Ray Tracing Scenery

CS11 Advanced C++ Lab 3

This lab continues the construction of our ray tracer.

The Ray Tracing Process

Ray tracing is conceptually very simple. Every object in a scene has a mathematical description of its surface. For example, a sphere with a center **C** and radius *r* includes all points **X** within a distance of *r* from its center: |**X** - **C**| = *r*. (Or, you can square both sides, which makes little difference in the solution, but definitely speeds up the math.)

Rays have two components, an origin **P** and a direction **D**. The direction is usually a normalized vector. The ray itself is modeled as another equation: **P** + **D** * *t*. The variable *t* only ranges over nonnegative values, since the ray starts from the origin point. When *t* = 0, the ray is at point **P**, and as *t* increases, the ray moves in the direction of **D**. (If **D** is normalized, notice that the ray has traversed a distance *t* at time *t*, which is sometimes useful to know...)

For simple objects like spheres, we can just solve for the intersection points of the ray's equation and the sphere's equation. (Actually, we solve for the values of *t* when intersections occur, then use these values to compute the intersection points.) In the case of a sphere, this may result in no intersections, one intersection (a grazing hit), or two intersections. Normally the closest intersection is taken, although for implementing features like Constructive Solid Geometry, you may want all of the intersection points.

Since this is a programming course and not a math course, we will give you the math equations for finding intersections with various simple objects. Your main job will be to implement the necessary classes in the ray tracer, and hopefully to implement the equations correctly! Each kind of scene object will provide its own intersection-test functionality, so that the ray tracer can render scenes with a variety of object types.

Once an intersection is found with an object, the color at that location must be properly computed. This involves, among other things, the surface normal at the point of intersection. Thus, scene objects must also provide a mechanism for determining the surface normal at a location.

If you want to learn much more about ray tracing, you can read the Wikipedia page on the subject. However, we'll tell you everything that's necessary to know for each lab.

Rays

It is very convenient to have a class to represent rays themselves, because they have multiple components, and a *lot* of operations will involve rays. So, you might as well make your life a little easier. Create a class to represent a ray being traced. You can call it **`Ray`**, or if you come up with a better name, use that.

As mentioned above, rays require two data members:

- the origin that the ray is emitted from
- the direction that the ray is headed

You can represent both of these values with your vector class from last week. (Yes, technically the origin is a point and not a vector, but representing these as separate types leads to a lot of headaches.)

Your ray's constructor should take the origin and direction values for the ray as its arguments. The one nuance of the `Ray` constructor is that sometimes you will want the constructor to normalize the direction vector automatically, and sometimes you will want to leave the direction value alone. Do this by having a third argument, a flag that controls whether the direction value is normalized. Make this flag default to normalizing the direction vector, since this will be the typical usage. For example, you might do this:

```
Ray(const Vector3F &orig, const Vector3F &dir,
    bool normalize = true);
```

You should provide accessors to the origin and direction values as well. However, you shouldn't need mutators on the ray object at all, since it doesn't need to be manipulated for any computations. When a new ray is fired from a particular location, you can just create a new `Ray` object with new values.

Your ray also need to provide one member function that returns the actual 3D point for a particular *t* value. During ray intersection tests, we don't want to store actual points of intersection because we really don't need to; we just use these values of *t* where intersections occur. But, ultimately, we will need to convert that into a 3D point. So, provide a function to do this. You might call it `getPointAtT(float t)`, for example. (Hint: Assert that *t* >= 0 in your code!)

Scene Objects

You will need to implement a class to represent each object in a raytraced scene. I recommend that you call them "scene objects" to clearly indicate their purpose. This class should be an abstract base class, since there is no well-defined way that generic scene objects should

behave. Eventually your scene object will include detailed information about its surface characteristics, but for now your objects will have one characteristic: their surface color. So, give your scene object class a single data-member, a "surface color" field, specified using the color class you created last week.

Once you have the basic framework for your class, you need to add the following functions:

- **An accessor and a mutator for the scene-object's surface color.** These should be very simple. Just return the current surface color, or take a color as an argument and store it. (While you are at it, you should probably create a default scene-object constructor that initializes your surface color to something other than black, such as gray (0.5, 0.5, 0.5).
- **A pure-virtual function that computes whether or not an intersection occurred.** If an intersection has occurred, the function should return the lowest *t* value for the intersection (this would be the intersection closest to the ray's origin). This is all the code needs to return; remember that we can get to the actual 3D point by using the combination of the ray and this particular value of *t*.

  So, your function might look something like this:

  ```
  float SceneObject::intersection(const Ray &r) const;
  ```

  The argument is the ray to test against, and the return value is the *t* value for the intersection.

  If *no* intersection occurred, you can return some known "invalid" value, such as *t* = -1. You should define a constant for this, and use the constant; don't just use -1 everywhere.

- **A pure-virtual function that returns the surface normal of a point on an object.** This function will be used when an intersection occurs, to determine what color should be assigned to the location, and what direction rays will bounce off of that point on the object. In this case, the argument should be a 3D point which is assumed to be on the surface of the object, and the return-value is a surface normal for that point. This function should be pure-virtual.
- **A function that returns the color of a point on an object.** This function also takes as an argument a 3D point on the surface of the object. The function can simply return the color passed to the scene-object constructor for now, but you can imagine how this would change if you were to implement texture-mapped objects.

Scene Object Subclasses

All objects in the scene being ray-traced will be subclasses of the scene-object base class. For now, you can create the following subclasses:

- **A plane object of infinite size.** Planes are specified by two values, a distance *d* from the origin, and a surface-normal **N** for the plane. Given these two values, the points in the plane satisfy this equation:

  $$f(\mathbf{X}) = \mathbf{X} \cdot \mathbf{N} + d = 0$$

  Thus, the plane class should have two data members:

  - A scalar specifying the distance of the plane from the origin. (Use the same type as your vector element type.)
  - A vector specifying the surface normal for the plane.

