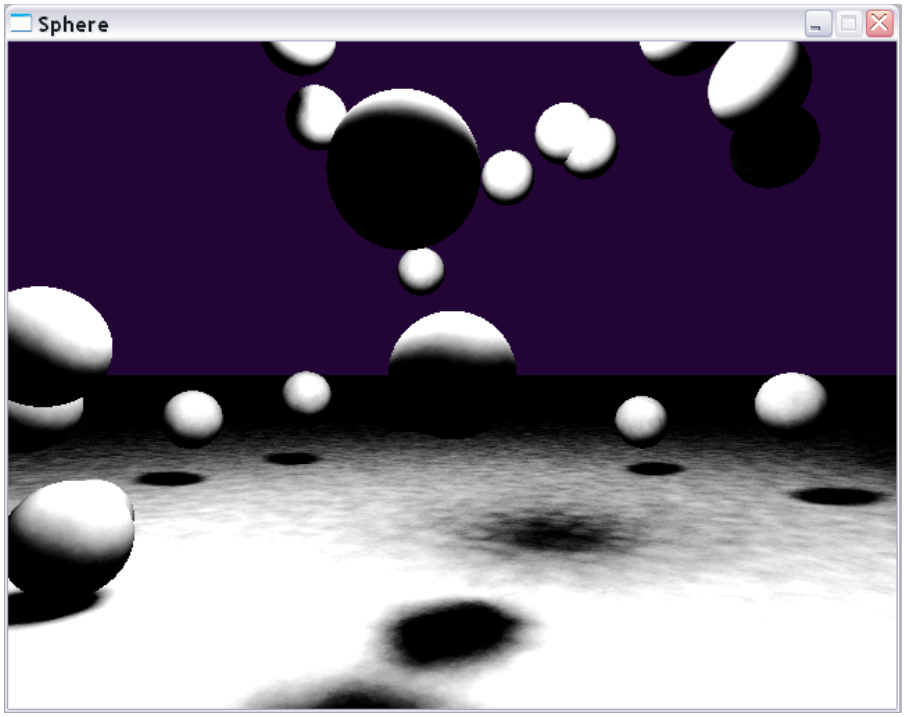
CS371 Spring 2007
Prof. Morgan McGuire

Assigned: Mon. 2/12/07
Due: Mon. 2/19/07 8:00 pm

Project 1: Spheres



This project renders an image of a 3D scene containing spheres, planes, and a light source. It produces this image by simulating the paths of about one million photons. Real photographs capture billions of photons; to produce a reasonable image the project uses numerical methods like random photons and averaging of photon intensities over neighborhoods.

Because most photons that are emitted do not enter the eye, the photon simulation technique is bidirectional to avoid simulating photons that will never be observed. Photons are first simulated "forwards" from the light source until they hit a surface. The photons are then stored in a data structure called a photon map. Next, rays are traced "backwards" from the eye (center of projection) through each viewport pixel into the scene. When a ray hits a surface in the scene the intensity of light from the nearby photons stored in the photon map is computed and written back to the corresponding pixel in the image.

You will continue to use and extend the infrastructure you write for this project in the following weeks. Well-structured, general purpose, and commented code will serve you well. Start this project early in the week and consider working with a partner. A minimum correct solution will probably take about 800 lines of C++ code, 200 of which are in common with your previous TRON project.

Sphere Specification

Your program must have the following features:

1. Simulate single-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. Handle arbitrary scenes containing spheres, planes, and light sources.
3. Support at least one light source.
4. Color the background, where backwards traced rays hit nothing.

Assume that all surfaces are perfectly diffuse (Lambertian) and are white.

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

This is advice on how I would implement this project. You don't have to follow all of my advice. All that you are graded on is how well your program meets (and exceeds) the specifications from the previous section.

You may want to design your program by creating the following classes (examples of several are below):

- Photon - Track the position and direction of a photon that is generated by a light source, and later use the same class to store the position and direction of a photon when it hits a surface.

