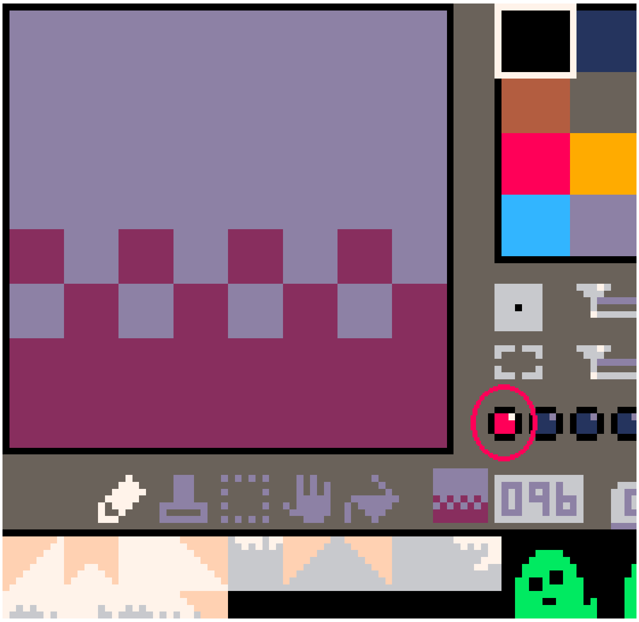
```
BULLETCONSTRUCT=FUNCTION(X,Y)
  ...
  OBJ.HITBOX={X=2,Y=4,W=4,H=3}
  ...
END
```

This hitbox array defines a rectangle on the sprite that will be used to detect collision. X and Y define the top left corner position of the rectangle. W and H define the width and the height. See image for a better understanding.



4 Mark the wall sprite

The game must know which sprites are walls. To do that, we will use the sprite flags in PICO-8. This allow us to manage different kinds of sprite in the game. To do that, simply check one or more of the little circles in the sprite editor. Here we will check only the first circle, this means that the flag selected is **0**. I suggest you to check the **FGET** documentation on PICO-8 manual in order to understand how the flag number is calculated.



So we will just define a global variable determining the **WALL** flag:

```
FWALL = 0
```

5 Detect collision

Now the difficult part: we want to detect if this hitbox hit a wall, so on each frame we will execute a function that detects if each corner of the bullet is positioned inside a wall. If at least one of the corner is in, then the bullet is considered on the wall and we destroy it. First, we create a global function that will detect if an object is on a wall, this object will of course need to have the ‘position’ and ‘hitbox’ attributes. The function **MGET** is a PICO-8 builtin function allowing us to get a sprite on the map. This function will check on the map at X and Y position and returns the sprite number. The function **FGET** is another builtin function that return the flag value of a sprite, so coupled with **MGET**, you can know if there is a wall at a X and Y position. The code of the top left corner is simplified for a better understanding. We will of course check other corners only if previous one are not detected as walls.

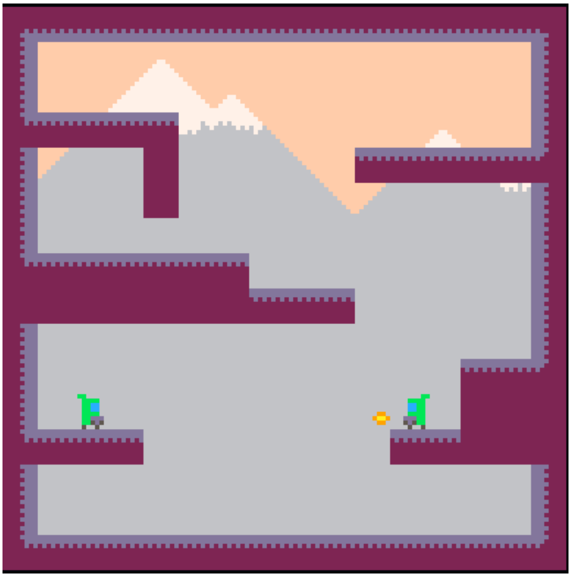
```
-- DETECT IF HITBOX OF OBJECT 'O' DOES HIT THE A WALL MAP SPRITE
FUNCTION CHECKWALL(O)
  --DETECT EACH CORNER OF THE HITBOX ONE BY ONE IF IT COLLIDE
  --THE <SKIN> ALLOW TO NOT DETECT FLOOR ON THE SIDE IF WE ARE
  --STANDING ON THE GROUND
  --TOP LEFT CORNER
  --POSITION OF THE TOP LEFT CORNER OF THE HITBOX IS CALCULATED
  --BY ADDING THE X POSITION OF THE OBJECT AND HIS HITBOX THIS
  --NUMBER IS DIVIDED BY 8, BECAUSE 'MGET' USE SPRITE POSITION AND
  --NOT PIXEL POSITION, AND ALL SPRITE IN PICO-8 ARE 8 PIXELS WIDE
  --FLR() ALLOW TO GET A INTEGER WITHOUT DECIMAL
  LOCAL XPOS = FLR((O.POSITION.X + O.HITBOX.X)/8)
  LOCAL YPOS = FLR((O.POSITION.Y + O.HITBOX.Y)/8)
  --GET THE SPRITE AT THE CALCULATED POSITION
  LOCAL FOUNDSPRITE = MGET(XPOS, YPOS)
  --STOCK IN 'D' VARIABLE IS THE FOUND SPRITE IS A WALL OR NOT
  LOCAL D = FGET(FOUNDSPRITE , FWALL)
  -- TOP RIGHT CORNER
  IF D == FALSE THEN
    D = FGET(MGET(FLR((O.POSITION.X + O.HITBOX.X +
O.HITBOX.W)/8),FLR((O.POSITION.Y + O.HITBOX.Y)/8)),FWALL)
  END
  --BOTTOM LEFT CORNER
  IF D == FALSE THEN
    D = FGET(MGET(FLR((O.POSITION.X +
O.HITBOX.X)/8),FLR((O.POSITION.Y + O.HITBOX.Y +
O.HITBOX.H)/8)),FWALL)
  END
  --BOTTOM RIGHT CORNER
  IF D == FALSE THEN
    D = FGET(MGET(FLR((O.POSITION.X + O.HITBOX.X +
O.HITBOX.W)/8),FLR((O.POSITION.Y + O.HITBOX.Y +
O.HITBOX.H)/8)),FWALL)
  END
  RETURN D
END
```

6 If you want more...

To complete the code, you can tell the bullet the direction you are shooting it, or even which player is shooting. You can also write a function that will detect collisions between objects instead of sprites on the map. To do that, simply loop on all the objects of the game, and check both hitboxes. Ok ok, I already made it, check this out!

```
--detect if 2 objects with hitbox are colliding
function collide(obj, other)
  if other.position.x+other.hitbox.x+other.hitbox.w >
obj.position.x+obj.hitbox.x and
    other.position.y+other.hitbox.y+other.hitbox.h >
obj.position.y+obj.hitbox.y and
    other.position.x+other.hitbox.x <
obj.position.x+obj.hitbox.x+obj.hitbox.w and
    other.position.y+other.hitbox.y <
obj.position.y+obj.hitbox.y+obj.hitbox.h then
    return true
  end
end
```

@schminitz
http://hauntedtie.be



MINIGAME COLLECTIONS

Deep, skill-driven games such as Street Fighter, Starcraft or Crusader Kings are great, but they're not always the best choice for a Friday night with friends. As a general rule, the more complex a game is, the steeper the barrier to entry is going to be. If you're looking for something more inclusive, you can't get much more accessible than a minigame collection - think Pokémon Stadium, WarioWare, Sega Superstars, Mario Party, etc.

