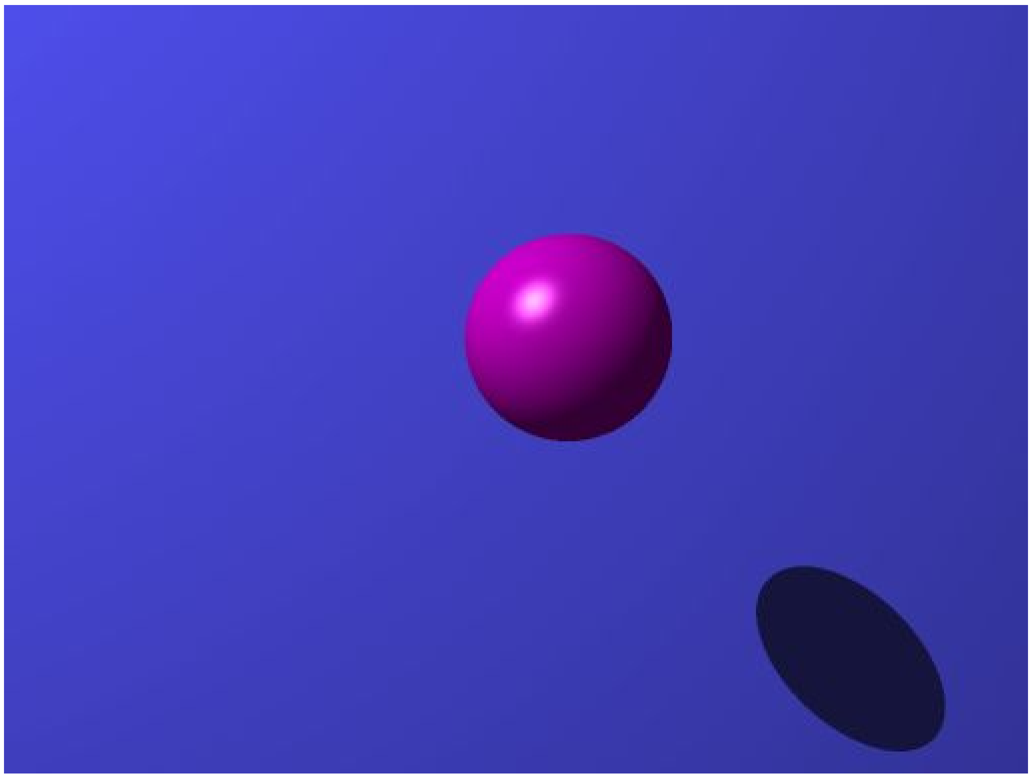
CPE101 WInter 2019 Optional Project 6
Ray Tracing

Due: 3/21 Midnight

Will be considered into the total course grade

For this first assignment you will define data representations (in the form of classes) that will be used for the remainder of the course project. In addition, you will provide __init__ functions that properly initialize the attributes of object conforming to these representations and you will implement test cases to verify the behavior of these functions.

These data representations will eventually be used in a program to create images like the following.



Part 1

You will develop your program solution over two files. You must use the specified names for your files.

- data.py - contains your class definitions with __init__ functions
- tests.py - contains your test cases

Once you are ready to do so, and you may choose to do so often while incrementally developing your solution, run your program with the command **python tests.py**.

Data Definitions

The specifications for data representations will be placed in data.py. This file will be imported by many different files as the term project progresses.

For each of the following add, to data.py, a class declaration with the specified name and the required __init__ function.

Point

Define a Point class that represents a point in three-dimensional space. Each Point object will have x, y, and z attributes. Define the __init__ function in Point to take arguments x, y, and z(in that order) and to initialize the attributes of the instance.

Vector

Definition
A vector represents a direction and a magnitude. For example, one might consider a force vector in a physics problem as a representation of the direction in which a force is being applied and the magnitude of that force.
In three-dimensional space a vector is represented as x, y, and z components. The direction of the vector is determined by the direction that must be taken from the origin to the point defined by these three components. The magnitude can also be computed based on these components.
As an example, the vector <1.0, 0.0, 0.0> is a vector with a magnitude of 1.0 pointing in the positive x direction.
You will study vectors and the associated operations in greater detail in later courses including physics, linear algebra, and computer graphics.
Data Representation
Define a Vector class. Each Vector object will have x, y, and z attributes initialized by the __init__ function in Vector (__init__ must take the arguments in order of x, y, and z).

Ray

Definition
The algorithm that will be used (in a later assignment) to generate the images shown earlier relies on "casting" rays from a virtual eye into a "scene" (which models the positions of spheres and a light). Each ray will begin at a specific point in three-dimensional space and will "move" in a single direction.
Data Representation
Define a Ray class. Each Ray object will have pt (expected to be a Point object) and dir (expected to be a Vector objet) attributes initialized by the __init__ function in Ray (__init__ must take the arguments in order of pt and dir).

Sphere

The ray casting program that you will implement over the course of this quarter will work with spheres as the only visible entities in the scene.
Define a Sphere class. Each Sphere object will have center (expected to be a Point object) and radius (expected to be a float) attributes initialized by the __init__ function in Sphere(__init__ must take the arguments in order of center and radius).

Test Cases

Many people tend to focus on writing code as the singular activity of a programmer, but testing is one of the most important tasks that one can perform while programming. Proper testing provides a degree of

confidence in your solution. During testing you will likely discover and then fix bugs (i.e., debug). Writing high quality test cases can greatly simplify the tasks of both finding and fixing bugs and, as such, will actually save you time during development.
In tests.py, write test cases for each of the above __init__ functions. You should place each test case in a separate, appropriately named testing function. Recall that the testing framework (unittest) requires that each testing function begin with test (in some form).
The following code snippet will get you started; you may also refer to the test cases from lab. You will want to use self.assertAlmostEqual to check for the expected value in the case of a floating point value. In addition, if an attribute is expected to be an object, then you will need to verify that it was properly initialized by checking the attributes of that object (i.e., verify that the attribute has the expected attributes; e.g., if r is a Ray, then you will want to check r.pt.x, etc.). We will simplify this sort of chained testing in a later assignment.

```
class TestData(unittest.TestCase):
   def test_point_1(self):
      pass  # create a point and verify that each attribute was properly initialized
```

Add the following code to the end of the tests.py. This will allow the test cases to be run from the command line.

```
if __name__ == "__main__":
     unittest.main()
```

Your test cases *must* use the unittest module used in lab and discussed in lecture. What are you testing exactly? For a given class, create an instance of the class and then verify that the attributes were properly initialized. At this stage of the project, your test cases are primarily used to verify that there are no typos or copy-and-paste coding errors in your __init__ functions.

Part 2

For this assignment you will define a number of utility functions on points and vectors. These functions will be used in multiple parts of the project, so you will define and test them now. You can think of this as defining your own library of functions. You will also define functions for checking the equality of two objects; these functions will simplify your test cases.

Files

Create a *hw2* directory in which to develop your solution. You will need to copy your files from the previous assignment to this new directory.
You will develop the new parts of your program solution over two files and you will make changes to data.py from the previous assignment. You must use the specified names for your files.

- vector_math.py - contains the function implementations
- tests.py - contains your test cases

Once you are ready to do so, and you may choose to do so often while incrementally developing your solution, run your program with the command **python tests.py**.

