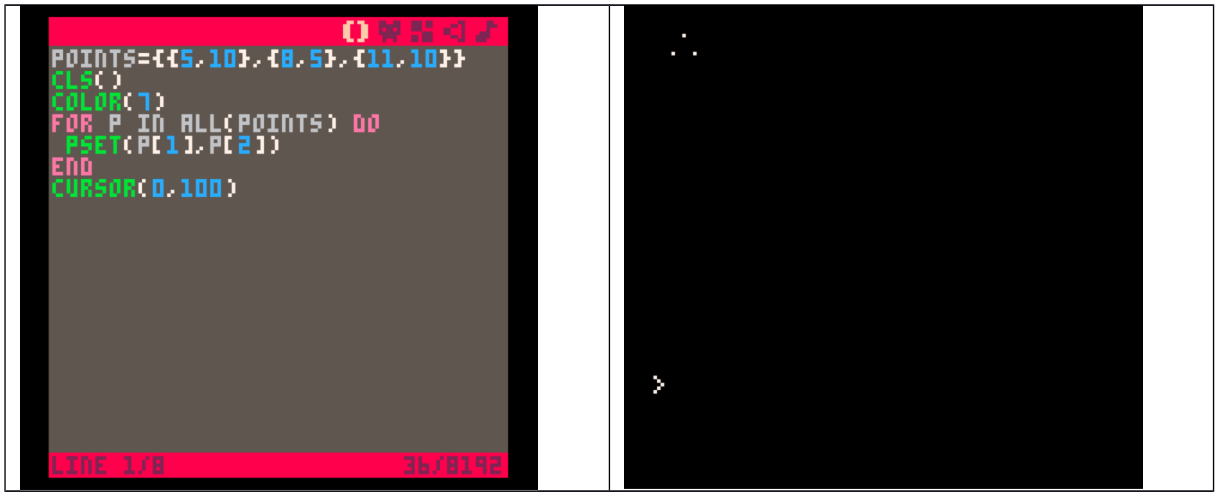
PICOZINE#5

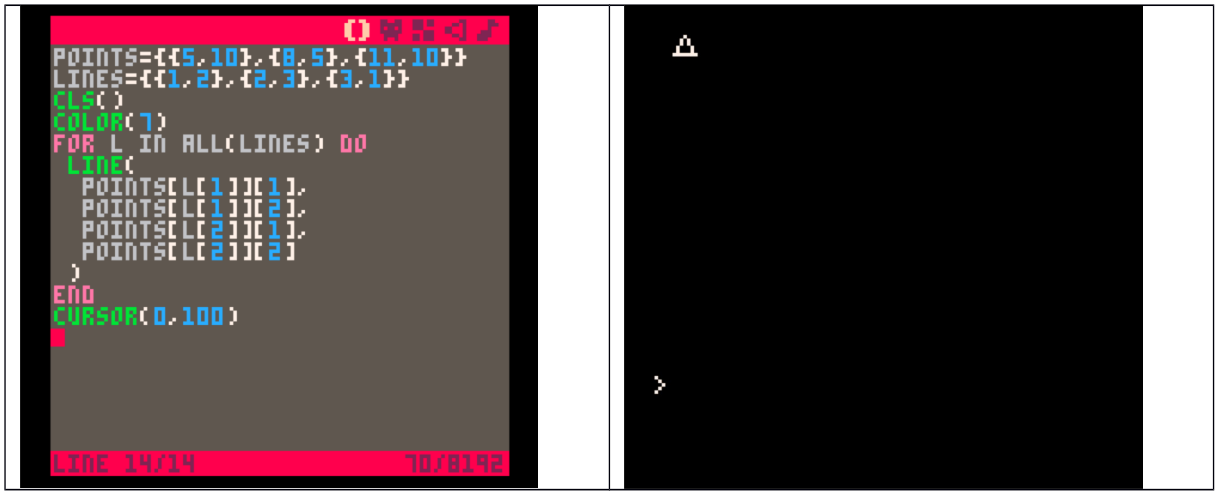
About math in pico8…

The beginning was a small part of energy in large space of nothingness: the point or in pico8 term, the pixel. The screen space is a 2D surface organized in the form of a plane with orthogonal axis. If you face the screen, one is going from left to right (x, abscissa) and the second from up to down (y, ordinate). The origin of our space (0, 0) is the upper left corner. The screen is a grid of 128x128 pixels. The two main differences between Euclidean geometry and the pico-8 screen is that the Y-axis increases going down and the value on each axis aren't continuous. This is a convention you must accept (as the use of 1 to index the first item of an array). I admit I had trouble with the last ;-)