They've been around since pretty much the dawn of video games - the Video Action machines made by Universal Research Labs in the '70s (clearly a time when SEO was not important!) allowed four players to choose from a range of Pong-like sports including Hockey, Volleyball and Soccer. Typically the games in these collections are very simple, consisting of a single mechanic and a few inputs. This means that anyone can join in - even people that don't usually play video games! They also have the benefit of still being entirely playable after a few beers.

1. Design

Over the last few weeks I've been working on my own minigame collection inspired by the minigames of Pokémon Stadium. Even now, we still brush the dust off the N64 every now and again to revisit such classics as 'Clefairy Says!', 'Snore War' and 'Rock Harden'! Looking at the systems behind these games, I was able to identify a few common mechanics that form the 'building blocks' of a good minigame. Search the examples on YouTube if you want to see them in action!

Timing

• Rock Harden, Snore War

Memory

• Clefairy Says

Dexterity

• Ekans' Hoop Hurl, Sushi-Go-Round

Button Mashing!

• Dig! Dig! Dig!, Thundering Dynamo

This is by no means an exclusive list - pretty much anything goes in a minigame, just keep it simple and bitesize. Combining these mechanics can be great too, for instance 'Run, Rattata, Run!'

uses both button-mashing and timing to great effect. As a rule of thumb, if you can explain how to play in a single sentence you're probably on the right track!

2. Implementation - Players

So let's get building! As of 0.1.2, PICO–8 supports up to eight players using gamepads (or getting very cosy around a keyboard!). I designed my minigames for four players, but I wanted to make sure that my code was flexible to allow two, three or four players to use the same application. We're going to use a table to store the game values we need for each player (e.g. colour, position), then another table to store all of our players. Here's a simple example that you can run in PICO–8:

```
-- CONFIG
N_PLAYERS = 2
START_POS = {8, 40, 72, 104}
COLOURS = {8, 11, 12, 10}
ALT_COLOURS = {2, 3, 1, 9}

-- CORE FUNCTIONS
FUNCTION _INIT()
  INIT_ONE()
END

FUNCTION _UPDATE()
  FOR P IN ALL(PLAYERS) DO
    UPDATE_ONE(P)
  END
END

FUNCTION _DRAW()
  CLS()
  FOR P IN ALL(PLAYERS) DO
    DRAW_ONE(P)
  END
END

FUNCTION INIT_ONE()
  CLS()
  CREATE_PLAYERS(N_PLAYERS)
END

FUNCTION UPDATE_ONE(P)
  -- VERTICAL MOVEMENT, ALTER THE PLAYER POSITION
  IF BTN(2,P.NUM-1) THEN
    P.POS[2]-=1
  ELSEIF BTN(3,P.NUM-1) THEN
    P.POS[2]+=1
  END
END

FUNCTION DRAW_ONE(P)
  -- DEFINE LOCAL VARIABLES
  LOCAL X = P.POS[1]
  LOCAL Y = P.POS[2]
  LOCAL COLOUR = COLOURS[P.NUM]
  LOCAL ALT_COLOUR = ALT_COLOURS[P.NUM]

  -- DO SOMETHING WITH THE VALUES STORED IN THE PLAYER TABLE
  RECTFILL(X-5,Y-15,X+5,Y+15,0) -- CLEAR SCREEN!
  CIRCFILL(X,Y,5,COLOUR) -- DRAW!
  PRINT(P.NUM,X-1,Y-12,ALT_COLOUR) -- PRINT!
END

FUNCTION CREATE_PLAYERS(N)
  PLAYERS = {}
  FOR I=1,N DO
    P = {}
    P.NUM = I
    P.POS = {START_POS[I],64}
    ADD(PLAYERS,P)
  END
END
```
This is all made possible because of the **ALL** iterator. This piece of code lets us call a function for **every player** in our players table, regardless of how many players we have:

```
FOR P IN ALL(PLAYERS) DO
SOMETHING(P)
```

```
END
```

This particular program is valid for a maximum of four players, though, as I've only specified four possible positions and colours in the config section. If you wanted to support more players, you'd just need to increase the number of items in those tables (or don't use them, or have duplicate colours, etc). From a design perspective, you'd also need to think about how you partition the screen space!

3. Implementation - Structure

So now you've got your program receiving input and drawing players. But we want more than one game in our collection, of course! There are lots of ways to structure a minigame collection, but the simplest approach I've found is to create each game as its own discrete application and then to have a management layer above the games that handles scoring, game selection, etc. **However**, in PICO—8 this isn't a viable option, so we have to get a bit clever! :]

As you know, PICO—8 has three main functions that form the game loop, _init, _update and _draw. When we're making a minigame collection, we want those functions to do different things depending on whether we're in the menu or in one of multiple different games. We can achieve this by creating a table that holds references to versions of these core functions for each of the games. Our menu then simply needs to set a variable that tells the game loop which of these sets of functions it should call. We can alter our existing program (above) to support this approach. First, let's set up our function tables in the _init function, and call init_menu to start the menu running:

```
FUNCTION _INIT()
  CLS()
  GAMES = {
    "GAME 1",
    "GAME 2"
  }
  INIT_FUNCTIONS = {
    INIT_ONE,
    INIT_TWO
  }
  UPDATE_FUNCTIONS = {
    UPDATE_ONE,
    UPDATE_TWO
  }
  DRAW_FUNCTIONS = {
    DRAW_ONE,
    DRAW_TWO
  }
  INIT_MENU()
END
```

Next, let's alter our core _update and _draw functions to point at our **function tables** instead of calling a function directly. We'll also use the menu boolean to control whether we run **menu** or **game** functions, and the complete boolean to return to the menu:

```
FUNCTION _UPDATE()
  IF MENU THEN
    UPDATE_MENU()
  ELSEIF COMPLETE THEN
    INIT_MENU()
  ELSE
    FOR P IN ALL(PLAYERS) DO
      UPDATE_FUNCTIONS[SELECT_GAME](P)
    END
  END
END

FUNCTION _DRAW()
  IF MENU THEN DRAW_MENU()
  ELSE
    FOR P IN ALL(PLAYERS) DO
      DRAW_FUNCTIONS[SELECT_GAME](P)
    END
  END
END
```

This is all we need to alter in the core program loop. Next, we will define the specific init, update and draw functions we're using for our menu. The menu functions allow the first player to view & select from our list of games:

```
function init_menu()
  cls()
  menu = true
  complete = false
  select_game = 1
end
-- if button is pressed, menu is false and the init function for the
--selected game is run. otherwise change selected game, if valid.
function update_menu()
  if btnp(4, 0) then
    menu = false
    init_functions[select_game]()
  elseif (btnp(2, 0) and select_game > 1) then
    select_game -= 1
  elseif (btnp(3, 0) and select_game < #games) then
    select_game += 1
  end
end

-- print out all of the games in our list
function draw_menu()
  local y = 32
  for i=1, #games do
    if i == select_game then
      print(games[i], 52, y, 12)
    else
      print(games[i], 52, y, 6)
    end
    y += 16
  end
  print("select a game!", 37, y, 7)
end
```

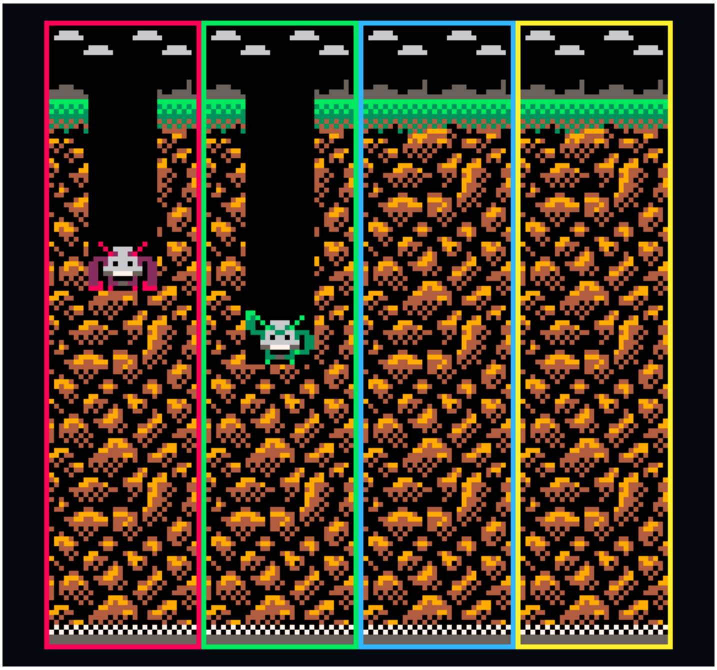
Next up, we need to create specific init, update and draw functions for the games we want to include. In this example, we need to add definitions for init_two, update_two and draw_two. For now, you can copy the code from the existing init_one, update_one and draw_one functions.

