PICO-8 zine #4

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

APRIL 2016

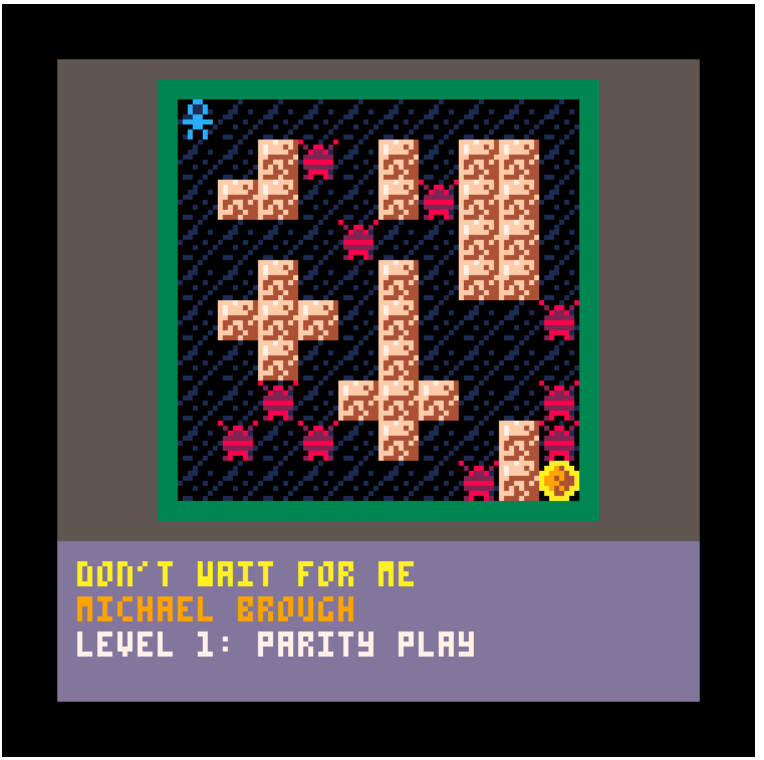
CONTENTS

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 @pietepiet
Special thanks to @dan_sanderson and @lexaloffle

DON'T WAIT

I made this to illustrate some of my reasoning behind a particular design decision in the roguelike games I've made. It's probably better to play it before reading this, it only takes a couple of minutes.

posted on the bbs: **http://www.lexaloffle.com/bbs/?tid=2991**



I hope it speaks for itself fairly well but I'll put it in context. In Rogue, and many other games following in its lineage, you could press a key (often ".") to skip a turn. In my games Zaga-33 and 868-HACK (and the unreleased Imbroglio) you can't (at least, not freely - there are ways). This goes against the genre expectations and so I sometimes get complaints - "why can't I wait in place?", "i can't find the wait button", "how do you wait?", "this game is bad because sometimes the enemy hits you", etc.

The three levels of this little game demonstrate three different approaches to "wait".

Level 1: you can't wait. All you can do is take a step or hit an enemy. Both player and enemies die in just one hit - this wouldn't be ideal for a more complex roguelike because a lot of the play in these games is making decisions about which resources to sacrifice ("do I take some damage or do I use up an item?") but it helps make everything very clear for this example. It means if an enemy is an even number of steps away it is guaranteed to get the first hit and kill you, if an odd number you can safely kill it. The twist is that you don't move when you hit an enemy, allowing you to convert those evens to odds if you're careful about the order you meet them in. (See Aaron Steed's Ending for a fully fleshed out game based around this idea.)

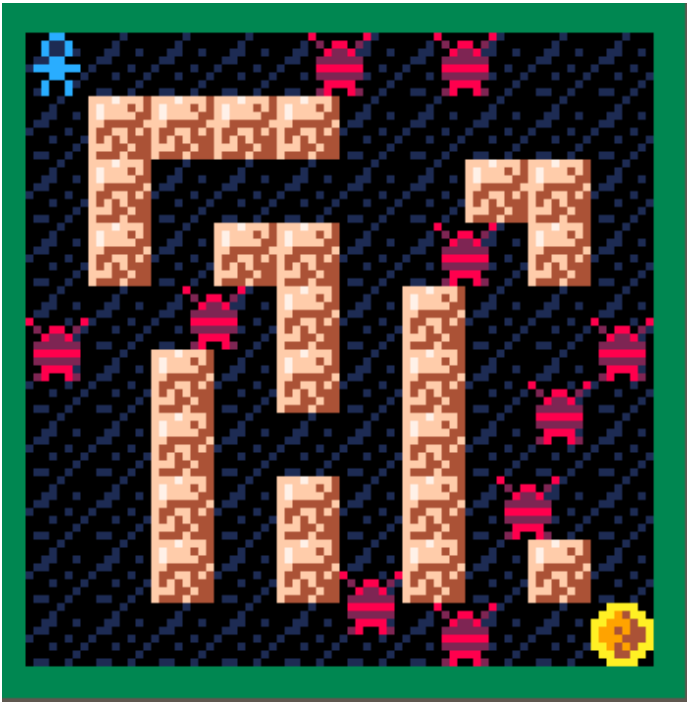
Level 2: you can wait. You can just press a key and all the enemies move closer, turn scary evens to nice safe odds. Depending on the order you hit them in you might need to wait a lot or very little, but it doesn't matter because you can complete it either way. The tactics in this level end up being a lot simpler; maybe it makes more sense that you can wait but something interesting has been lost. I'd also say there's a loss in simplicity - an extra control is an extra thing to tell you about, extra text on the screen. But a lot of games have this rule and it works well for them. (In Rogue and many classic roguelikes waiting to get the first hit isn't free the way it is here because there are various timers that tick up each turn even as you wait, governing hunger, enemy spawns, corruption, etc. - you'll starve to death in 100 turns so maybe you don't want to spend one of them standing still. These costs tend to be quite minor compared to being damaged, but still they do sometimes create situations where it's better to take an extra hit than an extra turn. But when you're making a smaller game that doesn't have so many ornate systems counterbalancing each other these costs might need to be made explicit.)

Level 3: the enemies can wait too. I've been told that it's unrealistic to not have a wait button, but if so then it's unrealistic for enemies not to wait as well - very unrealistic for the player

to always get the first hit. One of the elements cited in the Berlin Interpretation as a common characteristic of roguelikes is "Rules that apply to the player apply to monsters as well". And of course they do exactly what you've been doing in the previous level - waiting whenever you're two steps away, refusing to get hit first. Maximum realism! This fundamentally doesn't work with the "one hit point" rule, leaving the level impossible, but you can see how unpleasant it would be to have this tactic turned against you in a less strict game too.

Part of the enjoyment of roguelikes is outwitting enemies who outnumber you. The capacity to patiently wait while they blindly charge in can be one expression of this, but I don't think it's a very good one because it's always the right choice. Applying one simple method every time doesn't express much cleverness, better to have to think up new solutions to different situations.

