

Chapter 2

Graphics

Curiosity might ask *how* an image is built at all. In this chapter we consider three different representations: a bitmap assigning colors to each pixel, a scalable model requiring later calculations to render the image, and an even higher level specification where the underlying details are completely hidden from view.

As an example of our first distinction, PPM stands for “portable pixel map” and this format specifies literally hundreds of thousands of red-green-blue color values in very large image files. SVG is “scalable vector graphics” where images contain only a geometric description of lines and curves, thus reducing file size by delaying the pixel-by-pixel rendering process. This approach has the advantage that enlarging a vector image is immune from any pixelation issues the corresponding bitmap would face, but it also means the original image must be conceived in terms of geometric objects which is not trivial for, say, a photograph.

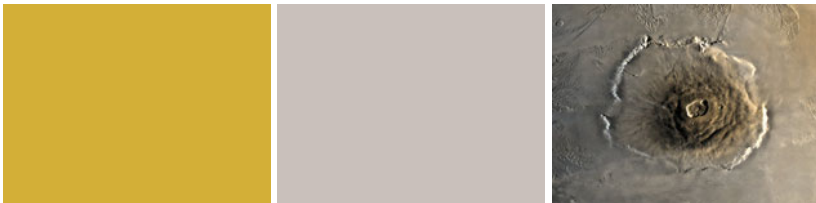


Fig. 2.1: Example image files: gold, silver, and Olympus Mons, Mars, the largest known volcano. Image courtesy of NASA from the Viking 1 mission, 22 June 1978.

Code Listing 2.1: A complete program, written in Python 2 with PIL.

```
#
from PIL import Image
img=Image.new('RGB', (100,75), (212,175,55)) # gold
img.save('gold.png') # formats: JPG, PPM/PGM, EPS
#
```

Table 2.1: A pixel map for a very small 5×5 image.

	[0]	[1]	[2]	[3]	[4]
[0]	•	•	•	•	•
[1]	•	•	•	•	•
[2]	•	•	•	•	•
[3]	•	•	•	•	•
[4]	•	•	•	•	•

2.1 Pixel Mapping

Table 2.1 shows a very small 5×5 image. If one byte is used to specify each color value and there are 25 pixels, then we require only 75 bytes to store all RGB data for this entire image. (Each dot stores red 0-255, green 0-255, blue 0-255 at the pixel center.) PPM files also include a header listing such information as the width and height of our image, but the header does not scale with image size in the same manner that the amount of color data will.

Lab211: Circle π

Figure 2.2 shows one quadrant of the unit circle within a unit square. Your assignment is to produce such an image. Code Listing 2.2 specifies a side-length m which in turn determines the total number of pixels $n = m^2$. The variable `count` will be used to count-up how many of these n pixels are also inside our unit circle.

In addition, Figure 2.3 shows pixelation when a smaller bitmap is enlarged, either by direct calculation or with interpolation of the color values. Free tools are available for this kind of image manipulation.

Code Listing 2.2: Initializing variables.

```
#
m=600
n=m*m
#
count=0
#
```

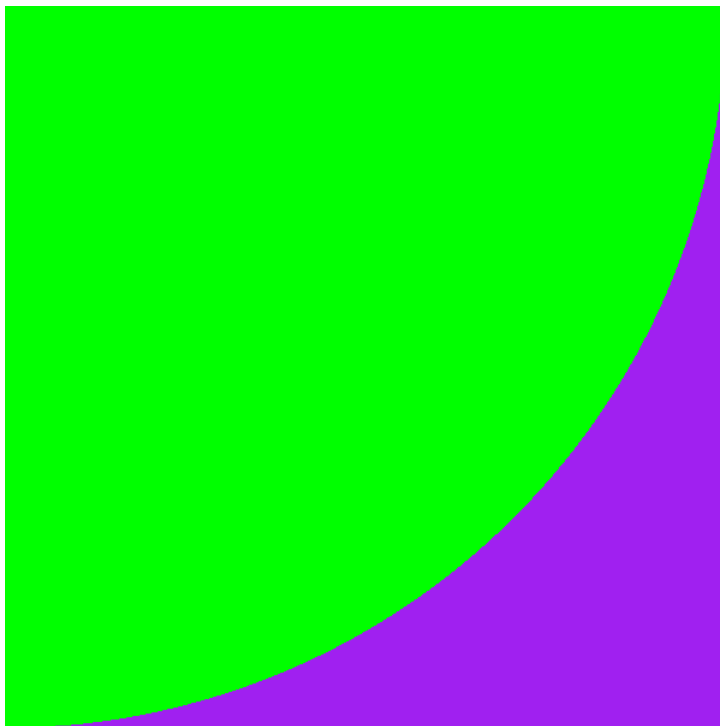


Fig. 2.2: One quadrant of the unit circle within a unit square. Note y-coordinates are often inverted in graphical systems, for either interactive windows or stored files.

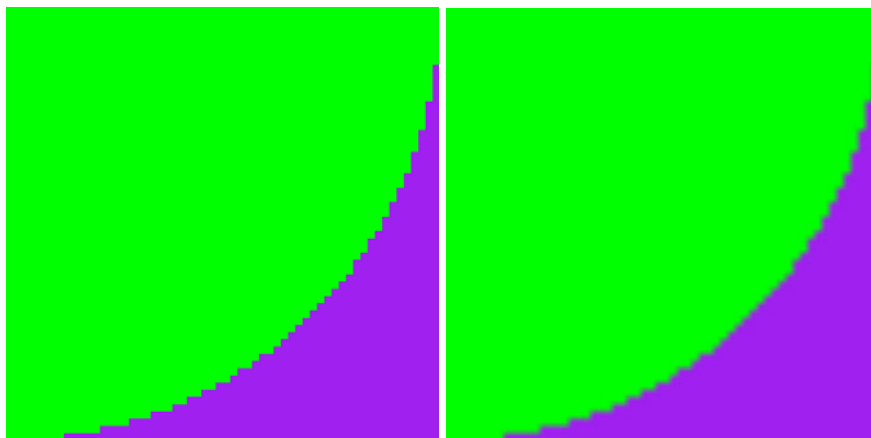


Fig. 2.3: Pixelation when a smaller bitmap is enlarged. Left: direct calculation. Right: linear interpolation of colors using the GNU Image Manipulation Program.

Table 2.2: Image size, computed value of π , and associated runtime.

size m	computed π	runtime, seconds
100	3.1428	0.05
1000	3.141676	1.32
10,000	3.14159388	132.31

Table 2.3: Size (no image), computed value of π , and associated runtime.

size m	computed π	runtime, seconds
100	3.1428	0.01
1000	3.141676	0.88
10,000	3.14159388	88.53
1,000,000	3.141592655988	11.55*

The area of a circle is $A = \pi r^2$ and so for the unit circle, where $r = 1$, area is $A = \pi$. For only one quadrant area is $A = \pi/4$. If the n pixels in our image represent a unit square with area $A = 1$, the number of pixels inside the circle will relate to n by a $\pi : 4$ ratio. Since we count these pixels in our code we may approximate π with improving accuracy as n increases, shown in [Tables 2.2](#) and [2.3](#) with runtimes for an Intel[®] Core i7-940 chip.

Code Listing 2.3 converts from pixel coordinates to unit coordinates, and also shows how the `putpixel` method is called on an image to set the RGB color value of a single pixel. Of course the value of (x,y) rather than (xp,yp) determines if a point is inside the unit circle or not.

Code Listing 2.3: Initializing variables.

```
#
x=(xp+0.5)/m # plus one-half --> pixel center!
#
img.putpixel((xp,yp),(160,32,240)) # purple
#
```

* Estimated runtime for size $m = 10^6$ is over ten days. Our eleven second runtime is based on a more efficient calculation suggested on the next page. A common story: the necessity of running a large problem is what compels us to consider a more sophisticated technique in the first place.