Object Equality Functions

Download utility.py and place the file in your *hw2* directory. You will use the epsilon_equal function defined in utility.py in your solution to this part.
Since you are encouraged to write test cases early, let's begin with some functions that will make writing these test cases less tedious. Modify data.py to add an __eq__ function to each class (i.e., Point.__eq__, Vector.__eq__, Ray.__eq__, and Sphere.__eq__). Each of these functions must be written to compare each of (and all of) the corresponding attributes of the self and otherparameters, returning True when all attributes match and False when they do not.
You can review the first part of Lab 3 for additional details.
To summarize, you will implement each of the following equality checking functons.
Point.__eq__(self, other)
Vector.__eq__(self, other)
Ray.__eq__(self, other)
Sphere.__eq__(self, other)

Vector Math Functions

You are to implement the following functions in vector_math.py. These provide basic operations on points and vectors (two of the data types defined in the previous assignment). Details for each function are given in the descriptions that follow. If interested, you will find additional discussion of vectors at http://en.wikipedia.org/wiki/Euclidean_vector.
scale_vector(vector, scalar)
dot_vector(vector1, vector2)
length_vector(vector)
normalize_vector(vector)
difference_point(point1, point2)
difference_vector(vector1, vector2)
translate_point(point, vector)
vector_from_to(from_point, to_point)

Scale

scale_vector(vector, scalar)
This function creates (and returns) a new vector with components equal to the original vector scaled (i.e., multiplied) by the scalar argument.
For example, vector <1, 2, 3> scaled by 1.5 will result in vector <1.5, 3, 4.5>.

Dot Product

dot_vector(vector1, vector2)
This function performs a type of multiplication (product) on vectors. A visualization (via a Java applet) of the dot product can be found here.
The dot product of two vectors is computed as follows.
<x1, y1, z1> * <x2, y2, z2> = x1 * x2 + y1 * y2 + z1 * z2.
As an aside, the dot product is quite useful in calculations between vectors. Specifically, the following is another expression of the dot product.

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \; \|\mathbf{b}\| \cos\theta$$

This formula relates the dot product to the angle between two vectors and the magnitude of the vectors. We will use this relationship in the next assignment.

Length

length_vector(vector)
The length of a vector (i.e., its magnitude) is computed from its components using the Pythagorean theorem.

Normalize Vector

normalize_vector(vector)
The function creates (and returns) a new vector by normalizing the input vector. This means that the resulting vector has the same direction but a magnitude of 1. In short, the new vector is the original vector scaled by the inverse of its length.

Point Difference

difference_point(point1, point2)
This function creates (and returns) a new vector obtained by subtracting from point point1 the point point2 (i.e., point1 - point2). This is computed by subtracting the corresponding x-, y-, and z-components. This gives a vector, conceptually, pointing from point2 to point1.

Vector Difference

difference_vector(vector1, vector2)
This functions creates (and returns) a new vector obtained by subtracting from vector vector1 the vector vector2 (i.e., vector1 - vector2). This is computed by subtracting the corresponding x-, y-, and z-components. (Yes, this is very similar to the previous function; the types, however, are conceptually different.)

Translate Point

translate_point(point, vector)
This function creates (and returns) a new point created by translating (i.e., moving) the argument point in the direction of and by the magnitude of the argument vector. You can think of this as the argument vector directing the new point where and how far to go from the argument point.
For example, translating point <9, 0, 1> along vector <1, 2, 3> will result in point <10, 2, 4>.

Vector From To

vector_from_to(from_point, to_point)
This function is simply added to improve readability (and, thereby, to reduce confusion in later assignments). A vector in the direction from one point (from_point) to another (to_point) can be found by subtracting (i.e., point difference) from_point from to_point (i.e., to_point - from_point).

Test Cases

In tests.py, write test cases for each of the above functions (including each of the __eq__ functions). You should place each test case in a separate, appropriately named testing function.
Your test cases *must* use the unittest module used in lab and discussed in lecture. Create values that are valid arguments to the functions, invoke the functions, and then check that the results are what you expect (as calculated by hand).

Part 3

In this assignment you will implement the core intersection mechanic for the ray casting project. More specifically, you will write a number of functions to determine the set of spheres with which a ray intersects.

Files

Create a *hw3* directory in which to develop your solution. You will need to copy your files from the previous assignment into this directory.
You will develop the new parts of your program solution over two files. You must use the specified names for your files.

- collisions.py - contains the function implementations
- tests.py - contains your test cases

Once you are ready to do so, and you may choose to do so often while incrementally developing your solution, run your program with the command **python tests.py**.
To check that your test cases are syntactically correct, you will need to write your test cases and then compile with **python -m py_compile tests.py**. This will compile your source file (and report syntax errors) without executing the test cases. You should do this before submitting your test cases for the first deadline.

Functions

You are to implement the following functions.
sphere_intersection_point(ray, sphere)
find_intersection_points(sphere_list, ray)
sphere_normal_at_point(sphere, point)
In collisions.py implement each of the functions as specified in the following descriptions.

Single Ray-Sphere Intersection

sphere_intersection_point(ray, sphere)
Partial Function?
The purpose of this function is to determine if the given ray intersects the given sphere. If they intersect, then the point at which they intersect is returned (as a Point object). If they do not intersect, then the function should return None. You should note that the function is returning a different type when there is an intersection than when there is not. This is because an intersection function is essentially a partial function (i.e., for some inputs there is no reasonable point of intersection). We will use None to indicate that there is no intersection and leave it to the invoker of this function to check for this case on return.
Functionality
In collisions.py define the sphere_intersection_point function using the algorithm described next.
Ray-Sphere Collision Algorithm
The reasoning behind the functionality.
Is a point on a sphere?
Consider a sphere, theSphere, somewhere in three-dimensional space. This sphere has a location (theSphere.center) and a radius (theSphere.radius). Consider a point **on** the sphere (and recognize that this holds for every such point). The distance between this point and the center is, of course, equal to the radius.

This fact is great. If we have a point (any point), we can determine if that point is on the sphere by comparing its distance from the sphere's center against the radius. But we do not yet have a point to check.
Recall the following definition of the dot product from the previous assignment.

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$$

This equation relates the dot product to the angle between two vectors and the magnitude of each vector. If we compute the dot product of a vector with itself, then the angle between them is 0 and, thus, the dot product gives the magnitude squared.

$$\mathbf{a} \cdot \mathbf{a} = \|\mathbf{a}\|^2$$

With this in mind, we can use the dot product to determine if a point p is on a sphere theSphere (we will do this because it turns out that using the vector functions simplifies the computation). Which vector do we want? The vector from the center of the sphere to the point. This gives the following.

(p - theSphere.center) $\cdot$ (p - theSphere.center) = theSphere.radius^2

Which point should be checked?

With the above it is relatively straightforward to determine if a point lies on a sphere (in fact, you have already implemented the dot product function). But now we need a point to check. It would obviously take far too much time to check for all such points, something we neither want nor need to do. Instead, the only points of interest are those along the ray being cast into the scene.
Recall that a ray (theRay) has both a point of origin (theRay.pt) and a direction (theRay.dir). All points along the ray can be found by transating the ray's origin in the ray's direction. This is a lot of points, but we can parameterize the equation as follows.

point$_t$ = theRay.pt + t * theRay.dir, $t \geq 0$

This equation states that a point is found by scaling the direction vector by some parameter t and then translating the ray's origin by the resulting vector. But this equation itself does not give a specific point of interest and there are far too many values to consider for t. Here's where the math comes in.