Finally, we need to add a way to **return** to the menu from the running game. To achieve this, we can add the following snippet to our update_one & update_two functions:

```
-- WHEN PLAYER ONE PRESSES BUTTON ONE, WE'LL RETURN TO THE MENU.
IF BTNP(4,0) THEN
  COMPLETE = TRUE
END
```

In this example I've included two games, but there's really no limit to how many games you could include this way (apart from the line/token count on the cart!). Persistent variables (such as player scores) can be created in the core _INIT function and altered by the game-specific functions as you go along.

4. Implementation - Dig! Dig! Dig!



I hope you're still with me! I thought I'd show you how I used these lessons in the development of an actual game. Building on our previous program once more, I'll create a clone of the classic Pokémon Stadium game **'Dig! Dig! Dig!'**. To keep things simple, I've avoided using any additional assets (e.g. sprites, sfx) outside of the code shown in this article.

First, we're going to make some alterations to our generic create_players function. The game requires some additional player variables, but it's pretty simple to add these in.

```
FUNCTION CREATE_PLAYERS(N)
  PLAYERS = {}
  FOR I=1,N DO
    P = {}
    P.NUM = I
    P.POS = {START_POS[I], 0}
    P.LAST = {FALSE, FALSE}
    P.TIMER = 0
    ADD(PLAYERS, P)
  END
END
```

Next up, we'll create another generic function that draws coloured borders around each player's section of the screen. These functions are great and should be used wherever possible, as they can be used across multiple games to save tokens!

```
FUNCTION DRAW_BORDERS()
  FOR I = 0, 3, 1 DO
    RECT(I*32, 0, (I*32)+31, 127, COLOURS[I+1])
  END
END
```

Next up, we'll rewrite our init_one function to set up the new game:

```
FUNCTION INIT_ONE()
```

```
  CLS()
  -- GAME DESIGN PARAMETERS
  PENALTY_TIME = 20
  FINISH_LINE = 107

  -- SIMPLE BACKGROUND, REPLACE WITH A NICE MAP IN YOUR GAME!
  RECTFILL(0,16,127,FINISH_LINE+16,4)
  RECTFILL(0,FINISH_LINE+16,127,127,9)

  CREATE_PLAYERS(N_PLAYERS)
  DRAW_BORDERS()
END
```

The update_one function is where all of our game logic sits. As before, it's called once for each player and we pass the player p to it.

First, we check for the win condition - has a player moved down below the 'finish_line' distance that we set in init_one? If so, we store the winning player number and mark the game as complete.

If not, we check whether the player is on **timeout.** In *Dig! Dig! Dig!*, clumsy mashers are penalised with a short timeout if they press the same button twice. If they are on timeout, we reduce the timer but don't let them move. Otherwise, we check the input received from that player and compare it with the last accepted input. If it's a repeat, we apply the timeout penalty, otherwise we increase the player's vertical position (let them dig!) and update the last accepted input.

```
FUNCTION UPDATE_ONE(P)
  -- CHECK IF FINISHED
  IF P.POS[2] >= FINISH_LINE THEN
    COMPLETE = TRUE
  -- IF ON TIMEOUT, REDUCE TIMEOUT
  ELSEIF P.TIMER > 0 THEN
```

```
    P.TIMER -= 1
  -- IF NOT ON TIMEOUT, CHECK FOR INPUT AND DIG
  ELSEIF P.TIMER == 0 THEN
    LOCAL BTNS = {BTNP(4, P.NUM-1), BTNP(5, P.NUM-1)}
    IF (BTNS[1] OR BTNS[2]) THEN
      -- IF IT'S A REPEAT INPUT, APPLY PENALTY, ELSE DIG!
      IF (BTNS[1] == P.LAST[1] AND BTNS[2] == P.LAST[2]) THEN
        P.TIMER = PENALTY_TIME
        P.LAST = {FALSE, FALSE}
      ELSE
        P.POS[2] += 1
        P.LAST = BTNS
      END
    END
  END
END
```

Drawing is relatively simple - instead of drawing circles as before, we're drawing rectangles. If the player is on timeout, we draw a flattened rectangle with the alternative colour.

```
FUNCTION DRAW_ONE(P)
  -- DEFINE LOCAL VARIABLES (THIS ISN'T NECESSARY, BUT IT MAKES
  --OUR CODE CLEARER)
  LOCAL X = P.POS[1]
  LOCAL Y = P.POS[2]
  LOCAL COLOUR = COLOURS[P.NUM]
  LOCAL ALT_COLOUR = ALT_COLOURS[P.NUM]

  -- SIMPLE BLOCK GRAPHICS - REPLACE WITH SPRITES IN YOUR GAME!
  RECTFILL(X+1, Y+2, X+14, Y+15, 0) -- CLEAR SCREEN!

  IF P.TIMER > 0 THEN
    RECTFILL(X+3, Y+12, X+12, Y+15, ALT_COLOUR)
  ELSE
    RECTFILL(X+4, Y+10, X+11, Y+15, COLOUR)
  END
END
```

At this point the game works, but it quits back to the menu as soon as a player reaches the finish line. Ideally, we'd want a short break to celebrate their win before moving on to the next game! We can implement this by creating a new variable, post-game_timeout, in our config section. Here, we will define the number of frames to wait after a player has won before returning to the menu (I've found **180**, which is about 6 seconds, to be a good amount). In our core _init function, we will define another new variable, `global_timer`. In our `update_one` function, as well as setting `complete = true` we will set the `global_timer` equal to our `postgame_timeout` value. We'll also record the winning player number in the winner variable.

```
-- check if finished
  if p.pos[2] >= finish_line then
    complete = true
    global_timer = postgame_timeout
    winner = p.num
```

Now, in our core `_update` function, we'll make a small change to make use of the `global_timer`. Instead of calling `init_menu` as soon as complete becomes true, it now has to decrement the `global_timer` down to zero first.

```
function _update()
  if menu then
    update_menu()
  elseif complete and global_timer > 0 then
    global_timer -= 1
  elseif complete and global_timer == 0 then
    init_menu()
  else
    for p in all(players) do
      update_functions[select_game](p)
    end
  end
end
```

We'll also define a function to display the winning player on a banner:

```
FUNCTION VICTORY_BANNER()
  RECTFILL(0,56,127,72,0)
  RECTFILL(0,58,127,70,7)
  PRINT("PLAYER "..WINNER.." WINS!",36,62,0)
END
```

All that remains is to add the following line to the core _draw function!

```
IF (COMPLETE) VICTORY_BANNER()
```

That's it! You've got a working minigame and a loop that allows you to jump into successive games from a single menu. Pretty cool! If you want to see what a more complex implemetation of this looks like, you can check out my full game here:
http://www.lexaloffle.com/bbs/?tid=2782
It includes four different minigames, complete with music, sfx and art!
I hope you're inspired to make your own weird, eclectic minigame collections for you & your friends. Please share them with me if you do!

