CHAIN // PixelLang Esoteric Language // Table of Contents [−] [×]

An esoteric language entirely in pixels.

Author: Ian Rash **Created:** 2019 **Total Entries:** 3 logs

0 - Intro [2019-09-05]

1 - More Instructions [2019-09-06]

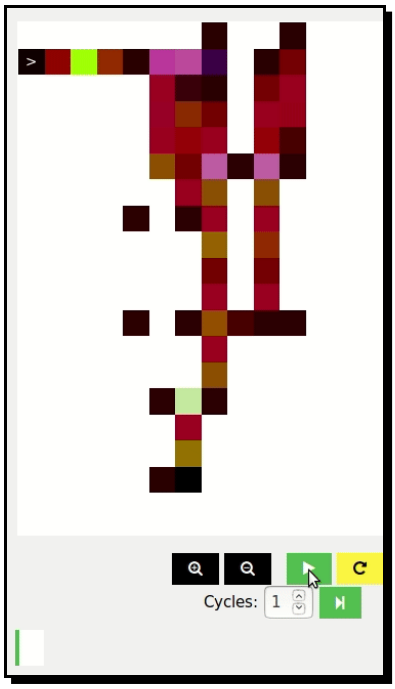
2 - Showcase [2019-09-07]

crystal | Esoteric Programming | Generative Art | raylib

ENTRY // PixelLang Esoteric Language #0 // Intro [−] [×]

Date: 2019-09-05 **Author:** Ian Rash **Index:** #0

Today I'm really excited to start a new series on pixel_lang, a pixel-based 2D esoteric language I wrote myself in Crystal. I've always really loved esoteric languages, and they can teach us a lot about programming. I enjoy the aspect of honing one's programming chops, and esoteric languages are filled with all sorts of interesting challenges. My goal in writing an esoteric language was to make a language with a large instruction set, that could be used to make art, programs, or some combination of the two.



You can find all project files located on the GitHub.

The purpose of this article, is to give a basic introduction on how to use the pixel_lang interpreter, as well as the web interface, to load and run programs, as well as a basic explanation of how the system works.

The pixel_lang interpreter takes an input PNG file, and uses that as it's program code. Each pixel in the PNG is read in, and stored into a 2D array of instructions. Any and all PNG files are valid program code and can be used with the interpreter without error but, may not start without the presence of a Start instruction, or will run infinitely. A program can be given input, either by providing an input string argument, or via the live interactive session (yet to be written). This interpreter is called the Engine.

Engines are comprised of a variable number of pistons. These pistons act as separate instruction readers and act mostly independently of each other, save for a few exceptions. The starting number of pistons is determined by a program code's number of Start instructions. The pistons then each take turns, reading an instruction, executing it, and moving one step forward in the direction it's facing. When all pistons have run an instruction, that is called a cycle. The instruction type (known as a Control Code or CC) is determined by the upper 4 bits of the color of the pixel, the arguments for the instruction (known as the Control Value or CV) are the bottom 20 bits of the pixel's color.

The full instruction list is as follows.

- 0x0 - End
- 0x1 - Start
- 0x2 - Pause
- 0x3 - Direction
- 0x4 - Fork
- 0x5 - Jump
- 0x6 - Call
- 0x7 - Return
- 0x8 - Insert
- 0x9 - Move
- 0xA - Arithmetic
- 0xB - Output Char
- 0xC - Conditional
- 0xD - Instruction Meta
- 0xE - Engine Meta
- 0xF - Blank

Some example instructions 0xA00000 - Arithmetic(MA(0) + MA(0) -> MA(0)) 0x100100 - Start(direction: up, priority: 0x100) 0x200010 - Pause(for 0x10 cycles)

A Piston is deleted when it reads an End instruction, and when an Engine has no more Pistons, it is no longer running.

Each Piston keeps track of a couple of things, it's current location, the values within it's own registers and memory, and a call stack (used with Call Return). The Piston has a total of 8 registers, which each hold a 20 bit integer, and each have special properties when read.

The MA register is one of the simpler registers, it takes a value that is written to it, and when read from provides the last value written to it. The default value of this register is always 0x0.

The MAV register is a little different. MAV is controlled by the MA register and when read from, provides the value in the Piston's memory at the location provided by MA. When written to it changes that memory's value. An uninitialized memory cell is always 0x0.

For example, if you wrote 0 to the MA register, then read from MAV you'd get 0. If you wrote 1 to the MA register and

then read from MAV you'd get 0. If you wrote 333 to the MAV register and read from MAV you'd get 333. If you wrote 0 to the MA register, then read from the MAV register, you'd get 0 again.

The next two registers MB and MBV operate the same as MA and MAV, they even use the same memory pool, so if MA is equal to MB then MAV is always equal to MBV. MA and MB can both be used to reference two different values in the same Piston's memory. The MB register's default is 1.

The next two registers are S and SV which work similar to MA and MAV except the memory pool they use is static, meaning all pistons can access this to communicate with each other.

Next we have the I register, which acts like a stack. If written to, it adds a new value to the stack, when read from it pops from the stack. You can choose also to peek this register instead, keeping the value on the top of the stack.

Lastly we have the O register, which is an output register. When read from, it provides the last character that was output from any piston, when set, writes the value to output.

When pistons move off the edge of the program space, it appears on the other side, asteroids style.

That's really all there is to the internals of the interpreter, the rest is pretty easy.

