CS 4620/5620 Ray 2

due: Friday 5 December 2014

1 Introduction

Ray 1 introduced you to the basic principles behind this relatively simple, and at the same time very powerful, algorithm for rendering images. In this assignment, we will enhance the previous algorithm to trace 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, the ray-object intersection.

2 Requirement Overview

In Ray 1, you implemented the basic ray tracing; we will provide a package that contains what you implemented from that assignment, plus some updates to the framework. You will extend this ray tracer as specified below. This new ray tracer should support all the features of the one in Ray 1 and the following features below:

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

2. *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 triangles. 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 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).

3. *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 volume hierarchy

(BVH) which is a simple and effective way of speeding up ray traversal. (Shirley 12.3). Before you have implemented BVH, you can add `<AccelStruct type="NaiveAccelStruct" />` in the scene file in order to test other parts of your implementation.

4. *Advanced shader: glazed*. A "Glazed" material that acts like a thin layer of dielectric over another material, and reflects somewhat like a mirrored surface. The glazed shader also calls another shader which computes the contribution from the substrate below the glaze (Shirley 13.1).

5. *Advanced shader: Cook-Torrance*. You should implement a Cook-Torrance shader which you are familiar with from the Shaders assignment. The only difference is that now you should compute the fresnel term from the refractive index.

6. *Advanced shader: glass*. A "Glass" material that simulates an interface between air and a dielectric material. The glass shader obtains its color from light intensity along two new rays —the reflected and refracted rays—and adds their corresponding contributions (Shirley 13.1).

7. *Environment mapping*. Our framework can load a pfm (a HDR image format) file as environment mapping. You are asked to implement a method that looks up the color of the cubemap in a given direction.

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

3 Implementation

A new commit has been pushed to the class Github page in the master branch. We recommend switching to your master branch, pulling this commit, and creating a new branch (e.g. A7 solution) and committing your work there. This new commit contains all the framework changes and additions necessary for this project.

We have marked all the functions or parts of the functions you need to complete with TODO#A7 in the source code. To see all these TODOs in Eclipse, select Search menu, then File Search and type TODO#A7.

3.1 Anti-Aliasing

To support anti-aliasing, modify the `renderBlock` method of `RayTracer` to make the ray tracer shoot multiple rays per image pixel. `renderBlock` method contains the `samples` variable which is the number of samples to take per one dimension of the image plane. (That is, the tracer should shoot $\texttt{samples}^2$ rays per pixel.) Note that we have abused the variable `samples`: in the xml file we use `samples` to refer to N^2, while here it refers to N.

3.2 Groups and Transformations

Each instance of the `Surface` contains fields that define the surface; for example, the sphere has a center and a radius, the box has its min and max points, etc. These fields are specified in *object space.* The surface is transformed into *world space* by a 4×4 transformation matrix also stored in the instance. The transformation is stored in three different fields:

- `tMat`: the transformation matrix,
- `tMatInv`: the inverse of the transformation matrix, and
- `tMatTInv`: the inverse transpose of the transformation matrix.

`tMatInv` is always equal to the inverse of `tMat`, and `tMatTInv` is always equal to the inverse transpose of `tMat`. These two matrices will be important for computing ray intersection with the transformed surface. We sacrifice memory for speed here because these matrices will be used multiple times.

The `Group` class is a subclass of `Surface`. It represents a transformation node in the tree hierarchy and can contain multiple children which can be groups themselves. 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`.

As a subclass of `Surface`, it also has the `tMat`, `tMatInv`, and `tMatTInv` fields. Note that `tMat` is a matrix that represent the final transformation matrix that will be applied to the surface, and it takes points from the object space to world space. However, `transformMat` stores the transform that takes points from the group's object space to the space of the group's parent.

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.

You need to implement the `setTransformation` method, which propagates the transformation matrix of `Group` to its descendants. The method takes three arguments `pMat`, `pMatInv`, and `pMatTInv`, which are the product of the transformation matrices from the root to the `Group`'s parent, its inverse, and its inverse transpose. The `setTransformation` method should multiply these matrices with the appropriate version of `transformMat`, and then call itself recursively on the `Group`'s children. The end result should be that the `tMat` of each `Surface` is the product of the transformation matrices from the root to that `Surface`.

