CS314
Lab 2
Winter 2015

Python

The main purpose of this lab is to allow you to explore the Python programming language. Python is a great language. It has a simple and clean syntax and lots of built-in features. One of Python's traits is that it is easy to write, and easy to remember how to write. It's the big comfy chair of programming languages. This characteristic leads us to this lab.

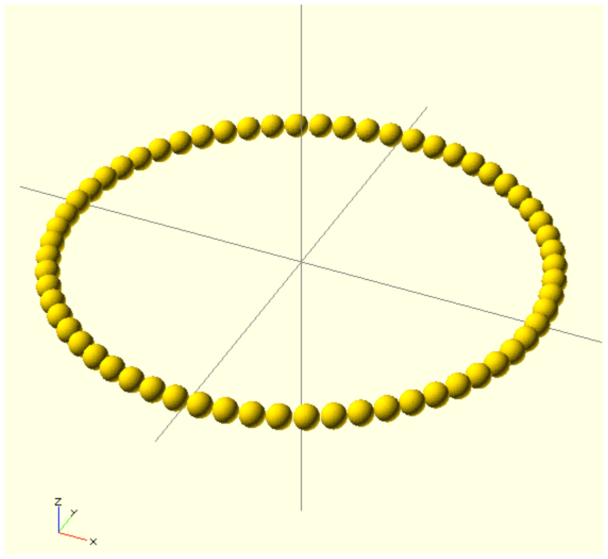


Figure 1: Pearl Necklace

We've just learned OpenSCAD. Nothing against OpenSCAD, but in all honesty, unless I'm going to do a bunch of work in it I'd rather not have to learn all its details. It would just be YAPLS[1] I have to remember. Couldn't I use a language that I already know and love instead? Sure. Let's take the basics of OpenSCAD and make them work from another language. And of course I have a perfect one in mind.

Let's take only the basics of OpenSCAD: Sphere, Cylinder and Cube, and union, difference and intersection. With just those primitives and operations we can make a lot of things. We'll write Python that has the capacity to *generate* OpenSCAD code for these things. Then we can write normal Python to create these objects, using OpenSCAD language and compiler as intermediate steps.

Your first goal is to write Python code, that generates the OpenSCAD source code for the two objects shown in Fig. 1 and Fig. 2. The first is a simple necklace of spheres, which we'll call the pearl necklace. The edges of the spheres just touch each other. They are laid out along a circle. Your code should calculate how many spheres there will be based on the diameter of the necklace and the size of each pearl. The next figure is what we'll call a crystal. It is a bunch of spheres arranged in a cubic close packing structure. This is similar to the face-centered cubic structure found in our favorite jewelry: gold, silver and platinum. With four spheres placed together on a plane, the next sphere up lies in the valley between the four below. Here we'll allow the spheres to meld together by some percentage.

To aid you in these tasks I've begun the Python implementation. Check out the `pyopenscad.py` module. I'll briefly take you through this. First we begin with wrapping the abstraction of a sphere within a class. Python has Object Oriented Programming features you may use if you like. This class has only a constructor and an ability to represent itself as a string. This is how we'll use it: create the object and then ask for it as a string. The string will be the valid OpenSCAD code for itself.

[1]'yet-another-programming-language-syntax'

Next up is the difference operation. This will be a function, written at module scope. Notice it is **not** *within* the `Sphere` class. It will give you the code, as a string, for the difference of a list of objects.

Lastly we need a way to output the entire OpenSCAD program to a file. This is where `save` comes in, another function at module scope.

Our module then becomes a library we can use to create OpenSCAD programs.