The math.

Let's take the two previous equations. The second is an equation for all points along a ray. The first is a means to check if a point is on a sphere.

(p - theSphere.center) $\cdot$ (p - theSphere.center) = theSphere.radius^2

point$_t$ = theRay.pt + t * theRay.dir, $t \geq 0$

Let's combine them.

((theRay.pt + t * theRay.dir) - theSphere.center) $\cdot$ ((theRay.pt + t * theRay.dir) - theSphere.center) = theSphere.radius^2

The resulting equation is a lot to look at but essentially asks which point along the ray intersects the sphere. Or, another view, what value of t gives a point along the ray that intersects with the sphere?
This equation can be transformed a bit into the more familiar form of the quadratic equation.

(theRay.dir $\cdot$ theRay.dir) t^2

+ (2 * (theRay.pt - theSphere.center) $\cdot$ theRay.dir) t

+ (((theRay.pt - theSphere.center) $\cdot$ (theRay.pt - theSphere.center)) - theSphere.radius^2)

= 0

Or.

A t^2

+ B t

+ C
= 0
Giving.
A = (theRay.dir $\cdot$ theRay.dir)
B = (2 * (theRay.pt - theSphere.center) $\cdot$ theRay.dir)
C = (((theRay.pt - theSphere.center) $\cdot$ (theRay.pt - theSphere.center)) - theSphere.radius^2)
The value of t can now be computed by solving the quadratic equation. As you may recall, this may give 0, 1, or 2 real roots. If there are 0 real roots (i.e., the discriminant is negative), then the ray does not intersect the sphere. If there is one real root, then the ray glances the surface of the sphere colliding at a single point. If there are two real roots, then the ray intersects the sphere at two points passing into and then out of the sphere; the point nearest the origin of the ray (but as computed only by non-negative t values) is the closest intersection point.
To do.
Wow, that was a lot of setup, but with the quadratic equation solved you can determine the nearest intersection point.
Implement the sphere_intersection_point function. If the ray intersects the sphere, then return a Point object with the coordinates of the nearest (to the ray's origin point) intersection point. If the ray does not intersect the sphere, then return a None indicating that the point is not valid.
Consider some of the possible (real) t values.

- Both t values are non-negative (i.e., zero or positive). This implies that the sphere is "in front" of the ray (or, in the zero case, that the ray starts on the sphere) and the ray passes into and then out of the sphere. The smallest of these values gives the desired point of intersection.
- Both t values are negative. This implies that the sphere is "behind" the ray. As such, there is no true intersection with the sphere (so the function should return None).
- One t value is non-negative and the other is negative. This implies that the ray originates inside of (or on) the sphere and intersects on the way out. The non-negative t gives the desired point of intersection.
- If there is only a single root, then the above still applies, it just means the ray intersects on the surface but does not pass through. If t is negative, then the intersection point is not of interest.

But we do not actually want the t value. Wait, what?
The computed t value, if there is a real one, is not what this function will return. Instead, it must return the point of intersection. This point can be found by using the equation given earlier (shown again below).
point$_t$ = theRay.pt + t * theRay.dir, $t \geq 0$
Recall that this equation states that the point is found by scaling the direction vector by t and then translating the ray's origin by the resulting vector.

All Ray-Sphere Intersections

find_intersection_points(sphere_list, ray)
After the discussion for the previous function, you may be thinking "oh, dear, here it comes." Fear not, this function uses the work of the previous function to gather all intersections for a given ray.
Functionality
In collisions.py define the find_intersection_points function. This function checks for an intersection between the given ray and each sphere in the sphere_list list. This function will return a list of pairs (i.e., a

tuple with two values) representing the found intersections; each pair will contain, in this order, a Sphere and a Point.
For each sphere in the input list, if the ray intersects (determined using the sphere_intersection_point function), add a pair to the list to be returned containing the sphere and the intersection point. The returned list will only contain pairs representing intersections and they should be ordered relative to the input list. As such, the first intersection found will be stored in the 0th position of result list, the second intersection found will be stored in the 1th position of the result list, and so on.
This function is really just an elaborate example of the filter pattern.

Sphere Normal

sphere_normal_at_point(sphere, point)
With a point on a sphere found, the normal vector at that point will be of use in the next assignment to determine if the viewer can see the point. The normal vector is a vector (with magnitude 1) along which the center of the sphere must move to reach the point (colloquially referred to as a vector **from** the center **to** the point on the surface). Conceptually, this vector is going "from the center out". This can (and should) be computed using functions from the previous assignment.
Functionality
In collisions.py define the sphere_normal_at_point function. Compute the normal vector for the provided sphere at the provided point. (This vector must normalized so that its magnitude is 1.)

Test Cases

In tests.py, write test cases for each of the above functions. You should place each test case in a separate, appropriately named testing function.
Your test cases *must* use the unittest module used in lab and discussed in lecture. Create values that are valid arguments to the functions, invoke the functions, and then check that the results are what you expect (as calculated by hand).
Hints

- When testing collisions, choose a few "easy" spheres and rays (e.g., those along an axis) and at least one "non-easy" sphere and ray.
- When testing a function that returns a list, it is often helpful to compare the result directly against an "expected" list that you create explicitly in your testing function. This assumes, of course, that the contents can be compared for equality using __eq__ (which is the case for our data values).

Part D

This assignment pulls together the work of the previous assignments to create a functional (though somewhat hard-coded) ray casting program. In order to do so you will need to provide a few more data definitions and implement some functions (two required, but likely a few more as helper functions).
Unlike the previous assignments, your solution to this assignment will be built up in parts.

Files

Create a *hw4* directory in which to develop your solution. You will also need to copy your files from the previous assignment. Since the solution to this assignment is developed in parts, you may also choose to create subdirectories corresponding to each part.

You will develop the new parts of your program solution over three files. You must use the specified names for your files.

- cast.py - contains the function implementations
- tests.py - contains your test cases for individual functions (unit tests)
- casting_test.py - contains a *main* function and a scene cast test (system test)

Once you are ready to do so, and you may choose to do so often while incrementally developing your solution, run your test cases with the command **python tests.py**. To run your system test (to generate an image) use the command **python casting_test.py > image.ppm**.

A Note on Color

The concept of a color appears in two different manifestations in this assignment. The majority of your code will work with a color as represented by red, green, and blue components that are float values. While, technically, such components should only range from 0.0 to 1.0, some of the arithmetic on colors (in the later parts) will allow for values greater than 1.0 (this is due to an "intensity" scaling factor).
The other representation of a color is *strictly for the output format* that is used in this assignment (the ppm P3 format discussed here). This representation requires integer values strictly in the range [0, 255] (for our purposes).
Converting between different representations is something that is often done in programming. For this assignment, all of your calculations are to be done in terms of float values. Then, just prior to printing, scale the color components so that they fall into the range [0, 255] (capping the values at 255), convert to an integer, and then print the components (you can write a separate utility function for converting).

Part 1

For this part, you will implement a program that casts rays into a scene and prints an image. This image will be printed in version *P3* of the *ppm* format. Details of this format are provided on a separate ppm P3 format page.

Functions

For this first part you will implement simplified versions of the two primary functions for this assignment. In the later parts you will extend these functions.

Single Ray Cast

You must implement the primary casting function.
cast_ray(ray, sphere_list)
In cast.py, implement this function to "cast" the specified ray into the scene to gather all intersection points (and the corresponding spheres); this will use the appropriate function from the previous assignment. If an intersection is found, then this function returns True, otherwise the function returns False.

