PHY307, Science and Computers I

Lab #9, October 14, 2002
(due by end of lab THURSDAY)

Forging fractals, Fiddling with Frames

Summary of what you will do:

This lab will be conducted over two days – extra time, if any, this week can be used for consulting on the physics and programming for your project. There will be a brief lecture to start each day. You will be generating fractals and computing their dimension.

RECURSION

1. Get set up. Start a lab report entitled "MyLastNameLab09.doc". That way you can e-mail it as an attachment. You will also have graph paper plots that you will submit.

2. You will use two tools we haven't used before: the programming concept of *recursion*, which is very useful for constructing certain fractals, and the VPython object **frame**, which is useful for drawing or moving complex objects. You will first experiment with recursion, then use recursion to draw fractals, then combine frames and recursion to draw some more pretty pictures. Besides drawing pictures, you will keep track of the volume and area of the objects you construct, in order to compute their fractal dimension.

3. Recursion. In the simplest form of recursion, a function calls itself. The function solves a problem or carries out a procedure that is defined using, in part, itself. For example, suppose you want to sum all of the numbers from 1 to n. Let $S(n)$ be this sum. The value for $n=1$ is easy: $S(n)=1$. When $n>1$, $S(n-1)$ is the sum of the numbers from 1 to $n-1$, so that $S(n)=S(n-1)+n$.

   You are to translate this into a computer program in **python**. Write a function **sum_up_to(n)** defined using a **def** statement. In this function, you will use an **if** statement to determine what value to return: if the argument **n** is equal to 1, return the sum of the numbers up to 1, that is, 1. If the argument is greater than 1, **return n** plus the sum up to **n-1.** Use this function and check the output values for a couple of values of **n**. INCLUDE YOUR CODE AND SAMPLE OUTPUTS IN YOUR REPORT.

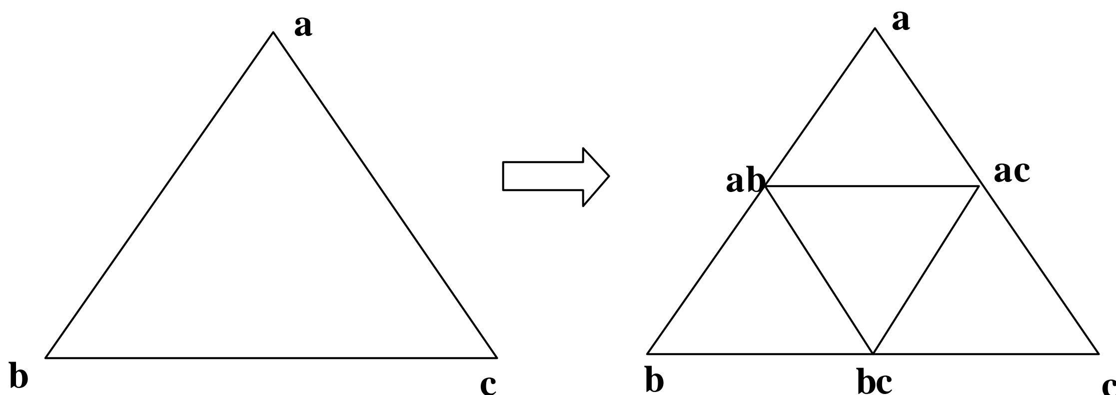
4. Recursion Example #2. A bit more complicated example of the magic of recursion is to make a function that list someone's ancestors, to a depth or distance of **m** generations. Make a program that does this by:
    a. Writing a function **list_ancestors(name, m)** using a **def** statement.
        1. In this definition, **print** the person's **name**.
        2. Note that you can paste strings together using the "+" operation. For example **"This is " + "a sentence."** has the value **"This is a sentence."**
        3. **if (m>0):** then apply the function **list_ancestors** to (**"the mother of " + name, m-1)** and also apply the function to **("the father of " + name, m-1)** to list the ancestors of **name**'s mother and father, to a depth of **m-1** generations. That is, the function will call itself, in two different ways, in the two lines following the **if** statement.
    b. Use this function. Apply it to find the ancestors back a distance of generations of someone named "Chris", for example.
    c. INCLUDE THE PROGRAM AND OUTPUT in your report.

RECURSION AND DRAWING

Now, let's use drawing and recursion to make objects with integer and non-integer dimensions. These objects will be made out of triangles. During the drawing of the objects, we will keep track of the number of triangles used, in order to compute the fractal dimension.

The idea is this: to draw a triangle, you divide it into subtriangles. If the subtriangle is small enough, just draw it. Otherwise, divide the subtriangles into smaller triangles.

Geometrically, this is how you can divided a triangle into subtriangles:



The triangle on the left has vertices **a,b,c.** The point **ab**, for example, is halfway between **a** and **b.** The points **bc** and **ac** are midpoints of the other two sides. The triangle on the left is composed of the four triangles (**a,ab,ac**), (**b,bc,ab**), (**c,bc,ac**), and **(ab,ac,bc)**.

