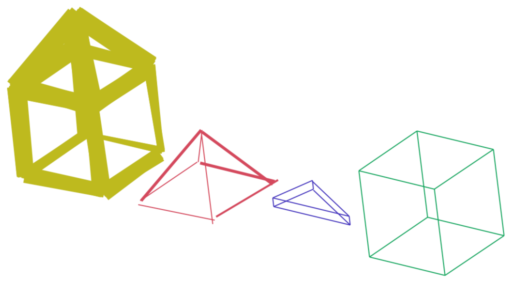
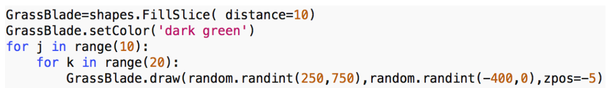
Zena's CS151 Project 11

For this project, we used the turtle to make drawings, but this time in 3D. The main goals of the project were to make L system-derived shapes in 3D and to use them to make a scene.

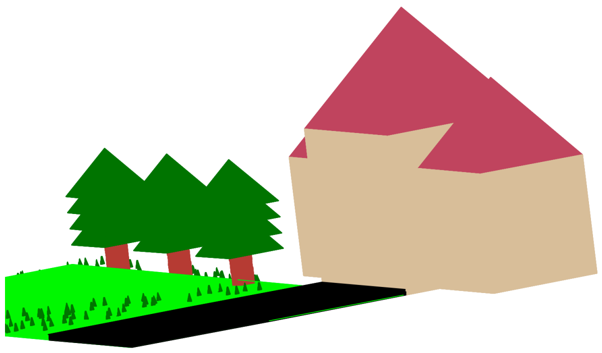
Task 1 was to make at least four 3D Turtle shapes and add them to our shapes library. In my case, I made a cube, a pyramid, a triangular prism, and a "house" shape. Making shapes in 3D is just like making 2D shapes, except now when the turtle turns, it is not only limited to "left" and "right". Rotation around the z axis (turning left or right like before) is called "yaw", "roll" is rotation around the x axis (leaning left or right), and "pitch" is rotation around the y axis (like turning to face up or down). We needed to add cases to the turtle interpreter, so "&" was given pitch up, "^" was given pitch down, "\" was given roll up, and "/" was given roll down. Keeping this in mind, I drew the shapes shown below. The characters used to represent a pyramid, for example, were [F+[(45)(45)&F]F[(45)(45)&F]F[(45)(45)&F]F(45)+(45)&F]. The turtle moves forward, turns left (90 degrees), remembers that position because it will continue from there to make the square base, then turns left 45 degrees, up 45 degrees (pitches up), and goes forward. Then it goes back to the remembered position, turns left 90 degrees, and continues on in the same manner until the pyramid is made. Note that if string needs to be repeated and you don't want to write it all out again, you can use the string multiplication property; writing 'F' * 4, for instance, makes the turtle go forward forward and right left times, drawing a square. The picture below also shows the jitter3 and jitter styles from last week's project.

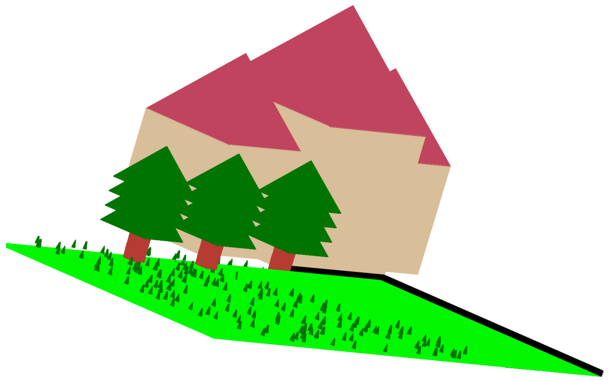


Task 2 was to make a scene using the shapes we made. I decided to fill the shapes I made in, which involved giving the turtle slightly different instructions. Since the fill function only works on flat objects, I had to treat each face of the shape as its own part to fill, meaning I couldn't take a shortcut to draw the shape, and had to have overlapping edges sometimes since the shapes shared them. At any rate, instead of drawing the main face(s) of a shape and then connecting that to the rest of the shape, I now had to do everything individually. Then I made the scene by arranging the shapes a certain way. The trees were stacked pyramids, for example, and their trunks were stacked blocks. The grass was just randomly scattered upright triangular prisms. The code below shows how first the shape is created, then the color is set, and then it is drawn at the specified location-- and we needed to input a z coordinate as well this time. There are 200 "grass blades" in total, since there are 20 drawn ten times. Every blade was randomly placed using random.randint with the desired range on the x and y axes input. Also worth noting is that I made the driveway by drawing a flat rectangle multiple times with the z coordinate being one more each time, giving the illusion of thickness.



```
GrassBlade=shapes.FillSlice( distance=10)
GrassBlade.setColor('dark green')
for j in range(10):
    for k in range(20):
        GrassBlade.draw(random.randint(250,750),random.randint(-400,0),zpos=-5)
```