All Rays Cast

You must implement the following function to cast all of the rays required for the entire virtual "scene".
cast_all_rays(min_x, max_x, min_y, max_y, width, height, eye_point, sphere_list)
In cast.py, implement this function. This function controls the ray casting for the entire scene. The function will cast a ray for each pixel in the output image. These rays are cast from the eye point to points on a

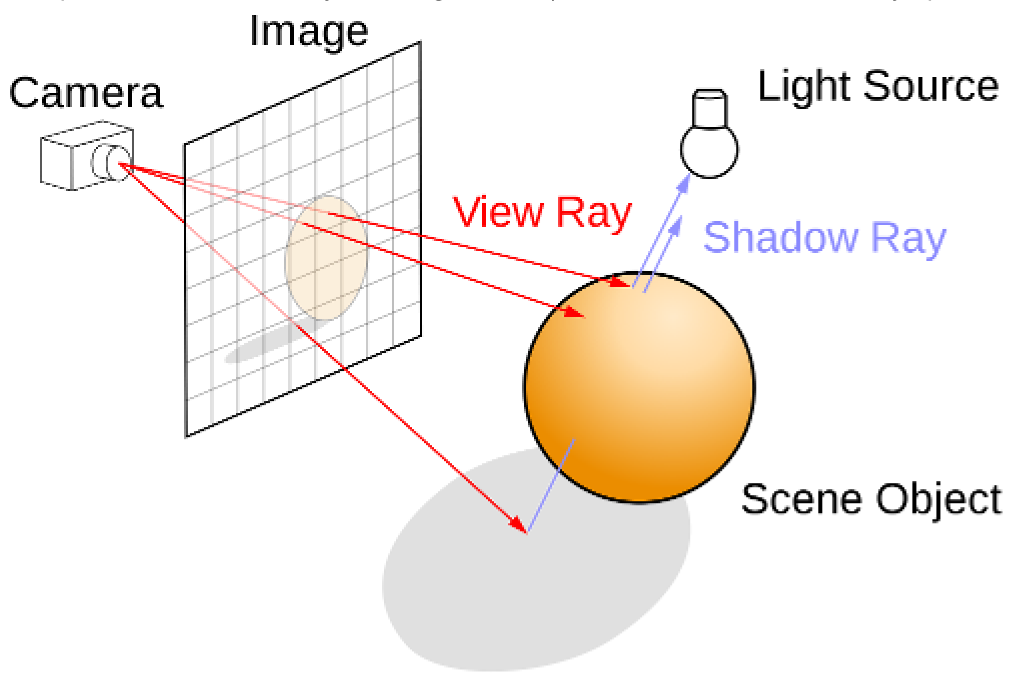
view rectangle as discussed below. Each ray cast will determine the color (black or white for Part 1) for the corresponding pixel.

The min_x, max_x, min_y, and max_y arguments specify the bounds of the view rectangle (with z-coordinate 0.0). The width and height specify the dimensions of the resulting image and, as such, the number of pixels. The pixels will be evenly distributed over the view rectangle: specifically, there will be width evenly distributed x coordinates in the interval [min_x, max_x) and height evenly distributed y coordinates in the interval [max_y, min_y) where the top-left corner of the view rectangle is at point <min_x, max_y>. You will likely note that this skews the image slightly to the left and top of the view rectangle, but that is fine.

Cast a ray from the eye through each of these evenly distributed points beginning with the top-left corner (<min_x, max_y>) and varying the x-coordinate before varying the y-coordinate (i.e., compute every pixel in a row before moving to the next row). For each ray cast, print to the screen (in the format discussed above for P3) black (red=0.0, green=0.0, blue=0.0 in the internal color format; red=0, green=0, blue=0 in the file color format) if the cast results in an intersection and white (red=1.0, green=1.0, blue=1.0 in the internal color format; red=255, green=255, blue=255 in the file color format) if it does not.

For instance, if min_x is -4, max_x is 4, min_y is -2, max_y is 2, width is 4, and height is 2 then the points through which the ray will be cast are as follows (in this order): <-4, 2> <-2, 2> <0, 2> <2, 2> <-4, 0> <-2, 0> <0, 0> <2, 0>.

This process is illustrated by the image below (where the camera is at the eye position).



Test Cases

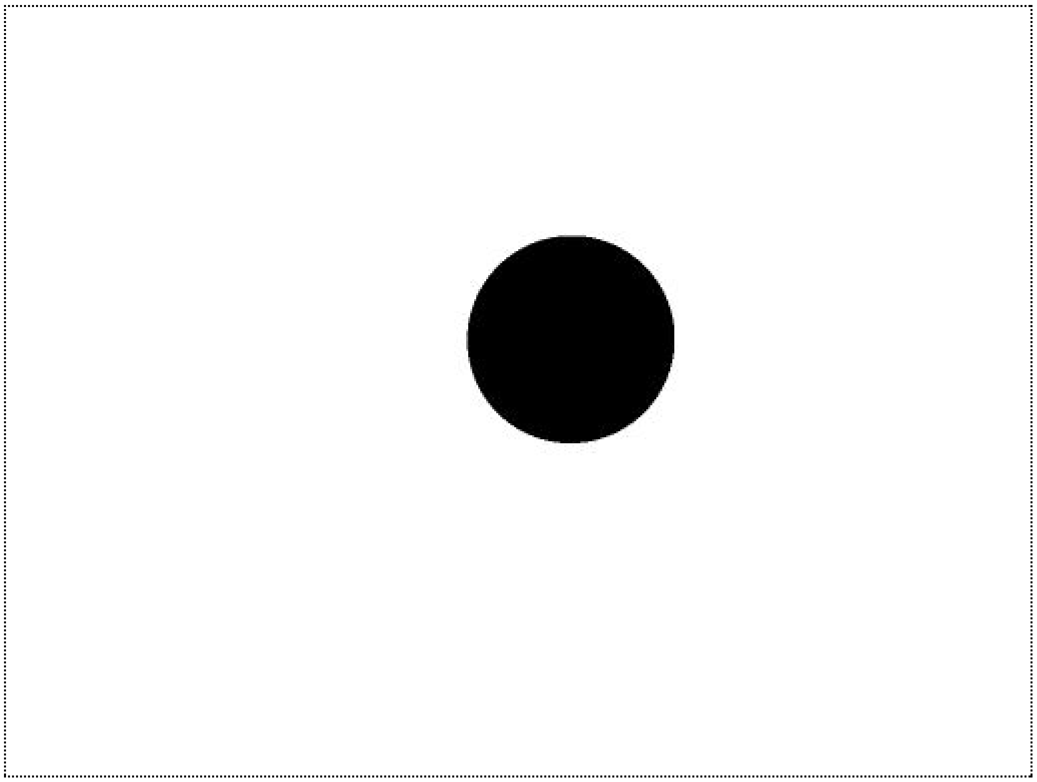
In tests.py, write test cases for cast_ray and any auxiliary (helper) functions that you have written. You need not provide test cases for cast_all_rays because this function prints to the console. This will be tested in a different manner as discussed below.