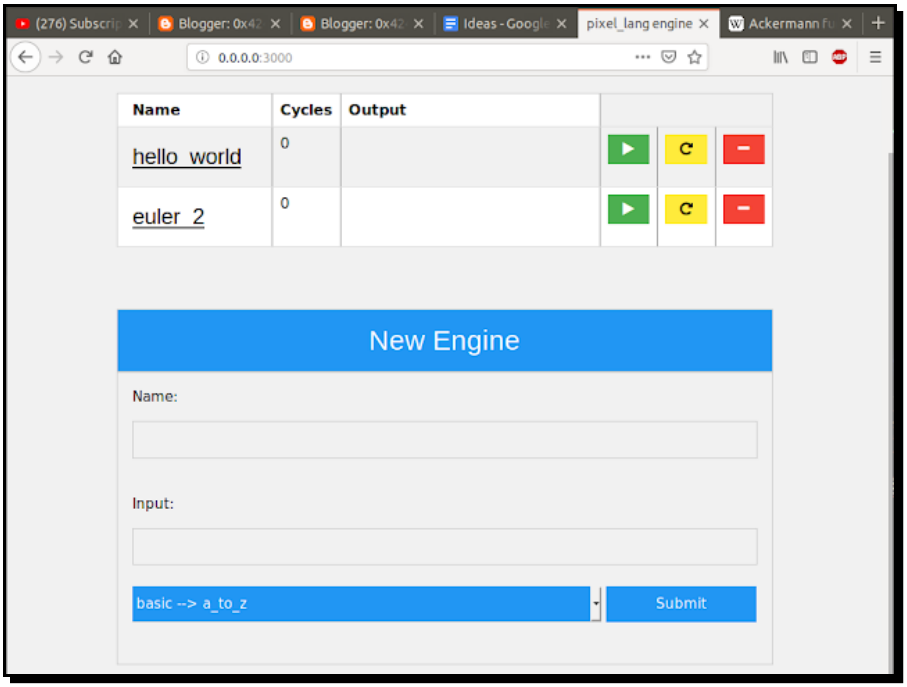
There are two ways to interact with programs right now, there is the runner, and the web interface. I'd highly recommend using the web interface, it's not bad to use, runs pretty smoothly, and allows you to watch the programs execute step by step.

To build the runner use `shards build runner`

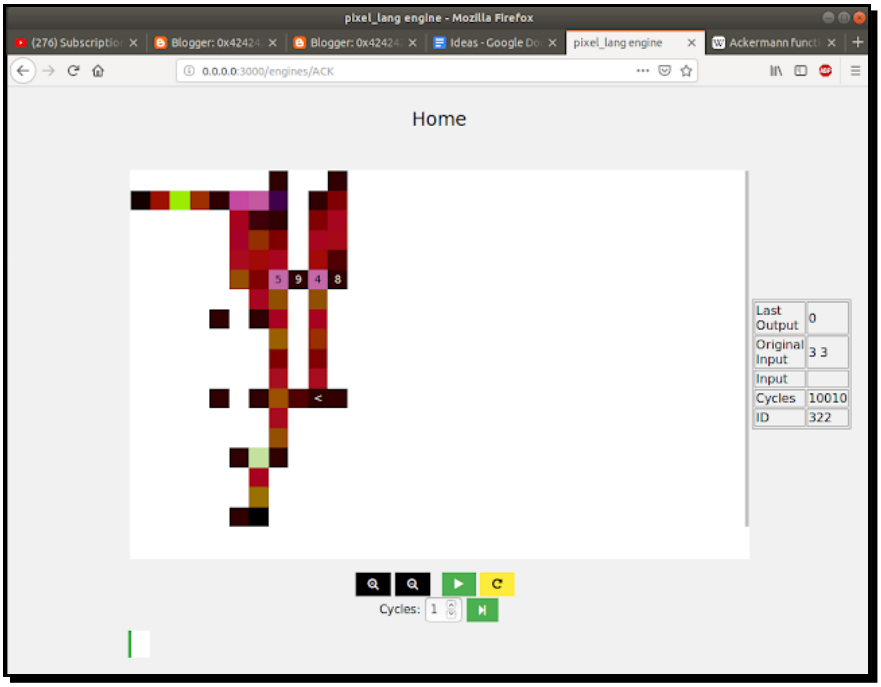
You can then run any program using `./bin/runner program_file "input text!!!!!"`

You can run the web app by using `shards build web_app && ./bin/web_app`

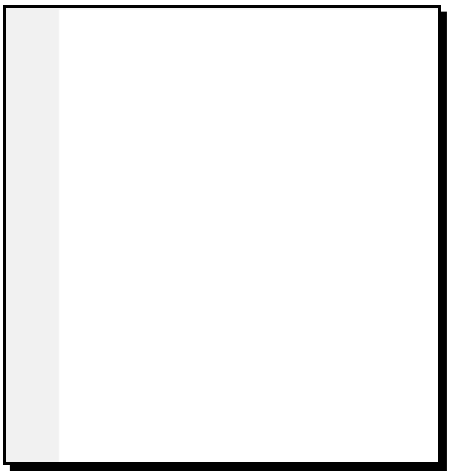
We will primarily focus on the web_app, as it is a lot easier to work with than the runner.



You can create a new Engine with a program loaded using the bottom box on the home page, You must specify a name, and a program. When you open up a program it should look like below.