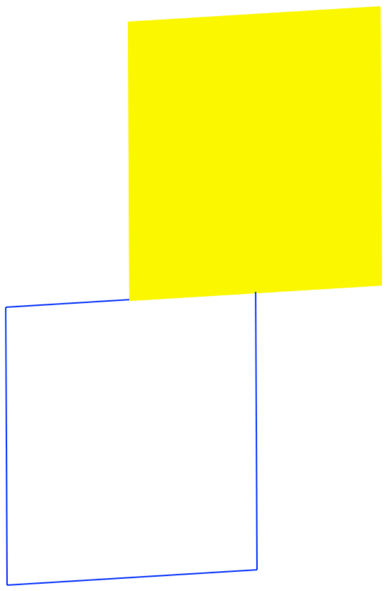
For Task 3, we needed to improve our code in some way, including doing one of the extensions, so I did extension 7, which was to make another "shape" that was actually a dynamic shape, drawing whatever the file it reads says, and, optionally, what the user inputs in the terminal. I didn't need to change the parent Shape class at all, I just needed to give this dynamic shape class instructions to read the file that was passed in in its init function, or to write the argv into another file (or possibly the same, depending on what is given) if it is given and read that instead. Then, once the parent init function is called, the only difference is that the string passed in is the string read from the file. This code is shown below:

```
if argv!=None:
    wr = file( outfile, 'w' )
    wr.write(argv)
    wr.close()
    rd=file(filename,'r')
elif argv==None:
    rd=file(outfile,'r')
lines=rd.readlines()
rd.close()
Shape.__init__(self,distance,90,color,''.join(str(lines)))
```

This "argv" is passed in through whatever line of code is calling the shape to be made; for example, the next pictures show what code I input in a file and into the command line to draw the picture below it. I wrote into the command line the two shapes that I wanted to be drawn. The place the yellow shape expects to get its text from is the first index of the command line, and the blue one expects it from index two. If the command line is not to be used, the argv should be given the value of None instead, so the file will just be read instead.

```
Yellow = shapes.DynamicShape('SquareText.txt',outfile='SquareText.txt',argv=argv[1],distance=200)
Yellow.setColor('yellow')
Yellow.draw( 0, 0, zpos=0)
Blue = shapes.DynamicShape('SquareText.txt',outfile='SquareText.txt',argv=argv[2],distance=200)
Blue.setColor('blue')
Blue.draw( 0, 0, zpos=0)
```

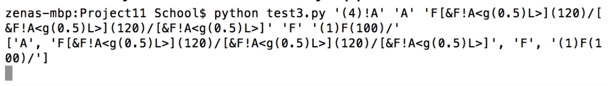
```
Zenas-MacBook-Pro:Project11 School$ python scene.py F-F-F-F- {F-F-F-F-}
```



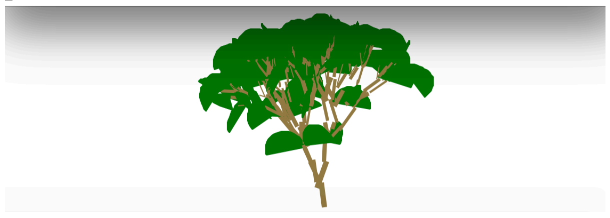
For my first extension, extension 5, I made a way for the user to type instructions for an L system into the command line and have that be drawn. This involved passing two arguments through the shapes class init, L system class init, readString init (which I made fresh), and the Turtle Interpreter's drawString function: base string (bstring) and rule string (rstring). First off, these two were necessary to differentiate between the base and rule(s) of the command line turtle instructions. The way I organized them was to have index 1 of argv be where the base is, then the ones after just follow the pattern of "base to replace" and "rule to replace it with", so I took advantage of that by using the following code when calling an L system to be made through the tree.py file. `ld = tree.Tree( 20, bstring=argv[1], rstring=argv[2:] )`
As you can see, it is organized into two categories: the base and all the rest, from which we will separate the bases to replace and the rules using the following code:

```
def readString(self,bstring,rstring):
        self.setBase(bstring)
        print rstring
        for index,rule in enumerate(rstring):
            if index%2==0:
                self.addRule(rstring[index:index+2])
```