- Hit – Information about the surface normal and position at which a ray intersects a geometric primitive.

- Raytracer – Class that maintains an array of the objects in the scene and has methods like "storePhoton(Photon&)", "getPhotonsNear(Vector3&, Array<Photon>&)", "emitPhoton()" and "intersect(Ray&, EntityRef, Hit&)". This can also be your GApplet class.

- ForwardTracer – Simulates photons forward from the light source and stores them in the photon map.

- BackwardTracer – Traces rays backwards from the eye through each pixel to find the surface that is visible and shade it appropriately.

- Entity – Base class for SphereEntity and PlaneEntity subclasses so that you don't have to write special-purpose code inside the Raytracer for intersecting different kinds of objects.

Start with your TRON code from the previous project. The basic App/Applet setup and converting an Image4 into a Texture will be identical. It will take about 30 seconds for anything to appear on screen this week, though, so don't expect your 60fps frame rate to do anything.

Work on your project in several phases so that you can test major components before moving on:

1. Backward trace a scene containing a single sphere. Set each pixel to Color3::white() its ray hits a surface and Color3::blue() if it misses everything in the scene. You should see a white disk on a blue background when your program runs.
2. Add a ground plane to the scene. Now you should see a white disk and a white bottom half of the screen.
3. Forward trace the scene into the photon map. Then balance the photon map tree and backward trace to generate an image. During backward tracing, set the intensity of the pixel equal to the number of photons in the photon map within a small radius of the ray hit. The top of the sphere should now be bright and it should cast a shadow. You will need to adjust the radius and the number of photons to get reasonable shading.
4. Instead of making your shading equal to the number of photons in a small radius, find the photons and compute intensity = max(0, –N dot L) for each one. Accumulate the intensities and set the pixel value to the total. You will need to introduce a scaling constant to make the intensity look good.
5. Increase the number of objects in the scene.

The GCamera class contains many convenient helper methods for ray tracing. For example, worldRay() returns the ray through a given pixel and project() converts a 3D location into the pixel that it will color. Use a GCamera (not the GApplet::debugCamera) to position the camera in the scene. The following code places a camera 5m from the origin on the Z-axis and makes it look back at the origin. This means that objects placed within about 5m of the origin will be visible to the camera.

```
m_camera.setPosition(Vector3(0, 0, 5));
m_camera.lookAt(Vector3::zero());
```

The CollisionDetection class contains helper methods to find the intersections between geometric primitives. Note that a photon is a "moving point;" i.e., the intersection of a ray and a surface is the same as the intersection between that surface and a point that starts at the ray origin and travels with velocity equal to the ray direction.

Define a Photon class and data structures for storing where photons hit surfaces. For example, the following code maintains a map over 3D space (PhotonMap) of all photons. The "typedef" statements create new names for parameterized types to use as shorthand. The name on the far right is the newly defined type name, the name in the middle is the type that it is shorthand for.

```
#ifndef PHOTON_H
#define PHOTON_H

#include <G3D/G3DAll.h>

/** Representation of a photon during simulation */
class Photon {
public:
    /** Unit-length direction in which the photon was traveling
        when it hit. */
    Vector3 direction;

    /** Location at which the photon hit the surface. */
    Vector3 position;

    /** Needed for AABSPTree. */
    inline bool operator==(const Photon& other) const {
        return (direction == other.direction) &&
               (position == other.position);
    }
};

/** Needed for AABSPTree. */
inline void getBounds(const Photon& photon, G3D::AABox& box) {
    box = AABox(photon.position);
}

/** Needed for AABSPTree. */
inline unsigned int hashCode(const Photon& photon) {
    return photon.position.hashCode();
}

typedef Array<Photon>     PhotonList;
typedef AABSPTree<Photon> PhotonMap;

#endif
```