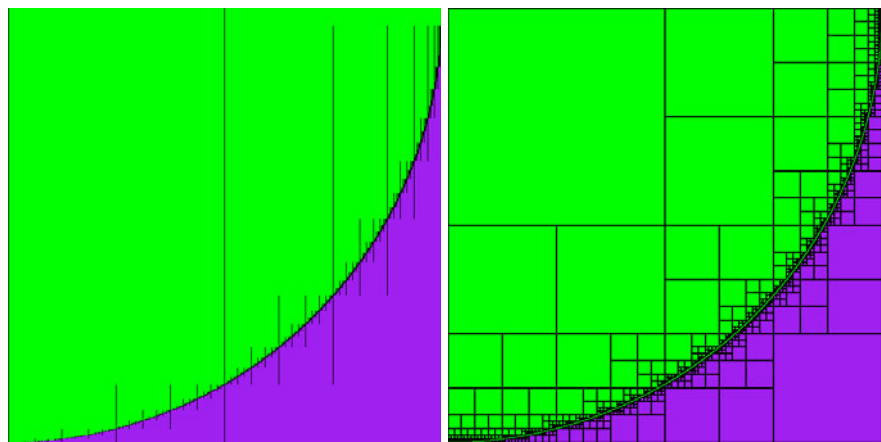


Fig. 2.4: Circle divide-and-conquer. Left: binary search, only the marked pixels are checked and all other pixels are classified automatically. Right: quadtree, a similar idea in 2-D where only the corners of each box are checked. In general our goal is to localize calculations for larger sizes along the edge of the circle, where they matter.

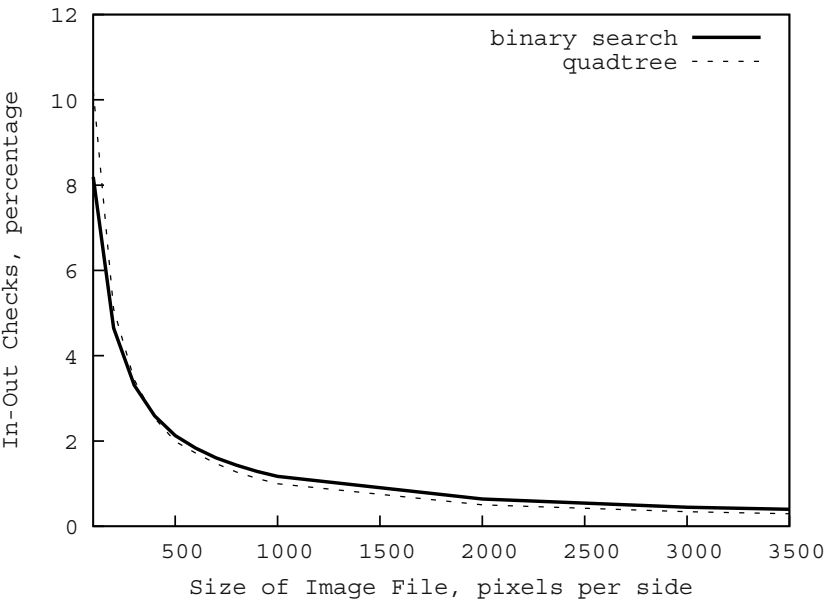


Fig. 2.5: Divide-and-conquer savings. For the previous size $m = 10^6$ result we might alternatively process the $10^6 \times 10^6 = 1$ trillion pixels in parallel using over 75,000 computers to achieve the same runtime performance, thus we tend to prefer a better algorithm to a bigger computer (or more computers) when possible.

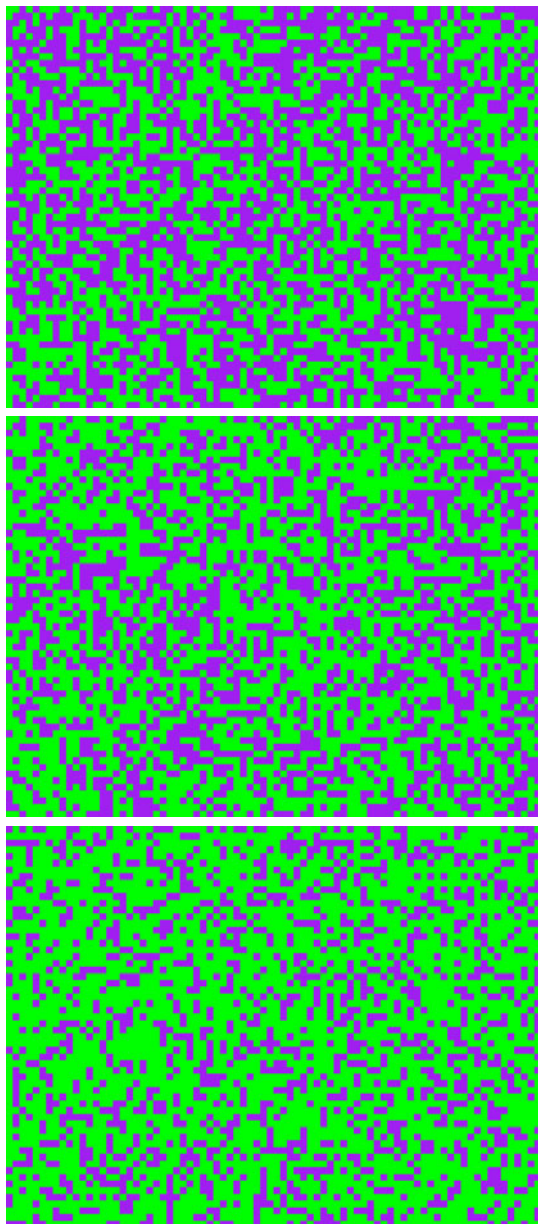


Fig. 2.6: Percolate pixelate. Top to bottom: $p = 0.5$, $p = 0.6$, and $p = 0.7$.

Lab212: Percolate Pixelate

Consider an 80×60 image where each pixel is colored green with probability p and purple with probability $1 - p$. Algorithm 2.1.1 enlarges this image to 800×600 by converting each pixel into a 10×10 block of 100 pixels. Figure 2.6 shows typical results for $p = 0.5$, $p = 0.6$, and $p = 0.7$. Later in Chapter 5 we will ask:

For what probabilities will there be a green pathway connecting all four sides?

If each pixel has at most four neighbors (i.e., not counting diagonals) then it appears the $p = 0.5$ image does not have a path while $p = 0.7$ clearly does. For $p = 0.6$ it is not at all obvious what will happen in general.

This question is related to “percolation” or the flow of fluids (e.g., groundwater) in porous material such as rock or a layer of sediment (or the flow of boiling water through coffee grounds in a percolator). Percolation theory applies graph algorithms and statistics to what was originally conceived as a physical science problem.

Also, in this context pixelation is actually helpful because it aids the human eye in tracing pathways across the image. It would not be desirable to use a “better” image with some form of interpolation, although eventually this will not matter once we have coded an automatic tool to determine if such a pathway exists.

Algorithm 2.1.1 Enlarging a smaller bitmap image.

```

1: while  $y = 0 \rightarrow 59$  do
2:   while  $x = 0 \rightarrow 79$  do
3:     if  $\text{random} < p$  then
4:        $\text{color} = \text{green}$ 
5:     else
6:        $\text{color} = \text{purple}$ 
7:     end if
8:     while  $i = 0 \rightarrow 9$  do
9:        $y_{\text{new}} = 10y + i$ 
10:      while  $j = 0 \rightarrow 9$  do
11:         $x_{\text{new}} = 10x + j$ 
12:         $\text{pixel}(x_{\text{new}}, y_{\text{new}}) = \text{color}$ 
13:      end while
14:    end while
15:  end while
16: end while
  
```

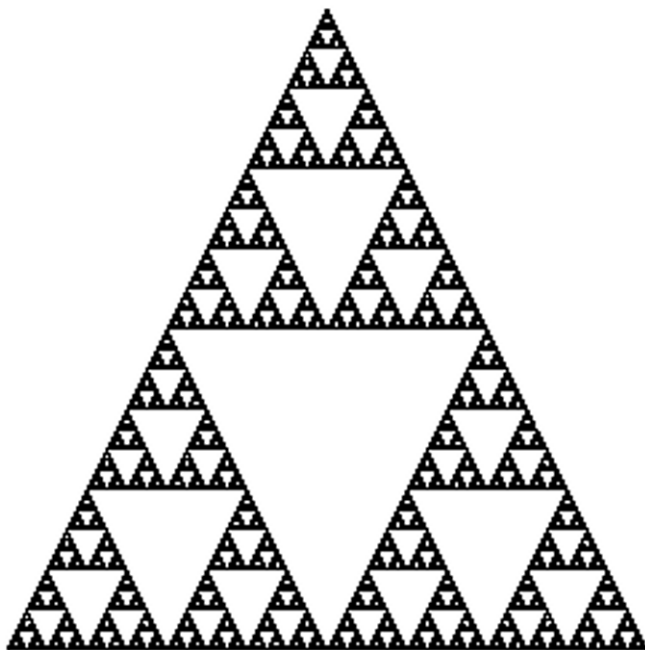


Fig. 2.7: Sierpinski's Gasket, formed by a Chaos Game.

