CS 4620/5620 Ray 2

out: Thursday 21 November 2013
due: Thursday 12 December 2013
NOTE: No late submissions can be accepted, even if you haven't used your late day yet.

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 part, 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.

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. We will pick the "best image" and the two runners-ups (on combined technical and aesthetic grounds).

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. *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).

2. *Triangle meshes*. Support triangle meshes in order to render more complex meshes such as the famous Stanford Bunny (Shirley 4.4)

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. However, you can also implement your own BVH from scratch, or if you want to implement another kind of acceleration structure, please ask the course staff (Shirley 12.3).

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: 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).

6. *Absorption*. Your ray tracer should also be able to take into account the absorption coefficient of the medium that the ray is going through, and scale down the light intensity along it according to Beer's law.

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

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

3 Implementation

Download the new Eclipse project from CMS. You do not need to port over any of your code from Ray 1. We have marked all the functions or parts of the functions you need to complete with `TODO` in the source code. To see all these TODO's in Eclipse, select *Search* menu, then *File Search* and type "`TODO`"

You will notice there are a few changes in the framework since Ray 1. Most significantly, `Workspace` has been removed from most of the code, since this is cleaner given the added complication of recursion. However, the class still exists in the framework; if you would like to re-implement it, we recommend making sure your project works fully before doing so. Secondly, the `BasicRayTracer` class and its abstract parent `RayTracer` have been combined, since there was not sufficient need to keep them separate.

3.1 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`. `setTranslate()` is given to you as an example. You need to complete `setRotate()`, `setScale()` to correctly set `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 also need to implement the `setTransformation` method, which propagates the transformation matrix of `Group` to its descendants. The method takes three arguments `cMat`, `cMatInv`, and `cMatTInv`, 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`.

Lastly, you 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.

3.2 Triangle Mesh

The implementation of triangle mesh is contained in two classes: `Mesh` and `Triangle`. Both classes are subclasses of the `Surface` class. The `Mesh` class contains the raw data of the mesh such as vertex positions, normals, and texture coordinates. An instance of `Triangle` represents a triangle in a mesh. You do not have to modify the `Mesh` class in this assignment, but you will need to access data contained in it.

The number of vertices and triangles in a `Mesh` can be obtained via the `getNumVertices` and `getNumTriangles` methods, respectively. If there are n vertices in a mesh, the vertices are numbered from 0 to $n-1$. The position, normal, and texture coordinate of Vertex i can be accessed calling `getVertex(i)`, `getNormal(i)`, and `getTexcoords(i)`, respectively.

An instance of `Triangle` contains a reference to the `Mesh` it belongs to and an array `index` of length 3, containing the indices of the three vertices of the triangle. Suppose the positions of the three vertices are $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$, respectively. The constructor of `Triangle` computes the fields `a`, `b`, `c`, `d`, `e`, and `f`:

$$\begin{bmatrix} \texttt{a} \\ \texttt{b} \\ \texttt{c} \end{bmatrix} = \mathbf{v}_0 - \mathbf{v}_1, \text{ and } \begin{bmatrix} \texttt{d} \\ \texttt{e} \\ \texttt{f} \end{bmatrix} = \mathbf{v}_0 - \mathbf{v}_2.$$

If the `Mesh` that contains `Triangle` does not have normal information stored with each vertex, the construnctor will compute the norm field to be:

$$\texttt{norm} = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{\|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\|},$$

which is the normal used in flat shading. You should use this normal as the normal of the ray-intersection point only when the `Mesh` does not store normals.

As with other subclasses of `Surface`, you need to implement the `intersect` method of the `Triangle` class. You will find Section 4.4.2 of Shirley and Marschner useful for this task. If the `Mesh` does not store normals, use `norm` as the normal at the intersection. Otherwise, you need to compute the barycentric coordinates of the intersection point, and use them to interpolate the normals of the three vertices.

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,

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

which is similar to the definition of the `Box` surface in Ray 1.

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, you guessed it, 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. You should reuse the code from the `intersect` method of the `Box` class.

Lastly, you need to implement the `intersectHelper` method of the `Bvh` class. This method checks whether the given ray intersects any surface in 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 Advanced Shaders

We ask you to implement two shaders: `Glazed` and `Glass`. These shaders are different from those in Ray 1 because they can demand the ray tracer to recursively trace more rays and use the light intensity along the ray to compute the materials' colors.

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 using the Schlik approximation from lectures.

The rough material part of the `Glazed` shader is stored in its `substrate` field, which can either be a `Lambertian` or a `Phong` 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.

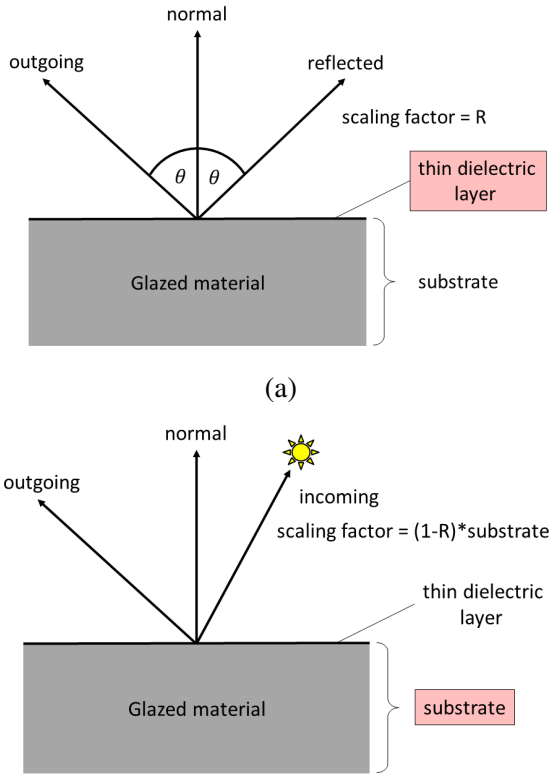

Figure 1: (a) The `shade` method should implement the behavior of the thin dielectric layer by generating a reflected ray whose scaling factor is R. (b) The `shade` method should also implement the behavior of the substrate by calling `substrate`'s `shade` method and scale the result by $1-R$.