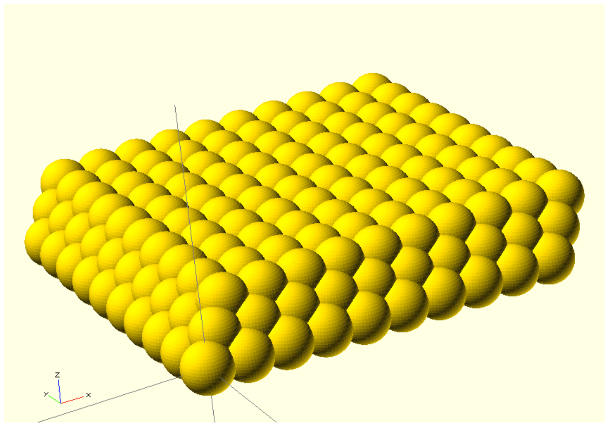


Figure 2: Close packed cubic crystal

Part I

Implement the `union` operation in the `pyopenscad`
module and then write two separate Python programs (`necklace.py` and `crystal.py`) to generate objects similar to those in Fig. 1 & 2. Each of these programs should run from the command line as in:

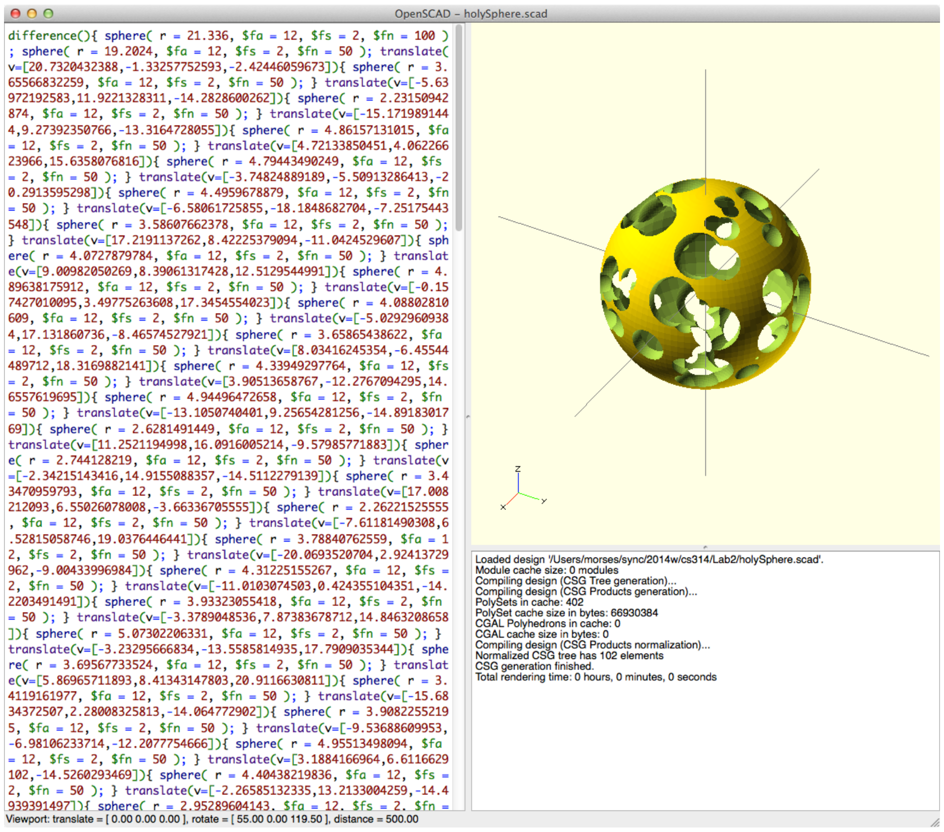
```
python necklace.py
```

and generate the appropriate OpenSCAD file that can be opened manually in the OpenSCAD application.

Part II

Now that you've got things down, let's give more power to our library. Implement a `Cylinder` class and a `Cube` class. Then implement `intersection`, `scale` and `rotate` operations.