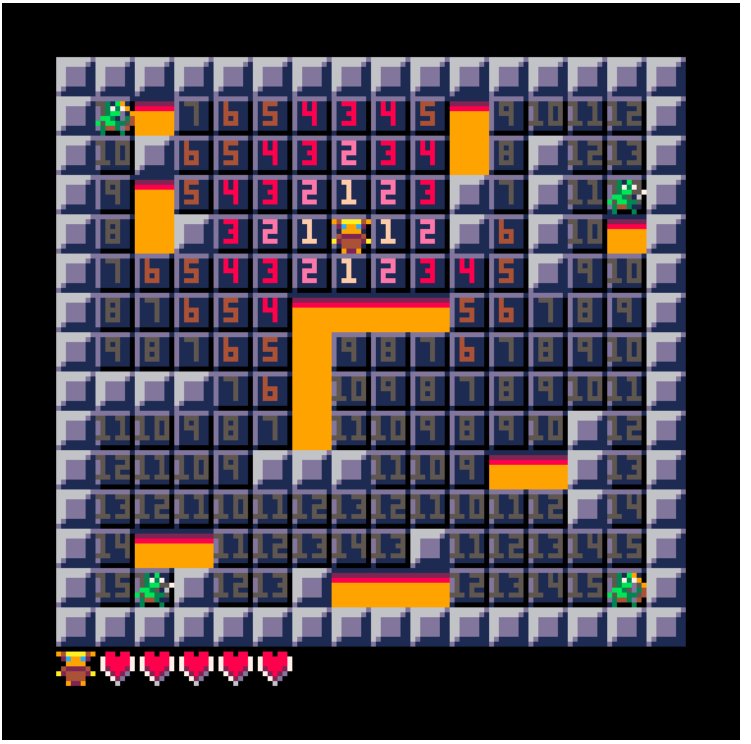
Michael Brough
@smestorp



AI MOVE SPECIAL ROGUELIKE

Here's a tutorial on how design a basic AI for monsters moving on a grid.

http://www.lexaloffle.com/bbs/?pid=18367&tid=2986&autoplay=1#pp



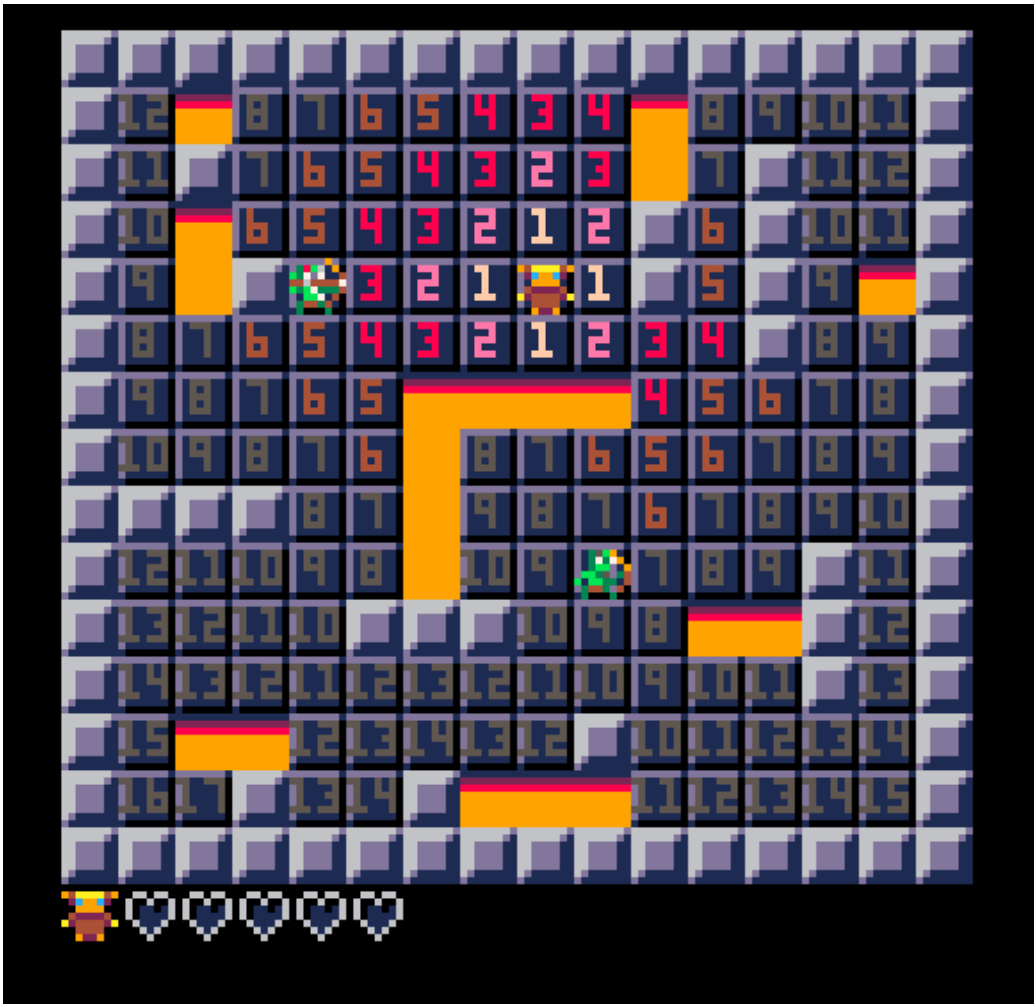
We will focus on two behaviors :
- fighters : follow hero to hit him
- archers : try to reach a line of sight and aim at hero

In the `_INIT()` function I read the map information to draw level, spawn hero/monsters and store useful data in a squares collection.

The `EXPAND(A,DID)` function takes all squares in `<A>` as a start position `(=0)` and then computes the distance from all others squares.

You can see the values by pressing `(2)` during the test. `RUN_MONCE)` will move the monster by checking the 4x nearby squares.If it finds a square with a lower distance it will move in this direction.Once they reach a square with a`DIST`of`0` archer will aim at hero by defining a `SHOOT_DIR` value.

Benjamin Soule
@benjamin_soule_



A* pathfinding in PICO-8

A* pathfinding is an efficient way to find the shortest path from one position to another if possible. It is widely used in video games but also in the transportation and computer networking industries.

We are going to implement A* pathfinding in PICO-8 in a simple single screen application by first implementing a simpler breadth-first pathfinding algorithm then extending it into an A* pathfinding algorithm.

1.The Map

Let's first draw a map in PICO-8 using this code:

```
function _init()
end

function _update()
end

function _draw()
  cls()
  mapdraw(0,0,0,0,16,16)
end
```

Next we will draw some basic sprite

They include:

- Blank - 000
- Wall - 001
- Goal - 016
- Start - 017
- Path - 018

The Walls block the path from the Start to the Goal, we will draw a path with the Path sprite.

Now we can create a map with a Start and Goal position somewhere and walls that will block the path.

2. Breadth-first Pathfinding

Next we want to search through the locations starting from the Start using breadth first search, this will find the neighbours of each location searched and then search the neighbours ignoring the walls. This code doesn't look for anything just yet, it just iterates through each map position in a kind of expanding circle outward starting at Start.

Update `_INIT()` with this code.

```
FUNCTION _INIT()

  START = GETSPECIALTILE(17)
  GOAL = GETSPECIALTILE(16)

  FRONTIER = {}
  INSERT(FRONTIER, START)
  CAME_FROM = {}
  CAME_FROM[VECTOINDEX(START)] = "NONE"
```

```
  WHILE #FRONTIER > 0 DO
    CURRENT = POPEND(FRONTIER)

    LOCAL NEIGHBOURS = GETNEIGHBOURS(CURRENT)
    FOR NEXT IN ALL(NEIGHBOURS) DO
      IF CAME_FROM[VECTOINDEX(NEXT)] == NIL THEN
      INSERT(FRONTIER, NEXT)
      CAME_FROM[VECTOINDEX(NEXT)] = CURRENT
    END
  END
END
```

