CS 4620/5620 PA 3: Advanced Ray Tracer

out: Tue 22, Nov 2011
due: Thu 8, Dec 2011 (due at 10am; demos from 11am scheduled on CMS)

1 Introduction

The first ray tracing assignment (PA1) introduced you to the basic principles behind this relatively simple and at the same time very powerful algorithm for rendering images. In the current assignment (PA3) we will add a "small" enhancement to the previous algorithm, tracing multiple light bounces, which will allow us to render essential physical effects, that were impossible to produce before (e.g., reflection and refraction of light).

You will quickly realize that tracing multiple ray bounces fits quite nicely in the provided framework and does not require much of an effort to implement, but it also reveals the major problem behind this approach, its performance. Therefore, one of your goals in this assignment will be to implement a hierarchical data structure, that will accelerate the bottleneck step in ray tracing, which is the ray-object intersection.

Being able to produce more physically plausible effects, we want to give voice to your creative side, by running a final "Rendering Contest". You should create a custom scene and render it using your own ray tracer implementation. There will be bonus points awarded to the "best image" and the two runners-ups we receive (on combined technical and aesthetic grounds). We will judge this during the demos on Thursday Dec 8th.

Due to this semester's tight schedule, a more populated framework code is provided to save you from spending time on class hierarchy design, and rather to have you focus on implementing the core ray tracing components. However, you also have the freedom to redesign the system as long as your ray tracer meets our requirements and produces the same images for given scenes.

2 Requirement Overview

Your ray tracer will read files in an XML file format and output PNG images.

You have already implemented the "basic ray tracing" and now you will build on top of your previous code, to create more "advanced ray tracing features". You should reuse your own part A implementation, by copy-pasting your PA1 implementation at all places marked with `TODO(A)`. You don't have to place any code for the cone (Cone.java), as we will not test it in the current assignment. In this document we will focus on what you need to implement for part B.

In addition to part A, your ray tracer should support the features given below.

1. *Advanced shader*. A "Glazed" shader that acts like a thin layer of dielectric over another material. Glazed shader also calls another shader which computes the contribution from the substrate below the glaze.(Shirley 13.1)

2. *Advanced shader*. A "Glass" shader that simulates an interface between air and a dielectric material. The glass shader traces two different paths, for the reflected and refracted rays, and adds their corresponding contributions. (Shirley 13.1)

3. *Group and transformations*. You should be able to transform any object using translation, rotation, and scaling. In a XML file, your scene will be represented as a tree structure. The leaf nodes are the actual surfaces that will be rendered in the scene such as spheres, boxes, cylinders, or triangle meshes. The inner nodes or non-leaf nodes are represented by a class called "Group" which can have multiple children and can contain a transformation. This transformation is specified as a sequence of rotations, scales, and translations, which are combined in the order to define the transformation that is applied to all children of the group. The transformations will be applied from the bottom to the root of the tree.(Shirley 6.2, 13.2)

4. *An acceleration structure*. Your program should 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. In this assignment, we provide a framework for axis-aligned bounding box hierarchy (AABB) which is a simple and effective way of speeding up ray traversal. However, you can also implement your own AABB from scratch, or if you want to implement other kind of acceleration structure, please ask the course staff. (Shirley 12.3)

5. *Antialiasing*. You should implement antialiasing using regular supersampling. (Shirley 8.3, 9.4, 13.4)

6. We have provided a list of extensions that can be implemented for extra credit.

3 Implementation

3.1 Part B

We have marked all the functions or parts of the functions you need to complete in part B with `TODO(B)` in the source code. To see all these TODO's in Eclipse, select *Search* menu, then *File Search* and type "`TODO(B)`" The following explanations are to give you a rough overview of the tasks. Most instructions, however, are provided in the comments in the source code.