Casting Test

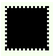
In casting_test.py, use cast_all_rays to create an image using the following configuration. Be sure to print the P3 header to the screen before casting the rays. The width and height are as specified and the maximum color value is 255.
Eye at <0.0, 0.0, -14.0>.
A sphere at <1.0, 1.0, 0.0> with radius 2.0.
A sphere at <0.5, 1.5, -3.0> with radius 0.5.
A viewport with minimum x of -10, maximum x of 10, minimum y of -7.5, maximum y of 7.5, width of 512, and height of 384.
When you run your program, redirect the output to a file named image.ppm. This is done by typing the following at the command prompt. (The first command, ulimit, which only needs to be run once per terminal session, is used to limit the size of the files generated. This will prevent a run-away program from filling your disk quota.)

```
ulimit -f 12000
python casting_test.py > image.ppm
```

The image generated from this part should look like the following.
Though this is not the image we ultimately want, it is a useful image. The image indicates where intersections occurred, but it does not give much information beyond that. In fact, it is hard to tell that the scene contains two spheres. On to the next part.
Note: You can compare your generated image to the "expected" image using tools discussed at the bottom of this page.



A smaller image can be had by changing the viewport to the following (this is only for debugging purposes; the casting_test.py file that you submit must have the viewport set as originally specified).
Minimum x of 0.05859357, maximum x of 0.44921857, minimum y of 2.03125, maximum y of 2.421875, width of 20, and height of 20.



Part 2

To this point there is really nothing to make the spheres stand out. Let's add some color.

Data Definitions

1. Modify data.py to add a new class to represent a Color in RGB (red, green, blue) format. The Color.__init__ function should take arguments r, g, and b and initialize the corresponding attributes of the object. Be sure to define the __eq__ function for this class.
2. Modify Sphere.__init__ to take color (as its last argument) and to set the corresponding attribute of the object.

Functions

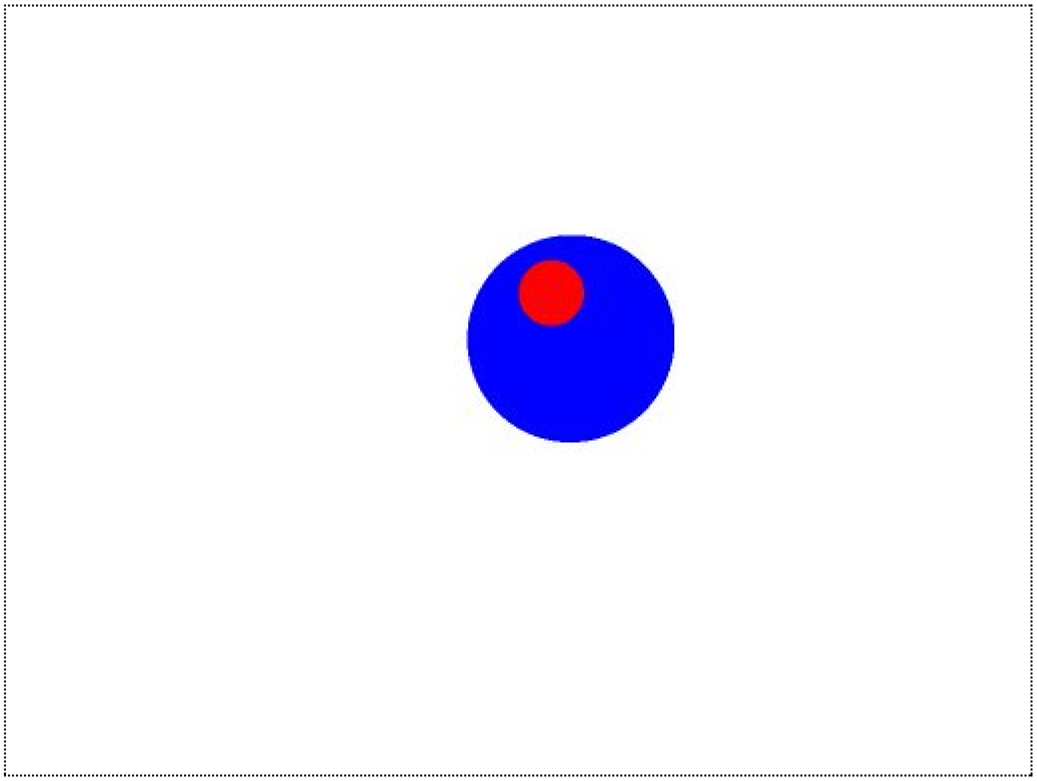
1. In cast.py, update the cast_ray function to return the color of the sphere with the nearest intersection point (to the ray's origin), if there is an intersection, or a default color of white (1.0, 1.0, 1.0) if there is no intersection. **Note:** Do not make any assumptions about the order of the spheres in the array as relates to the nearness of the spheres.

Test Cases

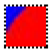
Update your test cases in tests.py to reflect the changes to these functions. Consider a test case for cast_ray that verifies it will find the nearest sphere regardless of the order of the spheres in the input array.

Casting Test

Update casting_test.py so that the smallest sphere is red and the largest sphere is blue. The image generated from this part should look like the following.
It is now apparent that there are two spheres in the scene (perhaps more, if some are hidden behind the others, but we know there are only two in this scene). Unfortunately, these spheres still look like circles. On to the next part.



And the smaller portion should look as follows.



Part 3

In order to give the scene some depth, and in so doing make the spheres actually look like spheres instead of circles, we need to add some notion of light. This will be done over these last three parts. Each part will add one component of the light model.
In order to implement the light you will need to add a finish to the spheres in the scene. The idea being that the finish "describes" how the object interacts with light (e.g., some objects are shiny while others are dull).

Data Definitions

1. Modify data.py to add a new class to represent a Finish. The Finish.__init__ function should, for now, take a single argument ambient and initialize the corresponding attribute of the object. This attribute represents the percentage of ambient light reflected by the finish. Be sure to define the __eq__ function for this class.
2. Modify Sphere.__init__ to take finish (as its last argument) and to set the corresponding attribute of the object.

Functions

1. In cast.py, update cast_all_rays so that it expects a Color as its last argument. This argument is the ambient light color.

