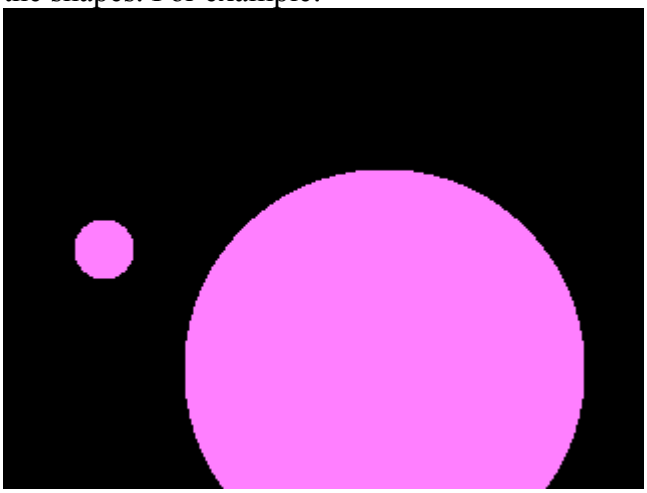
COSC 2P91 – Assignment 4 (Mini-Project 2) – Part Two

(Note: This is also a placeholder until the entire project is up)
Once you've gotten the first part done, and are comfortable with both writing image files and juggling tasks between both languages, the next step will be to establish the first basics of ray tracing.
Though we won't need to know the details about how graphics are rendered, we'll need to have a basic understanding of the math involved for things like shapes – e.g. rays, spheres, etc.

The task for this part is simple: to be able to load in an arbitrary scene file, and generate a silhouette of the shapes. For example:



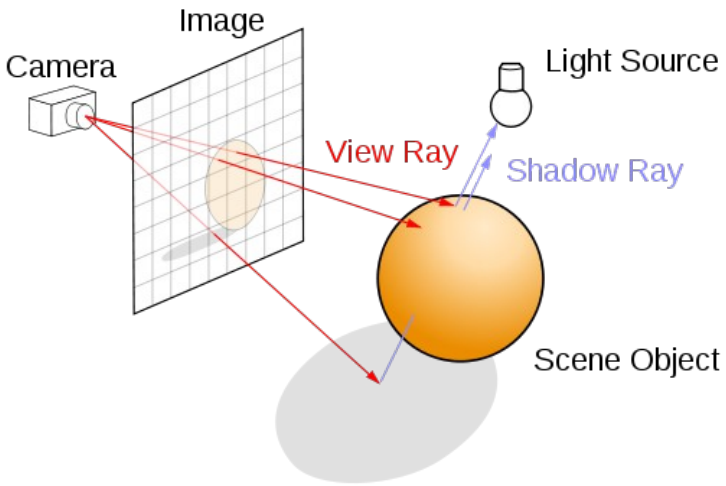
The requirements are simple:

- You are free to define your own *scene file* specification, but you'll have to be able to document it in the final part, so you may wish to go with something relatively simple
  - For example, perhaps the number of shapes, followed by the coordinates/size of each shape per line
  - You may need to augment your specification slightly for the third part, but it will be simple
- Your program must be able to load an arbitrary scene file
  - I'd suggest simply asking for the filename
- You may assume that only spheres exist for now
  - For space where no sphere exists, pick a default background colour (black makes sense)
- You can pick whatever colour you like for the spheres (white would be logical)
- A scene must be able to contain any (reasonable) number of spheres
  - Of course, since pixels only have two possible states, you won't necessarily *see* them all
- You're welcome to assume that the size of the image also defines the dimensions of the orthogonal projection (see below)
  - e.g. for a 320×240 image, you can assume coordinates of (0,0) to (319,239) or (-160,-120) to (159,119)
  - You don't have to; you simply can. Again, you'll be defining your own data file anyway
- This time, you do need to use threading
  - Though it's up to you whether you do threading in Python or C, my suggestion is still pthreads in C
  - You may assume a maximum of 8 threads, if you like, but still let the user choose

Rendering an image

Most ray tracing uses a conical projection; we'll just be using orthogonal. Both terms are simpler than they sound.

