CS 465 Program 5: Ray II

out: 14 November 2004
due: 29 November 2004

1 Introduction

In the first ray tracing assignment you built a simple ray tracer that handled just the basics. In this assignment you will build a more capable ray tracer that can handle more substantial models and can produce much more interesting renderings.

This assignment is rather open-ended relative to the earlier ones. You are encouraged to start from your Ray I solution, or the solution we provide, but you are free to design and implement the extensions in any way you like. You also choose which extensions to implement.

2 Requirements

Your ray tracer will read files in a standard file format and output PNG images (like the first ray tracer). It has to support the basic features given below, plus several extensions that you choose from a list.

2.1 Framework

The frame work for this assignment is a slightly extended version of the solution to the first ray tracing assignment. The solution code contains the following elements that were not in the original solution:

1. The code has been changed to read the Exported files produced by the Modeler assignment. This changed is completely contained in the `Model` class that has been added to the `mesh` package.
2. The `MeshTriangle` intersection method was changed slightly to correctly handle the degenerate triangles produced as part of many Modeler meshes. This changed only part of the `MeshTriangle` intersection method.
3. A new material called Textured Phong contained in the `TexturedPhong` material class to support the textured objects from the Modeler
4. The `math` package has implementations of 2D points and vectors to support texture coordinates.

5. An implementation of basic sub-pixel anti-aliasing with jittered samples in the `renderBlock()` method of the `RayTracer` class.

You should feel free to use either your solution or the provided solution as you see fit. However, if you use your own you will have to incorporate these changes into your code as part of this assignment.

2.2 File format

This assignment will require you to make several extentions to the existing code. Of course, you will need to be able to make test cases that can exercise the new features you will be adding. The framework's `Parser` class is designed to support this type of extension without change, but it requires that you implement the new features in a certain way. The requirements are:

1. Any class that will be instantiated by the Parser must implement a public constructor that takes no arguments.
2. Any class described by a block of xml that includes sub-tags must have public methods called either `setXXX()` or `addXXX` where XXX is the exact name of the sub-tag used in the description. These methods must take exactly one argument. The data will be parsed as if it is the same type as the argument. The `Parser` can correctly parse all primitive types, `String`s, `Color`s and sub-classes of `Tuple3`.

For example, if you wanted the following input to parse correctly:

```
<foo>
    <bar>
        <cat>Lucky</cat>
    </bar>
</foo>
```

You would need to have classes with the minimum definitions:

```
class FooClass {

    public FooClass() {}

    public setBar(BarClass inBar) {
        //Do something with inBar
    }
}

class BarClass {

    public BarClass() {}

    public setCat(String inName) {
        //Do something with inName
    }
}
```

Finally the most common case is that you will be adding new materials or lights. In this case, in addition to the requirements above, the new classes should be children of `Material` or `Light` and their input tags should use the type argument. For example:

```
<material type="ray1.material.MyMaterial">
</material>
```

is the correct way to specify a new material sub-class named `MyMaterial`.

There is a more detailed description of the Parser in comments contained in the header of the Java file.

2.3 Basic features

Your ray tracer must implement the following features beyond what the first ray tracer did:

1. An acceleration structure. Your program must be capable of rendering large models (up to several hundred thousand triangles) with basic settings in a few minutes. Achieving this requires a spatial data structure that makes the time to trace a ray sublinear in the number of objects. We recommend implementing an axis-aligned bounding box hierarchy (AABB), which is a simple and effective way of speeding up ray traversal. To implement an AABB you will need to do the following:

   (a) Create a class that defines an axis aligned bounding box. Design the class as you feel appropriate, but you will likely, at minimum, need methods to: determine if the box intersects a ray, determine if the box intersects another box, grow the box to include a point, and grow the box to include another box. You will also need to extend the `Surface` class to include a method that can grow a bounding box to include each surface type.

   (b) Design a class to represent a node in the heirarchy. Each node should have a pointer to a bounding box object, pointers to two children and a list of pointers to objects contained in the box.

   (c) Implement a method of building the heirarchy. The most straight forward approach is to create an AABB that encloses all the objects in the scene and then create a bounding volume node for this box and that includes all objects. Then recursively split this box and its children until the number of objects in each node is less than a constant (usually around 10). To split a node, choose an axis to split along, sort all the objects in the box along the axis and put each half in each of the children. The sort will require that you implement a method of sorting `Surface`s. A good method is to sort by an approximation of their center: the center for Spheres and the average of the vertices for triangles.