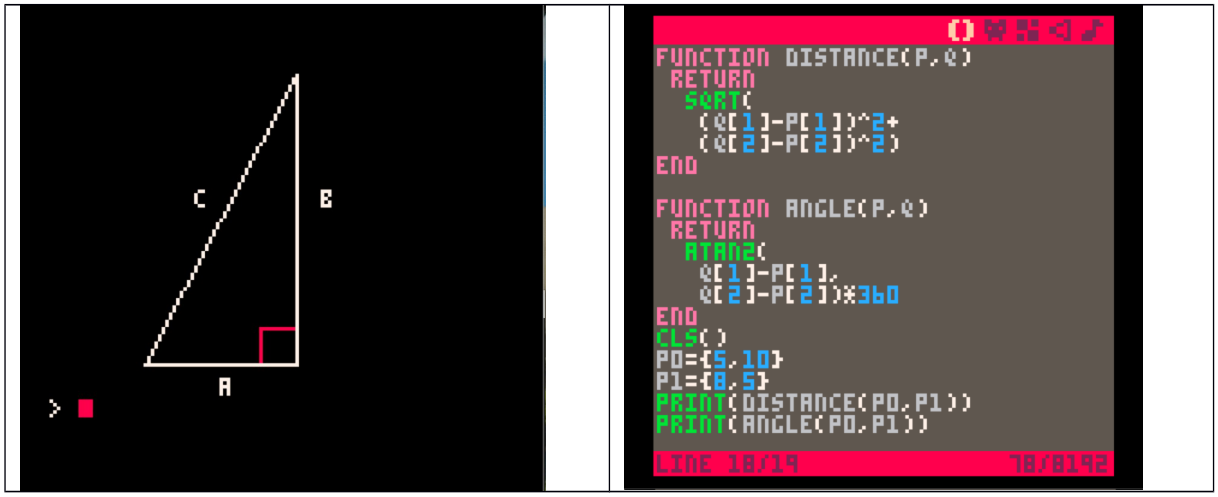
You can light up a pixel only with coordinates using integer. If you submit float values, pico-8 will convert them to integers (FLR). The **points** table contains the coordinates of 3 points. Let's draw the corresponding pixels.



The points are the summits of a nearly equilateral triangle. Let's draw line between the summits.