You also need to modify the `intersect` method of each surface to account for these changes:

1. Transform the given ray to object space.
2. Intersect the ray *in object space* with the surface.
3. Transform the resulting intersection point **and** normal to world space.

Note that the `Surface` class has a useful method called `untransformRay()` which should make step 1 very simple, and step 3 should just be simple matrix-vector calculations.

3.3 Acceleration Structure

You will implement an acceleration structure called *axis-aligned bounding volume hierarchy* (BVH). It is a tree whose leaves are surfaces in the scene. Each of its internal nodes contains an *axis-aligned*

bounding box (AABB) that contains all the surfaces in the subtree rooted at that node. An axis-aligned bounding box is defined by two 3D points $(x_{\min}, y_{\min}, z_{\min})$ and $(x_{\max}, y_{\max}, z_{\max})$. The box itself is the Cartesian product of three intervals in each dimension,

$$[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}].$$

Before constructing the acceleration structure, you need to complete the `computeBoundingBox` method of all the subclasses of `Surface` except `Mesh`. This method should modify three fields: `minBound`, `maxBound`, and `averagePosition`. After calling this method, the AABB defined by `minBound` and `maxBound` in world space should contain the represented surface entirely in world space, and `averagePosition` should be a 3D position in world space inside the AABB that can be regarded as the "center" of the surface in world space. We stress that `minBound`, `maxBound`, and `averagePosition` are defined **in world space**; you need to take into account the transform stored in `tMat` when implementing the method because all other fields are defined in object space.

After implementing all the `computeBoundingBox` methods, you are ready to complete the `Bvh` class. Each node in the BVH is represented by an instance of the `BvhNode` class. The root node is stored in the `root` field of the `Bvh` class. The surfaces the BVH manages are not stored in the nodes, but they are stored all in one place in the `surfaces` array in the `Bvh` class. Each `BvhNode` contains the `surfaceIndexStart` and `surfaceIndexEnd` subjected to the following invariant:

> Surfaces stored in `surfaces` with indices from `surfaceIndexStart` to `surfaceIndexEnd-1` are contained in the subtree rooted at the node.

`BvhNode` also contains the `minBound` and `maxBound` fields that define the AABB correponding to the node. Moreover, it has `child` array which contains the references to its left child and right child.

You need to implement the `createTree(int start, int end)` method of the `Bvh` class. This method should create and return a `BvhNode` that contains surfaces from `surfaces[start]` to `surfaces[end-1]` in its subtree. To do so, you need to divide the surfaces into two groups by sorting their `averagePosition` and modifying the `surfaces` array so that the first group is in the left half and the second group is in the right half of the array. Then, you call `createTree` recursively on the left and the right half to get two `BvhNode`s that will become the two children of the `BvhNode` that you construct for the original call to `createTree`. More details can be found in the comments in the code base.

Then, you will implement the `intersects` method of the `BvhNode` class. This method only checks whether the given ray intersects the AABB of the `BvhNode` or not. There is no need to compute the intersection point or other associated information.

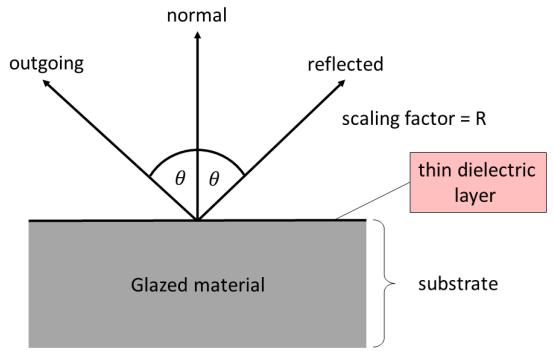
Lastly, you need to implement the `intersectHelper` method of the `Bvh` class. This method checks whether the given ray intersects any surface contained in the subtree rooted at the given `BvhNode`. First, you should check whether the ray hits the node or not by calling the `intersects` method of the `BvhNode` class. If not, the method can return immediately. (This check is the key why ray intersection becomes much faster with a BVH.) Otherwise, if the `BvhNode` is a leaf, you intersect the ray with the surfaces with indices from `surfaceIndexStart` to `surfaceIndexEnd-1` one by one. On the other hand, if the `BvhNode` is an internal node, you call `intersectHelper` recursively on its two children.