I've added a few helper functions as well. Some of these such as `POPEND()` and `REVERSE()` are added because pico-8 uses a subset of the lua language. I won't go into detail about these because this article is about the A* search.

```
-- FIND ALL EXISTING NEIGHBOURS OF A POSITION THAT ARE NOT WALLS
FUNCTION GETNEIGHBOURS(POS)
  LOCAL NEIGHBOURS={}
  LOCAL X = POS[1]
  LOCAL Y = POS[2]
  IF X > 0 AND (MGET(X-1,Y) != WALLID) THEN
    ADD(NEIGHBOURS,{X-1,Y})
  END
  IF X < 15 AND (MGET(X+1,Y) != WALLID) THEN
    ADD(NEIGHBOURS,{X+1,Y})
  END
  IF Y > 0 AND (MGET(X,Y-1) != WALLID) THEN
    ADD(NEIGHBOURS,{X,Y-1})
  END
  IF Y < 15 AND (MGET(X,Y+1) != WALLID) THEN
    ADD(NEIGHBOURS,{X,Y+1})
  END
 RETURN NEIGHBOURS
END

-- FIND THE FIRST LOCATION OF A SPECIFIC TILE TYPE
```

```
FUNCTION GETSPECIALTILE(TILEID)
  FOR X=0,15 DO
    FOR Y=0,15 DO
      LOCAL TILE = MGET(X,Y)
      IF TILE == TILEID THEN
        RETURN {X,Y}
      END
    END
  END
  PRINTH("DID NOT FIND TILE:"..TILEID)
END

-- INSERT INTO START OF TABLE
FUNCTION INSERT(T,VAL)
  FOR I=(#T+1),2,-1 DO
    T[I]=T[I-1]
  END
  T[1]=VAL
END

-- POP THE LAST ELEMENT OFF A TABLE
  FUNCTION POPEND(T)
    LOCAL TOP = T[#T]
    DEL(T,T[#T])
    RETURN TOP
END

FUNCTION REVERSE(T)
  FOR I=1,(#T/2) DO
    LOCAL TEMP = T[I]
    LOCAL OPPINDEX = #T-(I-1)
    T[I]=T[OPPINDEX]
    T[OPPINDEX]=TEMP
  END
END

-- TRANSLATE A 2D X,Y COORDINATE TO A 1D INDEX AND BACK AGAIN
FUNCTION VECTOINDEX(VEC)
  RETURN MAPTOINDEX(VEC[1],VEC[2])
```

```
END
FUNCTION MAPTOINDEX(X,Y)
  RETURN ((X+1)*16)+Y
END
  FUNCTION INDEXTOMAP(INDEX)
    LOCAL X = (INDEX-1)/16
    LOCAL Y = INDEX - (X*W)
  RETURN {X,Y}
END
```

Additionally add this code to the end of `GETNEIGHBOURS()`, just before the `RETURN`, to make the paths walk in a diagonal instead of in a square, this is the same distance but it looks shorter when walking a diagonal.

```
  IF (X+Y)%2 == 0 THEN
    REVERSE(NEIGHBOURS)
  END
```

Now we can add some code to stop searching if we find the Goal, add this just below `CURRENT = POPEND(FRONTIER)` in the `WHILE` loop.

```
  IF VECTOINDEX(CURRENT) == VECTOINDEX(GOAL) THEN
    BREAK
  END
```

This will break out of the `WHILE` loop if the current map position is the goal, no need to search the rest of the map.

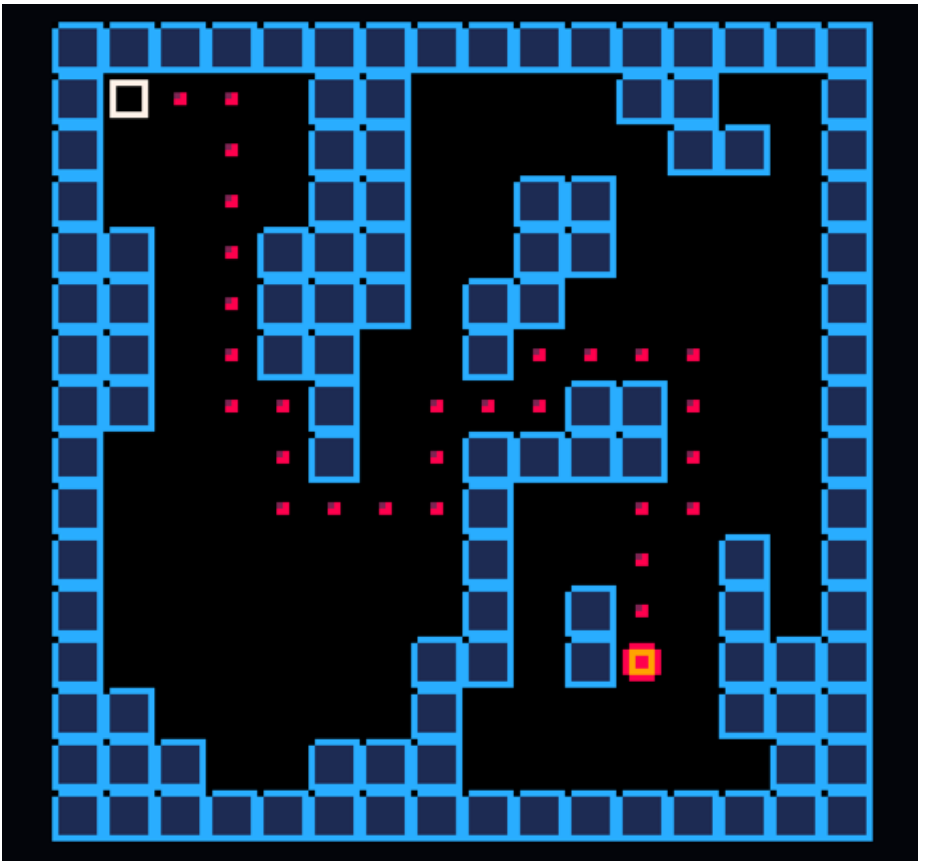
We can now draw a line using the Path tiles along the points in `CAME_FROM`. We will reverse them so they are drawn from Start to Goal.

Add this below the `WHILE` loop.

```
  CURRENT = CAME_FROM[VECTOINDEX(GOAL)]
  PATH = {}
  LOCAL CINDEX = VECTOINDEX(CURRENT)
```

```
  LOCAL SINDEX = VECTOINDEX(START)
WHILE CINDEX != SINDEX DO
  ADD(PATH,CURRENT)
  CURRENT = CAME_FROM[CINDEX]
  CINDEX = VECTOINDEX(CURRENT)
END
REVERSE(PATH)
FOR POINT IN ALL(PATH) DO
  MSET(POINT[1],POINT[2],18)
END
```

Now we should see something like this when we run the game.



This is great! the path goes straight from the Start to the Goal but if we add a new sprite at position 19 and add this to the **WHILE** loop.

```
local nextindex = vectoindex(next)
if (nextindex != vectoindex(start)) and
  (nextindex != vectoindex(goal)) then
  mset(next[1],next[2],19)
end
```

Then we can see where the algorithm has searched.
You can see the algorithm has expanded outwards until it found the goal, not very efficient.

3. A* Pathfinding

A* Pathfinding will find the shortest path quicker by searching less positions and ranking each position found by how close it is to the goal. This is done with a priority queue data structure.

Since pico-8 doesn't have a priority queue built in we can make a simple insertion sort function for a lua table. This isn't the fastest method of sorting but it is fairly simple.

```
--insert into table and sort by priority
function insert(t,val,p)
  if #t>=1 then
    add(t,{})
    for i=(#t),2,-1 do

      local next = t[i-1]
      if p < next[2] then
        t[i]={val,p}
        return
      else
        t[i]=next
      end
    end
      t[1]={val,p}
    else
      add(t,{val,p})
    end
end
```

We also change the return line of `POPEND()` so we only return the value part.

```
  RETURN TOP[1]
```

Then we need to add a heuristic, this is how far away from the goal each point is. A* uses this heuristic to calculate the total cost of traveling to each node as it is searching and will check the lowest cost paths first.

This implementation of a heuristic function simply uses manhattan distance.