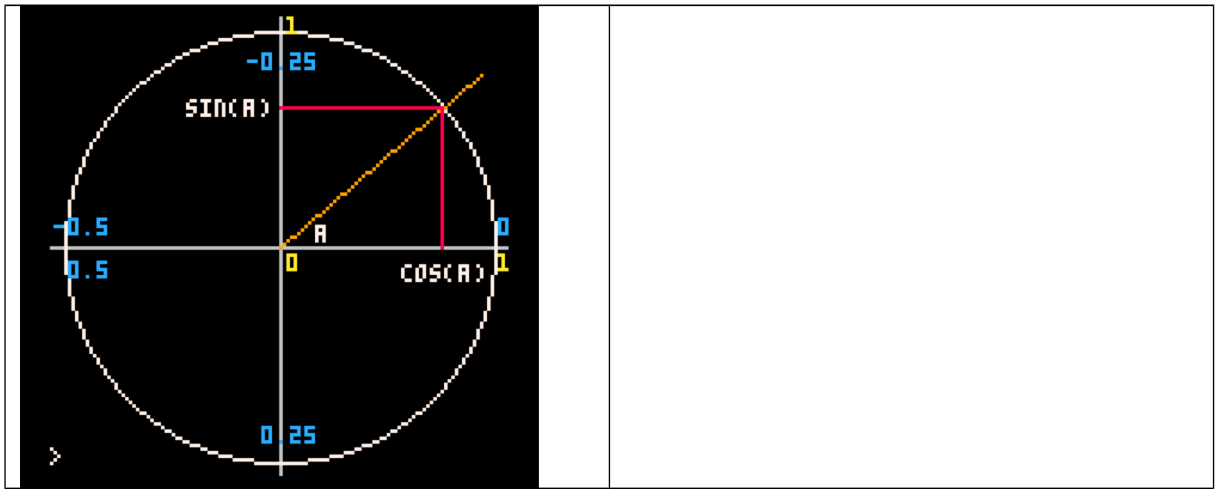


An equilateral triangle is a triangle in which all three sides are equal and all three internal angles are congruent to each other and are each 60° (360° of the full circle divided by 3 summit). The sum of all the angle of a quadrilateral (not crossed) is always 360°, e.g. square (4x90=360), pentagon (5x72=360),… Which bring us two questions. How can we measure distance? How can we measure angle? For the distance, Pythagoras gave us the solution (350BC) with his theorem relating the lengths of the sides (a, b) and the hypotenuse (c) of a right triangle: a^2 + b^2 = c^2.



If we put our points at each summit of the hypotenuse, the length of the hypotenuse C is the distance between them. The length of the side A is the absolute value of difference of their abscissas. The length of the side B is the absolute value of difference of their ordinates. This can be applied to our two first points P0(5, 10), P1(8, 5): (8-5)^2 + (5-10)^2 = C^2
If the parts of the equations are equal, their square roots is also: SQRT((8-5)^2 + (5-10)^2) = C
This give us the answer of the first question with the DISTANCE function.

Now we have to deal the second question which is a little bit more complicated. First, it's an angle question, so we have to introduce some trigonometry. Next, pico-8 has his own way to do with trigonometry. In pico-8 a full circle is… 1. To convert an angle in degrees to use in a trigonometric function, you have to divide it by 360 before. To do the reverse operation, you have to multiply the pico-8 angle by 360. Remind you this point each time you see the number 360 as divisor or multiplicator. The sine is inverted which is nice because like that it suits the screen space, e.g. PRINT(SIN(90/360)) returns -1.



It's time to meet an old friendly lady, a very useful function: arctangent with two arguments (nickname atan2). In short, the atan2 function return the direction (as an angle) from one point to another. In pico-8, you have to provide the abscissas difference and the ordinates difference between the destination and the origin points. This can be applied to our two first points P0(5, 10), P1(8, 5): ATAN2(8-5,5-10)*360 = 59.052 (so no it isn't an equilateral triangle even if it's look like).

The requisite groundwork has been covered and foundations laid to allow a little distraction. Why not rotate a square (or any described form) around the Z axis? What's about Z axis? We're in 2D! Yes, but… To rotate a form on the screen, we need to put a pin in it to turn around. We won't break the screen so we use the Z axis as substitute. The idea is to center the Z axis in the middle of the form. Then we will use a trigonometric circle having his center where the Z axis go through our screen. The radius of the trigonometric circle will be half the size of the form. In this context, rotating the form is equivalent to add an angle to his current angle and calculate the new coordinates of each points.

The angle transformation formulae we need are below (where a is the current angle and b the angle of the centered rotation):

$\cos(a+b)=\cos(a)\cos(b) – \sin(a)\sin(b)$
$\sin(a+b)=\sin(a)\cos(b) + \cos(a)\sin(b)$

For each point Pi(x, y) of our form, we can use a trigonometric circle that passes through this point. The radius of the trigonometric circle is 1 so we have to change the scale of our coordinates with a divisor (S). At this step, cos(a) = x/S and sin(a) = y/S :

$\cos(a+b) = x/S \cos(b) – y/S \sin(b)$
$\sin(a+b) = y/S \cos(b) + x/S \sin(b)$

The formulae give the coordinates of the point Pi(cos(a+b), sin(a+b)) after the rotation at the scale of the trigonometric circle. We need them at our current scale, so we have to change the scale of Pi coordinates:
Pi(S cos(a+b), S sin(a+b))