3.4 Glazed Shader

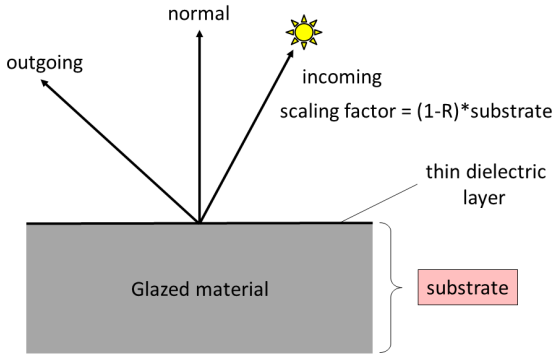
The `Glazed` shader models a rough material coated with a dielectric material which reflects light like a mirror. Therefore, the `shade` method should generate a reflected ray originated from the hit point whose direction is the reflection of the outgoing direction around the surface normal at the intersection point. `shadeRay` should be called again on this new ray. The returned intensity should be scaled by the Fresnel coefficient R. You should implement the `fresnel` function in `Shader.java` to compute R.

The rough material part of the `Glazed` shader is stored in its `substrate` field, which can either be a `Lambertian`, `Phong` or `Cook-Torrance` shader. For the implementation, you need to call the `shade` method of `substrate` and then scale the result by $1 - R$. Figure 1 provides a summary of the `Glazed` shader's behavior.

When implementing the shaders, note that something in the `Light` class are different from Ray1 because we support directional light in Ray2. Use `light.getDirection(record.location))` to get the direction from the shaded point to the light. Use `light.getRSq(record.location)` to get r^2 for shading (this function returns 1 for directional light, because directional light's source is at infinity and we use a different unit for its intensity, which is not affected by r^2).



(a)



(b)

Figure 1: (a) The `shade` method should implement the behavior of the thin dielectric layer by generating a reflected ray whose scaling factor is R. (b) The `shade` method should also implement the behavior of the substrate by calling `substrate`'s `shade` method and scale the result by $1-R$.

3.5 Cook-Torrance Shader

For detailed description, see the Shaders assignment pdf. There we used the Schlick's approximation to compute the Fresnel term. Now after you implemented the *fresnel* function in *Shader.java*, you should use that to compute the Fresnel term instead.