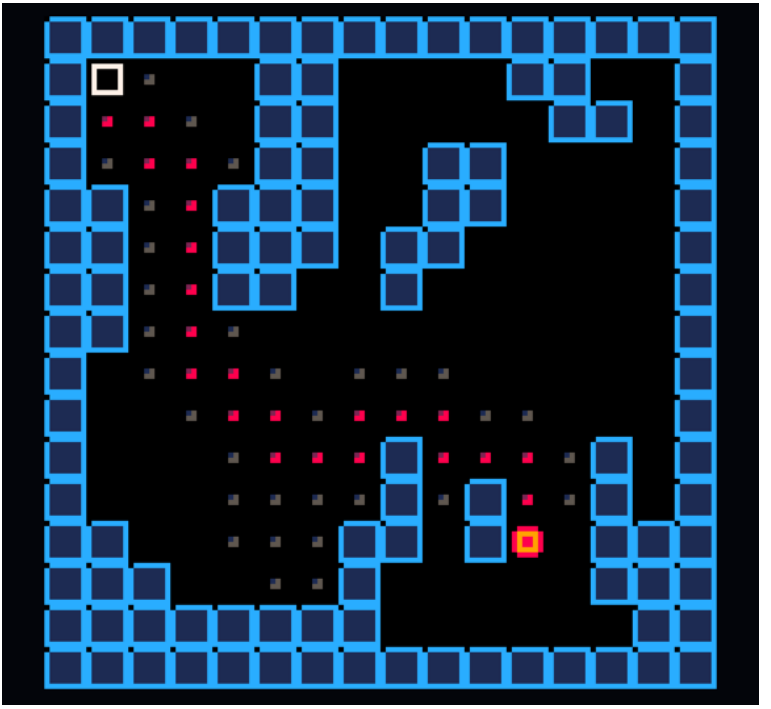
```
FUNCTION HEURISTIC(A, B)
RETURN ABS(A[1] - B[1]) + ABS(A[2] - B[2])
END
```

Finally we will update the while loop to use a running cost and prioritising the shortest paths. It also updates paths already checked if a faster way through that tile has been found.

```
FRONTIER = {}
INSERT(FRONTIER, START, 0)
CAME_FROM = {}
CAME_FROM[VECTOINDEX(START)] = NIL
COST_SO_FAR = {}
COST_SO_FAR[VECTOINDEX(START)] = 0

WHILE #FRONTIER > 0 DO
  CURRENT = POPEND(FRONTIER)

  IF VECTOINDEX(CURRENT) == VECTOINDEX(GOAL) THEN
    BREAK
  END

  LOCAL NEIGHBOURS = GETNEIGHBOURS(CURRENT)
  FOR NEXT IN ALL(NEIGHBOURS) DO
    LOCAL NEXTINDEX = VECTOINDEX(NEXT)
  LOCAL NEW_COST = COST_SO_FAR[VECTOINDEX(CURRENT)]
```

```
  IF (COST_SO_FAR[NEXTINDEX] == NIL) OR (NEW_COST <
COST_SO_FAR[NEXTINDEX]) THEN
    COST_SO_FAR[NEXTINDEX] = NEW_COST
    LOCAL PRIORITY = NEW_COST + HEURISTIC(GOAL, NEXT)
    INSERT(FRONTIER, NEXT, PRIORITY)

    CAME_FROM[NEXTINDEX] = CURRENT

      IF (NEXTINDEX != VECTOINDEX(START)) AND
      (NEXTINDEX != VECTOINDEX(GOAL)) THEN
        MSET(NEXT[1], NEXT[2], 19)
       END
     END
   END
END
```

We also added a `COSTPERMOVE` variable, this can be the same for all moves or can be based on different terrain e.g. walking in mud is slower.
And if we run it now the map shows a lot less map checked.



Congratulations! You have just implemented A* pathfinding! Get creative and apply it to something amazing and show everybody!

You can find the source for this at article at:

http://www.lexaloffle.com/bbs/?tid=3131

4.What else can we do with A* pathfinding?

-Use a smaller amount of nodes on the map rather than every point e.g. corners.
- Add different costs for different types of terrain, mud, ice etc.
- Try costs in one direction different to another e.g. a conveyer belt.
- Try a moving target!

Further reading

http://www.redblobgames.com/pathfinding/a-star/introduction.html
https://en.wikipedia.org/wiki/A*_search_algorithm
https://en.wikipedia.org/wiki/Priority_queue
http://www.raywenderlich.com/4946/introduction-to-a-pathfinding

Richard Adem
@richy486

Traps For Absolutely Every Imaginable Occasion

So if you are making a roguelike then you are going to put traps in it, because you must. You must because the only reason anyone ever makes or attempts to play roguelikes is because they want to bash against infinite unknowable brick wall universes over and over again until they find the one brick that is loose and then they can say, look, I did it, I won. This is the thing to say when you finally win a roguelike even though it really means "I won (that time)" and "I gave myself over to be entombed in a random superstructure but I was able to accurately observe the rules of its generation" and "I am drawing a map of where the loose brick is, slowly, blindly: this is what I am doing with my life." You are going to put traps in your roguelike because you don't just want a tall beautiful noble and unyielding maze: you want a mean maze that pushes back. Nobody can contend with mystery and the unknown without the horror of Unintended Consequences and so you will include traps.

What Is The Best Sort Of Trap

The absolute best sort of trap you can make for your game will be instantly recognizable to your player and they will want to step on it. I recommend a small switch on the ground that counts down how many steps you need to take in order to step on the trap and then a large sign that celebrates the player for stepping on the trap. It might be good to give the player some gold for stepping on the trap as well. You want the trap to be as exciting as possible.

Here is a short list of my favorite traps in roguelike games:

1. **The sink**, from NetHack. The ideal trap, taking a friendly form, having a handful of positive uses (identifying rings, free drinks) and many horrifying ones (black ooze!!!!), no player can resist the delightful sink.

2. **The statue trap**, also in NetHack. Most conventional NetHack traps just spring out of the darkness and are quite rude but the statue trap beckons you towards your own demise with sweet songs and curious rewards. Finally transforms prosaicly into A Real Monster Where You Assumed There Would Only Be A Statue, e.g., a Kiwi Statue would become Oh My God An Actual Kiwi. I love statues.

3. **The corruption trap**, from Ancient Domains of Mystery. Promises to eventually transform one into a "writhing mass of primal chaos", which is exactly the reason we all play computer games isn't it.

4. **The doppelganger staff**, from Shiren the Wanderer. Another example of Maybe The Best Kind of Trap Ever, the staff resembles a Useful Item that will transform a monster into the likeness of the protagonist, which will cause all other monsters to attack the target of the staff. But! Ta da! A victorious monster levels up and becomes even stronger after victory, often causing Wonderful Problems.

5. **The bloodwort seed pod**, from Brogue. Rendered in a forbidding crimson, the bloodwort stalk dares the player to approach before bursting into a cloud of red mist. "Surprise!" it says. "I'm healing you!"

6. **Siphoning** in 868-HACK. My favorite trap. Siphoning is necessary and desirable for so many reasons (new programs, points, money, energy) but it always produces Bad Things. The player falls gleefully again and again into the arms of death.

Other kinds of traps that launch arrows or fireballs or boulders or poison gas are good too even if they are a bit on the nose. Your players will learn how to mitigate even the most random and unfair circumstances, which they will confuse with a kind of

prickly complexity or they might think that you have a sense of humor and that you are attempting to communicate with them. Traps make a world snap and snarl with brutish life so make as many of them as possible.

Making As Many Traps As Possible in VNOOFIS

My game VNOOFIS (**http://www.lexaloffle.com/bbs/?tid=3039**) is about feeling your way through a slimy writhing fungus cave and finding luminous spores to press into your flesh. In an effort to include All Of The Positive Feelings of Traps I decided to make three kinds of traps:

1. Treasure Trap

2. Boulder Trap

3. Arithmetic Trap

The specific form I chose for Treasure Trap was Every Single Wall in VNOOFIS. Touching a wall allows a player to See (or Feel) What's Going On which is a Lovely Treasure that cannot be abided without beautiful anguish so the wall produces a hostile Slime. The Slime will hurry along to the place where the player is but the player can retreat with Powerful Knowledge. This gives a very nice feeling of Impending Doom That Can Perhaps Be Avoided, so everyone feels Clever.