-- Jack Harrison
-- @jhrrsn

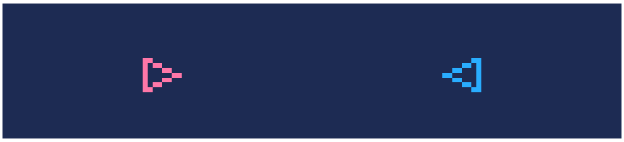
BLASTEROIDS

Hey folks, I'm Lulu Blue.
I've made a bunch of games (even some for the Pico-8) and you can check them out at **bluesweatshirt.itch.io** if you want. More importantly, I'm here to guide your hand in creating a little game I've came up with called Blasteroids. Imagine Asteroids, if you're familiar with that, except it's two-player and you're trying to blast each other.

If you're not familiar with it, Asteroids is one of those "classic games", it originally came out in the arcades in '79. The gist of it is that you're a lone spaceship in a field of floating asteroids, and your goal is to shoot down all of them while not being hit by the errant debris. Check out a video of it or something if you're curious, it's super neat.

Our goal here is not just going to be adding a multiplayer layer on top of Asteroids, it's to explore what it's like to design a multiplayer game. Adding an extra layer on top of something familiar is a perfect place to start. I arbitrarily consider this an intermediate-advanced guide, so you should be pretty comfortable with Pico-8 and a code editor before tackling this, just to get the most out of it. Instead of focusing on the act of construction we're going to be focusing on how each bit we add contributes to the design of the game.

1. FROM NOTHING TO SOMETHING: THE PLAYER



The first thing we should do to have our game is to have the thing we're controlling. From there, we can design all the other game's mechanics around that. To start off, we need to first write a bit of initializing code.

```
FUNCTION P_MAKE(X,Y,R)
 RETURN{
  X=X,--POSITION
  Y=Y,
  VX=0,--VELOCITY
```

```
   VY=0,
  ROT=R,--ROTATION
  COL=14,--COLOR
  P=0--CONTROLLER INDEX
  }
END
```

We need to track the player's position, velocity, rotation, color and player index - we'll use that for identifying whether you're Player 1 or Player 2 and for input handling. We're using tables here to define players instead of writing out variables manually, such as “p1_x”. It saves us from writing all the same player code twice over when they're going to behave the same anyway. We can initialize our players like this:

```
FUNCTION _INIT()
  P1=P_MAKE(32,64,0)
  P2=P_MAKE(96,64,0.5)
  P2.COL=12
  P2.P=1
END
```

In addition to being a lot cleaner, it also saves cartridge space. If this were made outside the Pico-8 though, it would easily allow for supporting more than 2 players if you stored their tables in an index.
Before moving on, let's write some constants the player will use. Just stick this at the top of your code.

```
P_LEN = 7  --LENGTH, FRONT TO BACK, OF THE PLAYER'S SHIP.
P_WID = 3 --WIDTH, FROM CENTER TO SIDE, OF THE PLAYER'S SHIP.
P_SPD = 1-- MAX SPEED THE PLAYER CAN TRAVEL.
P_ACL = 0.075 -- HOW MUCHTO INCREASE VELOCITY EVERY FRAME
P_DRG = 0.025 --HOW MUCH FRICTION TO ACT ON THE PLAYER EVERY FRAME
P_RAD = 2.5 --PLAYER HITBOX RADIUS
```

This is all pretty simple stuff, but it's going to get more complex from here on out. There's some vaguely involved math we'll be using to calculate drag for very purposeful ends. In competitive games, the nuance of every little action becomes important. It's not just the fact things move, but the way they move that makes the difference. We'll go deep into all the game's minutia and see how they contribute to the whole.

Before jumping headfirst into play though, let's get our ships drawing:

```
function p_draw(p)
-- how much the ship extends in front
local lenx = cos(p.rot)*p_len
local leny = sin(p.rot)*p_len
-- how much the ship extends to the side
local sidex = cos(p.rot+0.25)*p_wid
local sidey = sin(p.rot+0.25)*p_wid
-- point at front of ship
local pfx=p.x+lenx*0.6 -- 60% of the ship's length is in front of its center
point (x,y)
local pfy=p.y+leny*0.6
-- point at left of ship
local plx=p.x-lenx*0.4+sidex -- 40% of the ship's length is behind its center
point (x,y)
local ply=p.y-leny*0.4+sidey
-- point at right of ship
local prx=p.x-lenx*0.4-sidex
local pry=p.y-leny*0.4-sidey
-- thats a lot of math! easy part now!
line(pfx,pfy,plx,ply,p.col)
line(pfx,pfy,prx,pry,p.col)
line(plx,ply,prx,pry,p.col)
end
```

Quite a bit just for a little triangle, huh? But that's not just any triangle, it's our triangle. And it rotates. Most importantly, it rotates around roughly the center of the triangle, which will let us fit a circular hitbox inside it pretty snugly.
Let's get it on screen:

```
function _update()
--empty :)
end

function _draw()
rectfill(0,0,127,127,1)--clear the screen
rectfill(0,0,127,8,0)--draw the status bar
p_draw(p1)
p_draw(p2)
end
```

Now run it, and you should see an image that resembles the image at the top of this chapter. Wonderful! Seeing the first few things appear on the screen in a game never stops being exciting for me. Now let's write update code:

```
FUNCTION P_UPDATE(P)
  -- TURNING
  IF BTN(0,P.P) THEN
    P.ROT+=0.0125
  END
  IF BTN(1,P.P) THEN
    P.ROT-=0.0125
  END
  -- ACCELERATION
  IF BTN(2,P.P) THEN
    P.VX+=COS(P.ROT)*P_ACL
    P.VY+=SIN(P.ROT)*P_ACL
    -- VELOCITY CAPPING
    IF ABS(P.VX)+ABS(P.VY)>1 THEN
      LOCAL D=ATAN2(P.VX,P.VY)
      P.VX=COS(D)
      P.VY=SIN(D)
    END
  END
  -- DRAG
  LOCAL VEL=(ABS(P.VX)+ABS(P.VY))/P_SPD
  LOCAL SX=SGN(P.VX)
  LOCAL SY=SGN(P.VY)
  P.VX-=P_DRG*P.VX*VEL
  P.VY-=P_DRG*P.VY*VEL
  IF (SX != SGN(P.VX)) P.VX=0
  IF (SY != SGN(P.VY)) P.VY=0
  -- MOVEMENT
  P.X+=P_SPD*P.VX
  P.Y+=P_SPD*P.VY
  -- WRAPPING
  P.X=P.X%127
  IF (P.Y>127) P.Y=8
  IF (P.Y<8) P.Y=127
END
```

And then add it to the update loop:

```
function _update()
  p_update(p1)
  p_update(p2)
end
```

While writing this you probably got a good idea of what this code's doing, but let's go through it anyway. As you saw in the draw function, we're using `cos` and `sin` to rotate our coordinates. You'll see more of these as we go on.

The first thing we do after accelerating is cap the velocity. This serves two functions: It keeps the ships from accelerating to light speed, and it keeps the velocity capped the same no matter which direction you're moving or facing.

Next is drag. First, we're taking a proportional readout of the ship's speed, where 0 is still and 1 is maximum velocity. We use this to apply more drag as the ship speeds up, giving a nice curve to the acceleration and sparing us the logistics of applying more drag than acceleration at any given moment —then we couldn't move! Next we take the sign, "`sgn()`", of each velocity component, which is whether the given number is positive, negative, or zero, expressed as 1, -1, or 0. First we use this to make sure drag is always reducing the velocity value, whether it's positive or negative, then we use it to check if the drag has completely decelerated the ship. If the sign has crossed a threshold, then it's done its work.

To wrap it all up, we're using a couple simple statements to loop the ship around the screen, so that when you leave it from one end you come out the other. The differing Y looping is to account for the health bar display we'll be making later on. Looping contextualizes the space in a really interesting way, and it gives room for some really tricksy competitive strategies.