3.6 Glass Shader

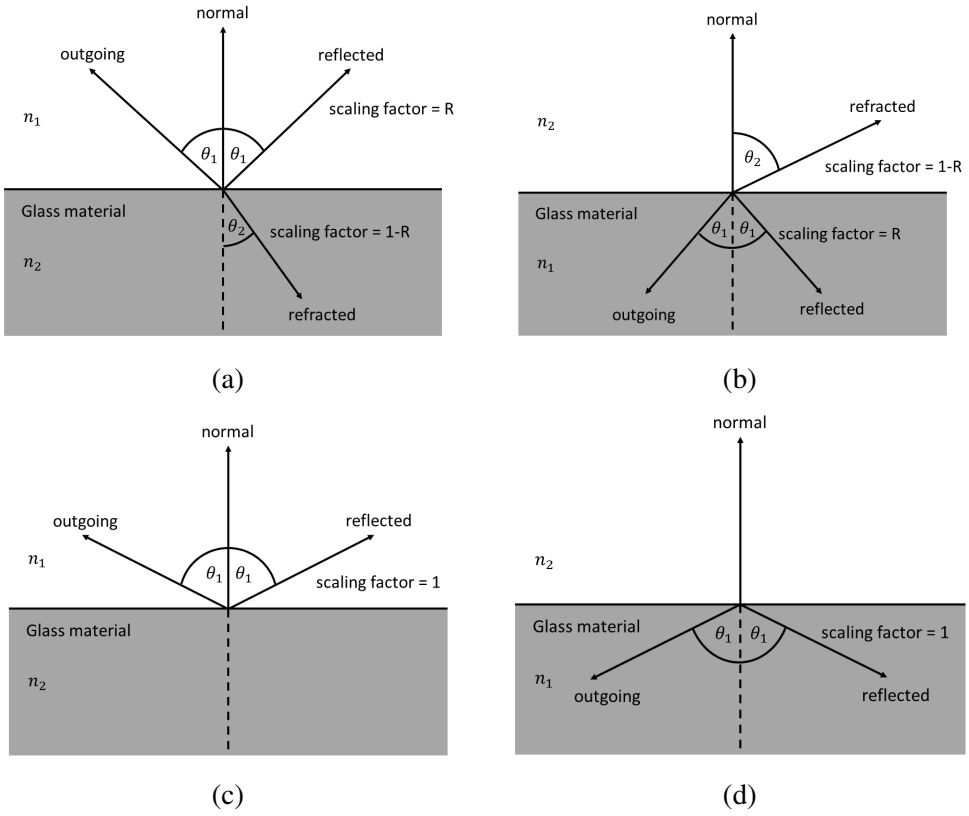
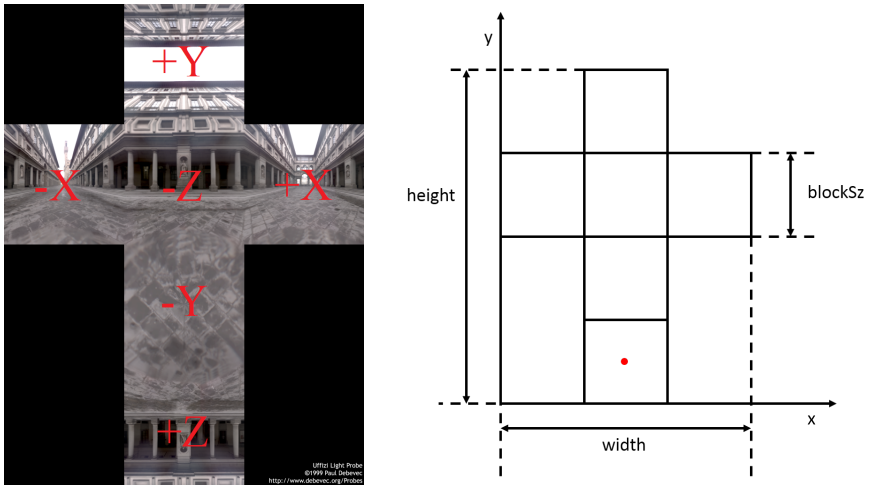


Figure 2: Additional rays to be generated for the `Glass` shader. In (a) and (b), one reflected ray with scaling factor R and one refracted ray with scaling factor $1 - R$ are generated. Note that the outgoing direction can either be inside or outside the surface. For the `Glazed` shader, there is only one reflected ray with scaling factor R. The angles θ_1 and θ_2 should be related by Snell's law: $n_1 \sin\theta_1 = n_2 \sin\theta_2$. In (c) and (d), total internal reflection occurs. Only one reflected ray should be generated with scaling factor 1. Note that total internal reflection can happen on the *outside* of the material despite its name because the ratio between n_1 and n_2 can be arbitrary. Your code has to work correctly in all these cases.

The `Glass` shader simulates an interface between air and a dielectric material. For its `shade` method, it should compute the directions of the reflected and refracted rays using Snell's law. `shadeRay` should then be called recursively on each of these rays. The factor for the reflected ray should be R, and the factor for the refracted ray should be $1 - R$. You need to check for total internal reflection in which case you only generate the reflected ray and set its factor to 1. One caveat is that the method must work for rays coming from both sides of the surface; you can tell

which side is outside by the fact that the normal always points outside. Figure 3 summarizes the behavior of the `Glass` shader.

3.7 Environment Mapping



(a) (b)

Figure 3: Example of Cubemap

The framework supports environment mapping as a background for the scene. For a scene with a cubemap, rays that don't hit any objects should look up the corresponding pixel value in the cubemap. The framework can load a pfm (an HDR image format) file as the environment map. An example of such a file is shown in figure 3. It's just an unfolding of the six faces of the axis-aligned cube into a cross. The corresponding directions for each of the faces are shown in figure 3.

