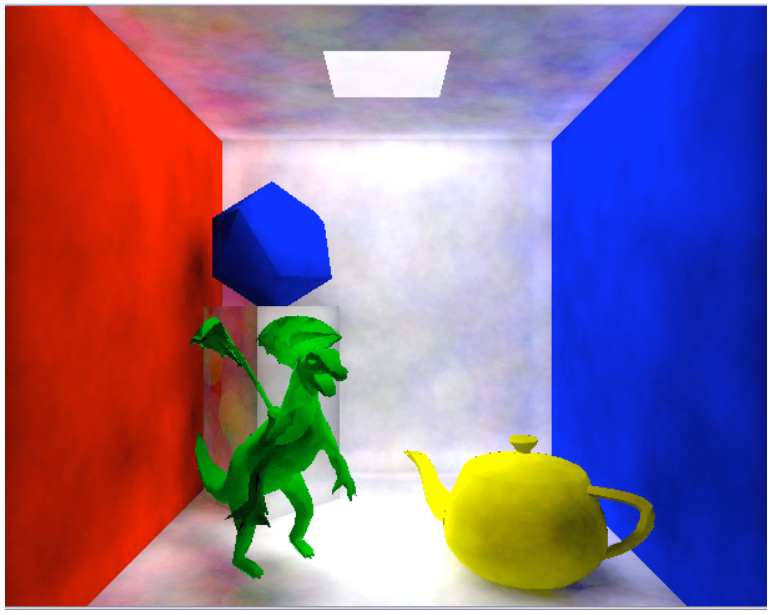
CS371 Spring 2007
Prof. Morgan McGuire

Assigned: Wed. 2/21/07
Due: Mon. 2/26/07 8:00 pm

Project 2: Scatter



For this project you will extend your bidirectional ray-tracer to handle more interesting scenes and more realistic indirect illumination.

A minimum correct solution will probably take about 1600 lines of C++ code, 600 of which are in common with your previous Spheres project and 400 of which are provided for you as sample code. Recall that, with that person's permission, you may use anyone's code from the previous project. As with all programming projects in this course, you may work with a partner on this project.

This project should only take you half the time of the Spheres project; it is a fairly straightforward extension. The goal is to optimize, organize, and generalize your code for the next project, which will simulate all kinds of light paths.

Scatter Specification

Your program must have the following simulation features:

1. Multiple-bounce photons by:
   a. Simulating photons forward from a light source and storing them in a photon map, and
   b. Tracing rays backwards from the eye to find the surface that colors a pixel and shading it using the photons stored in the photon map.
2. Arbitrary scenes containing:
   a. two-sided triangle meshes with both smooth and not smooth normals (loaded from IFS files),

   b. spheres,
   c. planes,
   d. and rectangles.
3. At least three wavelengths of light (e.g., r, g, b) and "colored" surfaces with non-uniform spectral response.
4. Comments explaining the correct physical units for all relevant quantities (e.g., What does your "image" store? What are the units on the fields of the Photon class?)
5. A BSDF class with LambertianBSDF subclass used to create scattering
6. Post-process the image in a method named tonemap that:
   a. Rescales the intensity range so that values *approximately* range from 0 to 1
   b. Inverts the monitor's gamma function by raising each pixel value to the (1/2.2) power *after* intensity rescaling
7. Images of two sample scenes:
   a. A Cornell box with smooth polygon, unsmoothed polygon, and rectangle primitives in it, demonstrating color bleeding and indirect illumination
   b. A scene of your choice

You may choose to use any language, platform, and libraries you wish, although only C++ and G3D on the department's FreeBSD systems are supported. If you use C++ as I recommend, please use Java (yes, Java) coding conventions for your C++ code.

Implementation Tips

Before adding any new features, you need a very fast single-bounce renderer. I recommend taking your code from the Spheres project and optimizing it so that it can render scenes with 2000 spheres in 30 seconds or fewer with one light source and 40000 photons.

The most significant optimization that you can make is using an AABSPTree for storing the scene entities (note that this is separate from your photon map!) The AABSPTree::intersectRay method can find the nearest intersection of a ray and your entire scene in expected amortized O(log *n*) time in the number of objects in the scene. A naïve array search would take O(*n*) time, which won't scale well for the large scenes you need to render in this project. Here's how to use AABSPTree for entities:

```
class Hit {
public:
        ...
        void operator()(const Ray& ray,
                        const EntityRef& entity,
                        float& distance);

};
```

```
...

void Hit::operator()(const Ray& ray,
                     const EntityRef& entity,
                     float& distance) {
    entity->intersect(ray, distance, *this);
}

...

AABSPTree<EntityRef> entitySet;

...

entitySet.balance();

...


entitySet.intersectRay(worldRay, hit, distance, false);
```

Overloading the function call operator (parentheses) on the Hit class allows an instance of the Hit class to be used as if it were a function. The AABSPTree::intersectRay method takes a callback function as its second argument. Passing a Hit means that Hit's operator() method will be called for each object in the BSP tree that is near the ray. When that operator is invoked, it just performs the Entity-ray intersection from the Spheres project.

You'll also need to add the following functions to your Entity.h file outside the Entity class definition. These are required by AABSPTree:

```
inline void getBounds(const EntityRef& entity, G3D::AABox& box) {
    entity->getBounds(box);
}
inline unsigned int hashCode(const EntityRef& entity) {
    return entity->hashCode();
}
```

The second most significant optimization you can make is to change your PhotonMap from an AABSPTree to a PointAABSPTree. PointAABSPTree is specially optimized for storing objects that have no volume and will make your photon gathering about twice as fast. To use it you need to implement

```
void getPosition(const Photon& photon, Vector3& pos);
```

The BSDF class is so that you can later add other kinds of BSDFs. I created all BSDFs once at initialization, stored explicit C pointers to them in my Entitys, and deleted all BSDFs on cleanup. I also added an EmissionFunction class, which is a probability distribution for an emitting source. Extend your Hit class with fields storing pointers to the BSDF and EmissionFunction for the surface hit. Make sure that you initialize these to NULL and that all of your intersection methods set these explicitly, otherwise you'll end up with the BSDF/EmissionFunction for some other surface still sitting in your next Hit.

Note that 90% of the triangle meshes code is implemented for you in the starter code. You just have to implement the triangle intersection algorithm in MeshEntity.cpp.

Innovate!

Some ideas for extending the Scatter project include:

1. Make a really interesting scene. The /usr/cs-local/371/data-files/ifs directory contains hundreds of IFS models.

2. Create an animation by moving the camera or scene objects over several frames.

3. Add an implicit surface primitive. See "metaball" or "isosurface" in RTR or Google for more information.

4. Shoot multiple (jittered) rays per pixel for antialiasing.

5. Display progress information while rendering (time, number of photons, number of primitives in the scene) and show the image as it is appearing during backtracing.

Submitting Your Solution

1. Put your name, e-mail address, and the name of the file in a doxygen comment at the top of **each file**.

2. Create a "doc-files" directory that contains a **readme.html** file with your name, e-mail address, partner's name (if you worked in a pair) and anything you'd like to point out to me when I'm reviewing your program.
    - This HTML file must display at least two JPG screenshots (stored in the same doc-files directory) of your program in operation.
    - If there are known bugs, extra credit features, or design points of note, list them here.
    - Credit any code that you used from someone else's Spheres project or found on the internet.
    - If you are using your 2-day grace period or a prearranged deadline extension, explain that in the readme file.

3. Delete all generated files using "icompile --clean". Do not hand in a build this week.

4. Change to the **parent** directory of your project and run the FreeBSD command:

```
submit371 scatter
```