CMSC 635 programming assignment #1:

Due: Feb 13 2013 11:59 pm

Ray

1. Introduction

IN this project you will implement a Whitted ray tracer in C++, using a graphics package called G3D. Wikipedia has a surprisingly good overview of the Whitted ray tracer (http://en.wikipedia.org/wiki/Ray_tracing_(graphics)).

G3D is a graphics package that supports in a wide array of graphics tasks. The library was spearheaded by Professor Morgan McGuire, who earned his PhD at Brown University. In this and further assignments, we will be using a subset of G3D to assist with physically-based rendering. G3D provides acceleration data structures, ray intersection algorithms (yeah!! You don't need to do the tedious (or exciting ☺ computation yourself), and abstractions for materials and lights.

The primary purpose of this project is to give you experience developing using G3D. We expect the actual ray tracing to come easily, and the bulk of the work will come from learning your way around the support code.

G3D will make ray intersection, reflection, and refraction easy. You shouldn't need to write more than 50 lines of code for this project. However, please start earlier, because the reading and thinking will take a lot of time.

2. Requirements

We will assume you are familiar with recursive ray tracing. If you are not, you can get up to speed by reading some of the CMSC 435 lectures slides or by consulting the faculty assistant (Keqin@umbc.edu), and her office hours is Thursday 11:00 am-12:00noon.

Heckbert's path notation [Heckbert 1990] describes the way recursive ray can bounce. Whitted's recursive ray tracer finds (LD*S*E). Yours must support:

- Direct lighting from point lights (you may assume all light sources are point lights)
- Shadows
- Reflection
- Refraction
- The output of your

The output of your ray tracer should match the demo (runs on Linux machine only): http://www.csee.umbc.edu/courses/graduate/635/spring13/bin/ray_demo.

3. Support Code

Support code can be downloaded from http://www.csee.umbc.edu/courses/graduate/635/spring13/asgn/01ray/01Ray.tar . You may develop using any editor you like, although we strongly suggest Qt Creator for its auto-complete ability. Qt Creator can be found at http://qt-project.org/ .

You can run your ray tracer in the command line. If you would like to use a scene other than the default, you can specify the path to a scene file as the first argument to the program. Several scene files (ending in .scn.any) are provided in http://www.csee.umbc.edu/courses/graduate/635/spring13/data/g3d/scene.tar .

The stencil consists of two classes: App and World. App contains the core program logic, while World stores details about the scene. You just need to implement App::raytrace, which is the recursive ray tracing function. Everything else is already set up for you.

The support code is documented in app.h and world.h. You can find G3D documentation online http://g3d.sourceforge.net/manual/9.00 .

Particularly useful G3D documentation pages include:

A shortlist of primitives for physically-based rendering: http://g3d.sourceforge.net/manual/9.00/topicindex.html#raytrace

A full list of classes G3D provides: http://g3d.sourceforge.net/manual/9.00/namespace_g3_d.html

The following classes were used extensively in implementing the demo – you may find them useful: Surfel, Light, Ray, Camera, CoordinateFrame (aliased as CFrame), Vector3.

G3D makes extensive use a C++ smart pointer class called shared_ptr. We expect some students may not have run into smart pointers before, so we provide a brief overview here. If you are familiar with smart pointers, you may skip this section.

A smart pointer is an object that can be used like a normal pointer, but provides some useful feature in addition. shared_ptr 's 'useful feature' is automatic memory management, using a technique called reference counting.

A shared_ptr 's reference count is the number of shared_ptrs that currently refer to the same thing. The refcount is incremented when one shared_ptr is copied to another; the refcount is decremented when a shared_ptr is destroyed or changed. When the refcount reaches zero, the shared_ptr deletes the memory it points to.

shared_ptr assumes that you only access the memory through a shared_ptr : therefore, if there are no shared_ptr s referring to the memory, then your program no longer knows about the memory, so it's safe to delete it. This means you should never directly access the pointer wrapped by the shared_ptr , even if the shared_ptr API allows you to do so.

You can _nd more about shared_ptrs on Google. Note that G3D aliases some shared_ptr types as Type::Ref . That is, you can use Type::Ref and shared_ptr<Type> interchangeably. The ::Ref alias isn't supported on all G3D objects, however.

TL;DR - shared_ptr<Type> (or, sometimes, Type::Ref) acts like Type* , except you don't need to call delete yourself. Never mix shared_ptr<Type> with a raw Type* .

Finally, you may also get weird-looking errors if you accidentally use someptr.foo instead of someptr->foo .

4. Handing in

You are required to hand in a README file with your project, as a plain text file. It should contain anything that could make the faculty assistant's life easier (e.g., major design choices, known bugs, kludges).

Do not hard code any paths in your program!

To hand in, email your code **in one tarball** to keqin@umbc.edu .

Reference:

Heckbert, P. S. [1990]. Adaptive radiosity textures for bidirectional ray tracing, *Computer Graphics (SIGGRAPH 90 Proceedings)*, Vol. 24, pp. 145–154.