This is a lot of backbone for just flying around space, right? You could omit all the velocity and acceleration and drag stuff and just leave it at moving forward when you press forward. I think it's worth it though, a little effort in the right places can go a long way. The core of this game is maneuvering; you're dodging asteroids and the other player while positioning yourself for the best shot. To really flesh that out, the game needs interesting movement mechanics. Here, it's very floaty. And while there's a lot of inherent fun to that, it also adds to the feeling of flying

a ship in space and adds another layer to the game's strategy:
A key part of multiplayer games is being able to read your opponents, (the fighting game community calls this “yomi”) to get in their head, predict them, and ultimately outwit one or the other. Floatiness adds a predictable trajectory to your ship's motions, and this creates a dynamic of being able to mislead and juke your opponents as well. These kinds of mind-games are the backbone of competitive games, and we'll add more elements to the game to flesh this out as we go on.

If you haven't already, run the game. Revel in the fun movement mechanics you just made! Get a real feel for them, and then start thinking about the sorts of things you could add to make an interesting versus game. Grab a nice warm beverage, chill out, and get ready for the next chapter.

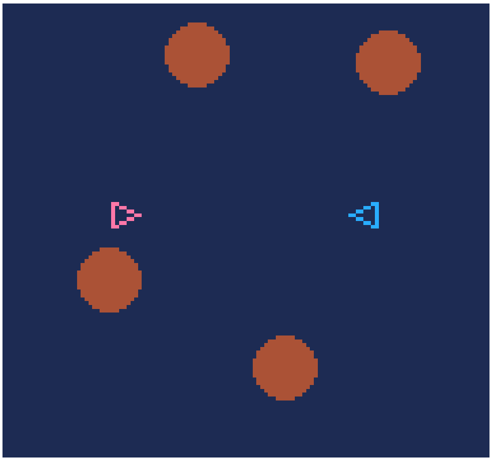
2. AGENTS OF CHAOS: THE ASTEROIDS

Most multiplayer games have some element of chaos to keep things interesting play-to-play. For a game like Street Fighter it's the sheer size of the possibility space. For a card game like Magic: The Gathering it's the randomness of which card you and your opponent will draw next. (and also the sheer size of the possibility space. Magic is a lot.) Think of the possibility space as the total range of theoretical outcomes produced from your game's mechanics interacting with each other. Learning to keep the possibility space of your game in mind is kind of like seeing the Matrix.

But more importantly, what possibility spaces and card draws have in common is impredictability, and that is your chaos element. Impredictability does not necessarily mean randomness, though it can. Many consider too much randomness counter-productive for a competitive game because it makes the outcome of the game more dependent on luck. The key here is flavor. Interest. Variety. Pizazz. In some multiplayer games, this might mean having different stages with different level geometry, or a roster of characters to play as, or a million-and-one different cards which all have unique mechanics associated with them. (terrifying!).

In our game, that chaos element is the asteroids themselves. Not only do they enter the play field differently each game, they break apart into smaller chunks that get sent flying around.
For the sake of competitiveness, you'll notice we take specific measures to add a deterministic element to rocks splitting apart,

adding a layer of complexity that skilled players may harness competitively. Competitive scenes often call this a “skill ceiling”, where the higher the ceiling is the more potential there is to harness the game's mechanics competitively. Games with high skill ceilings are things like Chess, Street Fighter and League of Legends, where games with low skill ceilings are ones like Tic-Tac-Toe and Rock, Paper, Scissors.



Let's dive into some code:

```
FUNCTION A_NEW(A,X,Y,S,VX,VY)
  A=A OR {}
  A.ON=TRUE
  A.X=X
  A.Y=Y
  A.S=S
  A.VX=VX
  A.VY=VY
  RETURN A
END
```

You'll notice we're handling asteroids a bit differently—you can pass in a pre-existing table.
Here's why:

```
FUNCTION A_INIT()
  --INITIALIZE ASTEROID POOL
  --WE DO THIS FOR CONSISTENT PERFORMANCE
  ROCKS={}
  FOR I=1,20 DO
    LOCAL ROCK=A_NEW(NIL,-16,-16,5,0,0)
    ROCK.ON=FALSE
    ROCKS[#ROCKS+1]=ROCK
  END
  ROCKS_ON=0
END
```

Here, we're using a pool of asteroids. There will always be at least 20 in memory, but it will expand the collection if necessary. Rather than initializing a new table when we want to put a new asteroid on the screen, we reuse old ones first. This way we're not constantly initializing and freeing data at the whims of the garbage collector, the part of Lua that cleans up no-longer used data now and then. It's good not to lean on the garbage collector too much, activating it comes with a performance overhead and memory is so limited on the Pico-8. It's important for competitive games to run smoothly and consistently.

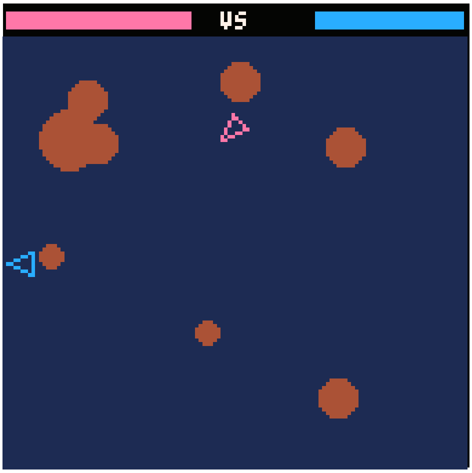
```
FUNCTION A_RECYCLE(X,Y,S,VX,VY)
  -- RECYCLE AN OLD PIECE OF DATA
  -- BEFORE MAKING A NEW ONE
  FOR A IN ALL(ROCKS) DO
    IF A.ON == FALSE THEN
      A = A_NEW(A,X,Y,S,VX,VY)
      ROCKS_ON+=1
      RETURN A
    END
  END
  ROCKS[#ROCKS+1]=A_NEW(NIL,X,Y,S,VX,VY)
  ROCKS_ON+=1
  RETURN ROCKS[#ROCKS+1]
END
```

Now we have a really convenient way to recycle asteroids. Instead of using `A_NEW()` to add an asteroid into the play field, we'll be using `A_RECYCLE()`

```
FUNCTION A_UPDATE()
  IF ROCKS_ON < 4 THEN -- KEEP THE PLAY FIELD FROM BEING EMPTY
    LOCAL R=RND(1)
    LOCAL X=64+COS(R)*100 -- CHOOSE A POINT OUTSIDE THE SCREEN, RADIALLY
    local y=64+sin(r)*100
    a_recycle(x,y,8,cos(r+rnd(0.1)-0.05)*-0.6,sin(r+rnd(0.1)-0.05)*-0.6)
-- add some random deviation so they don't all move directly towards the center
end
for rock in all(rocks) do
  -- very simple behaviour for the asteroids
  rock.x+=rock.vx
  rock.y+=rock.vy
  --wrapping
  if(rock.x>127 and rock.vx>0)rock.x=0
  if(rock.x<0 and rock.vx<0)rock.x=127
  if(rock.y>127 and rock.vy>0)rock.y=0
  if(rock.y<8 and rock.vy<0)rock.y=127
  end
end

function a_draw()
  for rock in all(rocks) do
    if rock.on then
    circfill(rock.x,rock.y,rock.s,4)
    end
  end
end
```

That took a bit of scaffolding, but we finally did it. Make sure to add `a_init()`, `a_update()` and `a_draw()` into your code, and then give it a run! Asteroids should start barreling in from the edges of the screen.