Pressing play will start an animation of the execution, showing the pistons moving and doing their work.



The above program is the prime sieve of Eratosthenes, which is a special way to filter prime numbers.

My other pride and joy program is the Ackermann function, I highly suggest checking it out.

If you want to write your own programs, any picture editor will do, as long as it supports hex color input (like #AABBCC). Pictures should be saved as PNG. To add them to the web interface, navigate to pixel_lang_crystal/programs and put your files in there, and then restart the web_app. I suggest Aseprite to edit any programs, as it has a useful palette feature to rip all the unique colors out of a picture.

The system also has a special way of helping make colors for you. For this, I would highly recommend opening crystal play in the pixel_lang_crystal directory and using that.

```
require "./src/pixel_lang_crystal"

Start.make_color(:right, 100)
Arithmetic.make_color(:mb, 0, :+, :mav, 0, :mbv, 0)
Conditional.make_color(:up, :turn_right, :s, 0, :<, :sv, 0)
```

```
"140064" : String
"a40118" : String
"c16134" : String
```

Basic Instructions

In this segment, we will cover some basic instructions, enough to make our own first Hello World program.

Start

The Start instruction is used to place pistons when starting the program. Each Start instruction can contain two arguments, what direction the Start is facing (up, down, left, right), and the priority, which describes in what order the pistons all run in. A piston with a priority of 0 runs before a piston with priority 100.

End

The End instruction removes a piston from the program space. If all pistons are gone off of the program space, the program has ended. Without an End instruction, a program will run indefinitely. End takes no arguments.

Direction

Direction changes the current direction the piston is going to one of 8, stored in it's arguments.

- Up
- Down
- Left
- Right
- Turn Left
- Turn Right
- Reverse
- Straight (No operation)

Jump

Jump moves a piston in the direction it's facing, a number of spaces determined by it's argument. (0x0 - 0xFFFFF).

For example, Jump 0 jumps only one space, Jump 1 jumps 2 spaces, etc etc.

Jumps that jump a piston off the program space wrap the piston back around.

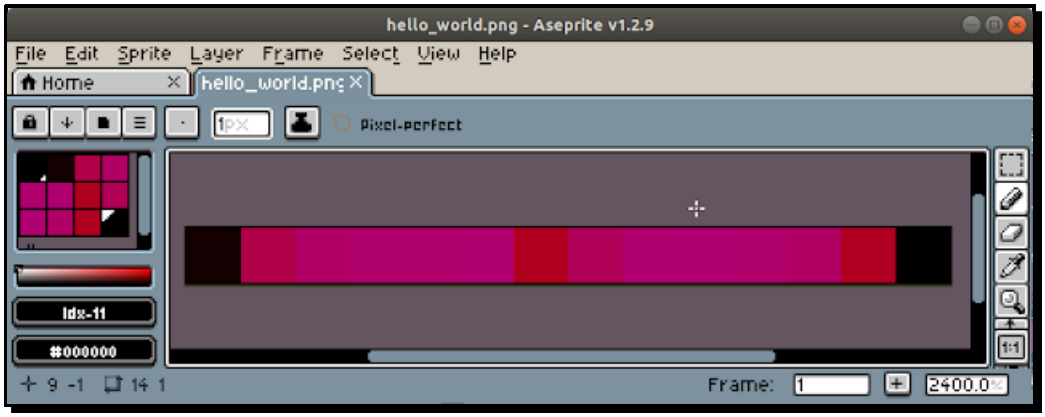
OutputChar

Outputs the char defined in the arguments. For example, OutputChar H would be 0xB00048.

Putting it all together

Now we have all the instructions we need to try a Hello World program! Let's put them all together.

The only three instructions we need are Start, End, and OutputChar. We can use crystal play to mix all our colors for us.



Next we just need to put them all into a PNG, open up your favorite pixel art editor and let's go!

If we open up the file and play it in the web app, we can see the output works!

```
require "./src/pixel_lang_crystal"

Start.make_color(:right, 0)                    "140000" : String
# One of each char
OutputChar.make_color('H')                     "b00048" : String
OutputChar.make_color('e')                     "b00065" : String
OutputChar.make_color('l')                     "b0006c" : String
OutputChar.make_color('o')                     "b0006f" : String
OutputChar.make_color(' ')                     "b00020" : String
OutputChar.make_color('W')                     "b00057" : String
OutputChar.make_color('r')                     "b00072" : String
OutputChar.make_color('d')                     "b00064" : String
OutputChar.make_color('!')                     "b00021" : String
# Jumps to help us move around
Jump.make_color(0)                             "500000" : String
Jump.make_color(4)                             "500004" : String
# Directions
Direction.make_color(:up)                      "300000" : String
Direction.make_color(:down)                    "300002" : String
Direction.make_color(:left)                    "300003" : String
Direction.make_color(:right)                   "300001" : String
Direction.make_color(:turn_left)               "300004" : String
Direction.make_color(:turn_right)              "300005" : String
Direction.make_color(:reverse)                 "300006" : String

End.make_color                                 "0" : String
```

TAGGED SKILLS:

Crystal | Esoteric Programming | Generative Art | Raylib

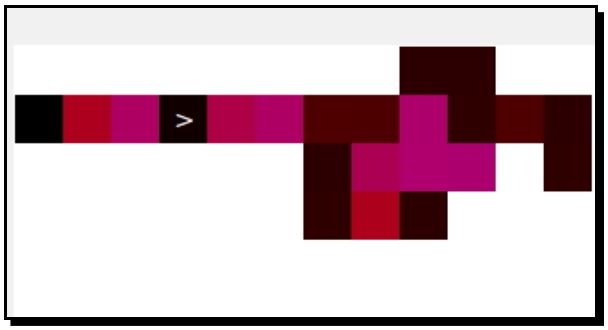
ENTRY // PixelLang Esoteric Language #1 // More Instructions [−] [×]

Date: 2019-09-06 **Author:** Ian Rash **Index:** #1

We covered Start, End, and Output Char in the last section, next we want to use Direction and Jump. Let's create a version of Hello World that does some crafty maneuvering to only use a single output char instruction for each character of "Hello World!". First we need to make the instructions we need to use.

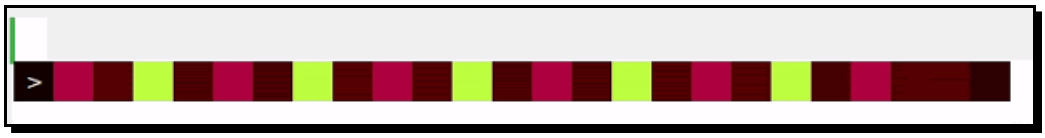
```
require "./src/pixel_lang_crystal"

Start.make_color(:right, 0)                    "140000" : String
# One of each char
OutputChar.make_color('H')                     "b00048" : String
OutputChar.make_color('e')                     "b00065" : String
OutputChar.make_color('l')                     "b0006c" : String
OutputChar.make_color('o')                     "b0006f" : String
OutputChar.make_color(' ')                     "b00020" : String
OutputChar.make_color('W')                     "b00057" : String
OutputChar.make_color('r')                     "b00072" : String
OutputChar.make_color('d')                     "b00064" : String
OutputChar.make_color('!')                     "b00021" : String
# Jumps to help us move around
Jump.make_color(0)                             "500000" : String
Jump.make_color(4)                             "500004" : String
# Directions
Direction.make_color(:up)                      "300000" : String
Direction.make_color(:down)                    "300002" : String
Direction.make_color(:left)                    "300003" : String
Direction.make_color(:right)                   "300001" : String
Direction.make_color(:turn_left)               "300004" : String
Direction.make_color(:turn_right)              "300005" : String
Direction.make_color(:reverse)                 "300006" : String

End.make_color                                 "0" : String
```

Then we just need to come up with an interesting layout. I had to specifically be aware of the OutputChar L, because it's used three times, and I need to go a different route each time.



I often times use Jump to allow a single directional line of code have multiple meanings.

For example, take this program, which only hits the pink spaces one way, and the yellow spaces other. This programs runs indefinitely.



Insert

Insert is instruction number 0x8, and it's 20bit argument is stacked onto the I stack of any Piston that executes it. It's used to introduce constants into your program.

Move

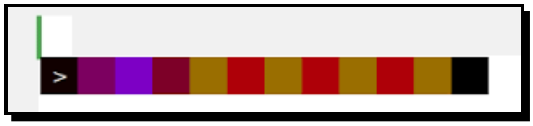
Move is an instruction which takes two register arguments, and moves the value from the source to the destination. When specifying a register, you also need to specify a register option which can change how the register is interacted with.

For example, a MOV(MA(0) -> MB(0)) will move the value in MA into MB, where MOV(MA(1) -> MB(0)) will move a random value where MA is the max of that number, and move it into MB.

Another good example would be MOV(I(0) -> MA(0)) which pops a value off I and puts it into MA, and MOV(I(2) -> MA(0)) which only peeks the value.

Register options can be very useful!

Let's give Move a try.



In the program above, 100, 200, and then 300 are placed on the I stack using Insert, then moved to output using MOV(I(0) -> O(0)), and a space is added between. We can see when the I stack runs out, it always returns 0 if there is no engine input to consume.

MOV can also be used with register options to change register behaviour. For example, if MA is equal to 1234 and MOV(MA(1) -> O(0)) is called, output will be a random number between 0 and 1234. Reading from I(2) doesn't pull the item off the I stack, while I(0) does.

Move also has two other options, swap and reverse. Swap swaps the values of two registers. The order it does that is very important, since register reads can trigger changes (like I or O for example). When swapping values, the source value is gotten first, then the destination value, then the source is set first, then the destination is set. This is important to understand especially when using the instruction MOV(I(0) -> I(0)), which will swap the top two values on the I stack.

You can also reverse the source and destination, this option isn't particularly useful, it's just to allow for more interesting colors to be made.