  The class' constructor should require these values, and you should provide accessors (but *not* mutators!) for these values.

  Given the above values, the relevant equations for plane computations are:

  - For a ray **P** + **D** * *t*, the intersection point is:

    $$t = -(\mathbf{P} \cdot \mathbf{N} + d) / (\mathbf{D} \cdot \mathbf{N})$$

    Only values of *t* >= 0 are considered intersections. Also, note that **D** · **N** can be 0, which also indicates no intersection.
  - The surface normal of any point in the plane is simply **N**.

- **A sphere object with a particular location and radius.**

  The sphere class should have two data members:

  - A vector specifying the sphere's center.
  - A scalar specifying the sphere's radius.

  The class' constructor should require these values, and you should provide accessors (but *not* mutators!) for these values.

  Given the above values, the relevant equations for sphere computations are:

  - There can be 0, 1, or 2 intersections between a ray and a sphere. For a ray with the same formulation as before, the intersections are simply the solution to the quadratic equation:

    $$a * t^2 + b * t + c = 0$$

    $$a = \mathbf{D} \cdot \mathbf{D}$$

$$b = 2 * (\mathbf{P} \cdot \mathbf{D} - \mathbf{D} \cdot \mathbf{C})$$
$$c = \mathbf{P} \cdot \mathbf{P} + \mathbf{C} \cdot \mathbf{C} - 2 * (\mathbf{P} \cdot \mathbf{C}) - r^2$$

You can use the discriminant to guide your computation, but remember the additional constraint that we only want solutions where *t* >= 0. Also, notice that *a* will never be zero since the magnitude of **D** will never be zero, so you don't have to check for that in your code.

To make your life easier down the line, you should implement this test in a public helper function that returns *all* of the sphere's intersection points, not just the closest one. You can write a helper function like this:

```
int getIntersections(const Ray &r, float &t1, float &t2) const;
```

This helper function can return the total number of valid intersections (0, 1, or 2), and can set `t1` and `t2` to the t-values of the intersection points, or to your "no intersection" value. (`t1` and `t2` are "out-parameters" since they are non-const references, and are used to return additional values back to the caller.)

Also, to make your life easier down the line, you should follow some conventions in how you return the values via `t1` and `t2`:

- When there are two valid intersection points, store the smaller one in `t1`, and the larger one in `t2`.
- When there is only one valid intersection point, always store it in `t1`, and set `t2` to "no intersection".
- When there are no valid intersection points, set both `t1` and `t2` to "no intersection" before returning 0.

This way, `t1` will always be the closest intersection point. In fact, when you use this helper-function to implement the `SceneObject` intersection-test function, it will simply be a matter of calling this helper function, then returning the value of `t1`:

```
// local variables t1, t2 to receive the results
// of the computation
float t1, t2;
getIntersections(r, t1, t2);

// t1 is either the closest intersection point, or
// it is our "no intersection" value.
return t1;
```

- The surface normal of any point on the sphere is:

$$n(\mathbf{X}) = (\mathbf{X} - \mathbf{C}) / |\mathbf{X} - \mathbf{C}|$$

In other words, subtract the point on the surface from the center, and normalize the resulting vector. Easy peasy.

In a few weeks you will add cylinders to your raytracer. This will involve a few new math operations, but the big win is that you can reuse your above sphere-intersection code, if you provide a mechanism to get all intersection points with the sphere. So, it's a bit of extra work right now, but it will save a lot of time down the line.

Lights!

In order to actually see anything in a ray-traced scene, there must be some light source. We can also use lights to render shadows that objects cast on other objects. For now, we will have a very simple lighting model, where all lights are point-lights of a specific color.

Create a class to represent point lights, with the following state:

- the position of the light
- the color of the light

The constructor should take arguments to store for these values. The class should also provide accessors (but not mutators) for these values.

We won't do anything else with lights this week, but we will incorporate them into the raytracing process next week.

The Scene

A scene is simply a collection of scene-objects being raytraced, and another collection of lights illuminating the scene. Although we could certainly be much more sophisticated than this, we will stick with the "simple" theme and represent scenes in this simple way.

Create a class to represent a scene. The objects in the scene will be dynamically allocated, as will the lights. (Since you only do this once to set up the scene, this will not hinder performance.) Use STL `vector`s to store the object-pointers and light-pointers. You should provide the following functionality:

- The default constructor should just create an empty scene.
- A member function to add a new scene-object, that takes a pointer to the scene-object to add. Assert that the pointer isn't 0. The scene-object will be heap-allocated outside the scene code, but the scene will still assume responsibility for cleaning up all scene-objects.
- A member function to add a new light, that takes a pointer to the light to add. The same comments apply here as for the previous member function.
- Write a destructor that goes through these lists of pointers and deletes all of the scene-objects and lights. (You might try using the

`for_each` algorithm with a delete-functor, as discussed in class.)

Next week we will take this scene object and implement the ray tracing process in it. For now, it will just function as a collection of objects and lights.

Testing

When you have finished writing all of these classes, you should write some test code to exercise your intersection code, to make sure that it works properly. Construct very simple tests, such as shooting a ray from (0,0,0) at a sphere of radius 1 at (2,0,0), and making sure you get back a result of (1,0,0). You can also try your sphere-intersection helper function, and check that you get both (1, 0, 0) and (3, 0, 0) as the results.

Once you are reasonably convinced that your code works properly, submit a tarball of your work on csman.

Last updated January 30, 2008.