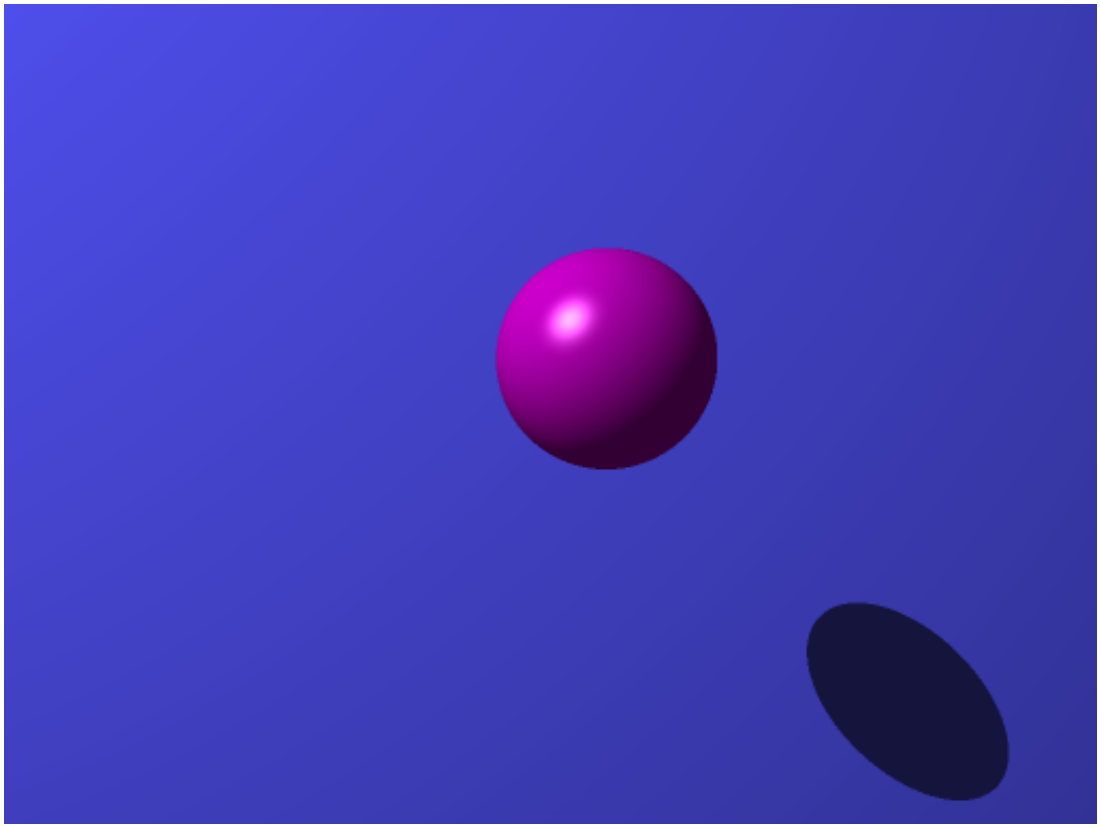
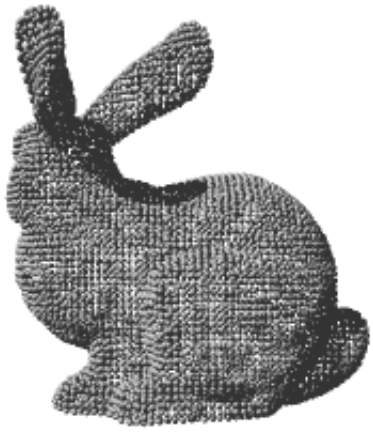
Assignment 1, CSC 101

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.





Files

Create a *hw1* directory in which to develop your solution.

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 `import`ed 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.

You **must** provide at least two separate test cases for each class. This means creating two instances of the class (using different input values) and verifying correct initialization.

Handin

You must submit your solution on PolyLearn via the related assignment by 11:55pm on the due date.

Note that you can resubmit your files as often as you'd like prior to the deadline. Each subsequent submission will replace files of the same name.

Grading

The grading breakdown for this assignment is as follows.

- **Clean Execution**: 10% — Program runs without crashing (and the submitted source demonstrates a legitimate attempt at a solution).
- **Test Cases**: 25% — Test cases are provided for each of the classes. The number of test cases is appropriate for the complexity of the corresponding function (with a minimum of two test cases).
- **Functionality**: 65% — Required functionality has been implemented.