# Ray Tracing Automation

## CS11 Advanced C++ Lab 4

This week the goal is to get the ray tracer to a point where it can render very basic scenes. Just as before, you should not work in your previous week's lab directory, but unlike before, we will begin to work with a Subversion repository to give us more power in managing our changes. Also, you will use the `make` utility to automate the build process.

## First Things First

Before really doing anything else this week, you should get a Subversion repository set up. This procedure should be relatively painless, if you follow the basic steps outlined in class. Here they are again:

1. Create a brand new Subversion repository using the `svnadmin create` command:

   ```
   svnadmin create ~/cs11/advcpp/svnrepo
   ```

   This will create a new Subversion repository in the `svnrepo` subdirectory of your CS11 Advanced C++ directory structure. *Use this repository location* so that we can also look at your repository.
2. Next, you need to import all of your ray tracer files into Subversion. Of course, before you do this you should get your files into a state that is worth importing, so create a temporary directory to hold the files you will import, and copy your files in from your `lab3` directory. Then, get rid of any object files (`*.o`), editor droppings (`*~`), any actual program binaries, etc. Also, if you want any particular directory structure, it's easiest to set this up right now too.

3. Once your files are ready for import, use the `svn import` command as shown in class. Remember that Subversion refers to repositories by URL, so you will need to use a `file://` URL for this. (You could also import to a remote repository using a `svn+ssh://` URL, but you will have to figure that out yourself.) You can run this command, from your temporary import directory:

   ```
   svn import file:///home/<username>/cs11/advcpp/svnrepo/raytracer
   ```

   *(Replace `<username>` with your own username, of course...)*

   This will recursively import *all* files in your local directory, into the repository. Note the "`raytracer`" at the end of the repository URL; this imports your ray tracer's sources into a "raytracer" project within the repository.

   **Don't delete the temp directory until you have verified that the import process completed successfully.**

   If Subversion complains about not being able to find an editor, you can shut it up by typing something like `export EDITOR=vi` before running your Subversion commands. (Replace `vi` with your preferred text editor, of course.)

4. Importing files from a directory does *not* automatically make that directory a working copy! So, after importing, you need to check out the repository's files into a local directory. Create a new `~/cs11/advcpp/lab4` directory, and do your checkout inside of this directory. Something like this:

   ```
   svn checkout file:///home/<username>/cs11/advcpp/svnrepo/raytracer
   ```

   This will create a local directory `raytracer`, containing the files you just imported. Check to make sure all the files you expected, actually showed up! Once you are sure everything is good, *then* you can go ahead and delete your import directory.

Once these steps are completed, you can go ahead and edit your working copy of the project without mangling the (hopefully) pristine source code in your repository. You can build it, test it, and make sure everything works properly before committing your work back to the repository.

As mentioned in class, you can issue an `svn commit` without specifying the repository URL because Subversion creates little `.svn` files in your working copy, that hold relevant details like which repository the files are from. (You can see them by typing "`ls -a`" in any directory of your working copy.)

If you want to work remotely, you can use a repository URL something like this:

```
svn+ssh://<username>@zappy.cs.caltech.edu/home/<username>/cs11/advcpp/svnrepo
```

This will cause the `svn` client to use the `ssh` client to establish a connection to the CS cluster, and access your repository. Secure and easy!

Remember that you can use `svn help` and `svnadmin help` for getting more information on what Subversion commands are available, and for looking at the details of specific commands.

## Real, Actual Ray Tracing

There is one more major feature to implement in your ray tracer, and that is a function that actually traces a ray in the scene to see what the color for that ray should be. You should implement this on your `Scene` class - you can call it `traceRay` or something similarly suitable. The function should take a single argument, a `Ray` to trace in the scene. The function's return-value should be a `Color` - the actual color value for that ray. This function should also be `const`, since it won't change the scene's contents.

The `traceRay` function's operation is actually quite simple:

1. First, you must find the object with the closest intersection point on the ray's extent. For each object in the scene, test the object for intersection, and get the *t* value for that intersection. As you go through all the objects in the scene, keep track of the closest object you have seen, and the *t* value of the closest intersection.

