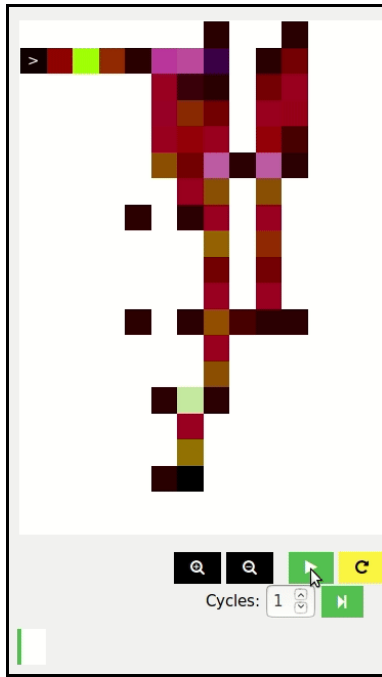


Date: 2019-09-05 Author: Ian Rash Index: #0

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.

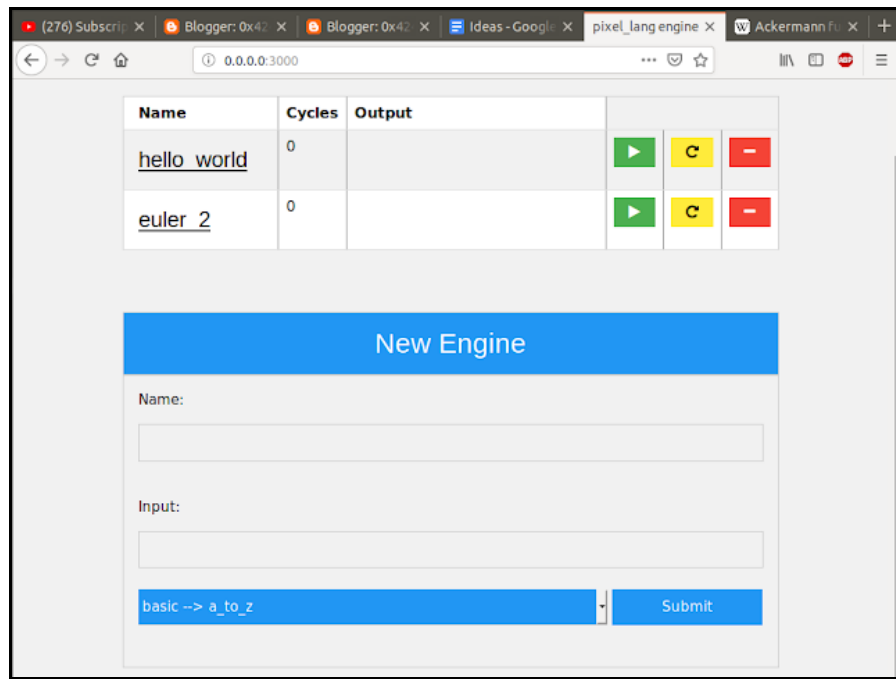
There are two ways to interact with programs right now, there is the runner, and the web interface. I'd highly recommend using the web interface, it's not bad to use, runs pretty smoothly, and allows you to watch the programs execute step by step.

To build the runner use shards build runner

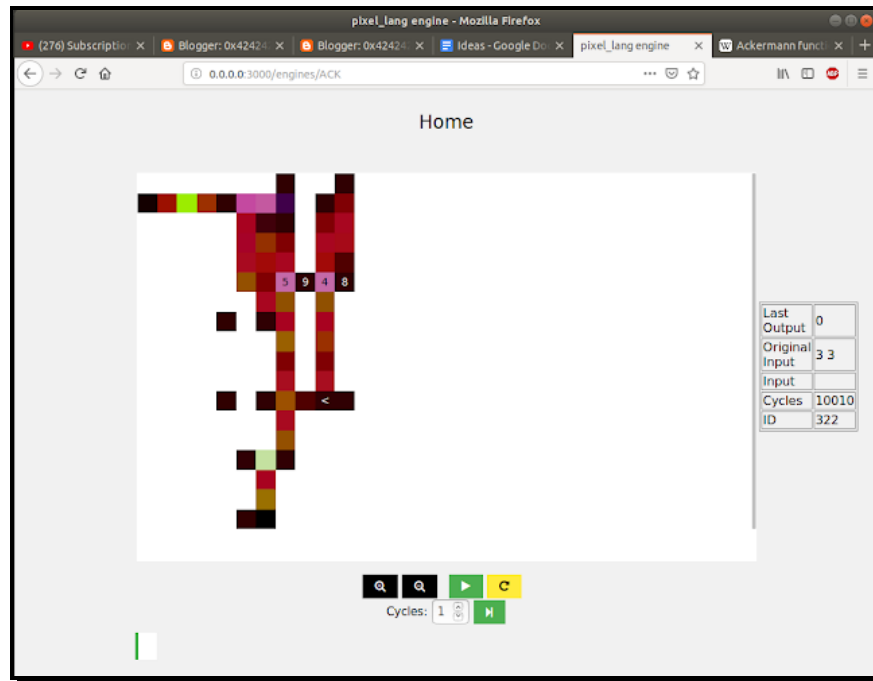
You can then run any program using `./bin/runner program_file "input text!!!!"`