   (d) Implement a method of traversing the heirarchy and finding the first object intersecting a ray. The method is described both in the lecture notes and in Shirley Chapter 10.

2. Adaptive sampling of the image using a quadtree sampling pattern. Rather than always tracing one ray for each pixel, your program will start by tracing one ray every few pixels, then recursively adding more rays where it detects additional detail. Thus you will trace many less than one ray per pixel in smooth, uninteresting areas of the image, and more than one ray where there are small features (such as edges) that require antialiasing.

   The adaptive sampling will be implemented in two stages performed once each for each sub-block of the image. First, you will build a quadtree covering the block's area in the image and then you will use the quadtree to calculate the values of the output pixels. To help you get started, the framework includes an abstract class that defines the members and principal methods to a quadtree. It also contains a method called `draw` which will produce a picture of the current quad tree. This can be fed to a `QuickViewer` object for debugging. You should refer to the class for more information; however we supply a short description of the tree here.

   The quad tree should be implemented as a linked heirarchy of nodes. Each node will represent a square block of the image. We recommend that you represent the image as covering the unit square $[0,1] \times [0,1]$. This will make the quad tree implementation considerably simpler because it will not have to be concerned with actual pixel positions or image sizes. Using this representation, each node will store the bounds of the cell in these canonical image coordinates, 4 color samples computed at each of the corners of the node's block, and 4 pointers to the children of the cell. The 4 children of a node are created by connecting the center of each block to the midpoints of each of the sides.

   To create an adaptive quadtree sampling of the image, you will need to start by creating a single node the size of the current image block. You will then recursively sub-divide this node, creaing new samples for the newly created children, until each node in the tree satisfies a error criterion. A good criterion is to stop when the difference between the minimum and maximum samples is less than some fraction of the average (say 5%) or if the cell is at some maximum depth in the tree. Since the samples are colors, you should compute the maximum difference among the three channels when checking the criterion. Finally when sub-dividing you will need to make sure that you do not duplicate samples for cells that share an edge (since you would waste a lot of time ray tracing samples twice). The best way to avoid this is to use the quad tree itself: before you create a sample, query the tree (see below) for the sample you would create.

   Finally you will need to use the tree to compute the output image. For this you will need to be able to use the tree to estimate the color of an arbitrary point inside the area covered by the tree. This query can be done recursively. For each call, check if the current node is a leaf node. If it is, compute the color of the output point by bilinear interpolation of the 4 corners of the cell. If it is not, determine which of the 4 children contains the point and recursively query that cell. The query mentioned above for avoiding sample recomputation is similar, but it should only check whether the query point is exactly one of the samples in the leaf cell; otherwise it returns that the sample was not found. For each pixel you should sample several points using a procedure very similar to the jittered sampling that is already implemented.

2.4 Extension features

Your ray tracer must also implement enough extensions from the following list to add up to 3 units. Features that are likely to take more time are assigned 2 units, and ones that require less effort are assigned 1 unit.

1. *Acceleration via k-d tree* (2 units). Instead of using an axis-aligned bounding box hierarchy, which is the easiest to implement but by no means the fastest spatial data structure, implement a k-d tree. For most models this will perform considerably faster than the bounding box hierarchy. Vlastimil Havran wrote a Ph.D. thesis comparing many different ray shooting algorithms. It is quite an involved work and most of it is completely unrelated to this assignment. However, he gives pseudo-code for the kd-tree traversal algorithm in Chapter 2 and several options for splitting criteria in Chapter 4. We strongly suggest that you stick with straighforward criteria, such as median splitting, when implementing your tree. The thesis can be found here: `http://www.cgg.cvut.cz/˜havran/DISSVH/dissvh.pdf`

2. *Distribution ray tracing* (2 units). Now that we have made the ray tracer much faster, we should do something nifty with all the extra rays we can trace. Using distribution ray tracing with random sampling, add soft shadows, glossy reflection, and depth of field to your ray tracer. Each image plane sample will now be rendered using a large number of random rays that are distributed to achieve particular kinds of effects. Details:

   - Area light sources: generalize your point light source by adding a radius, and treat the light source as a disc that faces the shading point. This approximates a spherical source but is much simpler.
   - Glossy reflection: generalize your glazed material by randomly perturbing the surface normal by a vector in the plane perpendicular to the normal. Choose the perturbation uniformly from a circle, or do something smoother (see Shirley's later chapters).
   - Depth of field: generalize your camera by making it generate rays with randomly chosen origins, but always pointing toward the appropriate point on the camera's focus plane.