$S \cos(a+b) = S(x/S \cos(b) – y/S \sin(b))$
$S \sin(a+b) = S(y/S \cos(b) + x/S \sin(b))$

Which can be resolved to:

$S \cos(a+b) = x \cos(b) – y \sin(b))$
$S \sin(a+b) = y \cos(b) + x \sin(b))$

Pi (x,y) after a rotation of an angle b become Pi (x cos(b) – y sin(b), y cos(b) + x sin(b))

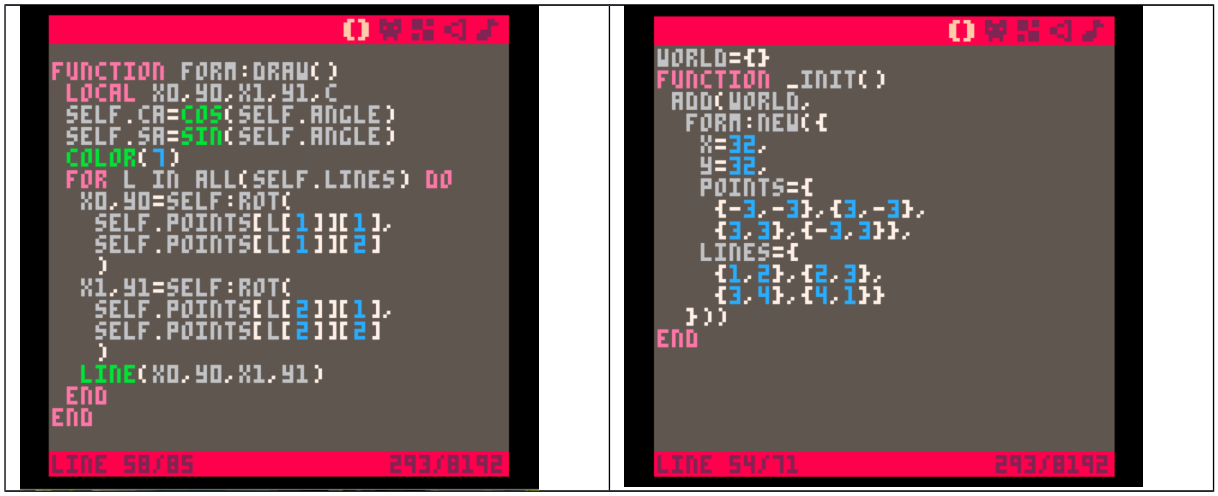
If I haven't lost your attention, you may be happy to get some practice. Seriously, if you understood or accept to use in state these tools (distance, direction, rotation), you could do whatever you want by assembling them. To ease the form manipulation, we introduce the use of the object oriented programming with FORM. The constructor accept a table as argument and uses its content to

initialize the properties of the new created object: abscissa (x), ordinate (y) of the screen position of the center of the described form, direction (angle) and its drawing (points and lines).

```
FORM={}
FUNCTION FORM:NEW(O)
 O=O OR {}
 O.X=O.X OR 0
 O.Y=O.Y OR 0
 O.ANGLE=O.ANGLE OR 0
 O.POINTS=O.POINTS OR {}
 O.LINES=O.LINES OR {}
 SETMETATABLE(O,SELF)
 SELF.__INDEX=SELF
 RETURN O
END
```

```
FUNCTION FORM:ROT(X,Y)
 RETURN
  FLR(
   X*SELF.CA-Y*SELF.SA+SELF.X),
  FLR
   (X*SELF.SA+Y*SELF.CA+SELF.Y)
END
```