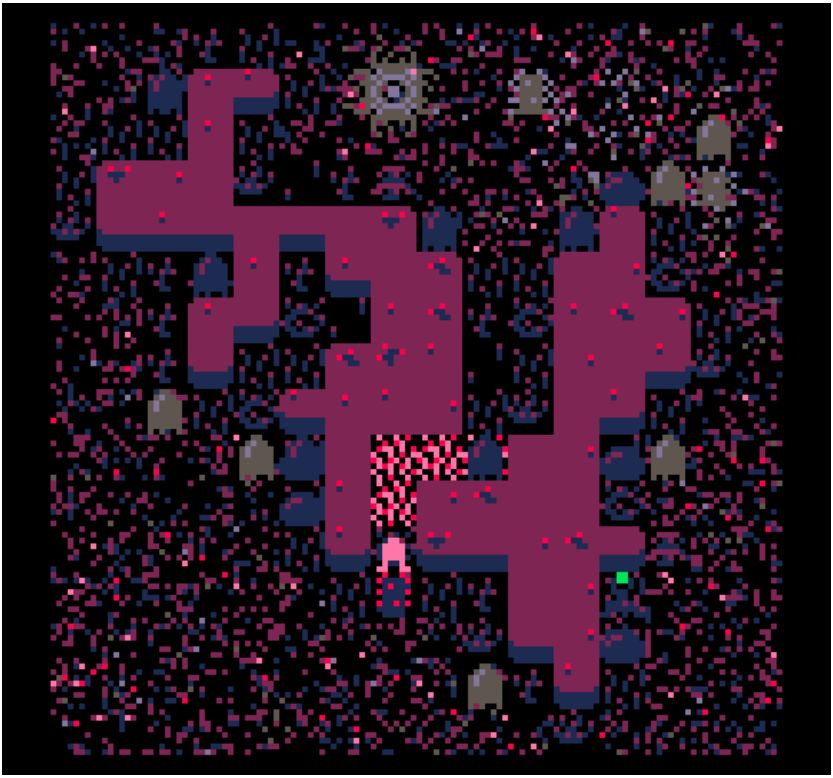
The Boulder Trap in VNOOFIS happens when you are confronted by Too Many Slimes, Maybe so you step into a wall, smashing it with your body into a carpet of nerves and flesh. You gain an Avenue of Escape but you must keep going because the wall is growing back in as an Angry Blister which will Explode if you attempt to return. You have traded Flexibility for Future Complexity and again you feel Clever.

The Arithmetic Trap is what I think is the most fun part of dying in Shiren the Wanderer, saying to everyone who can hear you, "oh no the enemies have killed each other in a fashion that was not

advantageous to me, creating an enemy that has statistics that are far beyond mine." In VNOOFIS there is the Segmented Worm, wandering around, difficult to avoid in their own right, but combined with a Hostile Slime! Well! You can only imagine the thrill of encountering this very special opportunity for defeat.

These are the ultimate trap combinations that I conceived for my gross cave game. They are the best ones but maybe you can think of another one!
Homework: make a game with FOUR kinds of traps (don't make any others though because making a PICO-8 game is an exercise in minimalism and restraint)!!

Kyle Reimergartin
@mooonmagic



The Roguelike*shiny*game-feel

What is the greatest feeling in roguelikes? Winning the game, in most cases. But what is another greatest feeling in roguelikes? Finding something ***shiny***. The most obvious example would be a piece of equipment, but new enemies, bosses, or any world design element has a very large *shiny* potential. Game mechanics like level-ups or a new ability can also feel *shiny*. Winning the game is *shiny*. ***Shininess*** is an amazing game-feel that we all **feel** and **love** and **the roguelike genre is the genre that has the best of it.**

1.Roguelikes and *shininess*

You may have understood from the introduction that a great factor of *shininess* is **discovery.** But roguelikes are not the only genre that has good discovery feels. Most RPGs regularly deliver new equipments and new environments to the player, well-made puzzle games generally are based on making you discover new ways of using whatever interaction you have with the game or even changing their own game-mechanics to give you brand new situations. What do roguelikes have that makes them so appropriate to the *shiny* feel?

Roguelikes are RPGs. They can easily use the classic RPG's equipment and environment renewing, and they should. But roguelikes are also the closest you can get to the arcade genre in RPGs. And the arcade genre (or the "try again" genre if you prefer) is the best genre for keeping you very close to the screen, fingers mindlessly pushing buttons, and indeed **having you the closest to the game.** Therefore arcade is the best genre for game-feels in general. And therefore roguelikes are the best at *shiny* feels.

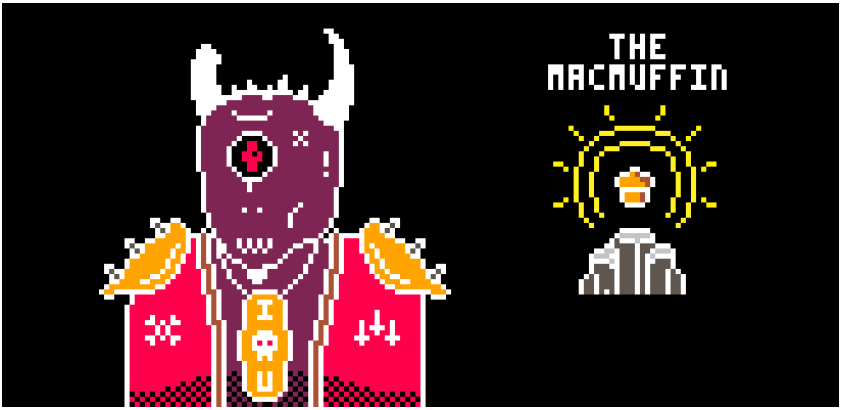
So how do you make it happen?

2.MacGuffins. MacGuffins everywhere.

A **MacGuffin** is a plot device that serves as a goal or a motivation for the main character. The princess Peach in Mario Bros is a MacGuffin. So are the coins to a further extent. In video games,

MacGuffins can be of different importance but they all share that the player will most likely go for them and when he gets one of them, it will make him happy. **MacGuffins are *shiny*.**

In roguelikes, any weapon, armor, potion, book, toilet paper or indeed any item can potentially serve as a MacGuffin. So, what makes a good MacGuffin?



First of all, MacGuffins are based on **anticipation.** If you give your player a new item without him asking for anything, you failed your MacGuffin. So first thing is **show the item to the player BEFORE giving it to him.** Then, at least make the player walk to it and pick it up. You can also make the item the reward for a quest or slaying a boss. If you chose either of these last ones, you may chose to not show the item before the deed is done or even make the reward random. That's ok but do build anticipation anyway, use your narration to make the player aware he will get something good. When the player can finally pick up the item, make it obvious. The item must catch the player's eye before he actually gets it. So use lighting effects, sound, flashing colors, text, **anything flashy.**

Then, **MacGuffins should not be deceiving.** If the player gets an item he thinks is really good and it's really bad, you broke it all. The item did feel *shiny* before he got it but now it is the darkest piece of trash he has ever wanted. There are solutions to that! One of the simplest is to make all the items tradable for something else, like money for example. In Dungeons of Dredmor by

Gaslamp Games, there is a point in the game where you get a lutefisk box, an item that can convert any other item in a quantity of lutefisk than you can then give to the lutefisk god who would give you a piece of equipment if you gave him enough lutefisk. Then, any object has **a minimal interest.** Another solution is funny/interesting tooltips. These can be information about the game's lore or bad puns or pop culture references, or anything you think the player will have some interest in. If you really want an item to be a waste of the player's inventory, make it so but then better put a very good pun on its tooltip or not make it shiny at all.

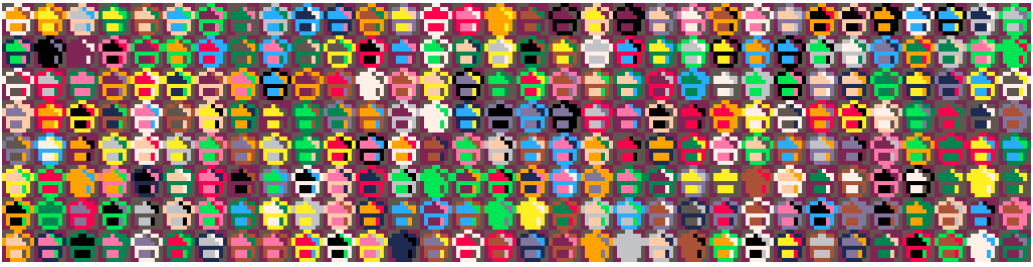
In the very successful roguelites The Binding Of Isaac by Edmund McMillen, Nuclear Throne by Vlambeer, Spelunky by Derek Yu and many other roguelites that are not turn-based, where we are even closer to the arcade genre, the shiny feels of a new item/mutation/weapon/trinket are often accompanied by an effective change in the mechanics of the player's game. Shooting works differently, you can hurt enemies by being close to them, etc... Which makes it even *shinier*!! **The more unique the trinket, the more *shiny* it is.**

And if you want ultimate MacGuffins, you have **achievements.** If one of your achievements is "find the Wooden Stick of the Doomed", you made a shitty weapon *shiny*. Well done. But if one of your achievements is "find the jeweled gold crown of the kingly royal King Bob Deluxe", you made an epically *shiny* item even *shinier*. Besides, Achievements are pretty easy to implement in a Pico-8 game: Have a table of strings, one for each achievement, and display one for a for a few seconds when the player achieves a thing!