Arithmetic

One of my favorite instructions, Arithmetic allows you to build simple mathematical expressions. Arithmetic takes two source registers, an operation, and a destination register. Here are some examples.

```
5 Arithmetic.make_color(:ma, 0, :+, :mb, 0, :mav, 0)      "a00208" : String
6 Arithmetic.make_color(:o, 2, :/, :i, 2, :mbv, 0)        "af1e98" : String
7 Arithmetic.make_color(:mb, 0, :*, :mbv, 0, :o, 0)       "a41338" : String
8 Arithmetic.make_color(:mbv, 0, :-, :ma, 0, :i, 0)       "a60830" : String
9 Arithmetic.make_color(:i, 0, :%, :o, 0, :mbv, 0)        "ac4718" : String
```

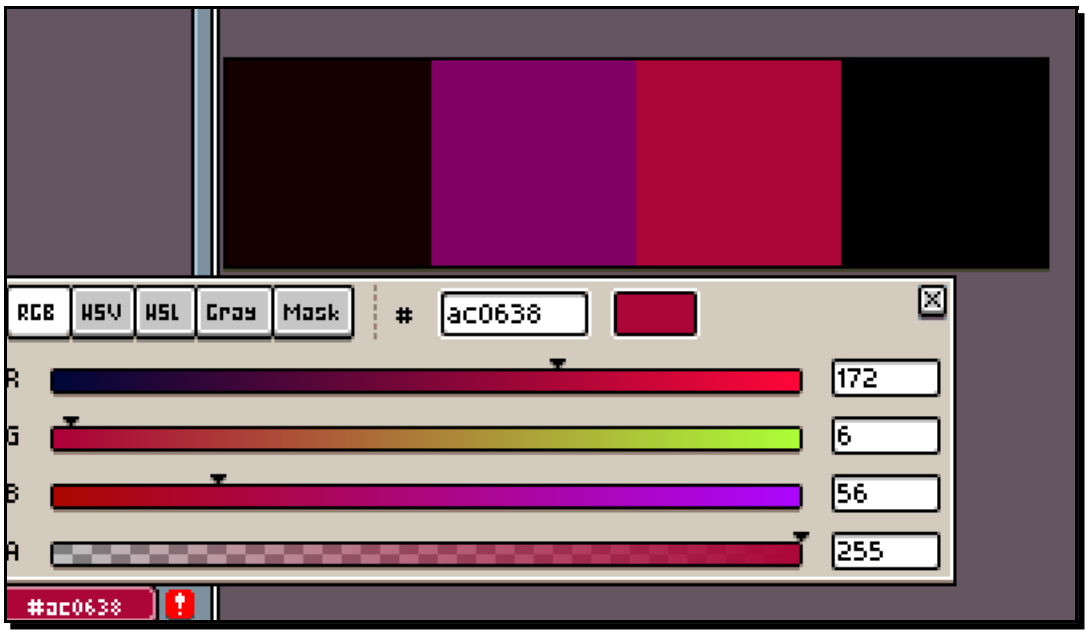
Here's a list of all the mathematical operators available.

BOOLEAN_OPERATIONS = [:<, :>, :<=, :>=, :==, :!=]
ARITHMETIC_OPERATIONS = [:+, :-, :*, :/, :**, :&, :|, :^, :%]

Using this instruction we can easily make a program to add 100 to an input number.

The program itself only needs 4 instructions. Start, Insert(100), AR(I(0) + I(0) -> O(0)), and End.

When running the program, you need to include an input number into the Engine before starting or it will always display 100, since I always deafults to 0 when there is no input.



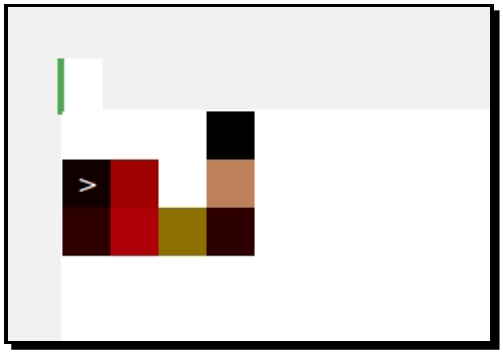
Conditional

Conditional allows pistons to make decisions on where they will go, based on a mathematical expression. You choose two directions, a true direction, and a false direction, and if the mathematical expression evaluates to 0 it goes the false direction, otherwise the true direction.

Boolean operators always produce either a 0 (false), or a 1 (true).

Conditional lets us create decisions and loops that will be the basic building blocks of our programs.

To show off what the Conditional can do, we are going to make a "count to" program which will take a number, and count up to that number.



In this example, the value of MB is always 1. We use that to increment MA each loop, and use the beige instruction (a Conditional) to determine if we have hit our max number yet. If not, we output the number, output a space, and start over again. Interesting note, Start instructions operate as Direction instructions when executed by a Piston. This allows us to restart programs easily if necessary.

Call/Return

These two instructions work in tandem to allow a Piston to return to a previous state, and choose what it takes along with it. Call and Return can be some of the most powerful instructions if used right.

Call takes a couple arguments, the first being an action, which is either :none, :push, :none_run, :push_run. This determines behavior of the Piston after running the Call, if it should push it's current frame to stack, or not, or if it should step once after the Call. For example, the none option does not push a frame to the call stack, where push does.

Call also takes a signed X and Y argument.

Call moves the Piston X and Y spaces away from the Call instructions. This is all relative spacing, there is no absolute values for Call.

When a Return instruction is read by a Piston, it checks to see if there is a frame on it's call stack. If there is a frame, the Return instruction chooses what values to copy back to the piston, for example, you can choose to restore the X position but not the Y, the direction, you can choose to keep MA the same, or restore it from the frame, that sort of thing. If there is no frame on the call stack, Return does nothing.