1. *Box.java*. In order for your axis-aligned bounding box to work, you need to determine a bounding box for each surface. Therefore, you need to complete `computeBoundingBox()` which will compute the bounding box and store it in two private variables called `minBound` and `maxBound`. These two points may not necessarily be the same as `minPt` and `maxPt` because a transformation may be applied on the box. You also need to figure out the average position and store it in another private variable called `averagePosition`. For box, this point is simply the center of the box.

2. *Sphere.java*. Same as *Box.java*, you need to complete `computeBoundingBox()`.

3. *Cylinder.java*. Same as *Box.java*, you need to complete `computeBoundingBox()`.

4. *TriangleMesh.java*. This class is a subclass of Surface and has the same interface as those of Box and Sphere. You need to implement `intersect()` and `computeBoundingBox()`. One way to implement the intersection test is to use barycentric coordinates. (Shirley 2.6)

5. *Group.java*. This class is a subclass of Surface. It represents a transformation node in the tree hierarchy and can contain multiple children which can be another group itself. This class has a transformation matrix called `transformMat`. The parser will automatically call `setTranslate()`, `setRotate()`, and `setScale()` which are public methods in this class to set the value of `transformMat`. `setTranslate()` is given to you as an example. You need to complete `setRotate()`, `setScale()` to correctly set `transformMat`. As a subclass of Surface, the group class (as well as box and sphere) has three important variables called `tMat`, `tMatInv`, and `tMatTInv`. `tMat` is a matrix that represent the final transformation matrix that will be applied to the surface. For example, suppose a sphere S is a child of a group G_1 whose `transformMat` is a translation for some amount in X-axis, and G_1 is a child of a group G_2 whose `transformMat` is a rotation around Y-axis, `tMat` of sphere S will be $R_{G_2}T_{G_1}$. Notice that we apply the transformations from the bottom up to the root of the tree i.e. for any vertex in our sphere, it will be translated first then rotated. `tMatInv` is always equal to the inverse of `tMat`, and `tMatTInv` is always equal to the inverse of the transpose of `tMat`. These two matrices will be important for computing ray intersections with the transformed surface. We sacrifice memory for speed here because these matrices will be used multiple times.

6. *AABB.java*. This class provides a framework for the axis-aligned bounding box acceleration structure. Again, you can start from scratch and write your own class for AABB. For this class, you need to complete `createTree()` which recursively constructs the tree and returns the root node, `intersect()` which performs a query on the tree, and `isIntersect()` which tells you whether a ray hits the bounding box. You can also simply reuse your code from box intersection test to implement `isIntersect()`.

7. *Glazed.java*. This class implements a recursive glazed shader. Unlike Lambertian and Phong, this shader should compute a reflected ray, trace it recursively through the scene, and use the Fresnel reflection factor R to weight its contribution. Having a shader that uses recursively computed rays means that your renderer will generate a chain of rays (or a tree of rays in the case of Glass shader), which is potentially slow and infinite in depth. In addition to a 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.

8. *Glass.java*. A glass shader simulates an interface between air and a dielectric material. It should compute the directions of the reflected and refracted rays using Snell's law, compute the reflection factor using Fresnel's formulas, then trace reflected and refracted rays recursively and combine the results using the reflection factor. It needs to work for rays coming from both sides of the surface; you can always tell which side is air because the air is on the outside, the side toward which the normal points.

9. *RayTracer.java*. To support antialiasing, you need to modify your ray tracer to shoot multiple rays per image pixel.

4 Framework

The rest of the framework is described at the end of this document in Appendix A (this is the same as in PA1). You do not need to spend time trying to understand it and it is not essential to read this to get started on the assignment. Instead, you can reference it as needed.

5 Extensions

Once you have all of the required features working, you can continue having fun and collecting bonus points at the same time, by implementing some of the extensions proposed below:

1. *Cube-mapped backgrounds*. 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 direction 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 technique 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 at `http://panda3d.org/wiki/index.php/Cube_Maps` and many actual maps can be found here `http://www.debevec.org/Probes/`.