Next, write a separate Python program to use these operations in a way that makes use of a number of Python features. In particular, let's make lots of spheres, or cylinders or cubes and combine them in some interesting way. Figure 3 shows an admittedly simple example that uses only the code we started with: the Holy Sphere!



```
difference(){ sphere( r = 21.336, $fa = 12, $fs = 2, $fn = 100 )
; sphere( r = 19.2024, $fa = 12, $fs = 2, $fn = 50 ); translate(
v=[20.7320432388,-1.33257752593,-2.42446059673]){ sphere( r = 3.
65566832259, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[-5.63
972192583,11.9221328311,-14.2828600262]){ sphere( r = 2.23150942
874, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[-15.171989144
4,9.27392350766,-13.3164728055]){ sphere( r = 4.86157131015, $fa
= 12, $fs = 2, $fn = 50 ); } translate(v=[4.72133850451,4.062266
23966,15.6358076816]){ sphere( r = 4.79443490249, $fa = 12, $fs
= 2, $fn = 50 ); } translate(v=[-3.74824889189,-5.50913286413,-2
0.2913595298]){ sphere( r = 4.4959678879, $fa = 12, $fs = 2, $fn
= 50 ); } translate(v=[-6.58061725855,-18.1848682704,-7.25175443
548]){ sphere( r = 3.58607662378, $fa = 12, $fs = 2, $fn = 50 );
} translate(v=[17.2191137262,8.42225379094,-11.0424529607]){ sph
ere( r = 4.0727879784, $fa = 12, $fs = 2, $fn = 50 ); } translat
e(v=[9.00982050269,8.39061317428,12.5129544991]){ sphere( r = 4.
89638175912, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[-0.15
7427010095,3.49775263608,17.3454554023]){ sphere( r = 4.08802810
609, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[-5.0292960938
4,17.131860736,-8.46574527921]){ sphere( r = 3.65865438622, $fa
= 12, $fs = 2, $fn = 50 ); } translate(v=[8.03416245354,-6.45544
489712,18.3169882141]){ sphere( r = 4.33949297764, $fa = 12, $fs
= 2, $fn = 50 ); } translate(v=[3.90513658767,-12.2767094295,14.
6557619695]){ sphere( r = 4.94496472658, $fa = 12, $fs = 2, $fn
= 50 ); } translate(v=[-13.1050740401,9.25654281256,-14.89183017
69]){ sphere( r = 2.6281491449, $fa = 12, $fs = 2, $fn = 50 ); }
translate(v=[11.2521194998,16.0916005214,-9.57985771883]){ spher
e( r = 2.744128219, $fa = 12, $fs = 2, $fn = 50 ); } translate(v
=[-2.34215143416,14.9155088357,-14.5112279139]){ sphere( r = 3.4
3470959793, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[17.008
212093,6.55026078008,-3.66336705555]){ sphere( r = 2.26221525555
, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[-7.61181490308,6
.52815058746,19.0376446441]){ sphere( r = 3.78840762559, $fa = 1
2, $fs = 2, $fn = 50 ); } translate(v=[-20.0693520704,2.92413729
962,-9.00433996984]){ sphere( r = 4.31225155267, $fa = 12, $fs =
2, $fn = 50 ); } translate(v=[-11.0103074503,0.424355104351,-14.
2203491491]){ sphere( r = 3.93323055418, $fa = 12, $fs = 2, $fn
= 50 ); } translate(v=[-3.3789048536,7.87383678712,14.8463208658
]){ sphere( r = 5.07302206331, $fa = 12, $fs = 2, $fn = 50 ); }
translate(v=[-3.23295666834,-13.5585814935,17.7909035344]){ sphe
re( r = 3.69567733524, $fa = 12, $fs = 2, $fn = 50 ); } translat
e(v=[5.86965711893,8.41343147803,20.9116630811]){ sphere( r = 3.
4119161977, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[-15.68
34372507,2.28008325813,-14.064772902]){ sphere( r = 3.9082255219
5, $fa = 12, $fs = 2, $fn = 50 ); } translate(v=[-9.53688609953,
-6.98106233714,-12.2077754666]){ sphere( r = 4.95513498094, $fa
= 12, $fs = 2, $fn = 50 ); } translate(v=[3.1884166964,6.6116629
102,-14.5260293469]){ sphere( r = 4.40438219836, $fa = 12, $fs =
2, $fn = 50 ); } translate(v=[-2.26585132335,13.2133004259,-14.4
939391497]){ sphere( r = 2.95289604143, $fa = 12, $fs = 2, $fn =
```

```
Loaded design '/Users/morses/sync/2014w/cs314/Lab2/holySphere.scad'.
Module cache size: 0 modules
Compiling design (CSG Tree generation)...
Compiling design (CSG Products generation)...
PolySets in cache: 402
PolySet cache size in bytes: 66930384
CGAL Polyhedrons in cache: 0
CGAL cache size in bytes: 0
Compiling design (CSG Products normalization)...
Normalized CSG tree has 102 elements
CSG generation finished.
Total rendering time: 0 hours, 0 minutes, 0 seconds
```

```
Viewport: translate = [ 0.00 0.00 0.00 ], rotate = [ 55.00 0.00 119.50 ], distance = 500.00
```