Return has a lot of arguments, a full list from the color_helper dev module:

```
Return Instruction
Returns a frame from the call stack.
0bCCCC00000000PPABSIIMMXYD
C = Control Code (Instruction) [4 bits]
P = Action bits [:pop, :peek, :pop_push, :peek_push]
A = Copy MA?
B = Copy MB?
S = Copy S?
I = Copy I action? [:keep, :restore, :clear]
M = Copy memory action? [:keep, :restore, :clear]
X = Jump back to X?
Y = Jump back to Y?
D = Change the direction?
```

First, the action specifies whether or not a frame should peeked, or popped off the call stack and/or if the current frame of the Piston should be added back to the call stack.

Next, should we copy MA, MB, S, X, Y, etc?

Lastly, I and Memory (since they are collections) both have special operations. Whether or not we should keep them, restore the frame's version, or clear altogether. When using :clear, even if there is nothing on the call stack, that item will still be cleared.

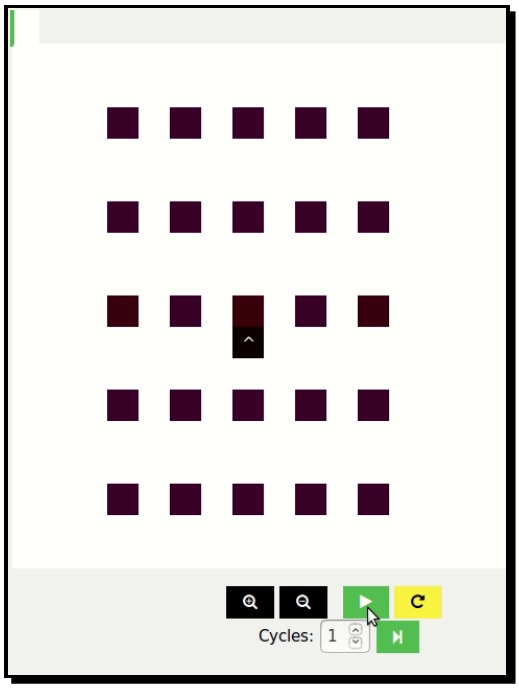
Again Return is super powerful, with it, we can set up all sorts of interesting interactions with the program code.

Fork

Fork is used to make exact duplicates of Pistons, facing in different (or the same) directions. One Fork instruction can

make up to 4 new pistons. The original piston always follows direction #1, and each piston created afterwards executes after the last one created. Imagine two pistons one the same space that hit a fork instruction. Let's call them P1 and P2, after their priority numbers. P1 creates a new Piston, sandwiched between it and P2 in the execution order, while P2's created clone will be below it in the execution order.

Here is an example program I wrote to test the limits of Fork (like how many times can you fork before the program crashes).



I also use Fork in my Ackermann implementation, since it's perfect for the job of expansion!

InstructionMeta

InstructionMeta is an instruction that houses four other functions, Get, Set, Resize, and Property.

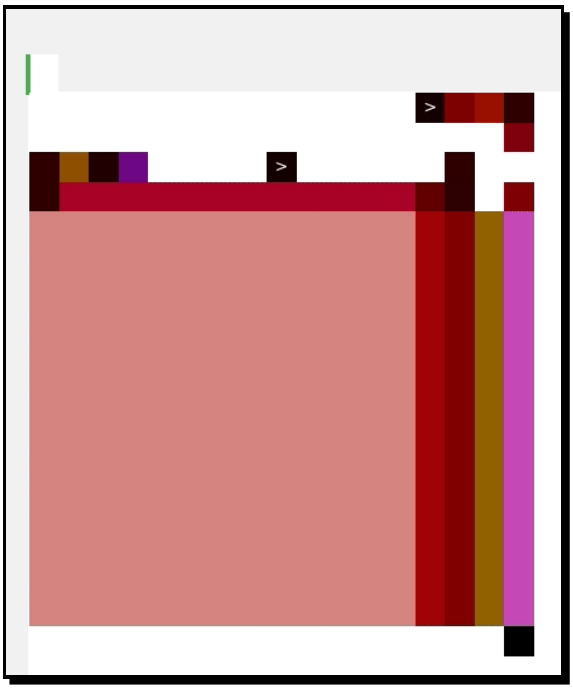
Get allows a Piston to get the value of a color located at the X and Y values specified by two registers. Puts the control code, then the control value on the I stack.

Set allows a Piston to set the value of a color located at the X and Y values specified by two registers, to the value located on the I Stack. Since the I stack can only hold values up to 0xFFFFF, the I stack is read twice and the values bitshifted and combined to for the color. For example, if 0xBBBBB then 0xA were on the I stack, the color would be 0xABBBBBB.

Resize allows a piston to resize the instructions width and height.

Property allows a Piston to get the width and height of the instruction set.

Using these instructions, we can modify the instructions on the board! Here is an example program, a painter bot.



The program above uses two pistons. One goes in a loop and counts up from 0 to the width of the "drawing canvas". The other piston reads the IMetaSet instructions, and executes them, changing the current square they are on, then moving to another. This gives the effect of a bot painting the ground behind it.

