I was more interested in creating my own Turing machine than using someone else’s, or simply researching. I’m also interested in generative art which pushed me towards this graphics based Turing machine. I wanted to create a kind of perpetual motion turing machine, I thought it would be interesting creating patterns that do this. I first had to make sure that my Turing machine framework actually processed instructions correctly.

I chose to use coloured squares rather than images because this would simplify the coding of the machine because there is often a lot of overhead required to display images. I made the squares fairly small, thirty pixels. At first I experimented with a small grid of five by three to get the basics of my machine rule processor working. I then extended it to a ten by ten grid, still with thirty pixel squares as the “image”.

I decided I would either write my machine in Processing or NodeBox. Processing a Java based graphics library and IDE while NodeBox is based on python. I chose to use Python because Processing uses a version of Java which does not include some nice iterators. The disadvantage of using Nodebox is that it is only currently stable on the OS X platform. The framework for the machine consists of 3 main functions, display(tape), draw() and processRules(set). State is maintained in a global variable state, as is the current head location. This is stored in two variables (xpointer, ypointer) rather than the single variable that would be required in a traditional Turing framework because of the two dimensions. The machine tape is stored in a two dimensional array.

display(tape) will render the tape passed to it. Colours and scale are set internally. As discussed, I had a scale of thirty pixels (square). I chose harmonious colours of light red and purple. Black was used to indicate a blank section on the tape. These colours were the external representation of the tape which used 1 and 2 as its internal language, with 0 representing blank. The tape and therefor the input string could be directly modified before running the program.

processRules(set) takes an array of strings which are in the correct “rule” format. These are then processed into a dictionary and returned. This dictionary is used in draw() to work out what state to change to on what input and what to output. A rule has five parts to it. An input state, an input symbol, an output state, an output symbol, and a movement. These are comma separated in a string.

For example, if you want the machine to move right, change to state q2 and output a 1 if the input is 2 and the machine is already in the state q1:

```
q1,2,q2,1,r
```

I found that any required transition can be expressed in this rule form. A “-” can be used to indicate no movement. Similarly, if no output is required, this behaviour can be simulated by actually outputting the same symbol as the input, i.e.:

```
q1,2,q2,1,-
```

The framework does not protect from useless rules, such as on that does not change state, change symbol or move. This however can be conveniently used to specify a final halt state.
I decided to make edges “wrap”, that is, if the head points to the right most “square” in a row and then moves one square right, the head will wrap back around to the first square on the row. Similar behaviour is implemented for vertical movement. If at the bottom most square and the head moves down, the head will wrap back to the top most square. I briefly experimented with creating a true infinite tape, but the data structure used to store the tape easily grew very large. I was also more interested in creating “self feeding” machines, which, while possible on a true infinite tape, they make more sense on a wrapping tape. This machine would process some input, and then create some output which could then be used as input for its self.

The first I made was the relatively simple, “string inverter”. This was created with the rule set;

```
set = ["q1,1,q1,2,r",
      "q1,2,q1,1,r",
      "q1,0,q1,0,-"]
```

The following is the beginning and end machine output.

![Machine Output](image)

This proved that my framework worked but did not take advantage of 2D space available. I then made a machine which would separate red and purple squares on to separate lines which was expressed with the rule set;

```
set = ["q1,1,q1,1,r",
      "q1,2,q2,0,d",
      "q2,0,q3,2,r",
      "q3,0,q1,0,u",
      "q3,2,q3,2,-",
      "q1,0,q1,0,-"]
```

This machine depended on the second row being empty. The following is the beginning and end of the machine output.

![Machine Output](image)
I attempted to create a sorting machine, which would split the two colours onto separate lines and then compress them (ie, remove the blanks between squares) but this is not possible without knowing the wrapping boundary, because the machine would need to know when its wrapped around to what is really the start of the compressed squares. Although I did technically know this value, it was not possible to create a generic machine that would handle this. Each time the wrapping boundary changed, a whole new set of states would have to be created to “count” the heads position along a row.

I then set about creating the circular machine. I decided that the machine should be self seeding, that is, the tape should begin as all blank.

With the following rules:

```
set = ["q1,0,q2,1,d",  
      "q2,0,q1,2,l",  
      "q1,1,q2,2,r",  
      "q2,1,q1,1,u",  
      "q1,2,q1,1,d",  
      "q2,2,q1,1,d"]
```

The machine produces the following,

However as shown in image three, this machine quickly falls into chaos, with no discernible self replicating pattern. It’s easy to create a this kind of circular “self feeding” machine, so this result was uninteresting.
I tried the following rules;

```
set = ["q1,0,q2,1,d",
      "q2,0,q1,2,r",
      "q1,1,q2,2,r",
      "q2,1,q1,1,r",
      "q1,2,q1,1,r",
      "q2,2,q1,1,r"]
```

This almost worked infinitely looping.

