Scene Descriptions

CS11 Advanced C++ Lab 6

Your raytracer should be functional at this point, allowing you to render very simple scenes including point-lights, spheres, and planes. The main problem with this is that you still have to specify a scene by programmatically constructing it, and that gets old pretty fast. This week you will add a very simple scene description language to your raytracer using C++ stream IO, so that your raytracer can take a scene description as a text file.

(As always, this week's work will go into your `~/cs11/advcpp/lab6` directory.)

Basic Data Types

As with all large programming tasks, you should always start with the simplest problems first, and build up to the more complex. So, your first task will be to implement stream-input operators for your vector and color classes. The stream-input operator must follow this calling convention:

```
istream & operator>>(istream &is, T &var)
```

The stream-input operator must be declared as a non-member operator overload. Ideally the public interface of type `T` will allow the stream-input operator to use mutators to store the values, but if this is not the case then the stream-input operator must be declared as a friend of type `T`. The value `var` will receive the values read off of the input stream, so of course it must be a non-const reference.

Add stream-input operators for both the vector and color data types. You should choose a simple format so that the code is reasonably easy to write; however, you need to use some kind of enclosing characters. For example, you might read vectors in this format:

```
(3.3, -2.5, 4.1)
```

Likewise, you might read colors in this format:

```
[0.5, 0.5, 0.9]
```

Make sure to update the stream state properly, in case bad input is handed to your program. You will need this to detect when a scene description is invalid. Also, you should make sure to handle leading or trailing spaces around any of the internal values, just to make it easier to write scene descriptions. (If you use the C++ stream-IO support for the primitive data types, following the example in the lecture notes, this will happen automatically.)

The Camera, Scene Objects and Lights

Once you have stream-input for the basic data types, use this functionality to add a simple scene description language to your raytracer. And "simple" is the key here; don't make your life too hard! One suggestion is to use a single line to describe each object, something like this:

```
camera (-1.5, 1, 3) (-0.3, 0.5, 0) (0, 1, 0)

light (-10, 10, 5) [0.8, 0.8, 0.8]
light (5, 3, 5) [0.3, 0.3, 0.3]

plane  (0, 1, 0) 0 [0.5, 0, 0.5]

sphere (-1.2, 0.5, 0) 0.5 [1, 0, 0]
sphere (0, 0.5, 0) 0.5 [0, 1, 0]
sphere (1.2, 0.5, 0) 0.5 [0, 0, 1]
```

Here is what each line means:

- The first line describes the placement of the camera, and consists of three vectors. The first vector is the camera position, the second vector is the "look-at" position, and the third vector specifies what is "up" for the camera.
- The next two lines describe lights. The first value for each light is the position of the light, and the second value is the color of the light.
- The third line describes a plane, with its surface normal and distance from the origin, and the color of the plane.
- The remaining lines describe three spheres. For each sphere, the values are the center and radius of the sphere, and then the color of the sphere.

(You may notice that this is the same test-scene specified for lab 4.)

Of course, you can construct a language that is a little different from this, or even a lot different if you want. Just make sure to keep it simple enough that it isn't too hard to implement.

Reading Scene Descriptions

The common theme of the above format is that each object is contained on a single line. The first value on the line describes the kind of object being read, and subsequent values are specific to that particular kind of object. Thus, you can set up your code to work something like

this:

1. Initialize an empty scene.
2. Read an entire line from the input stream, until there are no more lines to read.
3. Get the first value from the line.

   (You could do this by converting the string into an input stream.)
4. If the value is "light", create a light and add it to the scene.
   Otherwise, if the value is "plane", create a plane and add it to the scene.
   (etc.)
5. Go back to step 2.

Finally, you should check the scene, and report an error if there are no lights or no objects in the scene. (That would be an expensive way to generate a black box...)

Extensible Scene-Import Code

You could of course hard-code all of your tests into one large if-block that handles each line of the scene input, but your intuition would probably tell you that this is not going to be ideal in the long run. Specifically, if you want to add other kinds of lights or scene-objects *(hint hint)*, you would have a big headache to deal with.

A more extensible approach would make it easier to plug in new object types in the future. One idea to achieve this would be to have a mapping from object type strings, to a function that constructs that kind of object. For example, you might have something like this:

```
typedef boost::shared_ptr<SceneObject> SPSceneObject;

// Define a type for functions that take an input-stream, and construct a
// SceneObject from them.  (Don't forget the * even though it isn't strictly
// required by the C++ syntax for function pointers!  If you leave it out,
// the g++ STL implementation gets very crabby.)
typedef SPSceneObject (*SceneObjectReader)(istream &is);

// Functions for reading different kinds of scene objects, with same signature:
SPSceneObject ReadPlane(istream &is);
SPSceneObject ReadSphere(istream &is);

...

map<string, SceneObjectReader> readFuncs;

readFuncs["plane" ] = ReadPlane;
readFuncs["sphere"] = ReadSphere;
```

Then you can use the first value in each line of the scene description file to determine what function to call for that line, then pass the rest of the line to the function. Since all the functions have the same signature, you can write some pretty generic code to process scene objects from the input text:

```
string type;

// "is" is the input stream being read
is >> type;

if (readFuncs.find(type) != readFuncs.end())
{
  SPSceneObject newObj;

  // Retrieve and invoke the function.
  newObj = readFuncs[type](is);
  scene.addObject(newObj);
}
else if (type == "light")
{
  SPLight newLight;

  newLight = ReadLight(is);
  scene.addLight(newLight);
}
else if (type == "camera")
{
  ... // TODO:  Initialize the camera.
}
else
{
  ... // TODO:  Write error-reporting code.  :-)
}
```

Of course, cameras and lights aren't scene-objects! This means you do have a couple of special cases. This isn't too bad. (EXTRA CREDIT,

NOT REQUIRED: You could always create an `RTObject` "raytracer object" base-class, and derive `SceneObject`, `Light` and `Camera` from it, so that you can eliminate these special cases from your parsing code. However, only do this if you really want to; *it's not required!*)

Pulling It All Together

Notice that so far our stream input code is pretty generic; it isn't coded to only work with `cin`, for example. You should be very careful to make your code generic so that you can easily use it to read from `cin`, or from a `ifstream` or an `istringstream`, or whatever.

For this week's assignment, you should go ahead and update your `main` function to read the scene description from `cin`. (You can also have it read from an input file instead, but you must be sure to display usage information so that users of your raytracer can figure out how to use it.) After the scene description has been successfully read, go ahead and render the scene.

Once you have this working, you can create scene descriptions and then raytrace them! For example, if you stored your scene description in a file named `scene.txt`, you could render the scene into `image.ppm` with this command:

```
raytrace < scene.txt > image.ppm
```

That command redirects the contents of `scene.txt` into your program's standard input (i.e. `cin`), and the output of the program (i.e. `cout` will be stored into the file `image.ppm`.

Give your code a try, and make sure to test it with some "bad" scene descriptions as well, to make sure that the error handling works properly too. Once this is all finished, leave your code and your scene descriptions in your `~/cs11/advcpp/lab6` directory. Don't forget to update your Subversion repository too, of course.

Extra Credit!

Here are some additional things you could add to your scene description format:

- Update your scene-object input code so that it when it encounters an error, it actually reports what line number the error was found on. This is very helpful for figuring out how to get things working properly.
- Add support for comments. For example, you might ignore lines that start with the "`#`" character, or the character sequence "`//`".
- Add a command-line option `-f` *`filename`* to your program, that allows the user to specify a filename to read, instead of reading the scene description from standard input.

Last updated November 17, 2009.