Quests, value, lore, puns, achievements... Is there a way to make something shiny without affecting the game's mechanics or pouring lots of literary work in it?

3.The more litteral aspect of *shininess*

In Pokémon, the famous RPG by Game Freak where you have to capture pocket monsters and make them fight with other people's pocket monsters, there are "shiny" pocket monsters. These particular pocket monsters have that of different from the regular ones that they are extremely hard to find and they have **a different color palette.** Shiny pocket monsters are among the *shiniest* things to be found in all RPGs put together.

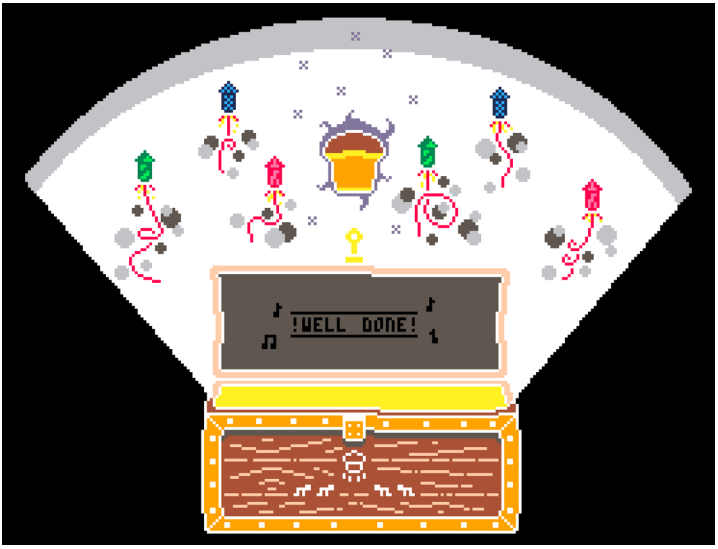


Guess what! Pico-8 has a wonderful function called **pal(c1,c2)** which will make it draw the color c2 instead of the color c1! Which means that with one 3-colored sprite and Pico-8's 16 colors, you have a potential **of 4096 variations of that same sprite!!** And sure some of them are trash and will hurt your eyes but a lot arenít and won't. Better than that yet, if your game has other animated sprites (which it definitely should), you can make your sprite's colors change over time! **Why not?**

Do keep in mind that these palette tricks work only once the player is accustomed to see the item with its original sprite. But such a simple trick can make any item *shiny*!! Also note that this can be used more extensively to make **A LOT** of items with a few sprites, take the MMO-roguelite Realm of the Mad God by Wild Shadow Studio for example, where most tiers of a same piece of equipment is the same sprite with different palettes. There, the tier palette is also an indicator that an item is better than what you currently have and, in direct consequence, *shinier*.

It also works with enemies, environments and probably other things so **use your imagination!!**

And palette swapping is not the only cheap graphical trick to make something *shiny*! In fact **any graphical or audio effect** you can add to an element of your game can make it more *shiny*. One very efficient graphical effect is **ridiculous lighting.**



A good example of ridiculous lighting is light coming from chests when you open them. Whatever is in there, someone put a lamp with it just so you know you found something *shiny*. Zelda games do that. They also make the items levitate out of the chest and put a catchy jingle over the whole thing. Keep in mind there are lots of possibility when it comes to lighting though. You could put light rays coming from the item, or it could be lightning, an aura, sparkly shapes, fireworks, flames, sparkles, circling sparkles, glitchiness, rainbows, fireflies, light bulbs, etc and all that with **all the different colors in the world.** Do pick one. **Don't be afraid to make it too extravagant, you won't, it's a video game.**

And finally, **add some animation** to the thing when it comes on screen. Make it hover up, make it roll, make it sway, make it zoom-out, make it **))shake((!** Chose whichever or make combos but when the item/enemy/npc/text/action/death-screen shows up, any animation will whisper *shiny* to the player's brain in a slightly

scary but exciting way. I have a preference for the shaking as it is generally very easy to implement. Just store two global values **shakeX** and **shakeY**, add them to the coordinates of all the things when you draw them, and multiply them both by **-0.8** each frame. Make a function **add_shake()** that gives random values to shakeX and shakeY and call that function **ANYTIME ANYTHING HAPPENS.**

4.To sum it up

Roguelikes are the best at ***shiny*** **feels** and to achieve making something *shiny* in one, you should make it a MacGuffin, based on **anticipation** and **in-game value**, and you should **add lots of graphical effects** to it, such as palette swapping, ridiculous lighting and simple animations. Along the way have also been mentioned that like game elements, **game mechanics** can also be *shiny* and that y**ou shouldn't be afraid to make things too extravagant.**

If you read all of the above, you should now be very capable of making anything *shiny* in your own roguelike or any other kind of game really. The *shiny* feel is the most adapted to roguelikes but you can certainly use it in most, if not all, games. **So make me proud and go do just that!!**

Dog Trasevol
@TRASEVOL_DOG

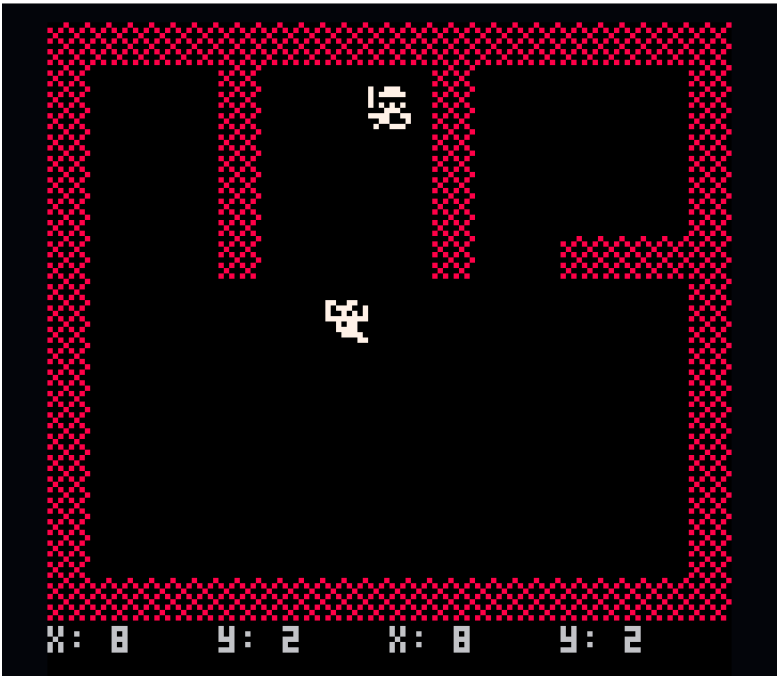
Dungeon walls

Hi. My name is Rodrigo Franco, and I'm not a game developer. I have been building web apps for over a decade — and playing video games for twice that — but I never got my hands dirty and tried to code a game. Then I found **Pico-8** (**http://www.lexaloffle.com/pico-8.php**) — its minimal approach was simple enough to make me face my own ineptitude. As a blank canvas, it was terrifying but also inspiring. I was sold.

Since I was a kid, my favorite gaming genre has been RPGs. From tabletop to early computer incursions, like Atari's adventure and the Ultima series, being able to live a classic "dungeon crawl" experience fascinated me. I think that's why I knew that when I finally started working on a game, it would be something like a roguelike.