Except there's no collision. Let's fix that right now. First, put this somewhere in your code:

```
FUNCTION DIST(X1,Y1,X2,Y2)
  RETURN SQRT((X2-X1)^2+(Y2-Y1)^2)
END
```

We'll use this to check the distance between any two entities, and if their circle hitboxes are intersecting (meaning, that distance is less than both of their radius put together) then we'll know they're touching.
Add this to the end of **P_UPDATE()**:

```
--COLLIDE ROCK
  FOR R IN ALL(ROCKS) DO
  IF R.ON THEN
    IF DIST(P.X,P.Y,R.X,R.Y)<R.S+P_RAD AND P.INV==0 THEN
    P_HURT(P,10)--HURT THE PLAYER
    P.VX=R.VX*2
    P.VY=R.VY*2
    --CAN ROCK
    R.ON=FALSE
    ROCKS_ON-=1
  --SPLIT ROCKS
  LOCAL SIZE=R.S*0.65--THE TWO ROCKS THAT COME OUT SHALL BE SMALLER
  IF SIZE>3 THEN--BUT THEY SHALL ONLY GET SO SMALL
    A_RECYCLE(R.X,R.Y,SIZE,R.VY,-R.VX)
     --ROCKS SPLIT PERPENDICULAR TO THE ROCK'S MOTION VECTOR
    A_RECYCLE(R.X,R.Y,SIZE,-R.VY,R.VX)
    END
    BREAK
    END
  END
END
```

As you may have noticed, there's a couple things in here we haven't defined yet. We'll do that in a minute, but first, I want to recall the ship hitbox size. Maybe it occurred to you in passing that it was a bit small. It totally is! A rule I like to follow is to always round down when deciding player hitboxes. Why? Because otherwise getting hit might feel less fair. Human perception is bound to misjudge time to time, especially when our eyes are trained on a screen and not a tangible object in front of us.

Instead of having people make snap judgements on the razor's edge, a little leeway makes everything feel a bit more well-rounded. We make up for the smallness of the player ship by making everything else in the game big. But it's not a zero-sum game. People are experiencing the visual and aural information of the game, and that informs how they understand your game's systems. They will not be acutely aware of the numbers acting behind the scenes like you might. I find it really important to keep this in mind.

Moving on, we've yet to implement damage-taking elements to the player, so let's do that now.

Add this to **P_MAKE**:

```
HP=100
INV=0
```

Add this to p_update:

```
P.INV=MAX(0,P.INV-1)
```

Add this to p_draw around the three line() commands:

```
IF P.INV%6<3 THEN
  --MAKE SHIP FLASH WHILE INVULNERABLE
  --LINE()
  --LINE()
  --LINE()
END
```

And finally:

```
FUNCTION P_HURT(P,DMG)
  P.HP-=DMG
  P.INV=45
END
```

That's not it though, we need to be able to see the hp, so add this to **P_DRAW()**:

```
-- HP BAR
IF P.P==0 THEN
  LOCAL BX1=1
  LOCAL BY1=2
  LOCAL BXW=56
  LOCAL BY2=BY1+4
  RECTFILL(BX1,BY1,BX1+BXW*(P.HP/100),BY2,P.COL)
ELSE
  LOCAL BX1=126
  LOCAL BY1=2
  LOCAL BXW=-56
  LOCAL BY2=BY1+4
  RECTFILL(BX1,BY1,BX1+BXW*(P.HP/100),BY2,P.COL)
  END
```

Now update your _draw():

```
FUNCTION _DRAW()
  RECTFILL(0,0,127,127,1)
  A_DRAW()
  RECTFILL(0,0,127,8,0)
  PRINT("VS",60,2,7)
  P_DRAW(P1)
  P_DRAW(P2)
END
```

You did it. Now run the game, and you should be able to experience to glory of being smashed by space debris in real-time and observing its effects. This was a big milestone, and that's fantastic! Give your brain a rest before we go into the final part. Everything's about to come together.

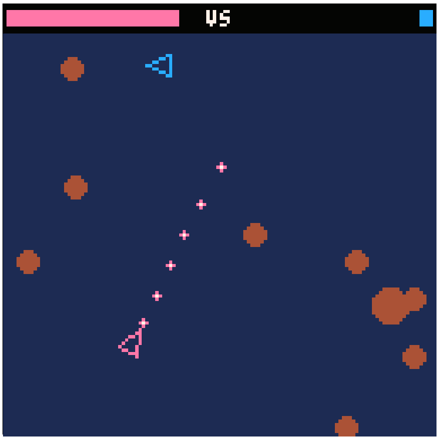
3. TOOLS OF WAR: THE SHOOTING

There's two major things you do in this game, move and shoot. We've got the moving, and the big rocks which make you really gotta move, so now we need shooting.

We're not just gonna smash a button to shoot though.

We're doing charged shots.

Okay, that isn't a huge enough deal to justify dramatic line-breaks, but it is important! The concept of charging up adds a whole new layer to the competitive dynamic—”When are they going to shoot?” Now shooting becomes a question of taking the time for a well-placed shot, or firing early to catch your opponent off-guard, or even barraging them with a hail of weak shots as a distraction mechanism for the big one-two. It creates a definitive line of strategy to think in terms of, and as a consequence, get into each others' heads over.



```
S_SPD_MIN = 2 -- MINIMUM SHOT SPEED
S_SPD_CHG = 1.5 -- SPEED TO ADD AS THE CHARGE GETS HIGHER

FUNCTION S_NEW(T,X,Y,S,D,P)
  T = T OR {}
  T.X=X
  T.Y=Y
  T.S=S--SIZE(0 TO 1)
  T.SZ=1+S*3--PIXEL SIZE IN RADIUS
  T.D=D --FACING DIRECTION
  T.DST=50-- DISTANCE TO TRAVEL BEFORE DISSIPATING
  T.ON=TRUE
  T.P=P--PLAYER WHO SHOT THIS
  T.SPD=S_SPD_MIN+S_SPD_CHG*S
  T.VX=COS(D)*T.SPD
  T.VY=SIN(D)*T.SPD
  RETURN T
END
```

And then we'll use the recycling method, just like before with the asteroids:

```
FUNCTION S_RECYCLE(...)
  SHOTS_ON+=1
  FOR S IN ALL(SHOTS) DO
    IF S.ON==FALSE THEN
    S = S_NEW(S,...)
  RETURN S
```

```
    END
  END
  LOCAL S = S_NEW(NIL,...)
  SHOTS[#SHOTS+1]=S
  RETURN S
END

FUNCTION S_INIT()
  SHOTS={}
  SHOTS_ON=0
  --INITIALIZE SHOT POOL
  FOR I=1,10 DO
    LOCAL S = S_NEW(NIL,0,0,0,0,0)
    S.ON=FALSE
    SHOTS[#SHOTS+1]=S
  END
END
```