Lab213: Sierpinski's Gasket

Fix three points P_1 , P_2 , and P_3 . As shown in [Figure 2.7](#) these are $P_1 = (0.5, 0.1)$, $P_2 = (0.1, 0.9)$, and $P_3 = (0.9, 0.9)$, if we map the image pixel coordinates to unit square coordinates. Randomly initialize a fourth point $P = (x, y)$, pick one of the three fixed-points also at random, then move P halfway toward that point and draw the corresponding pixel in your image. Now repeat: randomly pick one of the three fixed-points, move P halfway from its current position, and draw the pixel.

Our result is a famous fractal called Sierpinski's Gasket and this random drawing process is known as a Chaos Game. Experiment with the total number of loops for yourself but [Figure 2.8](#) provides some guidance for a 300×300 image. Note how we reach "carrying capacity" because there are only so many pixels to draw.

Alternative drawing techniques are suggested in [Figure 2.9](#).

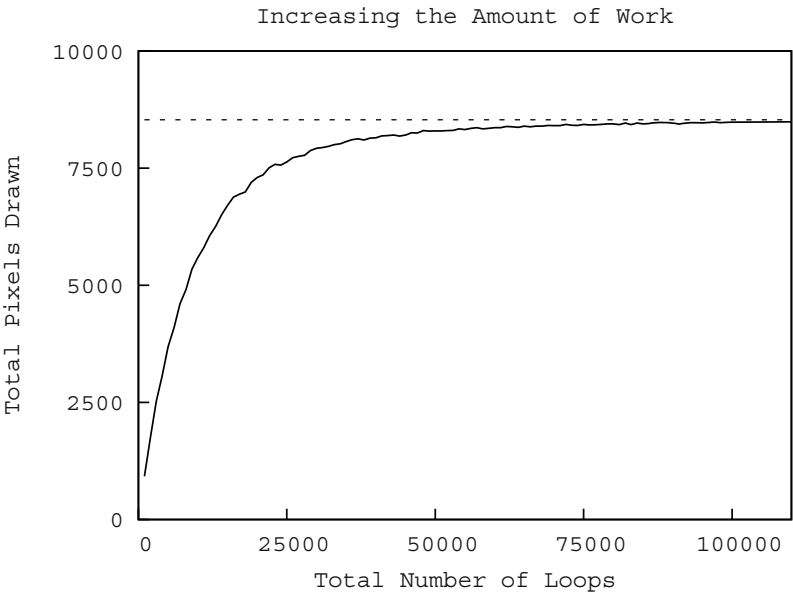


Fig. 2.8: Diminishing marginal returns. Since there are only so many pixels to draw eventually more-and-more looping introduces fewer-and-fewer new pixels.

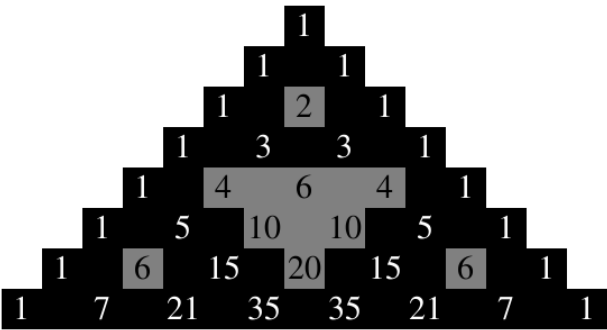


Fig. 2.9: Pascal's Triangle. Obviously we could generate a similar image geometrically by starting with a whole triangle and recursively discarding the middle quarter. Or, here we suggest a technique based on coloring the even-odd values in Pascal's much used triangle, albeit for a sample this small our image is quite pixelated.

Lab214: Draw a Line

A line may be drawn from a point $P_1 = (x_1, y_1)$ to another point $P_2 = (x_2, y_2)$ by looping from $t = 0.0$ to $t = 1.0$ and drawing pixels corresponding to:

$$x = x_1 + t \cdot (x_2 - x_1)$$

$$y = y_1 + t \cdot (y_2 - y_1)$$

Clearly when $t = 0.0$ we draw P_1 , when $t = 1.0$ we draw P_2 , and for $0.0 < t < 1.0$ the pixels in-between are drawn. However, if we choose dt too large then we may draw a “dotted” line and for dt too small we waste time re-drawing the same pixels over and over again.

As shown in [Figure 2.10](#) your assignment is to draw 50 random lines where P_1 is chosen inside a circle with radius $r = 75$ (all units are in pixels and the image size is 250×250) and P_2 is outside that circle but inside another concentric circle with $r = 125$. Note the lack of anti-aliasing here, as in Jack Bresenham’s famous line drawing algorithm but later addressed by Xiaolin Wu.

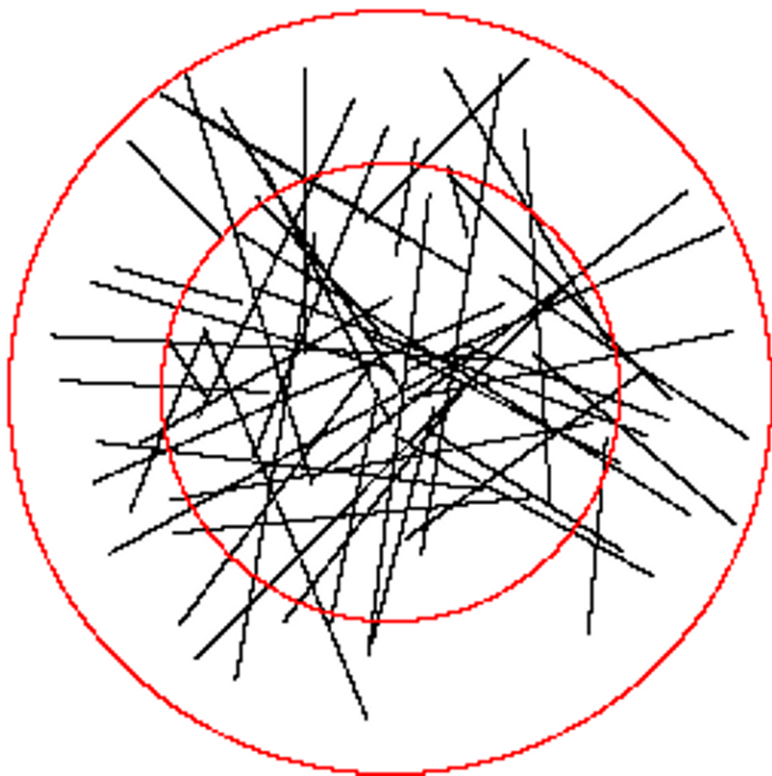


Fig. 2.10: Random lines in concentric circles.