   **Hint:** You might want to create a special helper-function that takes a ray and returns the closest scene object and time of intersection. You can use it to implement some cool features very easily, in the upcoming weeks. The easiest and fastest way to construct such a function would probably be to have a function signature like this:

   ```
   SceneObject * Scene::findClosestObject(const Ray &r, float &tIntersect) const
   ```

   The return value indicates whether an intersection occurs or not. If the return value is 0 then there is no intersection. Otherwise, if the return value is not 0, it points to the closest scene-object, and `tIntersect` is also set to the time of the closest intersection. (`tIntersect` is another out-parameter. There are several ways to pass multiple values back to the caller, but this is a simple and effective mechanism.)

   You might implement `findClosestObject()` using a "functor with state" that you write to work with `for_each()`. Or, you might implement the loop yourself. It's up to you.

2. Once you have checked every object in the scene, you either have an intersection or you don't! If there was no intersection, go ahead and return a suitable background color. You might choose black, or you might want some other color.

3. If there *was* an intersection, you need to compute the actual color at the intersection point. This is a pretty simple procedure to implement. Here is a description of the process for a single light:

   - Compute a vector **L**, which is the light's position minus the intersection point, as a unit vector. In other words:

     ```
     L = normalize(light_loc - intersect_loc)
     ```

   - Compute another vector **N**, which is the surface normal at the intersection point. This should be easy, since each scene object knows how to compute its own surface normals.
   - The color at that location of the object is simply:

     ```
     FinalColor = LightColor * SurfaceColor * max((N · L), 0)
     ```

     In other words, the light's color is filtered by the surface's color, and the angle of incidence will attenuate the magnitude of the lighting. Also, remember that **N** · **L** will be negative if the angle between them is more than 90° so we set a lower bound of 0. The backs of objects won't be lit at all by lights.

   If you happen to have multiple lights, you can repeat this procedure for each light, and just sum up each color generated by each light. (Just change `FinalColor = ...` to `FinalColor += ...`) As long as the lights aren't overwhelmingly bright, this will work great.

   (The `cmath` header contains `fmin()` and `fmax()` functions for doubles, and `fminf()` and `fmaxf()` for floats. Also, you *could* implement the light computation loop with STL using a carefully constructed stateful functor, but the functor would need to include the intersection details since they are part of the lighting computation. Thus, it's probably easiest to just implement the loop manually.)

Once this function is completed, you are getting close to rendering your first scene. The only thing remaining is to write the code that scans through every pixel of the scene, shooting a ray through that pixel to determine its color.

## The Camera

The camera in a ray tracer works in a pretty simple way. It simply shoots a ray from a particular origin, through every pixel in a grid, and generates a color value for that pixel. Then, the value is written out to an image somehow.

Create a class to represent a camera in your raytracer. The camera will need the following data members:

- A vector specifying the camera's location in the scene.
- A normalized vector specifying the direction the camera is facing.
- A "field of view" value, specifying the horizontal viewing angle of the camera. This value can be in degrees or radians, but be sure to clearly document which units you choose! (Also, keep in mind that the standard trig functions use radians.)
- A "camera-up" vector and a "camera-right" vector - these are perpindicular to the camera's direction vector, and they fully specify the camera's orientation. These should all be normalized.
- A scalar "distance" value, which the camera computes to achieve the desired field of view.

### Camera Constructor

Your camera constructor can take a camera-position vector and a "look at" position vector - this is a much easier way to position the camera, rather than specifying the direction vector explicitly. The camera's direction vector simply becomes the difference between the look-at position and the camera's position, normalized of course.

Besides the camera position and the "look at" position, your constructor also should take an "up" vector. This vector doesn't have to be exactly perpindicular to the direction vector at all; for example, it could just be (0, 1, 0). What you need to do is to take this "up" vector and generate the camera-up and camera-right vectors:

1. Compute the normalized **direction** vector from camera position and "look at" position.
2. **cameraRight** = **direction** × **up** (using right-hand rule)
3. **cameraUp** = **cameraRight** × **direction** (again, using right-hand rule)

Then, compute the "distance" value, based on the field of view. The camera will effectively shoot rays through a 1×1 region, over the range [-0.5,-0.5] to [0.5,0.5]. To achieve the desired field of view, this region must be at a specific distance from the camera's position. This distance *dist* is computed as follows:

```
dist = 0.5 / tan(fov / 2)
```

Note that the field of view *fov* ranges across the entire horizontal extent of the scene. Also, keep in mind that the `tan()` function takes radians. (The `cmath` header defines `tan()` to take and return `double`s, and the `tanf()` function to take and return `float`s.)