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 2 summarizes the behavior of the `Glass` shader.

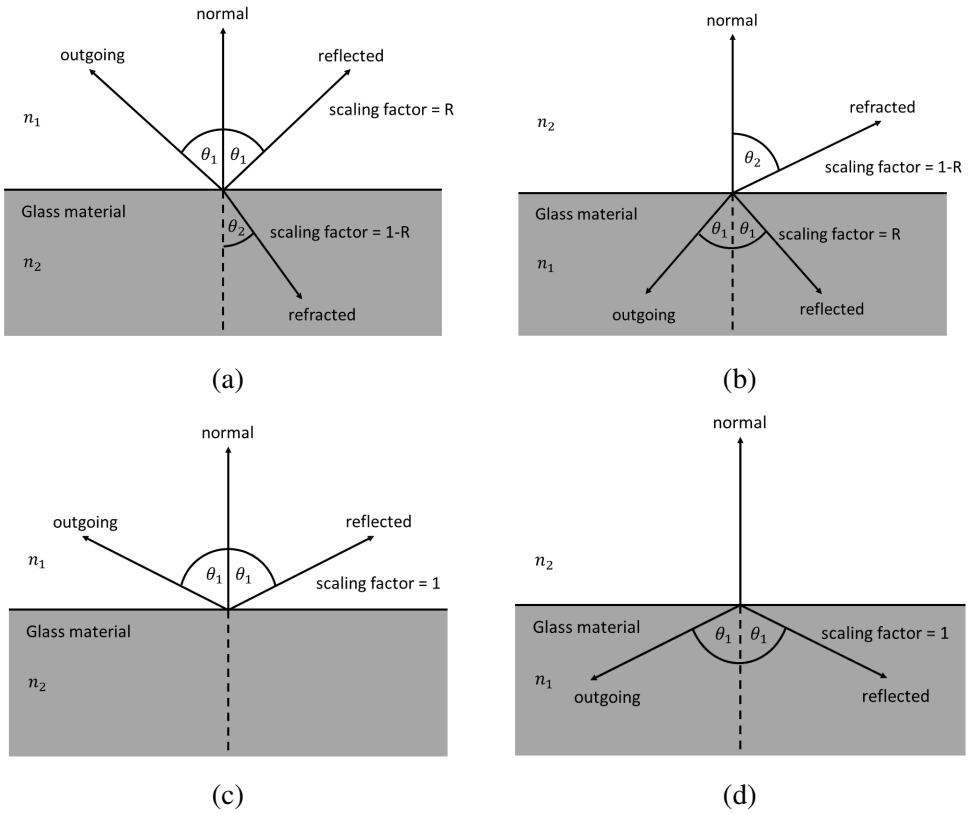

Figure 2: Additional rays to be generated for the `Glass` shader. In (a) and (b), one reflected ray with scaling factor R and one refracted ray with scaling factor $1 - R$ are generated. Note that the outgoing direction can either be inside or outside the surface. For the `Glazed` shader, there is only 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.

3.5 Absorption

Each shader needs to account for the attenuation by absorption; this is done by computing the distance ℓ that the original ray has traveled through, and scaling the color of the ray down by $e^{-\sigma\ell}$ where σ is `absorption`. Note that this formula is different from the one in Section 13.1 of Shirley and Marschner, which is $e^{-\ln(\sigma)\ell}$.

To make this easier, an `absorption` field (an instance of `Color`) has been added to the `Ray` class. You need to implement the `attenuate` method in `Ray` that takes an input color and some point along the ray and scales the color based on the distance between the input point and the ray's origin. Shaders then call this method to scale their final calculated color appropriately. Be careful with the `Glazed` shader; remember the substrate will already have attenuated its color!

3.6 Anti-Aliasing

To support anti-aliasing, modify the `renderImage` method of `RayTracer` to make the ray tracer shoot multiple rays per image pixel. `renderImage` 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.

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. *Cube-mapped backgrounds*. A ray tracer need not return black when rays do not hit any object. Commonly, background images are supplied that cover a large cube surrounding the scene. The direction of rays (that do not intersect objects) are used as indices into these images and the color of the image in the ray's direction is returned rather than black. The technique is commonly called cube-mapping. To implement cube-mapping in your 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/`.

5. *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`.

6. *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.

7. *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 on CMS. 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.

Figure 3: **"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!

6 Appendix A: 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>
```

Absorption coefficient. The absorption coefficient of the region inside and outside a surface can be specified by the `insideAbsorption` and `outsideAbsorption`. For example:

```
<surface type="Sphere">
    <shader ref="glass" />
    <insideAbsorption>0.1 0.2 0.3</insideAbsorption>
    <outsideAbsorption>0.0 0.0 0.0</outsideAbsorption>
    <center>1.5 5 1</center>
    <radius>0.4</radius>
</surface>
```

You can also specify the absorption of the entire scene:

```
<scene>
    <absorption>0.1 0.2 0.3</absorption>
    <!-- ... -->
</scene>
```

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>
```

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). 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>
```