You are asked to implement the `evaluate` method of the `Cubemap` class, which looks up the color of the cubemap in a given direction. In the `Cubemap` class, fields `width` and `height` store the width and height of the entire pfm image. The `blockSz` field stores the size of the square, which is `width`/3. The pixel values of the pfm image are stored in the float array `imageData`. The R, G, B values of a pixel are stored sequentially. And the pixels in the image are ordered row by row, from bottom-left to top-right. For example, if we want the color of environment light in direction (0, 0, 1), we need to look up the pixel value of at the red point in figure 3. Its coordinates in the image plane is (`x = 1.5 * blockSz`, `y = 0.5 * blockSz`). And the R, G and B values would be `imageData[3 * (x + width * y)]`, `imageData[3 * (x + width * y) + 1]` and `imageData[3 * (x + width * y) + 2]`.

4 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. If you implement any of these, please leave a description of what you did in your README file, and create an xml file or two to clearly demonstrate these features.

1. *Soft shadows.* Shadow rays need not go to a single point, but can be distributed over an area of the light source. Implementing soft shadows requires shooting more rays and distributing them evenly over the area light. You will need to extend the model of lights to have some area (a square area is convenient) for this problem.

2. *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.

3. *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.

4. *Bilinearly filtered texture mapping*. Implement bilinearly filtered texture mapping for *triangle meshes*. Use bilinear interpolation when you sample the texutre.

5. *Metal shader*. Implement a shader called "metal" that takes a complex refractive index, with separate values for R, G, B, and uses them to compute Fresnel reflectance for a conductor.

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.

5 Handing in

Submit a zip file containing your solution organized the same way as the code in the repository, together with your solutions for the written assignment. We strongly recommend you write your solutions for the written part neatly in a PDF file. Include a readme in your zip that contains:

- You and your partner's names and NetIDs.
- Any problems with your solution.
- Anything else you want us to know.

6 Appendix A: HDR output

The ray2 framework supports HDR output. In order to use HDR output, set the `writeHDR` variable in `RayTracer.java` to be true. We use the openEXR library to produce HDR output. You may need to configure your native library location. To do this in Eclipse, go to Project -> Properties -> Java Build Path -> Select Libraries -> Select the openexrjni-3.0.0.jar Drop Down Menu -> Modify Native Library Location. Modify this setting so that it matches your OS. Also in Project -> Properties -> Java Build Path -> Select Source -> Select the CS4620/src Drop Down Menu -> Modify Native Library Location. Modify this setting so that it matches your OS.

The HDR output file will be named as `*.exr`. You can download a tool from the following address to view the `*.exr` files as well as the `*.pfm` files. https://bitbucket.org/edgarv/hdritools/downloads

7 Appendix B: Notes on File Format

The code base still use the same framework that was released with Ray 1. In this section, we note some additional features that you can specify in the input XML file.

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. Translations and scales have components for x, y, and z; a rotation is a sequence of three rotations about the three coordinate axes, with the x rotation applied first and the z rotation applied last.

The file format can be defined by example. For instance, if a transformation is given as "T: 1 2 3; R: 40 50 60; S: 0.7 0.8 0.9," this 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, all transformed as defined above -->
</surface>
```

Cook-Torrance shader. You can specify the diffuse color, specular color, toughness and refractive index for Cook-Torrance shader in the scene file.

```
<shader type="CookTorrance">
    <diffuseColor>0.3 0.3 0.3</diffuseColor>
    <specularColor>0.2 0.2 0.2</specularColor>
    <roughness>0.8</roughness>
    <refractiveIndex>1.5</refractiveIndex>
</shader>
```

Glazed shader. 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>
```

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

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

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). Use any rounding technique that you'd like, as long as specifying a perfect square results in exactly that many samples. Note that the `samples` element here is approximately the square of the `samples` variable in the `renderImage` method.

AccelStruct. The type of AccelStruct used can be specified by a single line as a child of scene. This may be useful for testing your AccelStruct implementations, especially for scenes with large meshes.

```
<scene>
    <accelStruct type="Bvh" />
    <!-- ... -->
</scene>
```

Cubemap. You can use specify the directory of the cubemap in the xml file, just as follows.

```
<scene>
    <cubemap>
        <filename>data/textures/cubemaps/uffizi_cross.pfm</filename>
    </cubemap>
    <!-- ... -->
</scene>
```