Reflection and Cylinders

CS11 Advanced C++ Lab 7

This week you will have the opportunity to add reflective objects to your ray tracer. Also, you can add a new scene object type to the ray tracer using the same mathematical operations you will use for reflection computations.

Preparation for Reflection

It is actually surprisingly simple to implement reflection in a ray tracer. However, you do need one more vector math operation that you probably haven't written for your vector class yet, and that is the ability to project one vector onto another vector. If you have two vectors **A** and **B**, you can project **A** onto **B**, producing a new vector that is parallel to **B**, but with a length based on **A**'s length and the angle between **A** and **B**. Although this sounds complicated, it's actually very simple to define in terms of the dot product:

```
// Project A onto B
project(A, B) = B * ((A · B) / (B · B))
```

Notice that ((**A** · **B**) / (**B** · **B**)) produces a scalar, which is then used to generate a properly scaled version of **B**.

Reflection

Reflection is simply the situation where an incident ray bounces off of an object and then goes on to other objects in the scene. Of course, this doesn't happen with *all* objects in the scene; only reflective objects should have this characteristic. So, the first step in implementing this feature is that objects should have a scalar "reflectivity" value. This is a floating-point value in the range [0, 1]; 0 means "no reflection at all," and 1 means "completely reflective."

In a more general purpose ray tracer, this reflectivity value would go into a material object that is associated with each scene object. However, for your simple ray tracer, you can just leave this value in the scene-object itself for the time being. You can always refactor this appearance information into a `Material` class later on, along with the object's color and other visual characteristics.

For objects that have no reflectivity, it's business as usual; your current ray tracing code should handle those objects perfectly fine. But, for objects with a nonzero reflectivity, it is pretty straightforward to add the reflection computation, using the project operation you have implemented. Specifically, you can use the project operation to generate a "reflection" ray from the intersection point and surface normal of the object that was hit, as well as the original incident ray.

The computation goes like this:

```
$R_i$ = P + D * $t$      // Incident ray
X = point of intersection on object
N = surface normal at this point

// Compute direction of reflected ray using incident ray
$D_{par}$ = project(-D, N)
$D_r$ = D + 2 * $D_{par}$

// New reflected ray
$delta$ = 0.0001
$R_r$ = (X + $D_r$ * $delta$) + $D_r$ * $t$
```

Notice that our reflected ray doesn't start exactly at the intersection point **X**; rather, we add a small delta value to it. This ensures that the new ray doesn't immediately intersect with the object that it is reflected from! To avoid this, we simply nudge the reflected ray's origin slightly, in the direction that the new ray will travel. (Note that this operation, "reflect a ray off a position with a surface normal," would be a good addition to your `Ray` class, so that this functionality is encapsulated as a member function that can be easily reused. You can even make the *delta* value an argument to this function, that defaults to 0.0001, if you like.)

Now that you have a newly reflected ray, the next step is to find out the color for that ray. So, your `traceRay()` function needs to recursively call itself with the new ray to find out the color that the new ray will generate. To keep this recursion from going on indefinitely, you need to add a "recursion depth" value to the `traceRay` arguments. You can set a default value so that you can call the function without specifying the depth in your scene-scanning code.

For example, if your old trace-ray function was like this:

```
Color Scene::traceRay(Ray r)
```

You could update the function declaration like this:

```
Color Scene::traceRay(Ray r, int depth = 0)
```

The definition of the function will include the new depth argument, but it doesn't need to state the default; the default only appears in the function declaration.

For each reflection, the depth value should be incremented by one. When your recursion depth hits a certain maximum value, you shouldn't reflect the ray anymore; just report the color for the ray. A good value to try for the depth limit would be in the range of 6-10.

Lighting And Colors for Reflected Rays

For reflective objects, you must update the color computation code to add in the color of the reflected ray. The reflection color can be added in to the surface color computed so far, like this:

```
ray = the ray being tested
object = object that was intersected by the ray

intersection = point of intersection on the object
normal = surface normal at intersection point
result_color = result of lighting calculations computed so far

// If the object is reflective, and you haven't hit the maximum
// reflection depth, compute the new ray and then trace it here.
reflected_ray = ray.reflect(intersection, normal);
reflection_color = traceRay(reflected_ray, ray_depth + 1);
result_color += object.getReflectivity() * reflection_color;
```

Note that the above pseudocode doesn't include any of the tests, but it should give you a good outline of how to attack the problem.

Trying It Out

Once you have your reflection implementation done, you can try various nifty scenes such as multiple mirrored spheres next to each other. (You might take the original test scene, and set the balls' reflectivity to 0.7, and the plane's reflectivity to 0.1. That looks pretty nice.) Just make sure that you don't set your maximum reflection depth *too* high, or it could take a while.