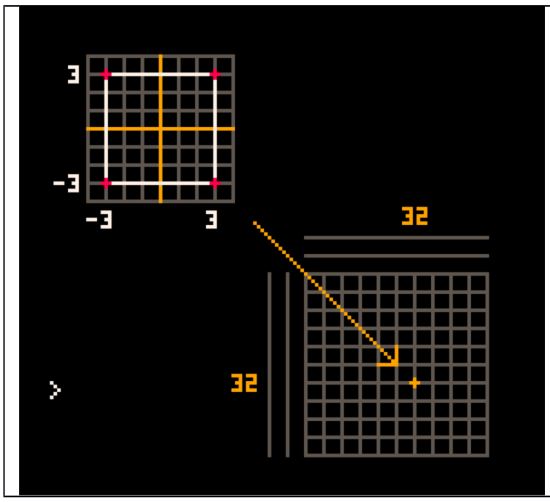
The ROT method calculates the coordinates of the point (x, y) on the screen with the rotation of ANGLE and the sliding to the FORM position. SELF.CA and SELF.SA are the cosine and the sine of the SELF.ANGLE. The rotation of the form is done with the same angle for each points of the form. To optimize, the cosine and sine are evaluated one time in the DRAW method and stored for reuse in the properties of the object.



```
FUNCTION FORM:DRAW()
 LOCAL X0,Y0,X1,Y1,C
 SELF.CA=COS(SELF.ANGLE)
 SELF.SA=SIN(SELF.ANGLE)
 COLOR(7)
 FOR L IN ALL(SELF.LINES) DO
  X0,Y0=SELF:ROT(
   SELF.POINTS[L[1]][1],
   SELF.POINTS[L[1]][2]
   )
  X1,Y1=SELF:ROT(
   SELF.POINTS[L[2]][1],
   SELF.POINTS[L[2]][2]
   )
  LINE(X0,Y0,X1,Y1)
 END
END
```

```
WORLD={}
FUNCTION _INIT()
 ADD(WORLD,
  FORM:NEW({
   X=32,
   Y=32,
   POINTS={
    {-3,-3},{3,-3},
    {3,3},{-3,3}},
   LINES={
    {1,2},{2,3},
    {3,4},{4,1}}
  }))
END
```

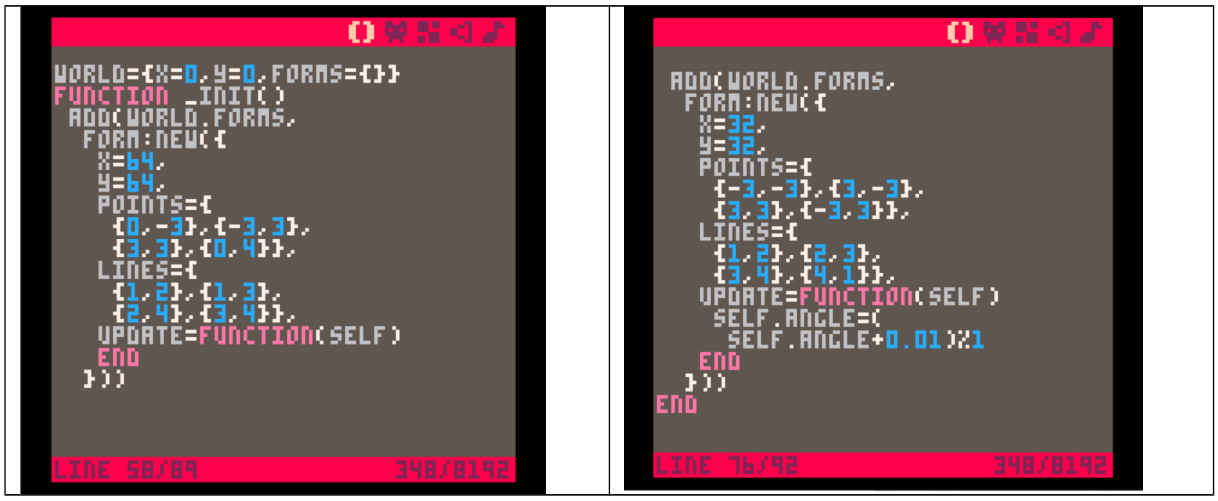
The DRAW method uses the same principle as the previously done for the triangle. It employs the ROT method to convert the coordinates before drawing the line on screen. The table WORLD is dedicated to the forms storage. In the pico-8 _INIT callback method, a new form (square) is created and added to the world. You can change the form position, direction and drawing or add new one.



The two last pico-8 callback functions handle respectively the drawing and update events. For each frame, the screen is erased and all DRAW methods of the forms in the world are called. The update increments the angle of each form. The modulus 1 limits the angle range to the interval [0 , 1[.