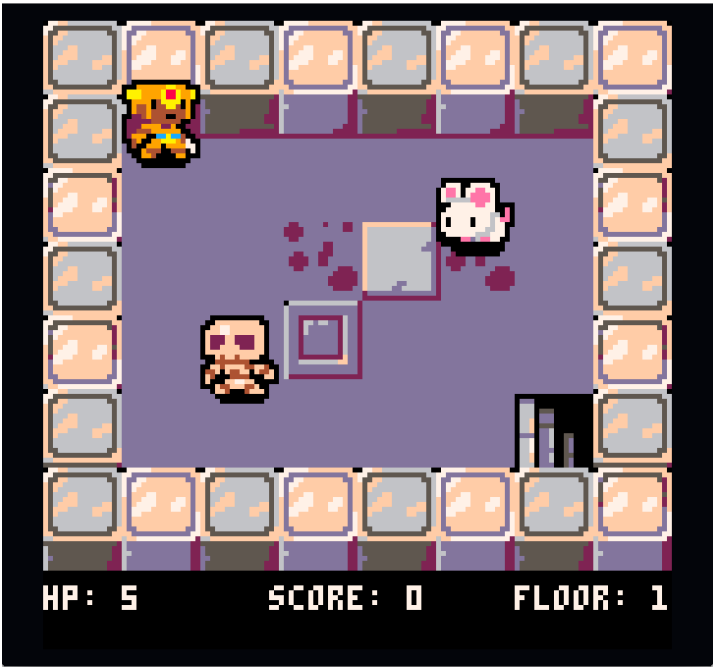
Inspired by Pico-8's minimalism, I decided to strip down my game to the bare essentials, but not graphically — I was lucky enough to be able to commission Christina Antoinette (**@castpixel**) to conjure lovely sprites for me — but since I had no idea how I would implement the game, I imposed severe restrictions on the mechanics to actually accomplish something. With that in mind, I came up with a random one-room-per-floor dungeon romp.

Like many before me, I started my gaming career (fancy eh?) by studying Zep's collision demo (**http://www.lexaloffle.com/bbs/?tid=2119**). From it I understood the "pico way" of map drawing and the use of sprite flags to define and check for solid areas. This approach worked well for my early prototypes, since everything was made of crude 8x8 sprites like this:



Everything was fine, until Christina sent me the final art:

These are not your grandma's 8x8 sprites. How could I handle that? Besides everything being 16x16, I wanted just part of the wall tiles to be solid. This would allow me to accomplish an isometric feel:



The way I was doing this (using the sprite flags) didn't work well with that. I did some research, but never found a proper solution, so I decided to come up with something myself. Here's how I solved it.

1.Setting the stage

First, I created some tables to hold the props I was about to create, all the objects that would be added to the world and what I called 'solid regions':

```
SOLIDS = {}
PROPS = {}
WORLD = {}
```

Then I came up with an ideal way to add items to the world. I wanted it to be something like this:

```
add(world,props.muddy_wall(0,15))
```

This call would add a muddy_wall to the world and position it on the screen at `x=0` and `y=15`.

To make this work, I added the following to `props`:

```
props={
muddy_wall=function(_x,_y)
  local prop={sprite=64,width=16,height=24,solidwidth=16,
  solidheight=16,x=_x,y=_y}
  add(solids,{prop.x,prop.y,prop.solidwidth,prop.solid-
height})
  return prop
end
}
```

`props.muddy_wall` is the code representation of a graphical element store at position 64 of the sprite sheet. The element has 16px of width and 24px of height, from what 16 by 16 are solid. By calling it with 0 and 15 as params, the function also adds a new element into `solids`, like this:

```
add(solids,{0,15,16,16})
```

By returning local variable `prop`, the function also allows it to be added to the world table.

From there I could just add elements to my game by passing `muddy_wall` multiple times with different `x` and `y` coordinates. I could also create new props by creating functions like `bright_wall`, `greek_column` or `stairs`. The sky's the limit!

2.Drawing the elements

With everything in the memory, it was now time to draw things. Again I tried to come up with the method signature first, then come up with a strategy on how to implement it. Here's what I wanted to do:

```
FOREACH(WORLD.PROPS._DRAW)
```

To be able to do that, I added the following method to props:

```
_DRAW=FUNCTION(P)
 SPR(P.SPRITE,P.X,P.Y,(P.WIDTH/8),(P.HEIGHT/8))
END
```

A simple `SPR` method call with the prop params. That was easy!

3.Making things solid

Now the real deal: How can we make props solid? As I said earlier, for each prop added to the `WORLD` table, we also added an item to `SOLIDS` with `X` and `Y` coordinates and width and `HEIGHT` of a solid area. It's time to put this to good use.

I'm not going to enter in details about how my actors traverse the dungeon. Just imagine I have a function called `MOV` that expects an `ACTOR` as param. This function calculates the actor's attempted position, and if the position is valid, moves the actor there. Something like this:

```
FUNCTION MOVE(ACTOR)
 IF VALID_MOVE(ACTOR.ATTEMPTED_X,ACTOR.ATTEMPTED_Y)
  DO_MOVE(ACTOR)
 END
END
```

Now imagine this other version:

```
FUNCTION MOVE(ACTOR)
 IF SOLID_AREA(ACTOR.ATTEMPTED_X,ACTOR.ATTEMPTED_Y)
  DO_NOT_MOVE(ACTOR)
 ELSE
  DO_MOVE(ACTOR)
 END
END
```

How would I implement `SOLID_AREA`? Here's how I did it:

```
FUNCTION SOLID_AREA(X,Y,A)
 LOCAL X = X + A.H
 LOCAL Y = Y + A.W

 FOR ELEMENT IN ALL(SOLIDS) DO
 IF (X >= ELEMENT[1]) AND ((ELEMENT[1]+ELEMENT[3]) >= X) AND
 (Y >= ELEMENT[2]) AND ((ELEMENT[2]+ELEMENT[4]) >= Y) THEN
   RETURN TRUE
 END
 END
END
```

Let's break this down. Here's what I'm doing, bullet by bullet:

* Add the actor's height to the attempted `X`
* Add the actor's width to the attempted `Y`
* Iterate all items in `SOLIDS`, and for each one check if the `X` and `Y` point is inside the solid area
* If we find the point inside of at least one element, returns `TRUE`

If `SOLID_AREA` returns true, I don't move the actor. Otherwise, `DO_MOVE` is executed.

3. Wrapping up

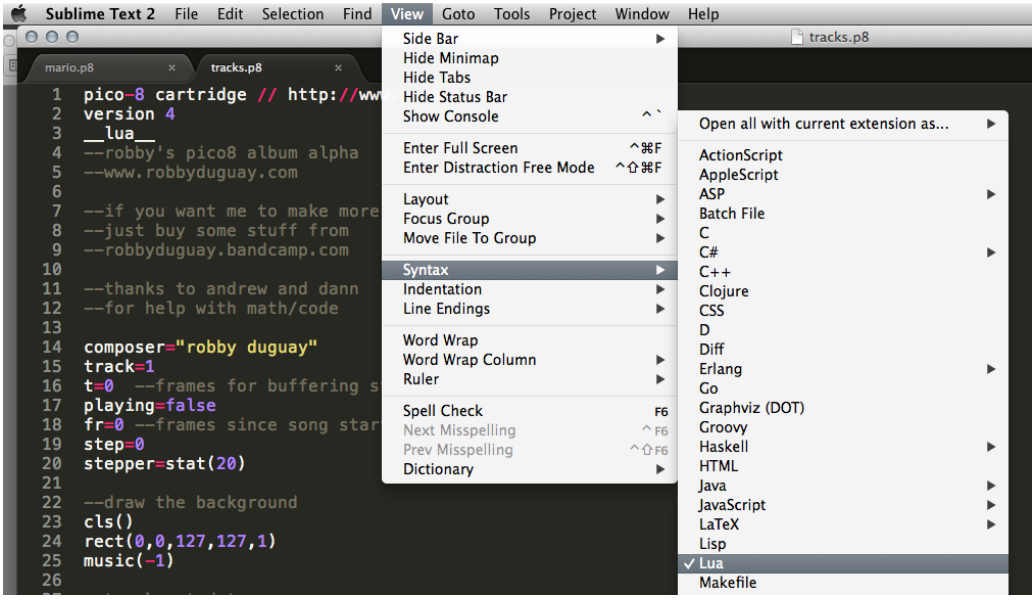
That is it. I know there are probably a million better or more "gamedev-y" ways to do it. For me, doing this was more for the challenge then anything else, and I'm happy with the result. Unfortunately, my game isn't ready yet—hell, I don't even know if I will be able to finish it, but if you follow me on twitter (**@caffo**) I will make sure to let you know when I make it available. Happy Pico-8'ing!