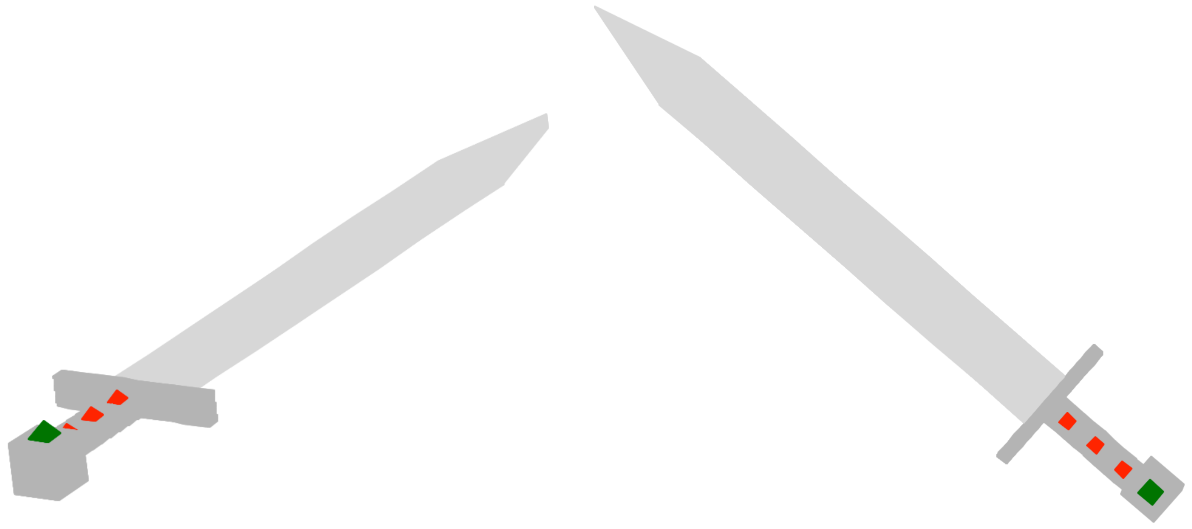
In the turtle interpreter, the same read string-reading method cannot be used, obviously, since the organization of the text on the command line is different from how it is in a file. To that end, I made a new way method to read string, in which the base is set to the bstring, and the rstring is separated into sections through a for loop. It loops over the rstring (and I used enumerate()) to force "index" to return the index of the rule. Only every other rule would be looked at, and in the same fashion as the file-reading method, the rule to be used later would be added as [the base to change] [space] [the rule], obtained by getting the index and contents of the current rule and the one after that. Then later, in the drawString method, if there are both a bstring and an rstring provided, it will use also use a slightly alternate way to process the text; I simply added the strings of the base and the rule together for the turtle interpreter to loop over. The image below displays the result of typing in the contents of the systemZ file (quotes need to be used around certain things in the command line to escape errors, such as '&' or "!"). As you can see, the tree is drawn exactly as it would normally appear.



```
zenas-mbp:Project11 School$ python test3.py '(4)!A' 'A' 'F[&F!A<g(0.5)L>](120)/[
&F!A<g(0.5)L>](120)/[&F!A<g(0.5)L>]' 'F' '(1)F(100)/'
['A', 'F[&F!A<g(0.5)L>](120)/[&F!A<g(0.5)L>](120)/[&F!A<g(0.5)L>]', 'F', '(1)F(1
00)/']
```



I also did Extension 4, which was to make another scene. It is not a "scene" per se, but I made a sword object by arranging objects from my shapes library in a certain way. The tip of the sword was the same 3D shape I used for my grass blades in the first scene; the blade was four sets of flat rectangles stacked on the z axis through a for loop to make it appear to have depth; the guard was also stacked rectangles; the hilt was eight stacked rectangular prisms; the pommel was a big cube. I also added some jewels along the hilt and pommel, made from my pyramid shape. Making it was a similar process to the first scene, but I had to be a lot more careful with the arrangement in this.



In this project, I learned a lot about how to think in three dimensions. I learned how to carefully plan turtle movements as well as placement of objects to give illusions of depth and create different-looking objects. I also learned a lot about how the command line works, how to handle text in it as well as in files, and how to formulate for loops to organize text from different formats.

**Worked alone