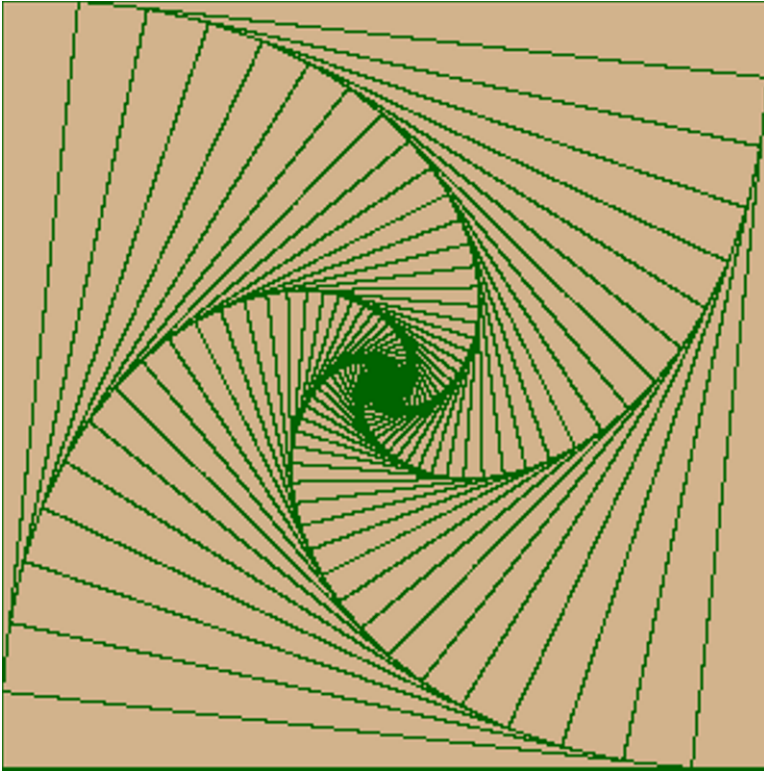


Fig. 2.11: Chasing turtles draw the “envelope” of four spirals.

Lab215: Chasing Turtles

We might imagine the spirals shown in [Figure 2.11](#) are the result of four bugs chasing each other, or four dogs, or even penguins. But as the next problem will use turtles we imagine them initialized at the corners of a box and looking only at their nearest clockwise neighbors. Steps “simultaneously” move the turtles 10% of the way toward these nearest neighbors while also drawing the lines of sight.

Code Listing 2.4: Color codes for tan and dark green.

```
#  
img=Image.new('RGB', (w,h), (210,180,140)) # tan  
#  
img.putpixel((x,y), (0,100,0)) # dark green  
#
```

2.2 Scalable Format

Like an SVG file our turtle programs will specify how shapes should be drawn, and both systems delay the actual pixel-by-pixel rendering process until either the vector image is displayed or the turtle walks along its path (carrying a marker, of course). Each format is a high-level representation of a drawing that may be scaled to an image of any size. Similar techniques are used to render 3-D models from CAD files.

Lab221: Turtle Square

In Code Listing 2.5 a general `drawline` function is defined on Line 11, essential turtle functions on Lines 25-45, and Line 46 onward is program specific. Output is shown in [Figure 2.12](#). Initialization of the turtle (Lines 54-56) is required for any drawing. Your assignment is to replace the ellipses with working code.

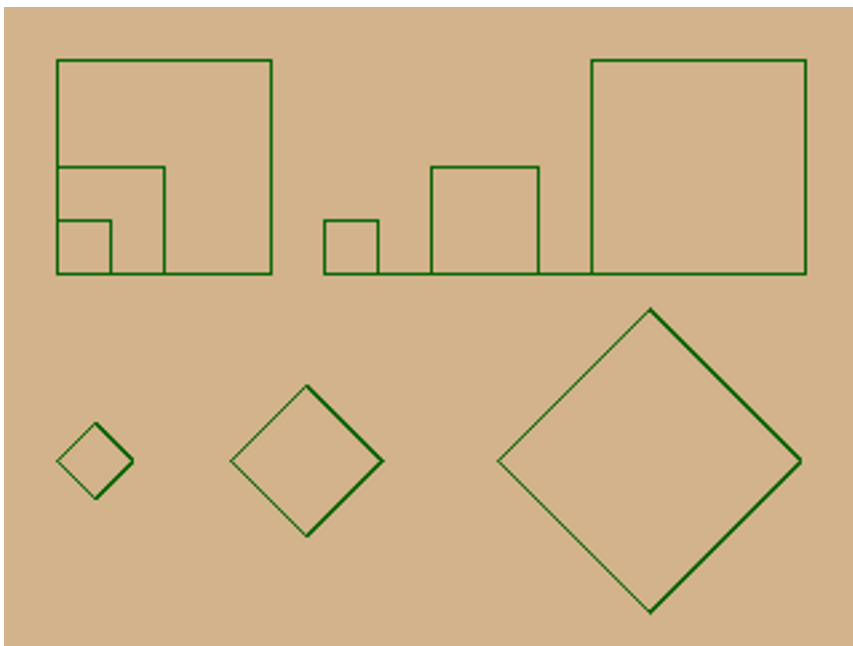


Fig. 2.12: Turtle square, rendered in a variety of sizes, sequences, and directions.

Code Listing 2.5: An incomplete program is known as a “shell.”

```

1 #####
2 #
3 # Chapter 2: Graphics
4 # Problem 2: Scalable Format
5 # Lab 2.2.1: Turtle Square
6 #
7 #####
8 from PIL import Image
9 from math import cos,sin,pi
10 #####
11 def drawline(x1,y1,x2,y2):
12     #
13     t=0.0
14     while t<=1.0:
15         #
16         ...
17         #
18         x=int(x+0.5)           # round to the nearest pixel
19         y=int(y+0.5)
20         img.putpixel((x,y), ...)
21         #
22         t+=0.001
23     #
24 #
25 def jump(xnew,ynew):
26     global xt,yt
27     #
28     xt=xnew
29     yt=ynew
30 #
31 def move(r):
32     global xt,yt
33     #
34     oldx,oldy=xt,yt
35     #
36     xt += r*cos(ht*pi/180.0)
37     yt += -r*sin(ht*pi/180.0)   # inverted
38     #
39     drawline(oldx,oldy,xt,yt)
40 #
41 def turn(dh):
42     global ht
43     #
44     ht+=dh                     # counterclockwise
45 #
46 def square(size):
47     #
48     ...
49     #
50 #
51 #####
52 img=Image.new('RGB',(320,240), ...)
53 #
54 xt = 20.0 # x-position of turtle
55 yt = 100.0 # y-position
56 ht = 0.0 # heading, in degrees
57 #
58 square(20.0)
59 ...
60 #
61 # end of file
62 #
63 #####

```

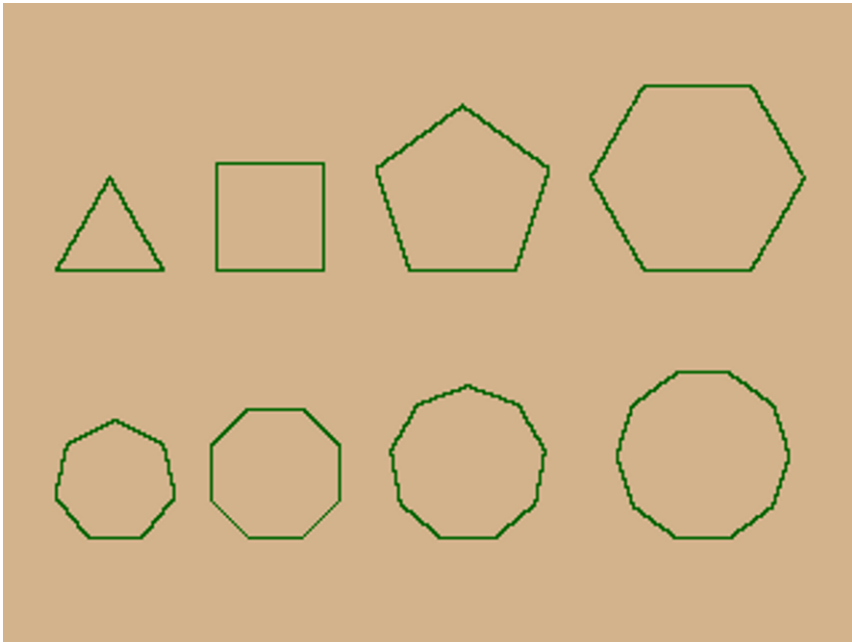


Fig. 2.13: Turtle poly, rendered for an increasing number of sides.

Lab222: Turtle Poly

Figure 2.13 shows polygons with $n = 3, 4, 5, 6$ sides and side-length 40 pixels (top) and $n = 7, 8, 9, 10$ of side-length 20 pixels (bottom). Code Listing 2.6 shows how easy `square` is to write once the `poly` function is working.