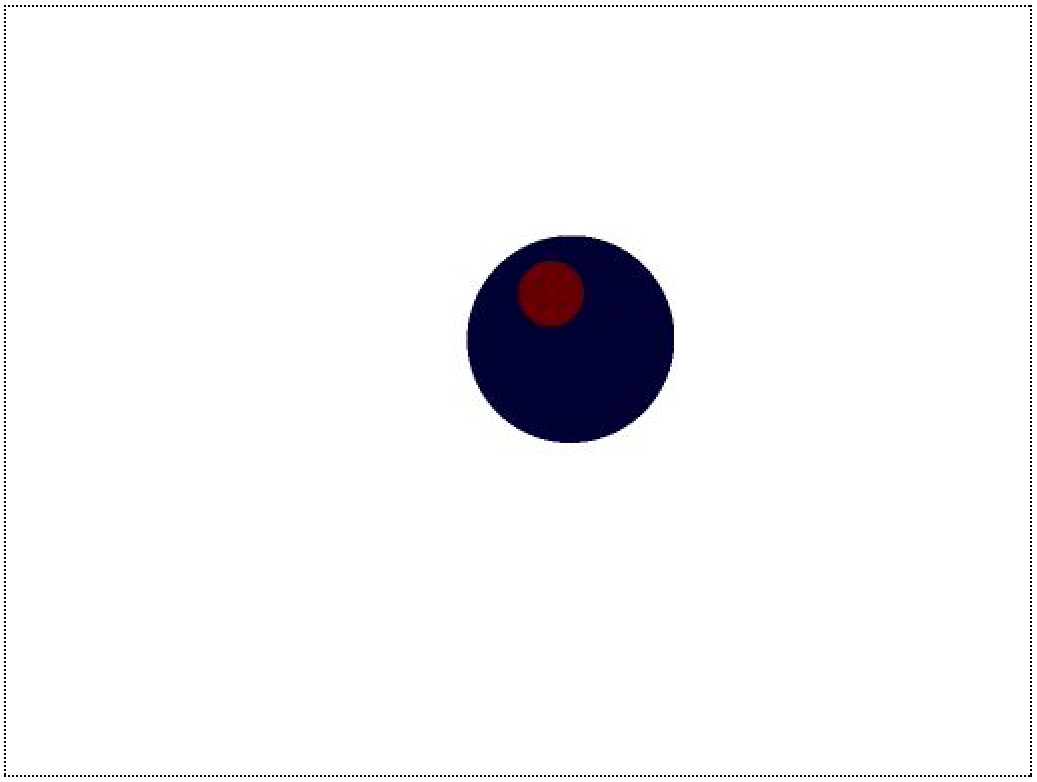
2. In cast.py, update cast_ray so that it expects a Color as its last argument. This argument is the ambient light color.
3. Update cast_ray to modify the calculation of the color of the intersection point. If the ray does not intersect a sphere, then the returned color will remain the white default (1.0, 1.0, 1.0). If, however, the ray does intersect a sphere, then the returned color will now depend on the light model. At this point, the only contribution to the color will come from the reflected ambient light (think of the ambient light as a light that reaches everything).
4. **Compute:** Each component of the returned color will be computed by multiplying the corresponding component of the sphere's color by the ambient value of the sphere's finish and by the ambient light's color.

Test Cases

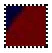
Update your test cases in tests.py to test any newly created functions and to reflect the changes to existing functions.

Casting Test

Update casting_test.py so that the scene's ambient light is white (1.0, 1.0, 1.0; though you may experiment with other light values, be sure to use this color for your submission). In addition, the large sphere should have a finish with ambient value 0.2 while the small sphere should have a finish with ambient value 0.4. The image generated from this part should look like the following.
This just looks like a darker version of the previous image but that is fine. The fact is, all that we really did in this step was set up the framework for the next steps and soften the color based on the ambient scale in each sphere's finish (if you increase the ambient scale, you can get the same image as in the previous step, but leave the values as directed for submission).



And the smaller portion should look as follows.



Part 4

For this step we will finally add some depth to the scene. This is accomplished by placing a point-light in the scene. Unlike the ambient lighting from the previous step, the point-light will emit light from a specified position.

Data Definitions

1. In data.py, update Finish.__init__ by adding diffuse as its last argument and by setting the corresponding attribute in the object. This attribute represents the percentage of diffuse light reflected by the finish. Be sure to update the __eq__ function to reflect this change.
2. In data.py, add a new class to represent a Light. The Light.__init__ function must take pt (a Point representing the position of the light) and color (a Color representing the color/intensity of the light) arguments and must initialize the corresponding attributes of the object. Be sure to define the __eq__ function for this class.

Functions

1. In cast.py, update cast_all_rays so that it expects a Light as its last argument.
2. In cast.py, cast_ray so that it expects a Light as its last argument.
3. Update cast_ray to modify calculation of the color of the intersection point. In addition to the ambient component of the color, a sphere intersection will now also compute a diffuse component that contributes (additively, i.e., the result of the following calculation is added to the result of the prior calculation) to the resulting color.
4. Consider the following options relating the intersection point on the sphere to the location of the point-light.
5. **Note carefully** that due to the imprecision of floating point values, using the computed intersection point in the following calculations can lead to unintended collisions with the sphere on which the point lies. To avoid these issues, these calculations will use a point just off of the sphere. This can be found by translating the intersection point along the sphere's normal (at the intersection point) by a small amount (use 0.01) (i.e., scale the normal by 0.01 to get a shorter vector and then translate the intersection point along this shorter vector). The following discussion refers to this new point as $p_ε$.
    - Compute $p_ε$ as just discussed.
    - If the light is on the opposite side of the sphere from $p_ε$, then it does not contribute to the color of this point. This can be determined by computing the dot product of the sphere's normal at the intersection point with a normalized vector from $p_ε$ to the light's position. If the light is in the general direction of the sphere's normal (i.e., on this side of the sphere), then the dot product will be positive. If the dot product is non-positive (i.e., 0 or negative), then we will consider the light to be on the opposite side of the sphere's surface and, thus, it will not hit this point (so the diffuse light contribution is 0).
        1. Compute the sphere's normal at the intersection point. I will refer to this as N. (This was already computed in order to compute $p_ε$ as described above).
        2. Compute a normalized vector from $p_ε$ to the light's position. I will refer to this as L_{dir}.

3. Compute the dot product of N and L_{dir} and use this value to determine if the light is "visible" from point p_ε.

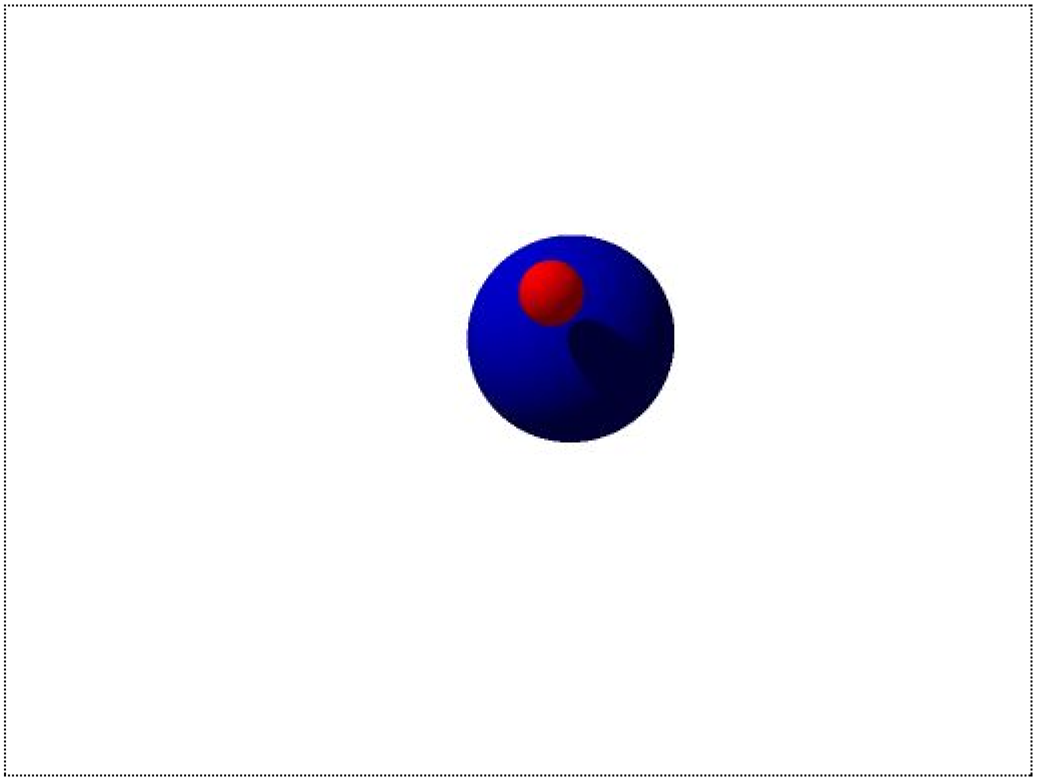
- If another sphere is on the path from p_ε to the light's position, then the light is obscured and does not contribute to the color of this point. This can be determined by checking if a ray from p_ε in the direction of L_{dir} (computed above) collides with a sphere <u>closer</u> to p_ε than the light is.
- If the light is not obscured, then it will contribute to the point's color based on the diffuse value of the sphere's finish and the angle between the sphere's normal (at the intersection point) and the normalized vector pointing to the light.
- More specifically, the light's diffuse contribution to the color of the point is (for each color component red, green, and blue) the product of a) the dot product between the sphere normal and the normalized light direction vector (i.e., $N \cdot L_{dir}$), b) the light's color component, c) the sphere's color component, and d) the diffuse scale of the sphere's finish.