TAGGED SKILLS:

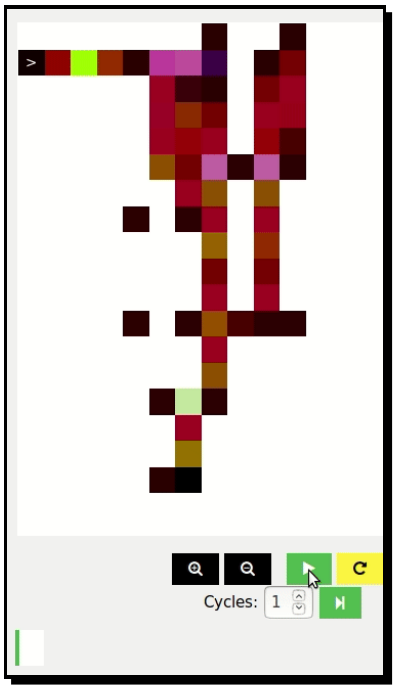
Crystal | Esoteric Programming | Generative Art | Raylib

ENTRY // PixelLang Esoteric Language #2 // Showcase [−] [×]

Date: 2019-09-07 **Author:** Ian Rash **Index:** #2

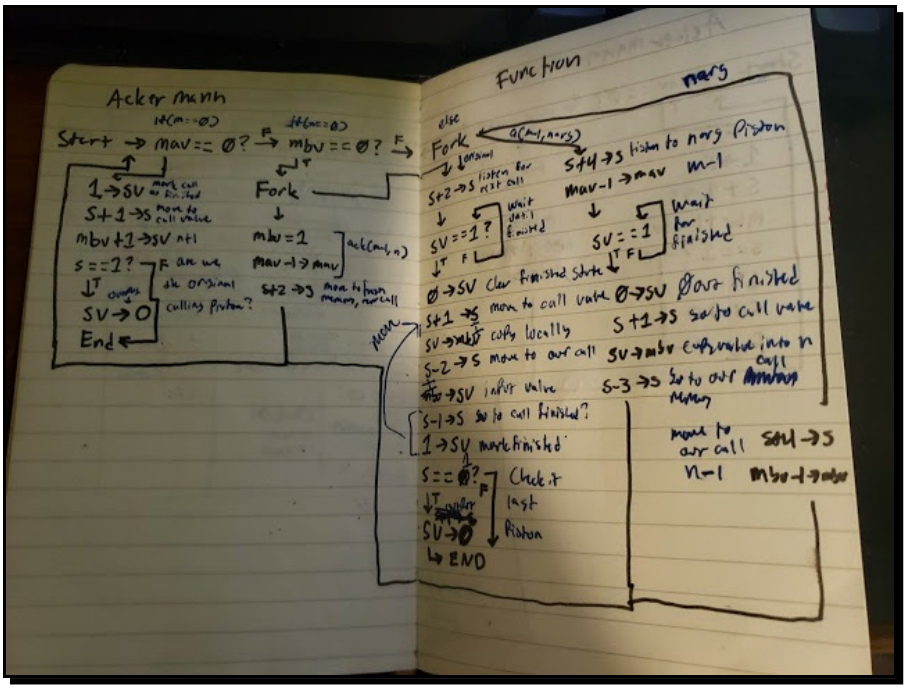
I'd like to take a little time today to showcase some of the programs I made with pixel_lang. I'm very proud of them, and I'm hoping someone will enjoy watching and dissecting them.

The first one I'd like to show off is my Ackermann function, which uses forking, and it's own in-memory call stack to orchestrate all the individual pistons to fire when necessary.



In the example above, there are three possible decisions made, if M is equal to 0, if N is equal to 0, and the recursive call, with the n-arg. That's why the fork splits the piston into three, one for the recursive call itself, and one for the n-arg, which is also a recursive call.

Here is the draft version of the function, that I made by hand with comments.



Since each call to ack() waits for the next call to finish, I created a basic lock system, that tests to see if there is a piston still executing the function. When a piston finishes, it either creates new pistons (the recursive n-arg call), or frees up old ones waiting for an answer. The n-arg always goes first, then the other calls, after the n-arg piston has finished.

The locks themselves listen directly to static memory, to make them efficient as possible. A locked piston only needs to run one instruction that way, as well as have a direction instruction that pushes the piston back into the conditional

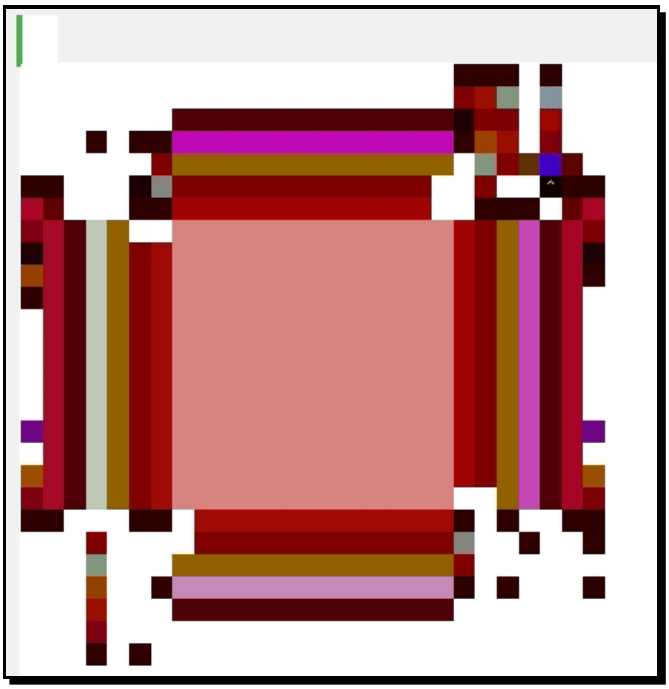
lock.