5. So the following program will divide a triangle into subtriangles **n** times, to draw the original triangle:

```
from visual import *
def tri(a, b, c, level):
    if (level == 0):              ## smallest subtriangle
        convex(pos=[a,b,c])       ## draws an object that contains 3 points
        numtriangles = 1
    else:                         ## find midpoints, delegate to next level
        ab = (a+b)/2
        ac = (a+c)/2
        bc = (b+c)/2
        numtriangles = tri(a,ab,ac, level-1) + tri(b,ab,bc, level-1) \
                     + tri(c,ac,bc, level-1)+ tri(ab,ac,bc, level-1)
    return numtriangles           ## The function returns the total number
                                  ## of triangles drawn.

## use the function
n=5
m = tri(vector(-1,-.5,0), vector(0,1,0), vector(1,-.5,0), n)
print m,"triangles at level", n, "at length scale",(1/2.)**n
```

Run this program for values of **n** up to 7 and plot the relationship between length scale (size of box in Thursday's exercise) and the number of triangles.

6. Now try drawing a 'triangle' recursively, but <u>*do not draw* the central triangle</u> (**ab,ac,bc**). Excise this triangle from the **tri** function. What do you observe? INCLUDE A SNAPSHOT and DISCUSS THE APPEARANCE of what you draw. WHAT IS THE SLOPE OF THE PLOT of number of triangles vs. size of the triangle? That is, what is the fractal dimension of this object (commonly named the 'Sierpinski triangle'.)

7. Save your code to a NEW FILE **pyramid.py.**
    a. Modify the **tri** function to take 5 aruguments: **a, b, c, d,** and **level**.
    b. Under the **if** statement, draw a convex object that includes **a, b, c,** AND **d** (just add **d** to the list of points in the **pos** argument.)
    c. Under the else statement, compute new midpoints **ad, bd,** and **cd**.
    d. Then draw four "triangles" (actually tetrahedra): **(a,ab,ac,ad), (b, ab, bc, bd), (c, ac, bc, cd),** and (fill in the ??) **(d,??,??,??).** WHAT IS THE FRACTAL DIMENSION OF THIS PYRAMID? INCLUDE A SNAPSHOT FOR SOME LEVEL OF DRAWING.

<u>FRAMES</u>

In VPython, the **frame** object can be used to group objects into a single object. **frames** have **pos** and **axis** attributes that give their orientation to the frame in which they are defined. Rotations and translations of the frame move the associated objects as a rigid

object. This is programmed by (a) creating a frame using the **frame()** function and (b) assigning or attaching objects to the **frame**, for example, when they are created. What you learn about frames here will be used to study more natural looking fractals next week.

For example, the following creates a frame **f** and associates a sphere and a cube with that frame:

```
from visual import *
scene.autoscale = 0
f = frame()
cylinder(frame = f, radius=0.3, pos = (-2,0,0), axis=(4,0,0), color=color.yellow)
sphere(frame = f, pos=(-2,0,0), color=color.red)
sphere(frame = f, pos=(2,0,0), color=color.red)
```

Check that when you run this, nothing spectacular happens (**frames** are invisible.) But now add this loop to the above:

```
while 1:
    rate(40)
    f.rotate(angle=0.05, axis=(0,0,1))
    f.pos = f.pos + vector(0.01,0.01,0)
```

8. RECORD WHAT YOU SEE WHEN you run this program.
9. You can also chain frames together: a frame can belong to a frame. You can make lists of frames. Note this about lists: you can refer to a member of a list by giving its position in the list: the first item is at position [0], the second item is at position [1], the third item is at position [2], etc. Negative numbers count from the end: position [-1] gives the last item in the list, position [-2] is next to last, etc. Try out the following examples (you should do this and understand this, but you don't *need* to put it into your report):

```
alist = [5, 9, 43, 27,25]
print alist[0]
print alist[1]
print alist[4]
print alist[-1]
from visual import *
boxlist = [box(), box(pos=(1,1,1)), box(pos=(3,3,3))]
boxlist[1].rotate(angle = 2, axis=(0,0,1))  ## WHAT DOES THIS DO?
```

10. Now use **frames** and **lists** together. Make a program by
    a. **import**ing **visual.**
    b. turning off **scene.autoscale**
    c. starting a list named **framelist** with a single frame (default values).
    d. Does your program work so far? Check it. Note that frames are invisible and you are simply seeing that you have made no errors so far.
    e. start a **for** loop – step through the numbers from 0 to 15. In this loop:
        1. set **prev_frame** to be the last item of **framelist**. Run your program to make sure there is no error.
        2. create **new_frame**, a **frame** that is a subframe of **prev_frame** (that is, **frame = prev_frame** should be an argument to the **frame()** function), has an **axis** attribute of **(0.9,0.,0.1)**, and a **pos** attribute of **(0, 0.5, 0).** Run your program again.
        3. **append new_frame** to **framelist.** Run your program again.
        4. create a **box** that has **new_frame** as its **frame** and has a **height** of 0.2.
    f. RUN YOUR PROGRAM – TAKE A SNAPSHOT and EXPLAIN WHAT YOU SEE.
11. Dynamically modify frames. Add a **while 1:** loop at the end of your program that repeats 50 times a second and rotates the *first* **frame** in **framelist** by an **angle** of 0.02 about the *y*-axis (**axis=(0,1,0)**). Remember that you can rotate an object in VPython by using the function **rotate** which belongs to an object. If **b** is a **box,** you can rotate an angle of 0.1 about the *z*-axis using the statement **b.rotate(angle=0.1, axis=(0,0,1).** EXPLAIN WHAT YOU SEE WHEN ROTATING THE 1st FRAME AND REPEAT THIS FOR A ROTATION OF THE 5th FRAME.