- A conical projection simply means that your field of vision expands as it goes outwards. That is, if you can see up to a width of 10 inches, from 1 inch away, then you might be able to see a width of a mile when viewing greater distances. In other words, it's what gives the notion of perspective, and that objects will appear smaller the farther away they are. It's relatively easy to do, but since this isn't a graphics course, we'll be going with something even easier.
- An orthogonal projection simply deals with what you'd see if you had massive retinas, and for each rod/cone on the retina, could only see what was directly in front. Of course, this means that a one-inch ball five feet away will still look one-inch wide from a distance of 37 miles.
    - Effectively, it takes a 3D world and flattens it into a plane.



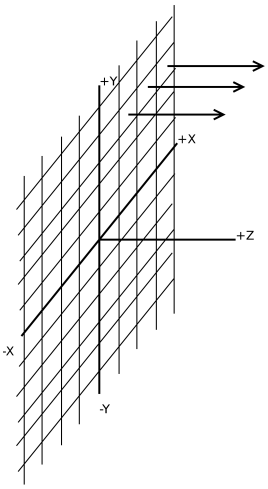
*Normal ray tracing, based on conical projection – i.e. what we **aren't** doing*
Source: http://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg

If we were doing a standard conical projection, ray tracing would be pretty easy:

- Arrange the pixels of an image into a virtual rectangle in space. For each pixel, fire out a *ray* from the same central point (representing the *camera* or *eye*), through that pixel, establishing a ray with both an origin and a unit vector for direction
- Calculate the intersections of that ray with each object in the scene. Whichever has the closest point of intersection must be the image the *eye* would "see"
- There's some extra steps, like calculating lighting, reflection, etc. later on, but we aren't there yet
    - If this affects your approach, we can eventually add extra features based on firing additional rays from that intersection point – e.g. if a sphere is mirrored, a ray bouncing off it would let you see another sphere in a different location. Or, if it was glass, you could see through it. (Or both)

However, as stated earlier, we're using orthogonal projection. That's even easier:

- Assume the image (pixels) to be arranged about some central point, aligned with the X and Y axes. Simply fire each ray into the positive Z axis
    - This means each ray starts with a directional vector of (0,0,1)
- Because you're only aiming for a silhouette, if the vector intersects a sphere in the scene, set the pixel's colour to one colour (e.g. white); if no intersection occurs at all, pick the other (e.g. black)



Example of orthogonal projection

Intersection with spheres
The only thing left to explain is the actual intersection itself:

- A ray has the formula of $\overline{X}=\overline{X}_0+t\overline{D}$ where *t* is a scalar for distance, and approaches infinity
- A sphere is defined according to the coordinates of its centre, and its radius
- The point of intersection is defined by the *t* such that a point in the ray has the same position as a point on the surface of the sphere
    - Of course, unless you're just grazing the very edge of the sphere, the ray will actually intersect a sphere *twice*: once to enter it, and once to exit it

There are numerous well-explained references for finding the intersection of a ray and a sphere (both related to ray tracing, and simply related to mathematics), but I'm partial to this one:
http://www.ccs.neu.edu/home/fell/CSU540/programs/RayTracingFormulas.htm

Just remember that our orthogonal projection will simply the first step – dx, dy, and dz will just be defined as 0, 0, and 1, respectively.

There is one small point not covered on that page: There are two solutions for *t*: one based on subtracting the square root of the discriminant (shown on the page), and one based on adding it. The only time this would matter would be for a ray originating from within the sphere (e.g. if the sphere was surrounding the eye). This isn't a concern for this part, but be aware of it, since it may matter for the next part.

Task
Implement the silhouette feature described above. Add it as a user-selectable option to your program from the first part (so now there will be two options).
Your only user-defined options are the dimensions of the image, the filename for the scene, the number of threads to use, and the filename for the output image.