Code Listing 2.6: A more general function.

```
#
def poly(size,n):
    #
    ...
#
def square(size):
    #
    poly(size,4)
#
```

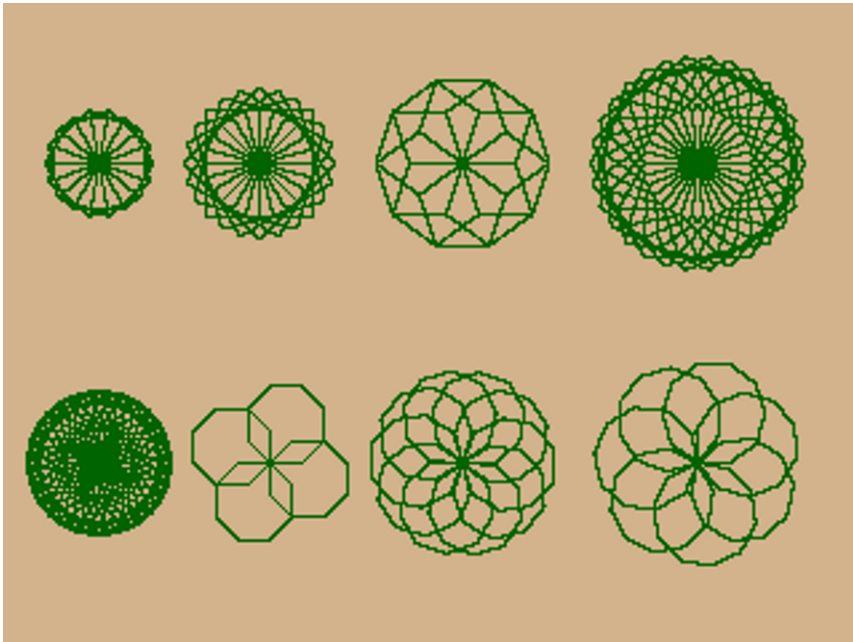


Fig. 2.14: Spin poly, rendered for a variety of patterns.

Lab223: Spin Poly

Figure 2.14 shows polygons spinning in beautiful ways. The idea of turtles drawing vector graphics is not new and versions for many different programming languages might be used in a variety of contexts and education levels.

Our code uses “local” variables in `drawline` because values of t , x , and y are unimportant once the function has finished executing. But we use “global” variables in `jump` and `turn` because the values of xt , yt , and ht are essential in tracking our turtle as it moves across the image over time.

Function `move` uses each kind of “scope” and note how Python treats arguments like r , dh , and $size$ as copies* of the original. Local copies. By default an assignment in a function will create a local variable as with t and x and y . When you see the `global` command it is telling Python not to do this, that the variable is not local, as with a turtle’s data xt and yt and ht .

* Variables stored with a “pointer” and passed to a function may be altered but not by assignment.

A Function that Updates Global Variables

```
#
def jump(xnew,ynew):
    global xt,yt
    #
    xt=xnew
    yt=ynew
    #
#
...
#
xt= 20.0
yt=100.0
#
jump(0.0,0.0)
#
print xt,yt      # output is 0.0 0.0
#
```

A Function that Creates Local Variables Instead

```
#
def jump(xnew,ynew):
    #
    xt=xnew
    yt=ynew
    #
#
...
#
xt= 20.0
yt=100.0
#
jump(0.0,0.0)
#
print xt,yt      # output is 20.0 100.0
#
```

Lab224: Spin Spiral

Figure 2.15 shows polygons spinning while side-length also changes. In particular, the circular spiral was inspired by an automated lawn mower project where the gap width between layers had to match the specific width of the real-life lawn mower. Can you draw this spiral with a *purposeful* gap width?

Turtle code may be organized into modules as Python does with its Tk library, math module, the Python Imaging Library (PIL), and even a built-in turtle:

<http://docs.python.org/library/turtle.html>

A `drawline` function might be in a general-use module for graphics separate from the turtle library. (In fact PythonWare[®] has `ImageDraw` in PIL.) Programs could access these resources with the same kind of `import` statements we have been using for “official” modules. We did not organize our programs this way only because the code is small and, especially when beginning, ease-of-use is highly valued.

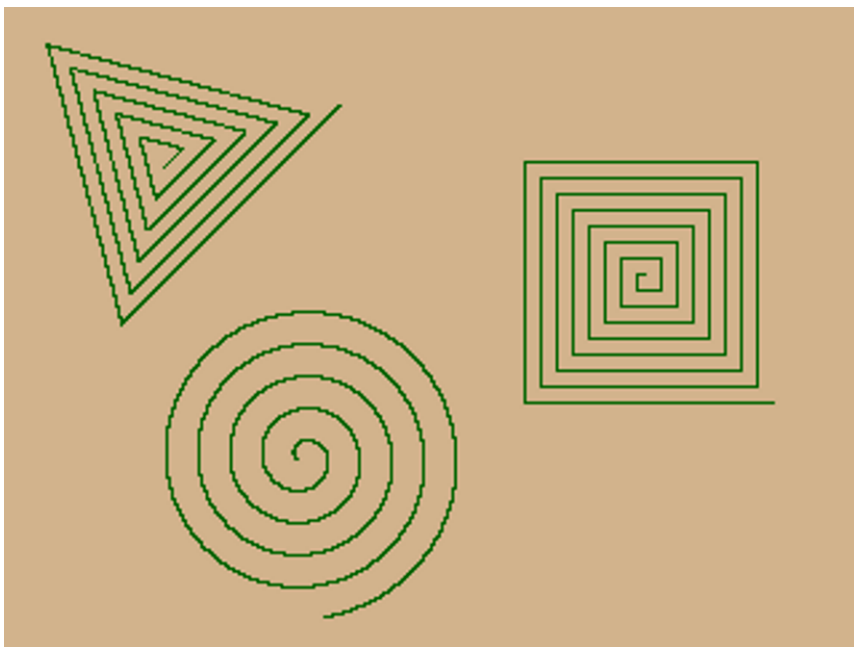


Fig. 2.15: Spin spiral, based on different polygons.

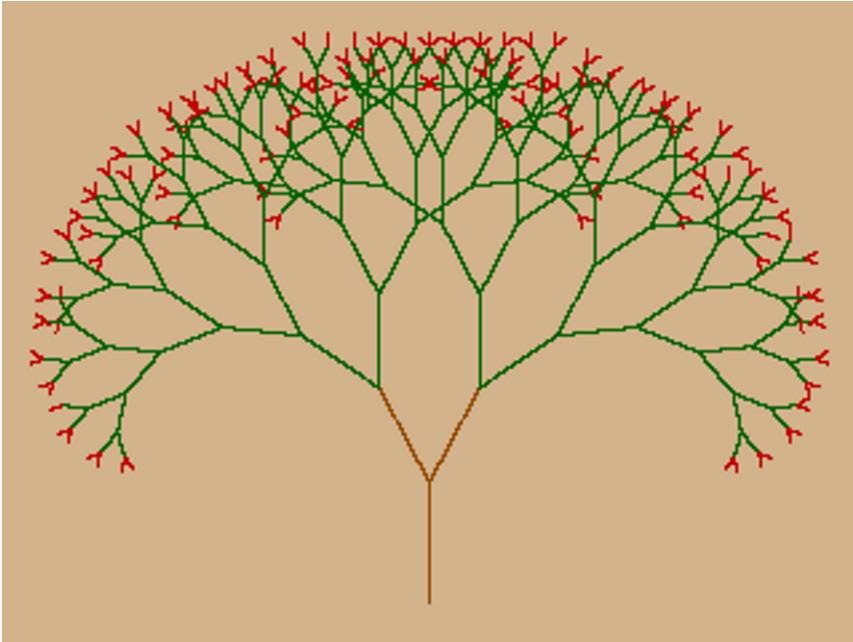


Fig. 2.16: Random tree, we might also use random angles at each branching.

Lab225: Random Tree

Figure 2.16 shows most, but not all, of a tree. The tree was drawn by 800 turtles, each beginning at the root and walking up the trunk, making 799 redundant lines just to start. Then, at random, half of the turtles branched left and half right.

This random branching process continued for nine total steps with size decreasing at each level. Some turtles walked the exact same total path as others, not contributing anything new to the overall drawing. This redundancy is more probable at all levels as time passes so we again have diminishing marginal returns, as shown in Figure 2.17. Depending on how much the size changes the details of these plots will vary but the overall characteristic remains the same.