2. *Output HDR images to OpenEXR format*. OpenEXR is an open standard file format for high-dynamic range images. Once your image is in the format, there are many tools for producing interesting images from the EXR file. There is existing code for outputting to it, but it is written in C++. So most of the work for this will be in making your Java code interact with the C++ library (using JNI). More information and a link to ILM's free code and tools can be found at `http://en.wikipedia.org/wiki/OpenEXR`.

3. *Bilinearly filtered texture mapping*. Implement bilinearly filtered texture mapping for *triangle meshes*. You can use the ship models from the Pipeline project since they have texture coordinates. You will need to interpolate these when you shade a ray that hits a mesh triangle and sample the texture. You may re-use your texture class from the Pipeline project.

4. *Camera depth of field*. A real camera exhibits depth of field effects, such that objects far away from the focal distance are blurry. This can be simulated in a ray tracer using distributed rays. Refer to section 13.4.3 in the book (Shirley et al., Third Edition) for more details.

5. *Spotlights*. Extend your point light source to be a circular spotlight. A spotlight has a direction, a beam angle θ_b, and a falloff angle θ_f, in addition to the usual position and intensity. For directions that make an angle less than θ_b with the spotlight's direction, it produces the same intensity as a regular point light. For directions that are more than an angle of $\theta_b + \theta_f$ from the spot direction, it produces no illumination. In the falloff zone it drops off smoothly according to a C^1 function of angle.

6. *Propose your own*. 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.

6 Handing in

All submissions will be through CMS. We will also use CMS to schedule demo slots to grade the ray tracer as we get closer to the due date. We will post clarifications on Piazza if needed.

Figure 1: **"Urban Oasis" by Rob Fitzel** from the Internet Ray Tracing Competition.

Rendering Contest! An exciting part of computer graphics is rendering things never seen or imagined before, and ray tracing contests are a great excuse to do so. To show off the best your program can do, please also submit:

1. one image rendered at high quality and at high resolution (1280 pixels across), and
2. the corresponding XML file used to generate the image.

Make the model interesting, and make the image aesthetically pleasing. There is no limit on CPU time to create your image, so have fun!

7 Appendix A

The framework for this assignment includes a simple main program, some utility classes for vector math, a parser for the input file format, and stubs for the classes that are required by the parser.

7.1 `Parser`

The `Parser` class contains a simple and, we like to think, elegant parser based on Java's built-in XML parsing. The parser simply reads a XML document and instantiates an object for each XML entity, adding it to its containing element by calling `set...` or `add...` methods on the containing object.

For instance, the input

```
<scene>
<surface type="Sphere">
<shader type="Lambertian">
<diffuseColor>0 0 1</diffuseColor>
</shader>
<center>1 2 3</center>
<radius>4</radius>
</surface>
</scene>
```

results in the following construction sequence:

1. Create the scene.
2. Create an object of class `Sphere` and add it to the scene by calling `Scene.addSurface`. This is OK because `Sphere` extends the `Surface` class.
3. Create an object of class `Lambertian` and add it to the sphere using `Sphere.setShader`. This is OK because `Lambertian` implements the `Shader` interface.
4. Call `setDiffuseColor(new Color(0, 0, 1))` on the shader.
5. Call `setCenter(new Point3D(1, 2, 3))` on the sphere.
6. Call `setRadius(4)` on the sphere.

Which elements are allowed where in the file is determined by which classes contain appropriate methods, and the types of those methods' parameters determine how the tag's contents are parsed (as a number, a vector, etc.). There is more detail for the curious in the header comment of the `Parser` class.

The practical result of all this is that your ray tracer is handed a scene that contains objects that are in one-to-one correspondence with the elements in the input file. You shouldn't need to change the parser in any way.

7.2 `RayTracer`

This class holds the entry point for the program. The `main` method is provided, so that your program will have a command-line interface compatible with ours. It treats each command line argument as the name of an input file, which it parses, renders an image, and writes the image to a PNG file. The method `RayTracer.renderImage` is called to do the actual rendering.