Of course, the field of view should also be an argument to the camera constructor. You can make it default to 60 degrees, and give callers the option to specify a different value if they want. (A larger field of view causes images to be distorted near the edges. But, if you want to look through the eyes of small prey-animals, feel free to crank it on up to 120 degrees or something...)

Once you have all of these values computed, you can use them to generate all rays into the scene. So, make sure to store them as data members, so that you don't have to recompute them everywhere.

## Generating Rays

The scene object will handle the process of rendering itself, but it will use the camera object to create a ray for each pixel. So, create a function like this:

```
Ray Camera::getRayForPixel(int x, int y, int imgSize) const
```

For now, we will assume that the image has the same width and height, although if you want to make it support different image sizes, you are free to implement that.

Given a pixel coordinate and an image size, you can generate rays as follows:

```
// dist = distance between camera location and the grid of pixels, as
// computed earlier.  direction = normalized camera direction vector.
Vector pixelDir = dist * direction +
                  (0.5 - (float) y / (float) (imgSize - 1)) * cameraUp +
                  ((float) x / (float) (imgSize - 1) - 0.5) * cameraRight;

Ray pixelRay(cameraLoc, pixelDir);
return pixelRay;
```

Note that this code expects pixel coordinates to be at least 0, and strictly less than the image size. Those would be some good assertions to write...

There isn't really anything clever about this code. It is mainly complicated because of the casts from integer pixel coordinates into floating-point values for use with the vector arithmetic. The only other subtlety is that increasing pixel x-coordinates scan from left to right, but increasing pixel y-coordinates scan *from top to bottom*. This is because increasing Y-values in the scene's coordinate system go "up", but lines in the generated images go *down* as the Y-value increases.

# Rendering a Scene

Once you have your camera class, you can implement the function that renders the whole scene. You can actually implement the rendering function on the scene class itself - you could call it "render," for example. The render function should take the following arguments:

- A reference to a camera object for generating rays into the scene. This should be const.
- An image size in pixels. Again, we are simplifying the code by assuming that the width and height are identical.
- An `ostream`-reference to write the image data to. This will be where the actual image data is written to. This *cannot* be const.

To render the scene, you must generate a ray through each pixel of the image, into the scene, in order to generate a color for that pixel. This will use your scene's "trace-ray" function extensively. What you want to do is to set up a loop that iterates through all pixels, like this:

```
// This code is rife with opportunities to optimize...
for (int y = 0; y < imgSize; y++)
{
  for (int x = 0; x < imgSize; x++)
  {
    Ray pixelRay = camera.getRayForPixel(x, y, imgSize);
    Color pixelColor = traceRay(pixelRay);
```

```
        ... // Output color value to output stream, in proper image format.
      }
    }
```

That's just about it! Of course, there is still one sizable hole in the above code - how to write each pixel's value to an image file. But that turns out to be pretty easy - just read on...

### Writing Out the Image

To keep the ray tracer simple, you can use the very simple and verbose ASCII PPM (Portable PixMap) image format. This format is as follows:

```
P3 width height maxval
red1 grn1 blu1
red2 grn2 blu2
...
```

The `P3` value is a "magic number" indicating that the image format is ASCII PPM. The `width` and `height` values are the image width and height in pixels. The `maxval` is the maximum color value; use 255 for your raytracer.

The rest of the file contains the color data for each pixel in the scene, in ASCII text format. That is, for a purple pixel, you would actually write out the string "`127 0 127`" to the file. One important requirement is that each line should be no more than 70 characters long. If you just write each pixel on its own line, you can easily avoid any issues here.

The other important caveat is that all color values must be in the proper range for the pixmap format; this is [0, *maxval*], which you specify in the PPM header to be 255. Failure to do so will confuse PPM parsers, and make things break in nasty ways.

