6.837 Introduction to Computer Graphics
Assignment 3:
Ray Tracing and Phong Materials
Due Wednesday October 1, 2003 at 11:59pm

This week, we add local and global illumination to our Ray Caster. Because we cast secondary rays to account for shadows, reflection and refraction, we now call it a Ray Tracer.

We clean up and improve shading calculation (local illumination). We define an abstract material class and derive a Phong Material class that implements a diffuse and a specular component (highlight).

You will encapsulate the high-level computation in a `RayTracer` class that will be responsible for sending rays and recursively computing colors along them.

To compute cast shadows, you will send rays from the visible point to the light source. If an intersection is reported, the visible point is in shadow and the contribution from that light source is ignored. Note that shadow rays must be sent to all light sources.

You will add reflection and refraction effects. For this, you need to send secondary rays in the mirror and transmitted directions, as explained in class. The computation is recursive to account for multiple reflections and or refractions.

1 Tasks

- Implement an abstract `Material` class and derive a `PhongMaterial` class that computes Phong shading.
- Update your intersection routines to return the Material of the intersected object, instead of just a simple color.
- Create a new class `RayTracer` that computes radiance (color) along a ray. Update your main function to use this class for the rays through each pixel.
- Treat the case of multiple light sources by accumulating shading contributions for all light sources.
- Compute cast shadows by sending rays toward the light sources.

- Compute perfect mirror reflection by sending rays recursively in the mirror direction.
- Compute transparency effects by sending rays recursively in the refracted direction.
- Provide a `README.txt` file that discusses any problems you encountered, how long it took to complete the assignment, any extra credit work that you did and how we can test the new features.
- Extra credit: semi-transparent shadows where the attenuation depends on the distance traveled in the transparent object; nested refracting materials; shadow ray acceleration using early termination and without returning the normal and material; Fresnel reflection term; other BRDF models such as Cook-Torrance or Ward; anisotropic BRDFs; solid textures.

2 Classes you need to update/write

The `Object3D` and `Hit` classes must now store more detailed information about the properties of the surface, encapsulated in a `Material class`. You must update your `Object3D` class (and the constructors of all subclasses) to replace the color by a pointer to a `Material`. You must also update your intersection routines so that they return a pointer to a Material and not a color in case of an intersection.

2.1 Material and PhongMaterial

The `Material` class has a pure virtual method `Shade` that computes the local interaction of light and the material. It takes as input the viewing ray, the Hit data structure, and light information.

```
virtual Vec3f Material::Shade
  (const Ray &ray, const Hit &hit, const Vec3f &dirToLight,
   const Vec3f &lightColor) const = 0;
```

In addition to the diffuse object color used in the last assignment, the `Material` class stores information about transparency and reflection. It must store two colors for the reflection and transparency coefficients, and an index of refraction. You must implement accessor functions for these fields.

```
Material::Material(const Vec3f &diffuseColor,
   const Vec3f &transparentColor,
   const Vec3f &reflectiveColor,
   float indexOfRefraction);
```

You must derive a subclass `PhongMaterial` that computes shading using both a diffuse and specular term. The class takes as additional parameters a specular color and a scalar exponent. The constructor expected by the parser is:

```
PhongMaterial::PhongMaterial(const Vec3f &diffuseColor,
   const Vec3f &specularColor,
   float exponent,
   const Vec3f &transparentColor,
   const Vec3f &reflectiveColor,
   float indexOfRefraction);
```

Implement the `PhongMaterial::Shade` function using the version of the Phong model taught in class based on the angle between the eye ray and the reflected ray.

2.2 RayTracer

The `RayTracer` class encapsulates the computation of radiance (color) along rays. It stores a pointer to the `SceneParser` for access to the geometry and light sources. The constructor is:

```
RayTracer(SceneParser *s, int max_bounces, float min_weight);
```

The main method of this class is `traceRay` that, given a ray, computes the color seen from the origin along the direction. This computation can be recursive because of reflection and refraction. We therefore need a stopping criterion to prevent infinite recursion. `traceRay` takes as additional parameters the current number of bounces (recursion depth) and a ray weight that indicates how much the contribution will be dimmed in the final image. The corresponding maximum recursion depth and the minimum ray weight are fields of `RayTracer`, which are passed as command line arguments to the program. Note that the weight is a scalar that corresponds to the magnitude of the color vector.

```
Vec3f traceRay(Ray &ray, float tmin, int bounces,
  float weight, float indexOfRefraction, Hit &hit) const;
```

First, implement `traceRay` so that it computes the closest intersection with the scene and performs shading. For ambient lighting, you should use the `getDiffuseColor` method of the `Material` pointer stored by `Hit` and multiply it by the ambient light of the `SceneParser`. Add the contributions of the light sources in the scene by iterating over the light sources stored by the `SceneParser` and call the `Shade` method of the `Material`.