Later in Chapter 5 we will see an alternative technique called *recursion* that can be used to draw the entire tree precisely with a single (!) turtle. Other recursive possibilities are shown in Figures 2.18 and 2.19.

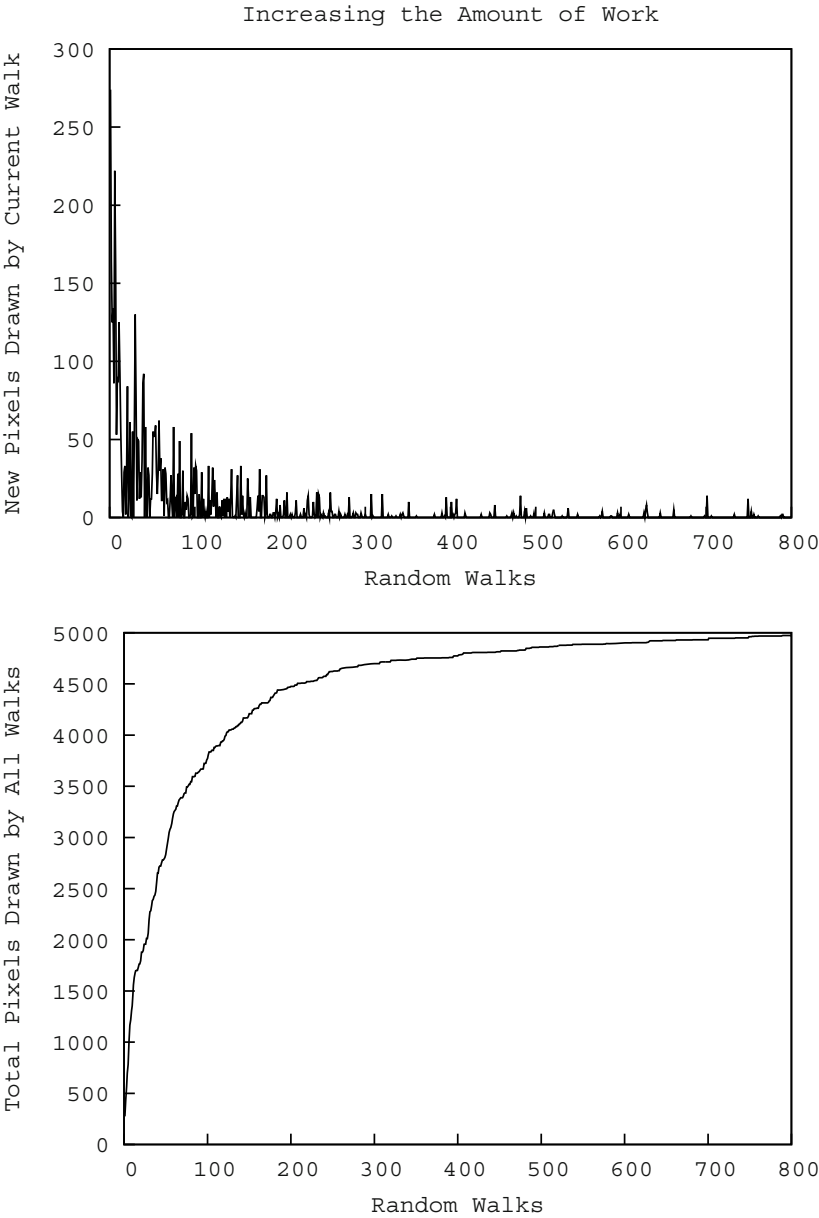


Fig. 2.17: Diminishing marginal returns. Top: new pixels drawn by current walk. Bottom: total pixels drawn by all walks, shown after each new walk.

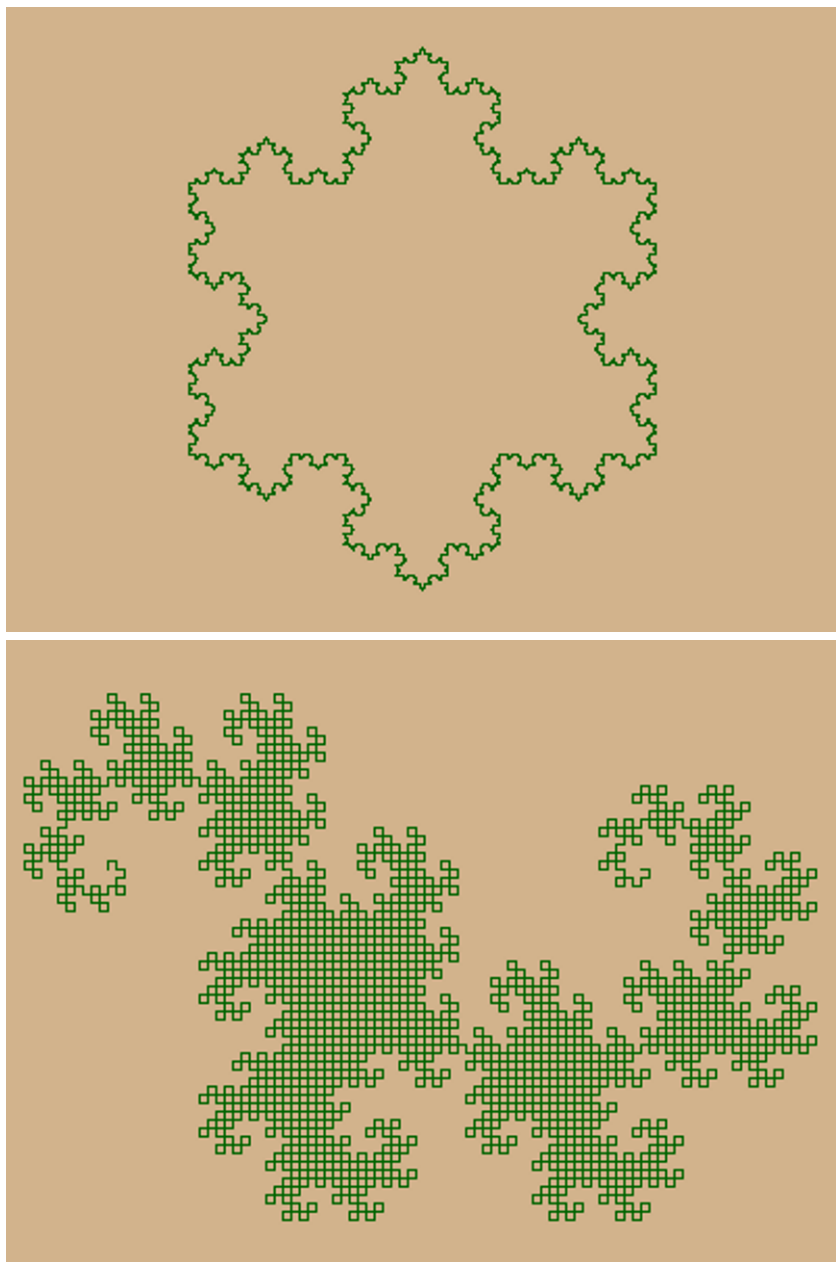


Fig. 2.18: Turtle recursion fractal gallery.