7.3 `Image`

This class contains an array of `float`s and the requisite code to get and set pixels and to output the image to a PNG file.

7.4 The `ray.math` package

This package contains classes to represent 2D and 3D points and vectors, as well as RGB colors. They support all the standard vector arithmetic operations you're likely to need, including dot and cross products for vectors and gamma correction for colors.

7.5 Other classes

The other classes in the framework all exist because they are required in order for the parser to decode files in the input format described above. Since the XML entities in the file correspond directly to Java objects constructed by the parser, there is a class for every type of XML tag that can appear in the file, including `Scene`, `Image`, `Camera`, `Light`; `Surface` and its subtypes `Sphere` and `Box`; and `Shader` and its subtypes `Lambertian` and `Phong`. These classes generally contain only the fields and set/add methods required to implement the file format.

7.6 File format

This assignment will require you to make several extensions 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 Foo { public Foo() {} public setBar(Bar inBar) { ... } }

class Bar { public Bar() {} public setCat(String inName) { ... } }
```

Finally, the most common case is that you will be adding new shaders, surfaces, or lights. In this case, in addition to the requirements above, the new classes should extend `Shader`, `Surface`, or `Light`. You can then instantiate the correct class using the `type` argument. For example:

```
<shader type="MyShader"> </shader>
```

is the correct way to specify an instance of a new shader class named `MyShader`.

You can find comments with a more detailed description of the Parser in `Parser.java`.

The input file for your ray tracer is in XML. An XML file contains sequences of nested *elements* that are delimited by HTML-like angle-bracket tags. For instance, the XML code:

```
<scene>
<camera> </camera>
<surface type=Sphere>
<center>1.0 2.0 3.0</center>
</surface>
</scene>
```

contains four elements. One is a `scene` element that contains two others, called `camera` and `surface`. The `surface` element has an *attribute* named `type` that has the value `Sphere`. It also contains a `center` element that contains the text "1.0 2.0 3.0", which in this context would be interpreted as the 3D point $(1, 2, 3)$.

An input file for the ray tracer always contains one `scene` element, which is allowed to contain tags of the following types:

- `surface`: This element describes a geometric object. It must have an attribute `type` with value `Sphere` or `Box`. It can contain a `shader` element to set the shader, and also geometric parameters depending on its type:
  - for sphere: `center`, containing a 3D point, and `radius`, containing a real number.
  - for box: `minPt` and `maxPt`, each containing a 3D point. If the two points are $(x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$ then the box is $[x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}] \times [z_{\text{min}}, z_{\text{max}}]$.
- `camera`: This element describes the camera. It is described by the following elements:
  - `viewPoint`, a 3D point that specifies the center of projection.
  - `viewDir`, a 3D vector that specifies the direction toward which the camera is looking. Its magnitude is not used.
  - `viewUp`, a 3D vector that is used to determine the orientation of the image.
  - `projNormal`, a 3D vector that specifies the normal to the projection plane. Its magnitude is not used, and negating its direction has no effect. By default it is equal to the view direction.
  - `projDistance`, a real number d giving the distance from the center of the image rectangle to the center of projection.
  - `viewWidth` and `viewHeight`, two real numbers that give the dimensions of viewing window on the image plane.

  - `image`: This element is just a pair of integers that specify the size of the image in pixels.

  The camera's view is determined by the center of projection (the viewpoint) and a view window of size `viewWidth` by `viewHeight`. The window's center is positioned along the view direction at a distance d from the viewpoint. It is oriented in space so that it is perpendicular to the image plane normal and its top and bottom edges are perpendicular to the up vector.

- `light`: This element describes a light. It contains the 3D point `position` and the RGB color `color`.

- `shader`: This element describes how a surface should be shaded. It must have an attribute `type` with value `Lambertian` or `Phong`. The Lambertian shader uses the RGB color `diffuseColor`, and the Phong shader additionally uses the RGB color `specularColor` and the real number `exponent`. A shader can appear inside a surface element, in which case it applies to that surface. It can also appear directly in the scene, which is useful if you want to give it a name and refer to it later from inside a surface (see below).

If the same object needs to be referenced in several places, for instance when you want to use one shader for many surfaces, you can use the attribute `name` to give it a name, then later include a reference to it by using the attribute `ref`. For instance:

```
<shader type="Lambertian" name="gray">
<diffuseColor>0.5 0.5 0.5</diffuseColor>
</shader>
<surface type="Sphere">
<center>0 0 0</center>
<shader ref="gray"/>
</surface>
<surface type="Sphere">
<center>5 0 0</center>
<shader ref="gray"/>
</surface>
```

applies the same shader to two spheres.

Recursive ray tracing.In the input file the glazed shader is specified by an index of refraction, through a parameter named `refractiveIndex`, and another shader for its substrate:

```
<shader type="Glazed">
<refractiveIndex>1.5</refractiveIndex>
<substrate type="Lambertian">
<diffuseColor>0.4 0.5 0.8</diffuseColor>
</substrate>
</shader>
```

In the input file the glass shader should be specified just by its index of refraction, through a parameter named `refractiveIndex`:

```
<shader type="Glass">
<refractiveIndex>1.5</refractiveIndex>
</shader>
```

Transformations.The transformation is specified as a sequence of rotations, scales, and translations, which are combined in the order given to define the transformation that is applied to all members of the group. The transformation that appears first in the file is on the outside. All transformations apply to all objects, even if the transformations and objects are intermixed in the file (it makes most sense to put the transformations first, then the objects). Transformations are described in exactly the same way as in the modeler: translations and scales have components for x, y, and z; a rotation is actually a sequence of three rotations about the three coordinate axes, with the x rotation on the inside and the z rotation on the outside.

The file format can be defined by example. For instance, if the transformation in the modeling assignment was given as "T: 1 2 3; R: 40 50 60; S: 0.7 0.8 0.9," the same effect can be specified in the ray tracer as follows:

```
<surface type="Group">
<translate>1.0 2.0 3.0</translate>
<rotate>40 50 60</rotate>
<scale>0.7 0.8 0.9</scale>
<surface type="Sphere">
<!-- ... -->
</surface>
<!-- more surfaces... -->
</surface>
```

Triangle meshes. Since meshes can be quite large, it is not practical to process them using the parser, so they are stored in a simple text format in separate files. These files contain standard indexed triangle meshes, with optional texture coordinates and surface normals at the vertices. If the mesh contains vertex normals, you should shade it with interpolated normals; otherwise you should shade it with the triangles' geometric normals.

The input format for a mesh is just a filename reference:

```
<surface type="Mesh">
<shader><!-- ... --></shader>
<data>filename.msh</data>
</surface>
```

The mesh file contains text as follows, one word or number per line:

- The number of vertices in the mesh.
  - The number of triangles in the mesh.
  - The keyword `vertices`
  - The 3D coordinates of the vertices, ordered by vertex number: $x_0, y_0, z_0, x_1, y_1, z_1, \ldots$.
  - The keyword `triangles`

- Three integer vertex indices per triangle.
- (Optional) The keyword `texcoords`, followed by u and v coordinates for each vertex.
- (Optional) The keyword `normals`, followed by x, y, and z components of a normal vector for each vertex.

Antialiasing.The number of samples is specified by the `samples` property of the `Scene` class. For example, the following input specifies a 640 by 480 pixel image rendered with a 3x3 grid of subpixel samples for each pixel.

```
<scene>
<camera>
<!-- ... -->
</camera>
<image>640 480</image>
<samples>9</samples>
</scene>
```

You are free to round the number of samples to a convenient number (for example, to the nearest perfect square). So if the user inputs 10, you should assume 9 samples - not 100.

Really, the file format is very simple and from the examples we provide you should have no trouble constructing any scene you want.