   These features require a few optional parameters in various places: lights should have a "radius" parameter; glazed surfaces need to have a normal perturbation angle "angle"; cameras need to have an aperture diameter "aperture" and a focus distance "focus". Counter to the tradition in photography, measure the focus distance from the viewpoint rather than from the film plane, because it's simpler that way.

   It's fine for you to use independent random samples to generate your points and directions. But you will consume less CPU time if you use jittered samples instead—but watch out to make sure they are not correlated with the image plane samples. See Shirley, ch. 20 and the solution's simple jittered anti-aliasing.

3. *Dielectrics and glazed materials* (1 unit). Implement a dielectric interface material that represents an interface between air (on the outside, the side toward which the normal points) and a denser dielectric. You should also implement a "glazed" material that acts like a thin layer of dielectric over another material—that is, it behaves like a dielectric, but rather than computing and tracing a refracted ray it just calls another material and scales its contribution by the Fresnel transmittance.

   A shader that uses recursively computed rays means that your renderer will generate a tree of rays, which needs to be pruned to keep the program from becoming too slow. In addition to the maximum-depth cutoff, you should also implement a maximum-attenuation cutoff by keeping track of how much a given ray will contribute to the image (i.e. what is the factor it is being multiplied by before it is added to the image). When that factor drops below a user-determined threshold, you should terminate recursion.

In the input file a dielectric material is specified just like the other materials; its single parameter is the index of refraction. For instance, glass could be specified as.

```
<material type="ray1.material.Glass">
    <eta>1.5</eta>
</material>
```

The glazed material is the same but also expects to see another material for its substrate, as its second parameter:

```
  <material type="ray1.material.Glazed">
    <eta>1.5</eta>
    <rho>0.4 0.5 0.8</rho>
  </material>
```

4. *Cube-mapped backgrounds* (1 unit). A ray tracer need not return black when rays do not hit any objects. Commonly, background images are supplied that cover a large cube surrounding the scene. The directon of rays that do not intersect objects are used to as indices into these images and the color of the image in the rays direction is returned rather than black. The techinique is commonly called cube-mapping. To implement cube-mapping in you ray tracer you will need to extend the `Scene` class to contain an image used as the cube map background. You will also need to write code that maps ray directions into cube-map pixels. A short introduction to cube-maps can be found here `http://panda3d.org/wiki/index.php/Cube_Maps` and many actual maps can be found here `http://www.debevec.org/Probes/`.

5. *Fancy lights and materials* (1 unit). Extend your point light source so that it can be switched between the default $1/r^2$ falloff of brightness, $1/r$ falloff, and no falloff (brightness independent of distance) and can also be used as a spotlight. For the spotlight, the input file specifies a direction and a Phong-like exponent; the intensity of the light is then scaled by $\cos^n \theta$ where θ is the angle between the spotlight direction and the shadow ray.

   In the file format, add optional parameters "falloff" (a boolean), "direction" (a vector), and "exponent" (a float).

   Also implement Ward's illumination model, described in the paper you can find in the readings on the course web site. Implement the isotropic version.

6. *Propose your own* (1 or 2 units). You can propose your own extension based on something you heard in lecture, read in the book, or learned about somewhere else. Doing this requires a little extra work to document the extension and come up with a good test case. If you want to do your own extension, email your proposal to the course staff list by 8am on Monday November 21.

3 Handing in

When you hand in your ray tracer, in addition to the code you need to hand in input files that demonstrate its abilities. A fraction of the grade for this assignment will be set aside for the quality of your test cases: do they test your features well, so that we can tell for sure that they work, and do the images just look nice. None of the test images should take more than about 10 minutes to compute on a recent PC (such as the ones in the lab). You are required to submit at least 2 test input files for each of the items you implement above. (Not assignment units. You only need two inputs for a 2 unit item.)

Also hand in a text file (a page or so) with simple user documentation that explains how to use your program. For example, we need to know how to set any options or parameters that are not set through the input file, and we need to know about any extra extensions you made to the file format.

Finally, hand in one image, rendered at high quality and at high resolution (1280 pixels across) that shows off the best your program can do. Make the model interesting, and make the image aesthetically pleasing. We will award 10 extra credit points to the best image (on combined technical and aesthetic grounds) we receive, and 5 points to each of two runners-up.