Shadows

Next, you need to implement cast shadows. Modify the code described above to test for each light whether the line segment joining the intersection point and the light source intersects an object. If there is an intersection, then discard the contribution of that light source. Recall that you must displace the ray origin slightly away from the surface, or equivalently set `tmin` as ϵ. Note that in this naïve version, semi-transparent objects still cast opaque shadows. Implement something better for extra credit.

Mirror reflection

You will then implement mirror reflection. This involves the use of secondary rays in the direction symmetric with respect to the normal. If the material is reflective (`getReflectiveColor()` $>$ (0,0,0)), then you must create a new ray with origin the current intersection point, and direction the mirror direction. For this, we suggest you write a function:

```
Vec3f mirrorDirection(const Vec3f &normal, const Vec3f &incoming);
```

Trace the secondary ray with a recursive call to `traceRay` using modified values for the recursion depth and ray weight. The ray weight is simply multiplied by the magnitude of the transparency color. Make sure that `traceRay` checks these stopping conditions. For the attenuation, check if the magnitude of the color is smaller than the maximum weight. Add the reflected contribution to the color computed for the current ray. Don't forget to take into account the reflection coefficient of the material.

Refraction

Finally, you will implement the computation of transmitted rays. For semi-transparent objects, you will trace a new ray in the transmitted direction. We suggest you implement a function `transmittedDirection` that given an incident vector, a normal and the indices of refraction, returns the transmitted direction.

```
bool transmittedDirection(const Vec3f &normal, const Vec3f &incoming,
   float index_i, float index_t, Vec3f &transmitted);
```

Be careful about the direction of the vectors and the ratio of indices. We make the simplifying assumption that our transparent objects exist in a vacuum, with no intersecting or nested refracting materials. This allows us to determine the incident and transmitted index of refraction simply by looking at the dot product between the normal and the incoming ray. Indeed, because we now consider transparent objects, we might hit the surface of a primitive from either side, depending on whether we were inside or outside the object. Note that the dot product of the normal and ray direction is negative when we are outside the object, and positive when we are inside. You will use this to detect whether the new index of refraction is 1 or the index of the hit object. (The index of refraction for the material surrounding the ray origin is passed as an argument to `traceRay`.) If you are inside the object, you must flip the normal before computing the transmitted direction. It is not necessary to flip the normal when computing the direction of the reflected ray (you'll get the same answer). For shading, use the original normal.

Recap

To summarize, the color returned by `traceRay` is the sum of the ambient contribution, the shading contribution of the visible light sources, the mirror reflection and the transmitted contribution.

3 Updated Utilities Provided

Hit.h

Hit has been modified to store the material of the intersection point. We provide you with the updated file.

Light (`light.h`)

Updated to include point light sources. Also getIllumination() now also provides the distance to the light source, which is important for shadow rays.

Parsing input files (`scene_parser.h, and scene_parser.C`)

These files have been updated to parse the point light sources and Phong materials. The `PhongMaterial` constructor you will write is called from the parser. Look in the `scene_parser.C` file for details.

Command line arguments

In addition to the parameters used in the previous assignments, your program should take the maximum number of bounces, and the cutoff for ray weight as command line arguments. Make sure the following example works, as this is how we will test your program:

```
raytracer -input scene.txt -output out.tga -bounces 5 -weight 0.01
```

4 Hints

- You do not need to declare virtual all methods in a class, only the ones which subclasses will override.
- Print as much information as you need for debugging. When you get weird results, don't hesitate to use simple cases, and do the calculations manually to verify your results. Perhaps instead of casting all the rays needed to create an image, just cast a single ray (and its corresponding ray tree).
- Modify the test scenes to reduce complexity for debugging: remove objects, remove light sources, change the parameters of the materials so that you can view the contributions of the different components, etc.
- Comment your code, we take this into account when grading.