The axis-aligned binary space partition tree (AABSPTree) above is like an array, but it can very efficiently extract the photons that lie within a 3D sphere or box.

```
#ifndef ENTITY_H
#define ENTITY_H

#include <G3D/G3DAll.h>
```

```
typedef ReferenceCountedPointer<class Entity> EntityRef;

/**
 Base class for objects in the scene.
 */
class Entity : public ReferenceCountedObject {
public:

    /**
      If worldRay intersects this object before hitDistance, sets
hitDistance to the distance
      to the intersection and fills out hitResult.  Otherwise
leaves hitResult and hitDistance
      unmodified.

      @param worldRay Direction must have unit length.
     */
    virtual void intersect(const Ray& worldRay, float&
hitDistance, class Hit& hitResult) = 0;

};

#endif
```

Emit random photons from your light source. That is, generate photons within a small area of random positions and send them out in random directions. Vector3::random() generates a random direction. The function uniformRandom() generates random floats; you can also use the methods on various classes like Triangle and Sphere that generate random points on their surface.

Generate about one million photons (fewer when testing early on), but only store photons that hit the scene reasonably close to the light source so that your program doesn't run out of memory. 30 meters is a reasonable radius. Don't bother generating photons that immediately move away from the scene. For example, if your light is above everything else in the scene, don't generate photons with a positive Y direction.

Although you don't have to use it (I didn't in my solution), it may help you to know that many C++ classes use a design pattern called an iterator for walking through their elements. A class that supports iteration has a begin() method and an end() method. You use them like this:

```
SomeClass::Iterator current = collection.begin();
SomeClass::Iterator end = collection.end();

while (current != end) {

      … do something with *current …

      ++current;  // DO NOT POST-INCREMENT (current++);
                  // IT IS REALLY SLOW!
```

```
}
```

You may want to switch to running optimized builds during the development process because the program is so computationally intensive. For a scene with 30 spheres and one plane, it should take 10 – 30s to forward trace, 30s – 1m to balance the tree, and less than 10s to backward trace in an optimized build.

Your program will allocate about 200 MB of memory. If you notice the system swapping to disk frequently and running slowly, then you might want to decrease the number of photons simulated. Decreasing the number of objects in the scene will not reduce the memory requirement noticeably (although it will speed up your simulation).

Innovate!

You are invited to go beyond the requirements for this and other projects. Extra work typically takes the form of new features or structuring your code in a particularly good design. Perfectly completing the requirements for each project with zero additional work earns you a B+. To receive an A or an A+ you must exceed the minimum requirements. The real reason to innovate in your projects is not the grade, of course, but instead that graphics is exciting so you'll want to add to your projects.

Some ideas for extending the Sphere project include:

1. Make your program render incrementally for the backwards tracing phase. You can do this either by breaking up your backwards tracer loop so that it renders a few rows of the image for every onGraphics call or by rendering using a separate thread (see GThread). Be aware that threads can be tricky to debug, but offer the possibility of launching multiple threads to take advantage of our dual-core processors.

2. Add color. Track the color of each object in the Hit class and multiply the color by the shading intensity.

3. Make an interesting scene. Using graph paper you can figure out a set of coordinates and radii to place spheres at so that the result is a group of snowmen (well, snowmen aren't very impressive, but you get the idea).

4. Add other primitives. Classes like Box, Triangle, Capsule, Cylinder support the same intersection operations as Sphere and Plane.

5. Replace the default background with an image by loading an image from disk and then NOT overriding the background color if no surface is hit during backward tracing.

6. Optimize the ray tracing using AABSPTree. Make another AABSPTree (different from your photon map). The beginRayIntersection method

allows you to cast a ray very efficiently through a scene containing many objects. You should be able to handle scenes with hundreds of objects using this technique.

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**, e.g.:

```
/**
 @file Raytracer.cpp
 @author Morgan McGuire, morgan@cs.williams.edu

 Implements the main applet and scene class for Spheres.
 */
```

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 one JPG screenshot (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 TRON 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. Make a clean build (using "icompile --clean") and then compile (and test) an **optimized** build of your program using "icompile –O".

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

```
submit371 spheres
```