Test Cases

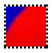
Update your test cases in tests.py to test any newly created functions and to reflect the changes to existing functions.

Casting Test

Update casting_test.py so that the point-light is located at <-100.0, 100.0, -100.0> with color component (1.5, 1.5, 1.5). In addition, the finish for each sphere should have diffuse value 0.4. The image generated from this part should look like the following.
You can now see how light provides a depth to the objects in the scene giving an illusion of a third-dimension. You can also see evidence of the shadow effect caused when a sphere partially obscures the light from hitting another sphere.



And the smaller portion should look as follows.



Part 5

This last step adds some additional character to the image through greater distinctions between different finishes by computing a specular component to the light model.

Data Definitions

1. In data.py, update Finish.__init__ to take specular and roughness as its last two arguments (in this order) and to set the corresponding attributes in the object. These attributes represent the percentage of specular light reflected by the finish and the modeled roughness of the finish (which affects the spread of the specular light across the object). Be sure to update the __eq__ function to reflect this change.

Functions

1. In cast.py, update cast_ray so that it expects a Point as its last argument. This is the position of the eye in the scene.
2. In cast.py, update cast_ray to modify the calculation of the color of the intersection point. In addition to the ambient and diffuse components of the color, a sphere intersection will now also compute a specular component that contributes (additively) to the resulting color.
3. The specular contribution is computed as follows.

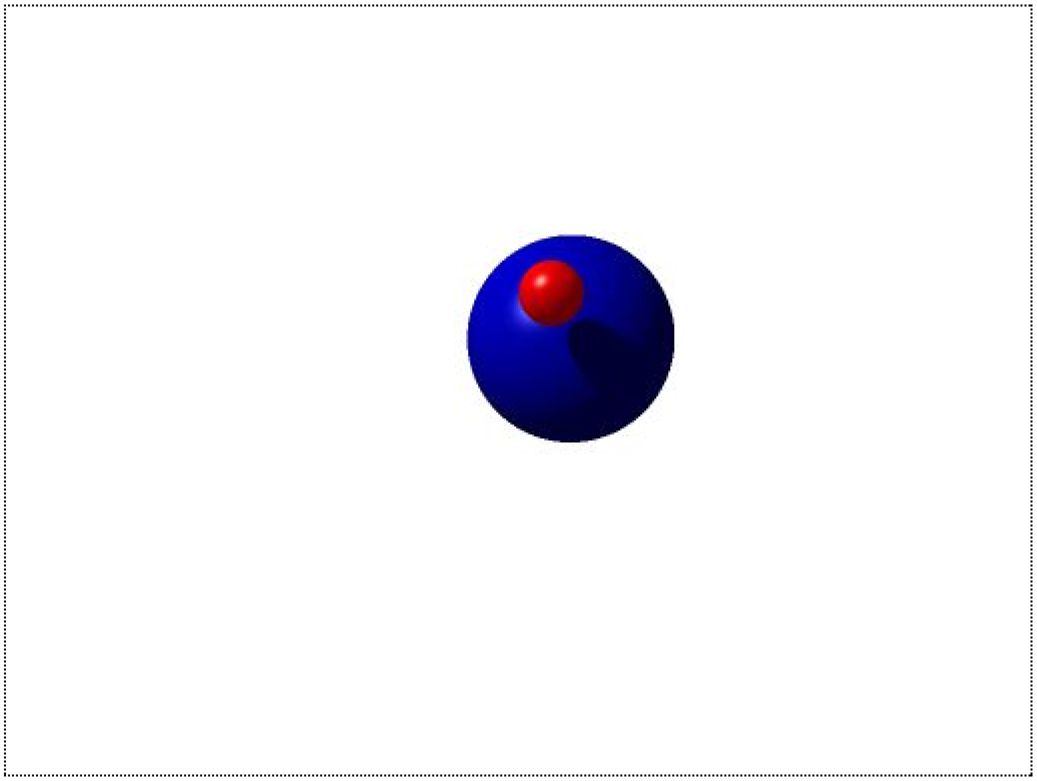
1. Compute the specular intensity by computing the vector that represents the direction in which the light is reflected and determining the degree to which that light is aimed toward the eye. You can do this as follows.
    1. Create a normalized vector from p_ε to the light's position (you already did this in the previous part). I will refer to this as the light direction, L_{dir}.
    2. Compute the dot product of the light direction and the sphere's normal at the point of intersection (you already did this in the previous part). I will refer to this as the light normal dot, $L_{dot}N$.
    3. Compute the reflection vector as L_{dir} - (2 * $L_{dot}N$) * N, where N is the sphere's normal at the point of intersection.
    4. Compute a normalized vector from the eye position to p_ε. I will refer to this as the view direction, V_{dir}.
    5. The specular intensity is the dot product of the reflection vector and V_{dir}.
2. If the specular intensity is positive, then the contribution to the point's color is the product of the light's color component, the specular value of the sphere's finish, and the specular intensity raised to a power of [1 divided by the roughness value of the sphere's finish]. If the specular intensity is not positive, then it does not contribute to the point's color.

Test Cases

Update your test cases in tests.py to test any newly created functions and to reflect the changes to existing functions.

Casting Test

Update casting_test.py so that the for each sphere has specular value 0.5 and roughness value 0.05. You can now see specular highlights from the light.



And the smaller portion should look as follows.

Part E

In this assignment you will complete the ray casting project by working with an input file and generating an output file. In addition, you will allow the user of your program to vary some of the characteristics of the casting through command-line arguments to your program.
Sample files are given near the end of this page.

Files

Create a *hw5* directory in which to develop your solution. You will need to copy your files from the previous assignment.
You will develop the new parts of your program over three files. You must use the specified names for your files.

- commandline.py - contains implementations of command-line processing functions
- tests.py - contains your test cases for individual functions (unit tests) - this will not be submitted for this assignment
- ray_caster.py - contains a *main* function and supporting I/O functions that implement the required functionality of the assignment

Once you are ready to do so, and you may choose to do so often while incrementally developing your solution, run your test cases with the command **python tests.py**. You will run **ray_caster.py** as discussed below to execute the program itself.

Functionality

There are effectively three parts to this assignment: generating an output file, reading from an input file, and processing command-line arguments. You may work on these parts in whichever order you prefer. For instance, you might find that generating a file is a relatively straightforward (and quick?) first step.
Once this assignment is finished, your program will take command-line arguments specifying the name of the input file (which gives the set of spheres in the scene) and optional flags that modify the default settings for the position of the eye, the view dimensions, the point light position and color, and the ambient light color. Your program will read the contents of the file to populate a list of spheres, cast all of the rays into the scene, and output the pixel values to a file named image.ppm.