Welcome in your world…

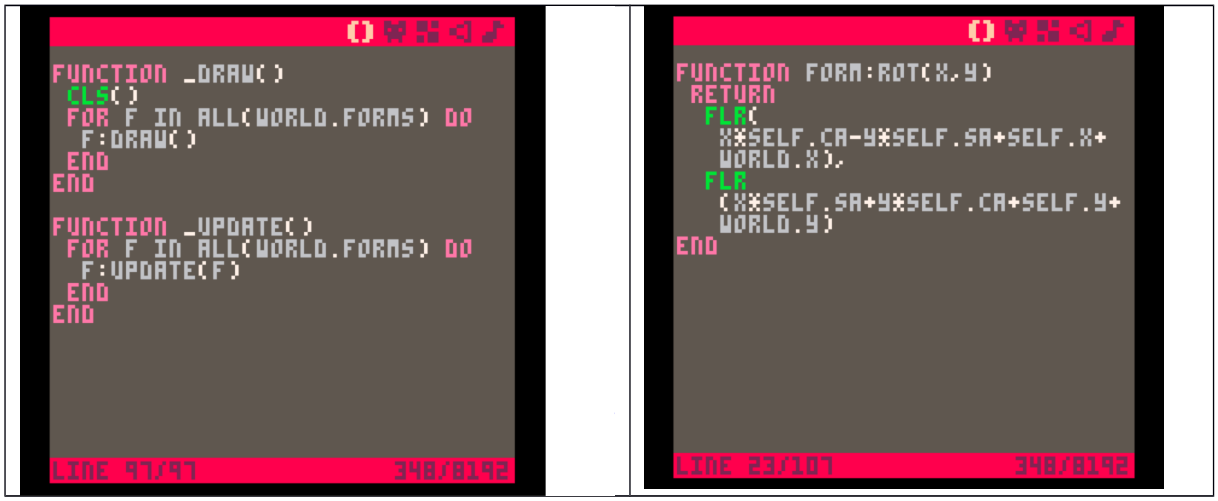
You are at one step to enter in your own world. We need just to add an avatar for the player to put you in the center of the world.



```
WORLD={X=0,Y=0,FORMS={}}
FUNCTION _INIT()
 ADD(WORLD.FORMS,
  FORM:NEW({
   X=64,
   Y=64,
   POINTS={
    {0,-3},{-3,3},
    {3,3},{0,4}},
   LINES={
    {1,2},{1,3},
    {2,4},{3,4}},
   UPDATE=FUNCTION(SELF)
   END
  }))
```

```
 ADD(WORLD.FORMS,
  FORM:NEW({
   X=32,
   Y=32,
   POINTS={
    {-3,-3},{3,-3},
    {3,3},{-3,3}},
   LINES={
    {1,2},{2,3},
    {3,4},{4,1}},
   UPDATE=FUNCTION(SELF)
    SELF.ANGLE=(
     SELF.ANGLE+0.01)%1
   END
  }))
END
```

The WORLD table evolves. It includes a screen offset (x, y) to scroll the axes when the player moves. The table containing the forms is now in the FORMS table of the WORLD table. In the _INIT callback, another FORM is added: the player itself. The forms are added to the WORLD.FORMS (rather than WORLD). A new property is added in the constructor call: UPDATE. In fact, this is a callback method where the code to manage the form can be put. The angle update of the square is moved in it. This allows the forms to have their own behavior.