You'll notice that our techniques mirror a lot of what we did with the asteroids. In a more complex game, you might even consolidate them into a single cohesive entity system. For this though, we have a very specific idea of what we want to do, so that would just eat up our time and introduce more moving parts than necessary. This is the last big code block there is. And it looks like more than it is! Most of what we're doing here will look familiar.

```
FUNCTION S_UPDATE()
  FOR S IN ALL(SHOTS) DO
  IF S.ON THEN
  --MOTION
  S.X+=S.VX
  S.Y+=S.VY
  --WRAPPING
  S.X=S.X%127
  IF(S.Y>127)S.Y=8
  IF(S.Y<8)S.Y=127
  --DISSIPATION
  S.DST-=S.SPD
  IF S.DST<0 THEN
    S.ON=FALSE
    SHOTS_ON-=1
  END
  -- COLLIDE ROCK
  FOR R IN ALL(ROCKS) DO
  IF R.ON THEN
  IF DIST(S.X,S.Y,R.X,R.Y)<R.S+S.SZ THEN
    SFX(3)
  --REMOVE OBJECTS
  S.ON=FALSE
  R.ON=FALSE
  SHOTS_ON-=1
  ROCKS_ON-=1
  --SPLIT ROCKS, JUST LIKE WITH SHIP COLLIDING
  S.VX*=0.3
  S.VY*=0.3
  LOCAL SIZE=R.S*0.65
  IF SIZE > 3 THEN
    A_RECYCLE(R.X,R.Y,SIZE,S.VY,-S.VX)
    A_RECYCLE(R.X,R.Y,SIZE,-S.VY,S.VX)
  END
  BREAK
  END
END
END
  -- COLLIDE PLAYER
  LOCAL P = NIL
  IF(S.P==0)P=P2--ENEMY PLAYER
  IF(S.P==1)P=P1
  IF DIST(S.X,S.Y,P.X,P.Y)<P_RAD+S.SZ AND P.INV==0 THEN
    S.ON=FALSE
    SHOTS_ON-=1
    P_HURT(P,10+15*S.S)
    P.VX+=S.VX*0.4*S.S
    P.VY+=S.VY*0.4*S.S
  END
  -- END SHOT UPDATE
  END
  END
END
```
There's a couple notable design choices in here. First, when asteroids collide with shots they don't split perpendicular to the asteroid's motion vector, they split perpendicular to the shot's. This adds a layer to the competitive dynamic where players can

manipulate crumbling asteroids as an offensive tactic. This adds to the mind-gaming of charged shots and sneaky screen-looping tactics.

We're also making it so that shots push around the ship they hit. Now you can interfere with the other player's maneuvering on a whole other level, possibly pushing them into unfavorable positions or overwhelming them with the mere threat of losing their bearings.

They're such small choices too, yet they can go a long way to expanding the possibility for interesting plays. A little effort in the right places! If I'm getting obnoxious with that phrase it's only because I see a lot of worth in it. Anyway, let's put the shots on-screen now.

```
FUNCTION S_DRAW()
  FOR S IN ALL(SHOTS) DO
  IF S.ON THEN
  LOCAL COL=0
  IF (S.P==0)COL=P1.COL
  IF (S.P==1)COL=P2.COL
  CIRCFILL(S.X,S.Y,1+S.S*3,COL)
  CIRCFILL(S.X,S.Y,S.S*3,7)
  END
  END
END
```

UPDATE_INIT(), **_UPDATE()**, and **_DRAW()** now to incorporate the shot functions. You can go ahead and test it, but we haven't added a way to shoot yet. Let's fix that. This is the last step, and then you have a game.

Add this property to **P_INIT()**:

```
CHRG=0
```

Add this to the top of **P_UPDATE()**:

```
--SHOOTING
  IF BTN(4,P.P) THEN
    P.CHRG=MIN(1,P.CHRG+0.05)
  END
```

```
 if not btn(4,p.p) and p.chrg>0 then
   local s=s_recycle(
     p.x+cos(p.rot)*p_len*0.6,--shoot it from the tip of the ship
     p.y+sin(p.rot)*p_len*0.6,
     p.chrg,
     p.rot,
     p.p
     )
   s.vx+=p.vx--in addition to the shot's own velocity
   s.vy+=p.vy--let it inherit yours
   --pushback
   p.vx-=cos(p.rot)*p.chrg*0.8
   p.vy-=sin(p.rot)*p.chrg*0.8
   --sfx
   if p.chrg==1 then
     sfx(1)
   else
     sfx(0)
 end
 --reset chrg
 p.chrg=0
 end
```

And this to the bottom of `p_draw()`, to represent charging a shot:

```
--charge graphic
if p.chrg>0 then
 circfill(pfx,pfy,p.chrg*3,10)
end
```

We could talk about this, but you could also just play it. You've earned it. Talk can wait. Pull aside a loved one and shout “Hey! I made a versus game and it's probably really cool play it with me NOW” and they shall surely answer your summons.

If it does not live up to their or your hopes, then I challenge you:
Make it better. This is yours, so own it and take it as far as you want it to go. I've shared as much as I can without lecturing your head off (hopefully), so if what I've offered has impacted you, take it forth with everything else you have in you and bring a cool new thing into this world.

4. HERE WE LIE AT JOURNEY'S END

You have a game now. It is Blasteroids, or maybe it is something entirely your own instead. Many would say it's missing things like a “win state” or a “title screen”, but do not listen to them, those are window dressings you may adorn as your please. Your game is only ever what you want it to be, so pay no mind to anyone who presumes your interest in one value system or form of presentation or another.

Here are some additional goals I've thought of that you can pursue if you like:

• Add visual and sound effects to make all the game's little interactions pop
• Tweak the physics around to create an entirely different play dynamic
• Add a new mechanic which takes advantage of btn5 in an interesting way
• Make objects loop around the screen more smoothly
• Create a simple AI for 1-player mode. The trick is to not confine yourself into thinking it has to behave just like a player! Let it exist on its own.
• Do something entirely outside the boundaries of my other suggestions.

Before we part ways, I'd like to offer a closing thought: Multiplayer games are wonderful things that bring people together. Much like a dinner with loved ones, they are never uncomplicated or without squabbles and tension, but that's what makes them special. You bicker, you laugh, your body probably does something gross at some point, and ultimately it brings us closer to one another. We close gaps and wounds we suffer from the frustrations of daily life that sometimes threaten to push us apart. At every opportunity we must hold close these precious rituals which remind us that we love each other.

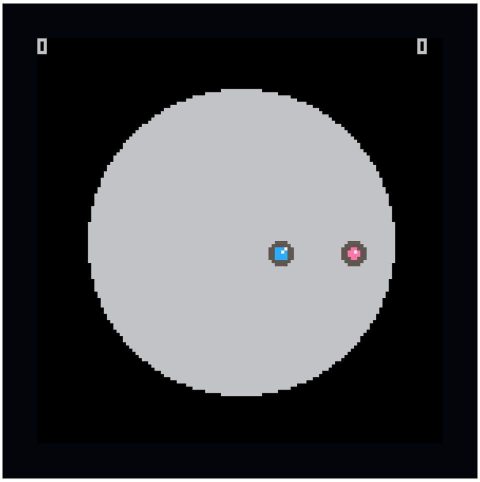
Yours,

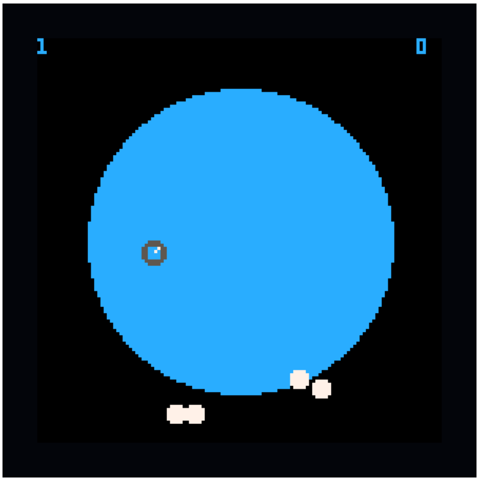
-- Lulu Blue
@luluisbluetoo

http://www.lexaloffle.com/bbs/?tid=2458

SUMO PICO

Sumo Pico is actually a demake of a small jam game I made with my partner, Britt, called "Sumo Puckii". Sumo Puckii is an inertia driven bumper-boat-meets-sumo-wrestling game that we put out to test a bunch of stuff in Game Maker, including how little friction we could put on something and still have the player feel in control, pythagorean distance, exporting to android, and of course, multiplayer. It is funny then that remaking Sumo Puckii has kind of become my benchmark for testing new development environments, especially the multiplayer side of things, particularly with PICO-8, which until a recent announcement, hasn't been branded so much as a local-multiplayer powerhouse.