We know a piston is the last piston alive, because the position in the call stack it's currently at will always be 0. We check that at the end to see when we need to output the answer.

In this program, MAV is M, and MBV is N. S is the current position of the call stack, and SV is the values on the stack, which are whether or not the piston has solved it's part of the equation, and what that answer was.

I also want to point out that I could have made this in less cycles using Jumps, but I like watching the flow of the program, so there is only one necessary Jump instruction in the whole program.

The next program is one I developed recently to show off for part 2, but didn't finish it in time.



This program uses the meta-programming instruction IMetaSet, to color the middle square two different colors each rotation. The internals on it I'm pretty proud of.

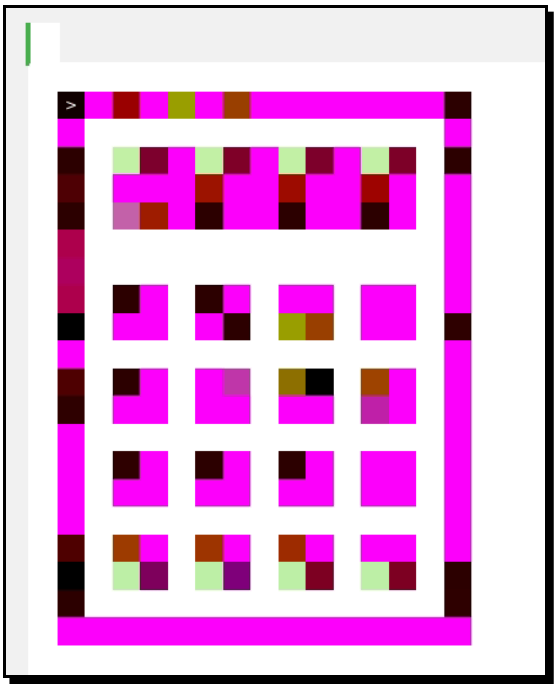
First of all, this program has an infinite loop with no possibility of memory leakage. In pixel_lang, a piston can potentially crash a program after millions of loops because of things like left-over values in I. Since I is a collection with no bounds, if it gets to full the program either crawls to a halt, or straight crashes.

If you look on the left and right sides, you can see two pistons, each going in a loop. The right side is an incrementer, and the left side is a decrementer. They count either up to 19, or down to 7, which is the exact coordinates of our center square, on two separate SV values. The inner piston changes it's S to read from either SV depending on what stage of the program it's at.

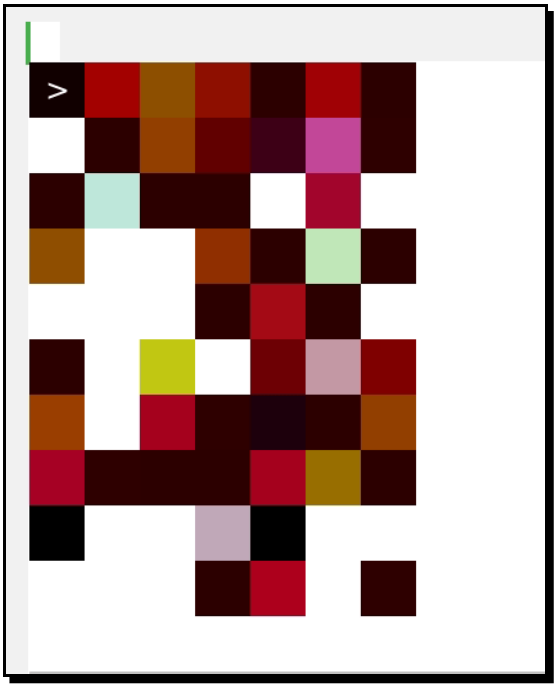
The painting bot itself uses a neat effect/tactic. The two colors it's changing itself between are IMetaSet(:sv, 0, :mav, 0, 2, :ma, 0), IMetaSet(:mav, 0, :sv, 0, 2, :ma, 0). SV and MAV are either X or Y, depending on the direction. MA is the control code register, which is always 0xD (for IMeta). On I we stack the color value, which in this case is either, 0x68480 or 0x49480. Whatever color is currently on the board, it paints the opposite of it.

The timing was really difficult on this program, because fork priority had to be respected at every turn. Since each piston has Read, Execute, and Move phases, the two pistons on the sides only increment/decrement after the painting piston has moved. This gives the effect of painting whats behind it, even though it actually changes the instruction it

was on during execute phase, and then moves forward one.



Next is my calculator program, this one takes a non-parenthesized math expression, the runs the operations from left to right (does not respect OoO). This was just a fun artsy project I wanted to do to show art was possible (If you can call it art).



Another program I wrote is the Sieve of Eratosthenes, a fun way to filter prime numbers. This program uses Call/Return to move the pistons into an "incremental tar pit", the higher the number of the current index, the higher the wait for a piston to be released from the tar pit. Basically, a generator forks pistons with the current index (which it increments every cycle), then checks to see if it went over the limit. The forked pistons go through the program. When a prime number is found, it is added to a list of prime numbers which is checked against all other future potential primes.This is list is output at the end of the program, once the last piston is done. This example of the program runs up to 30.

TAGGED SKILLS:

Crystal | Esoteric Programming | Generative Art | Raylib

+--+
© 2026 Ian Rash - Built with Crystal & Made into a PDF
+--+