Cylinders

Once you have the project operation implemented, you can also use it to implement a new scene object for your ray tracer: a cylinder that can be oriented along any axis. The cylinder object will require these data members:

- A position value, which specifies the center of the cylinder.
- A unit-vector that specifies the orientation of the cylinder's long axis. This vector can point in any direction, but it should be unit length.
- A radius value, which is obviously a scalar.
- A height value, again a scalar, which specifies the total height of the cylinder.

Performing intersection tests against a cylinder with an arbitrary orientation could conceivably be very expensive, except that you can actually cheat! What you do is to take the ray and transform it into a coordinate system where the cylinder is oriented along the +Y axis. Then you can just use the sphere intersection code, since you implemented the general-purpose intersection function. Just construct a sphere whose radius is the same as the cylinder's radius, and with the sphere's center set to a particular location in the X-Z plane:

```
$\mathbf{C}$ = cylinder center
$\mathbf{A}$ = cylinder axis (unit vector)

$\mathbf{P}$ = ray origin
$\mathbf{D}$ = ray direction (unit vector)

// Compute parallel and perpindicular components for various vectors

$\mathbf{C}_{\mathbf{par}}$ = $\mathit{project}(\mathbf{C}, \mathbf{A})$;
$\mathbf{C}_{\mathbf{perp}}$ = $\mathbf{C} - \mathbf{C}_{\mathbf{par}}$;

$\mathbf{P}_{\mathbf{par}}$ = $\mathit{project}(\mathbf{P}, \mathbf{A})$;
$\mathbf{P}_{\mathbf{perp}}$ = $\mathbf{P} - \mathbf{P}_{\mathbf{par}}$;

$\mathbf{D}_{\mathbf{par}}$ = $\mathit{project}(\mathbf{D}, \mathbf{A})$;
$\mathbf{D}_{\mathbf{perp}}$ = $\mathbf{D} - \mathbf{D}_{\mathbf{par}}$;

// Once you have these values, find the $t$ values of intersection
// using the ray $\mathbf{P}_{\mathbf{perp}} + t * \mathbf{D}_{\mathbf{perp}}$, against a sphere located at
// $\mathbf{C}_{\mathbf{perp}}$ with radius $r$ (the cylinder radius).

...
```

First important note: This code creates a test-ray for computing the intersection test, but you *must not* normalize this ray's direction vector. If the test-ray's direction vector is normalized then the resulting *t* value will have a different scale than all the other *t* values you use for finding the nearest object, and that wouldn't be good. (If you followed the earlier advice to give your `Ray` constructor a "normalize" flag, then this will be a piece of cake.)

Once you have the intersection test completed, you can take all (one or both) of the sphere's *t*-values and compare them to the cylinder's height value. If all intersections are beyond the cylinder's height, you can report "no intersection." Check the *t* values like this:

```
| $\mathbf{P}_{par}$ + $\mathbf{D}_{par}$ * t - $\mathbf{C}_{par}$ | < h / 2
```

Second important note: Because you are transforming the "real" ray into a "test" ray, you can easily have the case where the "closest" sphere intersection point is outside of the cylinder's height, and the second "further" intersection is actually the one you want. Therefore, you need to test all of the sphere-intersection values, rather than just the closest one. Otherwise things will look really weird and you won't know what's going on. (This is why the general sphere-intersection code was supposed to return *both* t-values, not just one.)

Surface Normal of Cylinders

Once you have computed an intersection point **X** on a cylinder, you need to find the surface normal of that point. This can be computed with the following equation:

```
V = X - C
V_perp = V - project(V, A)      // Project V onto A
n(X) = V_perp ÷ | V_perp |
```

In other words, take the vector between the intersection point and the cylinder's center, and subtract out the component of that vector that is parallel to the cylinder's axis. Obviously, removing that will produce the component that is perpindicular to the cylinder's axis. Then, normalize that result and you have the surface normal.

Scene Description Updates

Once you have reflection and cylinders implemented, you should also update your scene description language to support each of these features. It should be easy to add a "reflectivity" value to each object's description; you could put it right after the color value for each object.

Cylinders are a little more complicated because they actually have four values to specify: the position of the cylinder, a vector specifying the cylinder's axis, the radius of the cylinder, and the height of the cylinder. This shouldn't be too hard since you have stream-input support for your vector type by now. Also, don't forget the color and reflectivity value for cylinders as well.

All Done...

When you are finished, leave your work in your `~/cs11/advcpp/lab7` directory. Of course, update your Subversion repository, as usual.