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

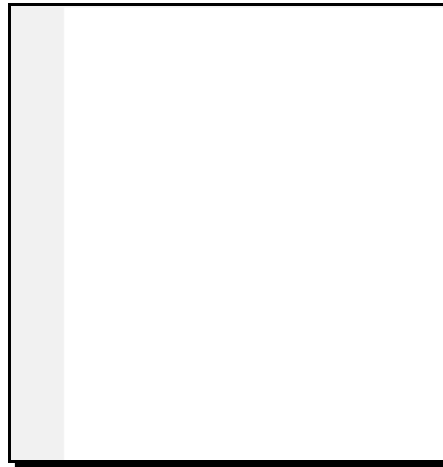
We will primarily focus on the web_app, as it is a lot easier to work with than the runner.



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.

<pre> 1 require "./src/pixel_lang_crystal" 2 3 Start.make_color(:right, 100) 4 Arithmetic.make_color(:mb, 0, :+, :mav, 0, :mbv, 0) 5 Conditional.make_color(:up, :turn_right, :s, 0, :<, :sv, 0) </pre>	<pre> "140064" : String "a40118" : String "c16134" : String </pre>
--	--

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.

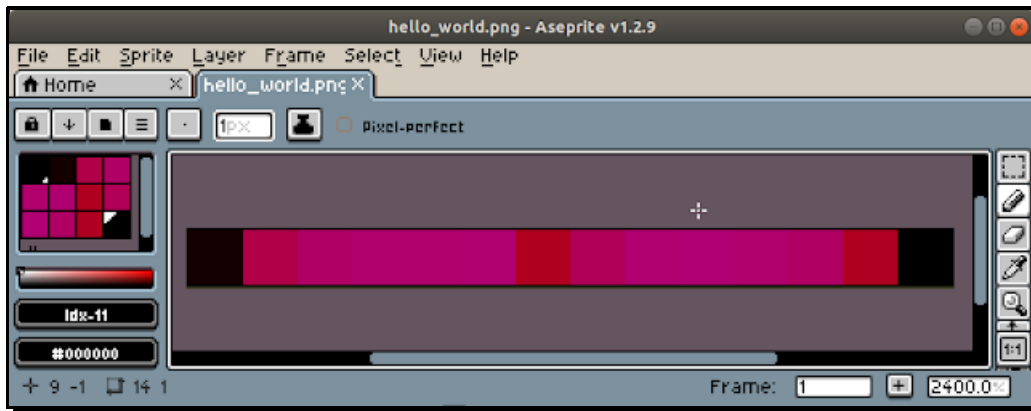
OutputChar

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!

```

1 require "./src/pixel_lang_crystal"
2
3 Start.make_color(:right, 0)
4 # One of each char
5 OutputChar.make_color('H')
6 OutputChar.make_color('e')
7 OutputChar.make_color('l')
8 OutputChar.make_color('o')
9 OutputChar.make_color(' ')
10 OutputChar.make_color('W')
11 OutputChar.make_color('r')
12 OutputChar.make_color('d')
13 OutputChar.make_color('!')
14 # Jumps to help us move around
15 Jump.make_color(0)
16 Jump.make_color(4)
17 # Directions
18 Direction.make_color(:up)
19 Direction.make_color(:down)
20 Direction.make_color(:left)
21 Direction.make_color(:right)
22 Direction.make_color(:turn_left)
23 Direction.make_color(:turn_right)
24 Direction.make_color(:reverse)
25
26 End.make_color

```

```

"140000" : String
"b00048" : String
"b00065" : String
"b0006c" : String
"b0006f" : String
"b00020" : String
"b00057" : String
"b00072" : String
"b00064" : String
"b00021" : String
"500000" : String
"500004" : String
"300000" : String
"300002" : String
"300003" : String
"300001" : String
"300004" : String
"300005" : String
"300006" : String
"0" : String

```

TAGGED SKILLS:

Crystal Esoteric Programming Generative Art Raylib