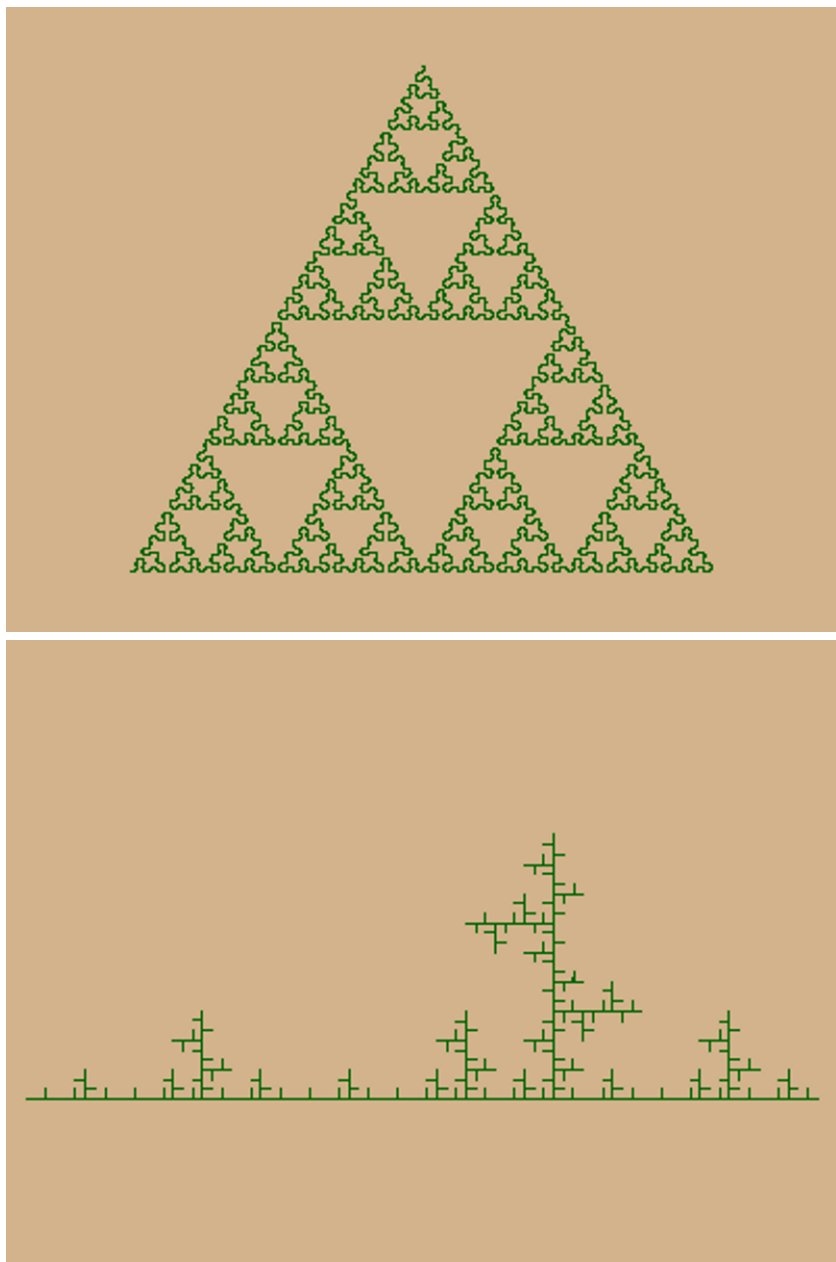


Fig. 2.19: Turtle recursion fractal gallery, continued.

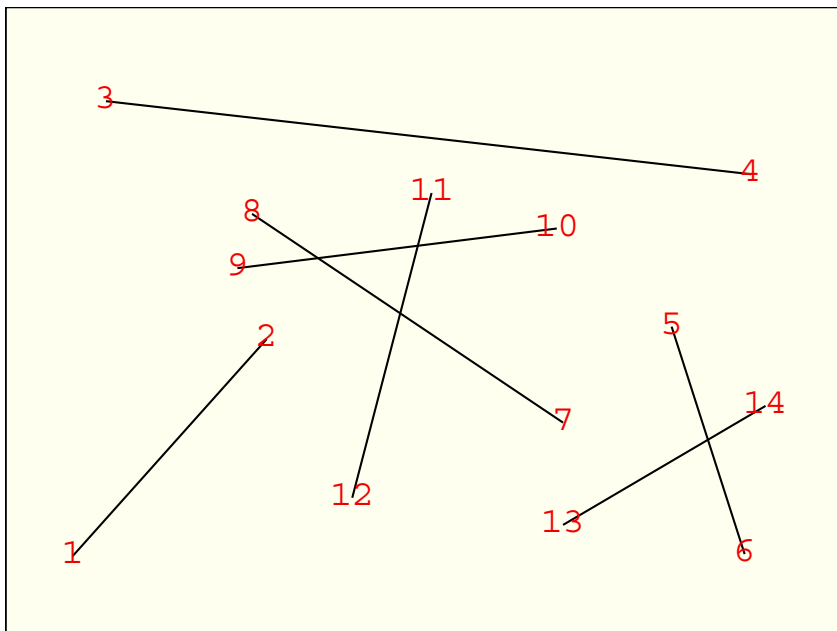


Fig. 2.20: Click count, where your own program need not show the counts.

2.3 Building Software

We return to interactive graphics with a goal to build drawing programs for non-commercial sketchwork. The final suggested version will include both tool and color selectors and is a starting point for a full piece of software. The user experience is foremost so we continually ask ourselves: “How would someone new to this react?” Feature-creep must be carefully guarded against as clutter and incoherence do not make a positive interface. Keep in mind what the application is supposed to be.

Writing “software” is not the same as writing “code” because code is written for only yourself to run. Software has to keep working even after you leave the room so we need to make assumptions about what a typical user would want, and also what they can actually do. Comparing vector graphics to pixel-by-pixel data, software is a representation at an even higher level where the user does not see *any* of the details. In the same way, as a coder you are like the user when importing a library you did not write, or a library that you wrote but just not recently.

Lab231: Click Count

Our first idea is to respond when the user clicks the mouse; every second click we draw a line connecting the two previous click locations. The click count numbers shown in [Figure 2.20](#) have been added only to show how the lines were drawn and they should not be present in your own version unless you really mean for them to be a part of someone's sketch.

Code Listing 2.7 is a shell where the event-object `evnt` knows the (x,y) location of each mouse click. Note again how we must specify that `count`, `x`, and `y` are defined at the global scope since function `click` contains an assignment statement for each of these variables. In this manner their values will persist between successive calls (i.e., between successive clicks). Line 30 assumes we want left-clicks rather than right-clicks, thus Button-1 rather than Button-3.

Code Listing 2.7: A shell of a program.

```

1 #####
2 #
3 # Chapter 2: Graphics
4 # Problem 3: Building Software
5 # Lab 2.3.1: Click Count
6 #
7 #####
8 from Tkinter import Tk,Canvas
9 #####
10 #
11 w,h=400,300
12 #
13 count = ...
14 #
15 def click(evnt):
16     global count,x,y
17     #
18     count += ...
19     #
20     if ...
21         x=evnt.x
22         y=evnt.y
23     else:
24         cvs.create_line(x,y, evnt.x, evnt.y, fill='black')
25 #
26 root=Tk()
27 cvs=Canvas(root,width=w,height=h,bg='#FFFFFF') # ivory
28 cvs.pack()
29 #
30 root.bind('<Button-1>',click)
31 root.mainloop()
32 #
33 # end of file
34 #
35 #####

```

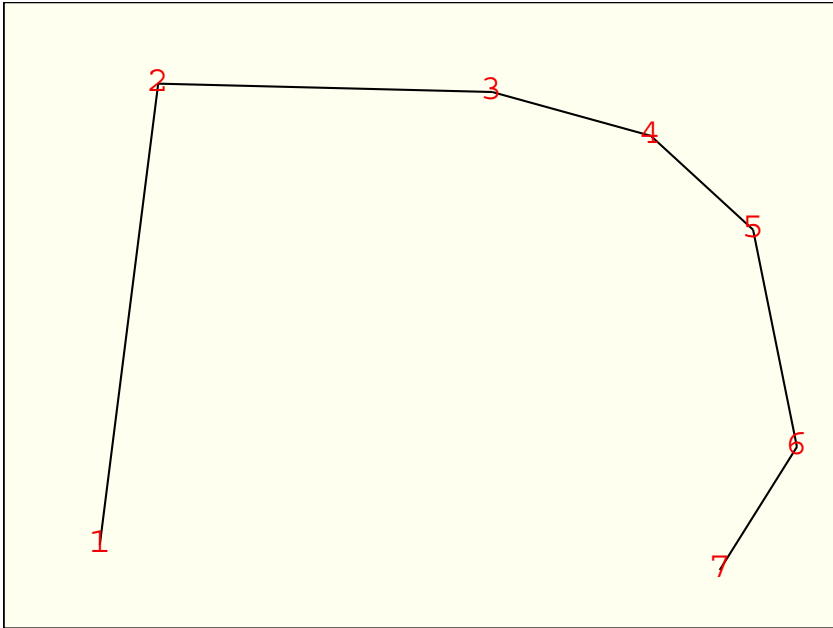


Fig. 2.21: Polyline, right-click ends each chain, left-click starts the next one.