Sumo Pico was my very first game made with PICO-8, so as such, if you were to dig through the source code, it would be an utter mess. Because I had essentially made this game before, I knew what to do, just not how to do it. In fact, before this game, I had never even touched Lua before. I will not pretend to be a professional, but the code's design did have a plan, and works a little like this:
A container for information about both players is initialized. PICO-8 calls them "tables" I believe.

```
ACTORS={}
```

This allows us to treat each object contained in the "ACTORS" table essentially the same, while still being able to refer to each individual one as a separate instance. In the INIT() loop, I initialize a variable for each player, and tell it to run a function that sets up a bunch of necessary variables for them.

```
FUNCTION _INIT()
P1 = CREATE_ACTOR(32,64,2,0)
P2 = CREATE_ACTOR(96,64,3,1)
END
FUNCTION CREATE_ACTOR(X,Y,IMG,N)
  LOCAL P = {}
  P.X = X
  P.Y = Y
  P.DX = 0
  P.DY = 0
    P.IMG = IMG
  P.N = N
  P.H = 3
  P.W = 3
  ADD(ACTORS,P)
  RETURN P
END
```
The function requires that we pass in an x and y position at which to create the player, an image (or the index of which sprite to use), and "N", the identifier of which player is in control of the object (0 being player 1). Then, it creates a new table "P", which will contain all of the variables about the instance of the player, adds a bunch of variables into that table, and then adds that table to our "ACTORS" table, finally returning "P" so that our variable "P1" or "P2" can easily refer to it.
In the UPDATE() loop, I use the "FOREACH" loop, which applies the function to all objects in a table, to call a function "MOVE_ACTOR()" which will check the input from a controller with the id number (N) of each of the players, and then make them move accordingly.
```
FUNCTION _UPDATE()
  FOREACH(ACTORS,MOVE_ACTOR())
END

FUNCTION MOVE_ACTOR(P)
  P.DX += (0-P.DX)*0.05
  P.DY += (0-P.DY)*0.05
  IF (BTN(0,P.N)) THEN
    P.DX += (-4-P-P.DX)*0.1
  END
  IF (BTN(1,P.N)) THEN
```

```
    P.DX+=(4-P.DX)*0.1
  END
  IF(BTN(2,P.N))THEN
    P.DY+=(-4-P.DY)*0.1
  END
  IF(BTN(3,P.N))THEN
    P.DY+=(4-P.DY)*0.1
  END
  P.X+=P.DX
  P.Y+=P.DY
END
```

For the sake of simplicity, I have removed the collisions from this code example, and because this is about multiplayer in PICO-8, I wanted to focus more on the input here. To get input in PICO-8, you use the built in function "`BTN(BUTTON,CONTROLLER)`".
Because the foreach loop automatically fed the variable "`P`" (which player) to the function "`MOVE_ACTOR`", and because we assigned each "`P`" a variable "`N`" to refer to which controller will be in control of them, when querying input, instead of individually checking if player 1's controller is doing xyz, and player 2's controller is doing abc, we can just check if p's controller is doing something, because p contains both the information about who is in control and the velocities and the x and y positions. This is particularly useful for expansion. When PICO-8 adds up to 8 player support, all that I would have to change is how many players I initialize in order to support it! In fact they could add up to an infinite number of possible players and it would still work (in theory). I've been making multiplayer games for quite some time now, and I've found that the best way to manage multiple player objects and inputs is to make them dynamic, meaning, if the objects are meant to behave the same way and only require a few changes between them, it will make your life easier to have them be the same, and make the changes based on an identifier number. PICO-8 is surprisingly accommodating with this, and I hope that everyone embraces the new multiplayer expansions with love and caring!

--Cullen Dwyer
@cullenddwyer
http://www.lexaloffle.com/bbs/?tid=2191

System

load filename

save filename

export filename.html

folder

ls

run

resume

reboot

stat x

info

flip

printh str

Graphics

clip [x y w h]

pget x y

pset x y [c]

sget x y

sset x y [c]

fget n [f]

fset n [f] v

print str [x y [col]]

cursor x y

color col

cls

camera [x y]

circ x y r [col]

circfill x y r [col]

line x0 y0 x1 y1 [col]

rect x0 y0 x1 y1 [col]

rectfill x0 y0 x1 y1 [col]

pal c0 c1 [p]

palt c t

sspr sx sy sw sh dx dy [dw dh] [flip_x] [flip_y]

Collections

add table val

del table val

all table

foreach table func

pairs table

#table

Input

btn [i [p]]

btnp [i [p]]

Audio

sfx n [ch [offset]]

music [n [fade [ch_mask]]]

Map

mget x y

mset x y v

map cel_x cel_y sx sy cel_w cel_h [layer]

Memory

peek addr

poke addr val

memcpy dest src len

reload dest src len

cstore dest src len

memset dest val len

Math

max x y

min x y

mid x y z

flr x

cos x

sin x

atan2 dx dy

sqrt x

abs x

rnd x

srand x

band x y

bor x y

bxor x y

bnot x

shl x y

shr x y

Strings

#str

str0..str1

sub str start [end]

Cartridge Data

cartdata id

dget inde   x

dset index val

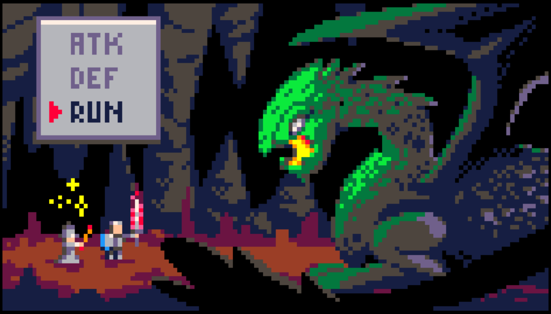
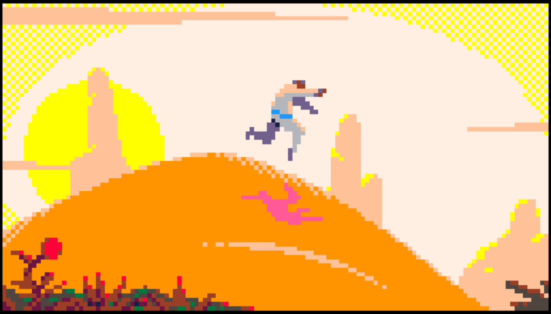
RAM layout

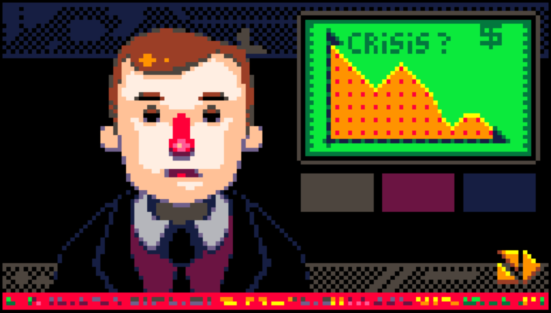
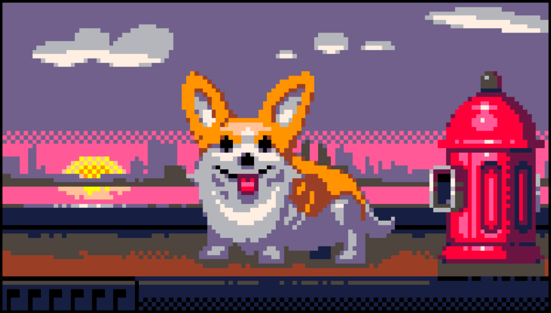
0x0000	gfx
0X1000	gfx2/map2 (shared)
0X2000	map
0X3000	gfx_props
0X3100	song
0X3200	sfx
0X4300	user-de-fined
0X5f00	draw state
0x5f80	persistent cart data
0x5fc0	(reserved)
0x6000	screen (8k)

Colour palette

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

@obono


ATK
DEF
▶RUN



CRISIS?
$


loop de loop
mrHk.tumblr.com

kinder