Figure 3: Make yours much better than this.

Here is the code that made it:

```python
from pyopenscad import *
from math import *
import random

def mag(x,y,z):
    return sqrt(x**2 + y**2 + z**2)

def normalizeToSphere(radius,x,y,z):
    # normalize to surface of sphere with radius. Return in spherical coordinates
    vlen = mag(x,y,z)
    x = (radius/vlen) * x;
    y = (radius/vlen) * y;
    z = (radius/vlen) * z;
    # convert to spherical coordinates (r,theta,phi)
    r = radius
    theta = atan2(y,x)
    phi = acos(z/r)
    return (r,theta,phi)
```

```
def getPointsOnSphere(radius = 1.0,count=20):
    '''
        Get random points on the surface of a sphere of arbitrary radius.
        Works by generating many points placed randomly in a unit cube and then
        choosing only those that lie in the unit sphere.  Then normalize them
        out to the surface of an arbitrary radius sphere.  Should be somewhat
        randomly distributed.
    '''
    n = 5000
    xs = [random.uniform(-1.0,1.0) for i in range(n)]
    ys = [random.uniform(-1.0,1.0) for i in range(n)]
    zs = [random.uniform(-1.0,1.0) for i in range(n)]
    pts = [normalizeToSphere(radius,x,y,z) for (x,y,z) in zip(xs,ys,zs) if mag(x,y,z) <= 1.0]
    return pts[0:count]

def main():
    ballRadius = 1.680*25.4/2       # regulation golf ball size
    holeRadius = 4

    # we'll difference this list
    objList = [Sphere(ballRadius,fn=100),Sphere(0.9*ballRadius,fn=50)]

    # generate a bunch of holes of random size, placed randomly near the
    # surface of the ball
    pts = getPointsOnSphere(0.95*ballRadius,100)
    for (r,theta,phi) in pts:
        bubbleRadius = random.uniform(0.5*holeRadius,1.3*holeRadius)
        hole = Sphere(bubbleRadius,fn=50)
        radius = random.uniform(0.8*r,1.15*r)
        x = radius * cos(theta) * sin(phi)
        y = radius * sin(theta) * sin(phi)
        z = radius * cos(phi)
        objList.append( translate(hole,x,y,z) )

    obj = difference(objList)
    save('holySphere.scad',obj)


if __name__ == '__main__':
    main()
```

Think up something you can make, using the power that is Python. Make hundreds of cubes, spheres or cylinders. Place them where you want with Python. Use the built-in Python libraries, such as random. Make something fun!

While you're doing it, please use the following Python features:

- lists, list slicing and list comprehension
- tuples
- functions with default parameters, that return lists, tuples or dictionaries; and calling functions with pass-by-name (keyword arguments) arguments
- for loop
- if-elif-else statements
- objects and operations(i.e. your Cylinder and Cube, intersection, union, scale and/or rotate)

Turning it in

Please print out your code as well as images of your objects. Turn these in on paper during class. Also, turn in your code electronically in one zip file. Details on where to submit it electronically will be given in class.