Output File

In the previous assignment the pixel information for the generated image was printed to the screen in the ppm P3 format. For this assignment, you must modify your solution to output the generated image (in the same format) to a file named image.ppm. Be sure to include the P3 header in the file.

Input File

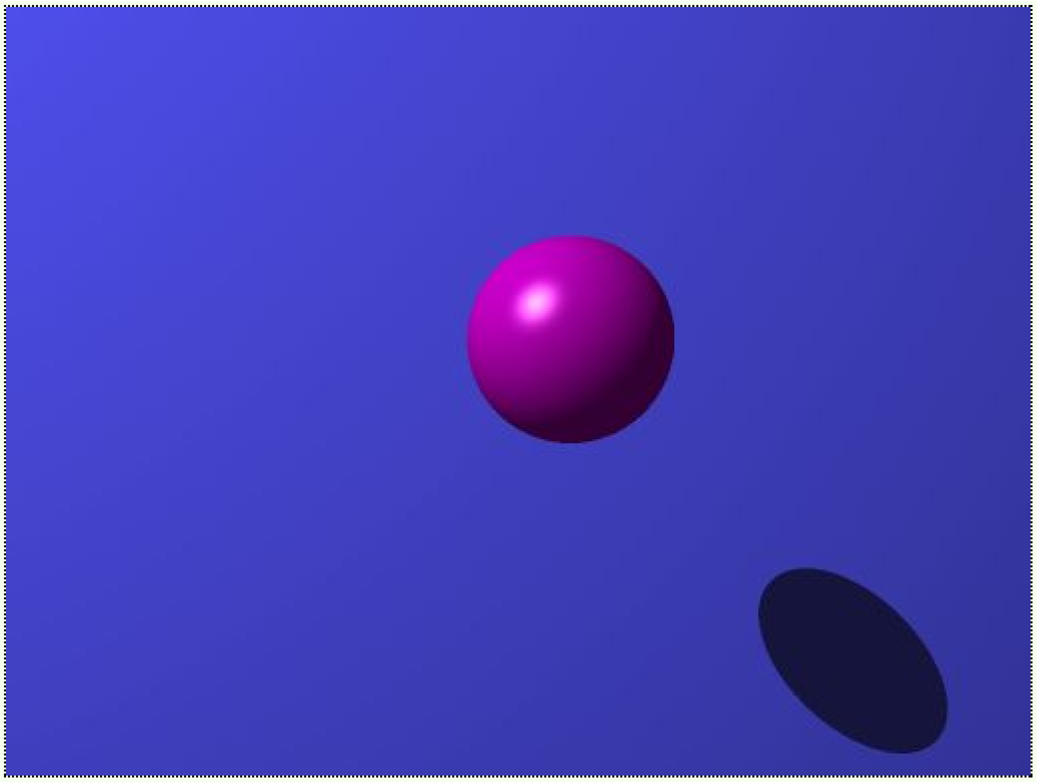
The file from which to read the set of spheres must be specified on the command-line (discussed below). Each line of this file will contain the values required for the creation of a sphere in the order x y z radius r

g b ambient diffuse specular roughness. Each value is of type float and they are separated by whitespace.
Your program must read every sphere in the file. Reading must stop when the end-of-file is reached.

File Format

For instance, the following specifies the characteristics for the two spheres in the image below. Note that the blue background in the image is actually the second sphere; this sphere is very large and serves as a surface for the shadow of the first sphere to land upon.

```
1.0 1.0 0.0 2.0 1.0 0.0 1.0 0.2 0.4 0.5 0.05
8.0 -10.0 110.0 100.0 0.2 0.2 0.6 0.4 0.8 0.0 0.05
```



Errors in the file

Your program *should not* assume a correctly formatted input file. More specifically, your program must verify that each line contains valid values for a sphere (i.e., there are exactly the required number of values and each is a number/float). If a line does not represent a valid sphere, then your program should output an error message. The following examples show an input file with invalid lines and the expected error message.

```
1.0 1.0 0.0
1.0 1.0 0.0 2.0 1.0 0.0 1.0 0.2 0.4 0.5 0.05
1.0 1.0 0.0 2.0 1.0 0.0 1.0 0.2 0.4 0.5 0.05 3
4.7 1.0 2.0 2.0 2.0 bob 2.0 020 412 000 0000
8.0 -10.0 110.0 100.0 0.2 0.2 0.6 0.4 0.8 0.0 0.05
```

8.0

```
malformed sphere on line 1 ... skipping
malformed sphere on line 3 ... skipping
malformed sphere on line 4 ... skipping
malformed sphere on line 6 ... skipping
malformed sphere on line 7 ... skipping
```

Command-line Processing

Your program will take one required command-line argument and up to four command-line arguments that the user may optionally specify (in any order).
The first command-line argument to your program will be the name of the input file. If this file does not exist or cannot be opened for reading, then an error should be reported and the program terminated.
If the file argument is not specified, then print a usage message like the following.

```
usage: python ray_caster.py <filename> [-eye x y z] [-view min_x max_x min_y max_y width height] [-light x y z r g b] [-ambient r g b]
```

The remaining arguments may be optionally specified. If any are specified, then they will begin with what are conventionally called flags (or switches). Each such flag will signify that additional arguments follow. These flags (grouped with their additional arguments) may appear in any order.
The flags and additional arguments are as follows. If an argument is missing, use the specified default values for the flag. If an argument cannot be converted into a numeric value, then use the default value for that argument.

- -eye If this flag is present, then the next three arguments specify the x, y, and z coordinates of the position of the eye from which rays are cast. If this flag is not present, then the eye defaults to position <0.0, 0.0, -14.0>. Note that many settings for the eye will result in skewed images because the view rectangle is always perpendicular to the z-axis and fixed at z=0; you can generate some bizarre images by moving the eye given our current configuration.
- -view If this flag is present, then the next six arguments specify the min_x, max_x, min_y, max_y, width, and height components of the view rectangle. If the flag is not present, then the view rectangle defaults to position min_x of -10, max_x of 10, min_y of -7.5, max_y of 7.5, width of 512, and height of 384.
- -light If this flag is present, then the next six arguments specify the x, y, and z coordinates of the light's position and the r, g, and b values of the light's color. If the flag is not present, then the light defaults to position <-100.0, 100.0, -100.0> and color/intensity (1.5, 1.5, 1.5).
- -ambient If this flag is present, then the next three arguments specify the r, g, and b values of the ambient light color. If the flag is not present, then the ambient light color defaults to (1.0, 1.0, 1.0).

Test Cases

You should write test cases in tests.py to verify the non-I/O functions in your solution. You will *not* submit these tests, so the degree of testing is up to you; make sure that you are confident in your solution.

Sample Files and Comparing

Some sample input and output files are given below (the images shown in the browser are in a different format and are scaled down; do not compare against these). You should run your program on smaller test files (like the sphere example) and then try the larger examples. Keep in mind that the larger examples will take a while to run (this is expected due to both the number of spheres and the overhead associated with the Python interpreter).

Each of the sample images was generated with the default settings for the eye, view, point light, and ambient light. **You can reduce the resolution (width and height) to reduce the execution time.**
sphere.in sphere.ppm