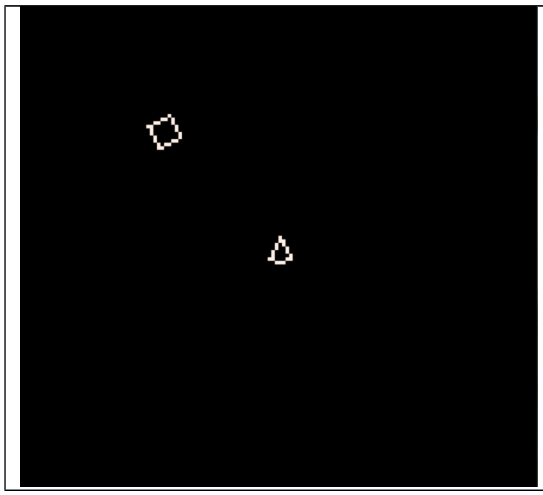


```
FUNCTION _DRAW()
 CLS()
 FOR F IN ALL(WORLD.FORMS) DO
  F:DRAW()
 END
END

FUNCTION _UPDATE()
 FOR F IN ALL(WORLD.FORMS) DO
  F:UPDATE(F)
 END
END
```

```
FUNCTION FORM:ROT(X,Y)
 RETURN
  FLR(
   X*SELF.CA-Y*SELF.SA+SELF.X+
   WORLD.X),
  FLR
   (X*SELF.SA+Y*SELF.CA+SELF.Y+
   WORLD.Y)
END
```

The pico-8 _DRAW and _UPDATE callback methods are modified to use WORLD.FORMS instead of WOLRD and call F:UPDATE(F). The WORLD offset management is done by the FORM:ROT method by adding it to the coordinates. In state, when the cartridge is launched, you can see the square who spins and the vessel of the player.

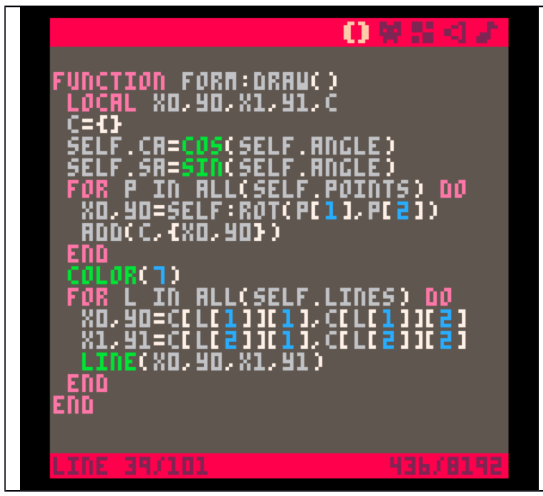


```
    {2,4},{3,4}},
  UPDATE=FUNCTION(SELF)
   LOCAL A,DX,DY
   IF BTN(0) THEN
    SELF.ANGLE+=0.01 END
   IF BTN(1) THEN
    SELF.ANGLE-=0.01 END
   IF BTN(2) THEN
    SELF.ANGLE=SELF.ANGLE%1
    A=SELF.ANGLE-0.25
    DX=1.3*COS(A)
    DY=1.3*SIN(A)
    WORLD.X+=DX
    WORLD.Y+=DY
    SELF.X-=DX
    SELF.Y-=DY
   END
  END
}}
LINE 64/101                 417/8192
```

The last - but not the least - action is to implement the update method of the player previously left empty. The left and right arrows control the direction of the vessel. The front is on the up summit (90° left from the drawing) so we have to subtract 0.25 (90/360) from SELF.ANGLE to get the moving direction. The distance of each move is 1.3 (pixels) is decomposed on each axis regarding the angle of the direction (A). The offset of the world and the position of the vessel are updated to move the world around the player (leaving his vessel in the middle of the screen).

Optimize

Has done before for the cosine and sine of ANGLE, we can store the results of the calls of the method ROT used while drawing the lines later. The table C in FORM:DRAW is dedicated to this usage.



```
FUNCTION FORM:DRAW()
 LOCAL X0,Y0,X1,Y1,C
 C={}
 SELF.CA=COS(SELF.ANGLE)
 SELF.SA=SIN(SELF.ANGLE)
 FOR P IN ALL(SELF.POINTS) DO
  X0,Y0=SELF:ROT(P[1],P[2])
  ADD(C,{X0,Y0})
 END
 COLOR(7)
 FOR L IN ALL(SELF.LINES) DO
  X0,Y0=C[L[1]][1],C[L[1]][2]
  X1,Y1=C[L[2]][1],C[L[2]][2]
  LINE(X0,Y0,X1,Y1)
 END
END


LINE 39/101                 436/8192
```

Jean-Marc "jihem" QUERE
http://picoscope101.fr
@wdwave