Lab232: Polyline

Figure 2.21 shows a “polyline” where a right-click ends each chain. Key events are also shown in Code Listing 2.8 and function `exit` is from the `sys` module.

Code Listing 2.8: Mouse click events and quitting the program.

```
#
def click(evt):
    ...
#
def rightclick(evt):
    ...
#
def quit(evt):
    exit(0)
#
root.bind('<Button-1>', click)
root.bind('<Button-3>', rightclick)
root.bind('q', quit)
#
```

Lab233: Pencil Draw

Of course when we draw a pencil sketch we do not press the paper only at the endpoints of a line but at every single point. We can implement this feature using drag events as shown in Code Listing 2.9 and [Figure 2.22](#), a quick “review” session.

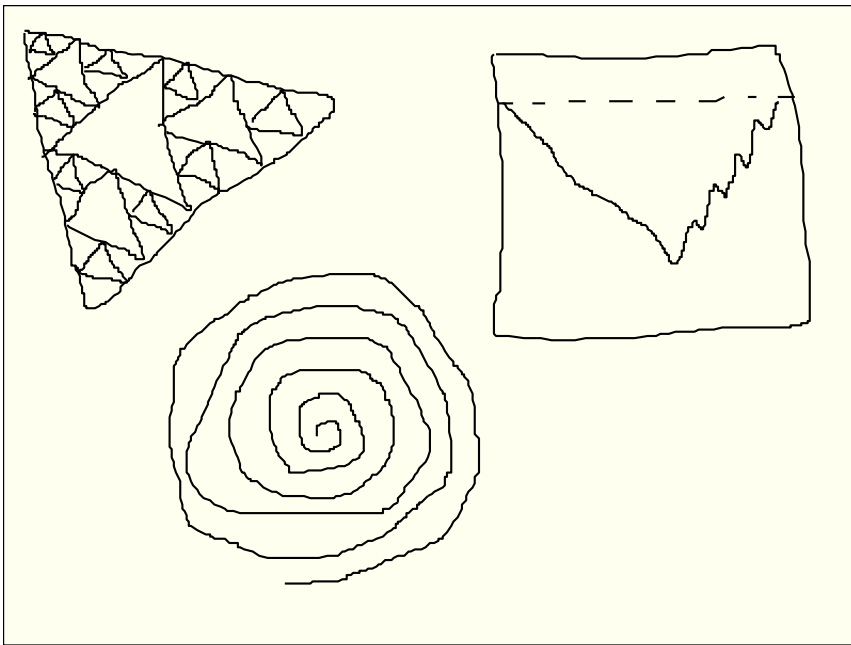


Fig. 2.22: A stroll down memory lane with highlights from our previous labs.

Code Listing 2.9: Mouse dragged event.

```
#
def click(evnt):
    ...
#
def drag(evnt):
    ...
#
root.bind('<Button-1>', click)
root.bind('<B1-Motion>', drag)
#
```

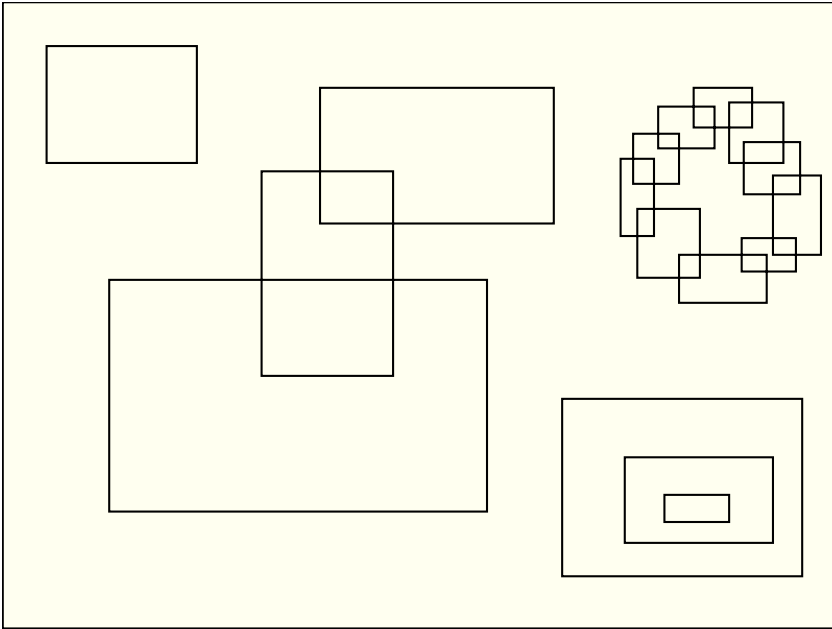


Fig. 2.23: A lot of rectangles, each is drawn so you see it grow as you drag.

Lab234: Rectangles!

The idea is that as the mouse is dragged you can see the rectangle changing size, updated dynamically. Each rectangle is fixed in place only as the button is released.

Code Listing 2.10: Rectangle commands.

```
#
tkid=cnvs.create_rectangle( ... ,fill='')
cnvs.coords(tkid, ... )
#
```

Code Listing 2.11: Mouse released event.

```
#
def release(evnt):
    ...
#
root.bind('<ButtonRelease-1>',release)
#
```

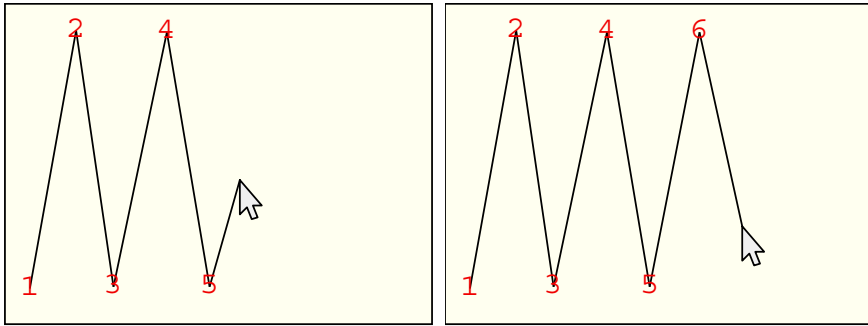


Fig. 2.24: Motion and more motion, toward a WYSIWYG system.

Motion and Double-Click

We might also like to see the growing line update as we move (not drag) in our previous line and polyline programs, before a click sets the second or next endpoint. The `coords` command works the same on lines as it does on rectangles, shown in [Figure 2.24](#) where a line is growing first from Point 5 to Point 6 and then, presumably, a future Point 7. Code Listing 2.12 shows how to respond to mouse motion events when no button is being pressed.

In addition, Code Listing 2.13 shows a double-click event we might use to connect the last endpoint of a polyline back to the first endpoint. In this case we must assume the location of our first click was remembered at the time it happened because it cannot be reconstructed later.

Code Listing 2.12: Mouse motion event.

```
#
def move(evnt):
    ...
#
root.bind('<Motion>', move)
#
```

Code Listing 2.13: Double-click event.

```
#
def doubleclick(evnt):
    ...
#
root.bind('<Double-Button-1>', doubleclick)
#
```

Lab235: Graffiti Tool

Figure 2.25 shows all our previous drawing tools as options, plus a color selector, and one last new feature: spray paint. (Disclaimer, vandalism is a crime.) This tool presents a number of coding challenges most notably that the spray paint should still work when the button is pressed even if the mouse is not moving, so drag events alone are not sufficient.

One solution is to use animation where a click sets some Boolean variable true, the subsequent release sets it false, and drag events update the (x,y) location. All the while our tick function is drawing random 1×1 rectangles somewhere in a circle centered at the current (x,y) and so long as the mouse button is still being pressed. In this context random points clustered near the center of a circle actually match the reality of a spray can.

The graffiti icon displayed among the tools will be different with each run of the program unless we set an explicit random number seed. (In the same way exact replication of simulation results may be obtained, an important requirement of any experiment.) You might also consider including a fill-the-area tool, a cut-and-paste option, and some facility for saving the current picture to an image file.



Fig. 2.25: Graffiti tool, one of many options our users enjoy.



<http://www.springer.com/978-1-4614-1887-0>

Applied Computer Science

Torbert, S.

2012, X, 202 p., Hardcover

ISBN: 978-1-4614-1887-0