Ideally, all components of your color values will be in the range [0, 1], but if they aren't, you should clamp them to be in that range. An easy solution would be to write a `clamp(minVal, maxVal)` function on your `color` class to call for this very situation. Then you could simply do something like this:

```
pixelColor *= 255;
pixelColor.clamp(0, 255);
... // output the pixel's components as integer values
```

You can dump the image data to the passed-in `ostream`-reference. This way the caller of the render function can either pass in `cout`, or a specific file output-stream. Note that when you create files of this format, use `.ppm` for the extension.

Once you have ASCII PPM images, you can convert them to a more "normal" format using, for example, `pnmtojpeg` for JPEG images, or `pnmtopng` for PNG images. Or, you can use the GIMP to open your image data.

Just make sure not to go over-quota on your CS account... Of course, as inefficient as ASCII PPM images are, they compress *really* well, especially with `bzip2`.

## Test Scene

Once you have your camera code all completed, you are ready to try your ray tracer on a really simple test scene. Create a file to hold your `main()` function, e.g. `rt.cc` for "raytracer." For now you will have to programmatically set up the scene. You can make something like the following:

- A plane through the origin: distance = 0, surface normal = (0, 1, 0). Set its color to [0.5, 0, 0.5], or purple.
- A sphere with center (-1.2, 0.5, 0) and radius 0.5. Set its color to red [1, 0, 0].
- A sphere with center (0, 0.5, 0) and radius 0.5. Set its color to green [0, 1, 0].
- A sphere with center (1.2, 0.5, 0) and radius 0.5. Set its color to blue [0, 0, 1].

You also need some lights:

- Add a light at position (-10, 10, 5) with color [0.8, 0.8, 0.8].
- Add a light at position ( 5, 3, 5) with color [0.3, 0.3, 0.3].

Finally, configure the camera with these parameters:

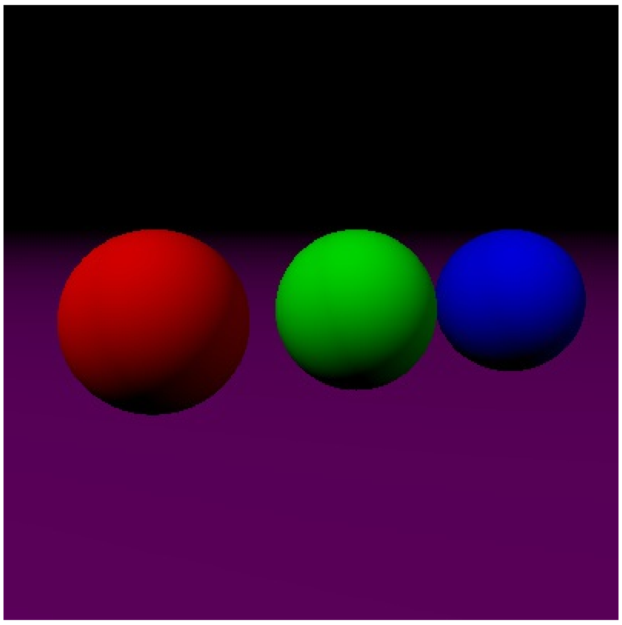
- Camera position: (-1.5, 1, 3)
- Look at position: (-0.3, 0.5, 0)
- Up vector: (0, 1, 0)

Once you have your scene initialized, you can render it, passing `cout` as the output-stream to write the image data to. Compile your program, and then you can run it like this:

```
./rt | pnmtopng > balls.png
```

This takes the "standard output" from your program and redirects it to the "standard input" of the `pnmtopng` program, which will convert the

ASCII PPM data into a PNG image file. If you did everything correctly, you should end up with an image like the following:



If this is what you have, congratulations!

## All Done, *Finally*...

Once you have your raytracer working, **don't forget to commit all changes to your Subversion repository!** Specifically, use the `svn status` command to make sure you add all new source and header files you created this week. (Don't add binary files, .o files, or other generated results. Also, there is no need to add your raytraced images to the repository, unless you want to save them for posterity...)

When you are done, submit one file `readme.txt` to csman, specifying the URL of your Subversion repository. If you have created it on the CS cluster as specified, we will be able to retrieve your work directly.

---