Rodrigo Franco
@caffo

Sharing music between carts

I've written an album, "9 Songs for PICO-8" with the intention that other people can use the songs in their PICO-8 games. They're in a bunch of different styles, but nothing far from what you've heard before. That being said, it's not as easy as just taking an audio file and dropping it into your own project. Let's have a look at how to take a song from one cart and put it in another!

The first thing we need to do is open up the cart in an external code editor. I like to use Sublime Text, and set to View the Syntax as "Lua" - this way everything becomes nicely colour coded. You could also just use Notepad or TextEdit if you wanted. In the file, you'll see a few sections denoted with double underscores before and after the title (e.g.lua). We're interested in the sfx and musicsections, but let's get into a little bit of the theory before we start moving things around.



Music in PICO-8 is made up of sound effects. The music player can play up to four sound effects at once, making your song. On a cart with no sounds, the sfx section is followed by a ton of zeroes. If you're looking at a cart which already has sound effects and

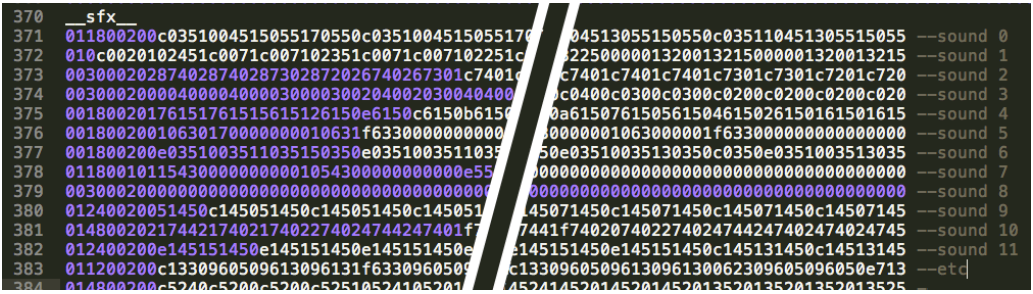
music in it, you'll see what looks like a bunch of gibberish. Each of these lines is one of sixty-four "sound effects," (numbered from 0 to 63). The numbers and letters are the code which tells the audio system what notes to play, in what order, using what articulation, and how fast.

Underneath the music section is much easier to understand. Again, it's 64 lines of information for the audio system, but because all the heavy lifting is taken care of by the sound effects themselves, this section is almost readable. The first two digits tell the tracker if the music is supposed to loop from that point, continue or stop, then the 8 digits following are hexadecimal for which sound effects will be triggered.

Sharing music between carts

```
__music__
01 08004243
00 08014300
00 03014300
00 02030500
00 02030500
00 03414300
00 08014500
00 03040500
00 03020500
00 03020500
02 08010706
01 0a4d0949
00 0a0d090c
```

Sharing music between carts. The easiest way to share music between carts is to open them both up in the external editor, and copy and paste the entire sfx and music sections onto the second cart. Then, when you open it in PICO-8, you'll see all the music and sounds from one cart on the other. This is great for collaborative work, but be careful because if two people are working on sounds separately, you certainly risk overwriting someone's work. That brings us to the more precise and careful way of doing this: copying specific lines. This is easy with the sound effects themselves, you just count from the first line after sfx (starting at 0, see diagram below). That number will correspond to the name of the sound effect when it was in PICO-8. You can pick and choose which sounds to copy over. Be careful to paste them on the same line they came from! Repeat this process for the relevant music lines as well, and you're all set.



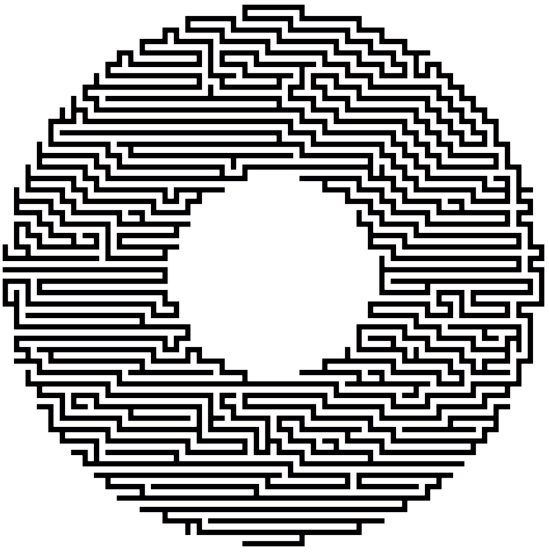
If you want to take it a step further, you can paste the sound effects to whatever line you want. This is good if you want to consolidate, or if you want to move a song from a high tracker number to a lower one. Remember how the music steps just point to sound effects? Well, if you copy over the relevant sound effects to whatever line you want (and you're like me and can't easily write out hexadecimal), it's not hard to just open the file in PICO-8 and point the tracker to the relevant sounds after the fact.

@RobbyDuguay.
Robby

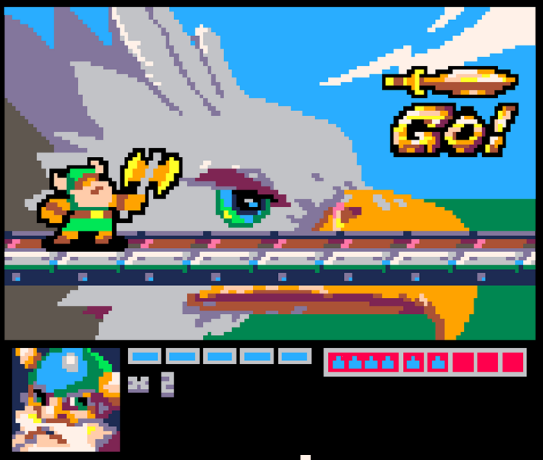
Editor's note: In PICO-8 v0.1.5, it is now also possible to copy and paste music between carts, but the following methods can be used for finer control when manipulating audio data.

DONUT MAZE

```
FUNCTION ADDCEL(X,Y,DX,DY)

X += DX*2 Y += DY*2 -- MOVE ALONG PATH

-- ALREADY VISITED OR OUTSIDE?
IF (PGET(X,Y) == 0) RETURN
IF (PGET(X-1,Y)==7 OR PGET(X+1,Y)==7) RETURN
IF (PGET(X,Y-1)==7 OR PGET(X,Y+1)==7) RETURN

-- DRAW THE NEW CEL AND A JOINING PIXEL
PSET(X,Y,0)
PSET(X-DX,Y-DY,0)
IF ((X+Y)%4 == 0) FLIP() -- SLOW DOWN!

-- VISIT NEIGHBOURS (FAVOUR RIGHT)
D = FLR(RND(4))/4
IF (RND(1.15)<1) D=0
FOR I = 0, 0.75, 0.25 DO
 ADDCEL(X,Y,
 COS(I+D),SIN(I+D))
END
END

-- SET UP DONUT SHAPE
PAL(6,7,1) -- WHITE WALLS
RECTFILL(0,0,127,127,7)
CIRCFILL(64,64,48,6)
CIRCFILL(64,64,16,7)

-- START MAZE FROM LEFT
ADDCEL(17,63,0,0)

WHILE (BTN()==0) DO END
```



--zep
@lexaloffle



@CASTPIXEL