but got stuck on row two,

I decided that this rule set was not going to work so I wrote out a new set to try.

```
set = ["q1,0,q1,1,r",
      "q1,1,q2,2,r",
      "q1,2,q1,1,l",
      "q2,1,q1,2,d",
      "q2,2,q1,1,d"]
```

With this rule set I thought I had actually created a kind of recursive behaviour, At first the machine would fill up the tape with a diagonal line as shown in the first image, and then it continued to create these diagonal lines, with the smaller ones “moving” from right to left.
and the larger one moving from left to right. I was quite excited at the prospect of this macro/micro combination, but unfortunately, it eventually devoles into image 3. I tweaked the code slightly, resulting in the following rules;

\[
\text{set} = [\ "q1,0,q1,1,r", \\
"q1,1,q2,2,r", \\
"q1,2,q1,1,l", \\
"q2,1,q1,2,u", \\
"q2,2,q1,1,r"]
\]

These rules actually presented good results and achieved what I wanted, a self replicating circular machine.

It initially filled up the tape with the pattern emerging in image one. It then made the purple line wider and wider until the entire tape was purple, after that it would begin making the purple line again, shifted to the right slightly.

This completed my goal of creating a self perpetuating Turing machine. Video is available online at [http://criticaleffect.net/self_h264.mov](http://criticaleffect.net/self_h264.mov) (230KB)

The ability of modern displays does not have any gain over the “old style” Turing machines, other than being able to display more of the current context at which the Turing machine is executing. Old Turing machines were capable of using multiple tapes, which can be used to simulate the a modern 2D display.
Appendix A: Code

Note: This code requires to be run from inside the software package “Nodebox” available here: http://nodebox.net

def display(tape):
c0 = color(0, 0, 0)
c1 = color(0.9, 0.3, 0.4) #0
c2 = color(0.4, 0.3, 0.5) #1
x = 0;
y = 0;
sk = 30;
for row in tape:
    for column in row:
        if(row == ypointer and column == xpointer):
            stroke(color(1,1,1))
        else:
            stroke(None)
        if(column == 0):
            fill(c0)
        elif(column == 1):
            fill(c1)
        elif(column == 2):
            fill(c2)
        else:
            fill(0, 0, 0);
        rect(x, y, sk, sk)
        x =  x + sk
        x = 0
        y = y + sk

def processRules(set):
    rules = dict()
    for rule in set:
        inner = rule.split(',')[0]
        try:
            rules[inner[0]]
        except KeyError:
            rules[inner[0]] = dict()
            rules[inner[0]][int(inner[1])] = {
                "state":inner[2],
                "output":int(inner[3]),
                "move":inner[4]
            }
    return rules

speed(25)
def setup():
    global state
    global xpointer
    global ypointer
    global tape
    global rules

    set = [  "q1,1,q1,0,r",
            "q1,2,q2,1,r",
            "q2,1,q1,0,u",
            "q2,2,q2,1,l",
            "q2,0,q2,1,u",
            "q1,0,q1,2,u"
    ]

    #inverter
set = [ "q1,1,q1,2,r",
      "q1,2,q1,1,r",
      "q1,0,q1,0,-"]

#separator
set = [ "q1,1,q1,1,r",
      "q1,2,q2,0,d",
      "q2,0,q3,2,r",
      "q3,0,q1,0,u",
      "q3,2,q3,2,-",
      "q1,0,q1,0,-"]

#perpetual motion machine
set = [ "q1,0,q1,1,r",
      "q2,1,q2,2,r",
      "q1,2,q1,1,l",
      "q2,1,q1,2,u",
      "q2,2,q1,1,r"]

rules = processRules(set)

#print rules

tape = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
]

size(len(tape[0])*30, len(tape)*30)
state = "q1"
xpointer = 0
ypointer = 0

def draw():
global rules
global state
global xpointer
global ypointer
    #print "State: ", state
    #print "Xpointer: ", xpointer
    #print "Ypointer: ", ypointer
    try:
        action = rules[state][tape[ypointer][xpointer]]
    except KeyError:
        print "Halted"
        display(tape)
        return

    #print action
    tape[ypointer][xpointer] = action['output']
    state = action['state']
    if(action['move'] == 'r'):
        xpointer = xpointer+1
    elif(action['move'] == 'l'):
        xpointer = xpointer-1
    elif(action['move'] == 'u'):
        ypointer = ypointer-1
elif(action['move'] == 'd'):
    ypointer = ypointer+1
elif(action['move'] == '- '):
    # No movement
    pass
else:
    print "Something went wrong, unknown movement"

# Wrapping
if(xpointer == -1):
    xpointer = len(tape[0])-1
if(xpointer == len(tape[0])):
    xpointer = 0
if(ypointer == -1):
    ypointer = len(tape)-1
if(ypointer == len(tape)):
    ypointer = 0
display(tape)
#export("turing_"+str(counter), extension=".png")