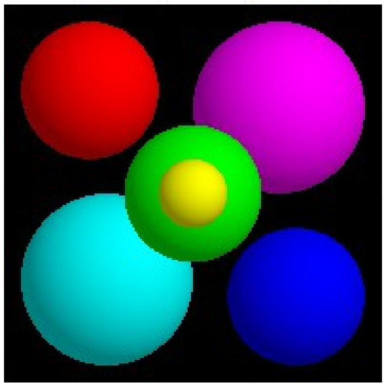
Homework #9

Ray Tracing



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 Ray Tracer 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 ray sin the mirror and transmitted directions, as explained in class. The computation is recursive to account for multiple reflections and or refractions.

Tasks

- Derive a `PhongMaterial` class from the `Material` class, that computes Phong shading.
- 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.

Classes you need to write/update

- Make the original `Material` class pure virtual by adding a pure virtual method `Shade` that computes the local interaction of light and the material:

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

It takes as input the viewing ray, the `Hit` data structure, and light information and returns the color for that pixel.
In addition to the diffuse object color used in the last assignment, the

`Material` class stores also 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.
- Create a new class `RayTracer` that computes the radiance (color) along a ray. Update your main function to use this class for the rays through each pixel. This class encapsulates the computation of radiance (color) along rays. It stores a pointer to the `SceneParser` for access to the geometry and light sources. Your constructor should have these arguments (and maybe others, depending on how you handle command line arguments):

```
RayTracer(SceneParser *s, int max_bounces,
          float cutoff_weight, bool shadows, ...);
```

  The main method of this class is `traceRay` that, given a ray, computes the color seen from the origin along the direction. This computation is recursive for reflective or transparent materials. 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 the percent contribution of this ray to the final pixel color. The corresponding maximum recursion depth and the cutoff ray weight are fields of `RayTracer`, which are passed as command line arguments to the program. Note that 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;
```

To refract rays through transparent objects, `traceRay` is also passed the `indexOfRefraction` (see below), and returns the closest intersection in `hit`, which is used to create the depth & normal visualizations. You can test your code at this point with examples from previous assignments.

- Add support for the new command line arguments: `-shadows`, which indicates that shadow rays are to be cast, and `-bounces` and `-weight`, which control the depth of recursion in your ray tracer.
- Implement cast shadows by sending rays toward each light source to test 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 naive version, semi-transparent objects still cast opaque shadows.
- Implement *mirror reflections* for reflective materials (`getReflectiveColor()`> (0,0,0)) by sending a ray from the current intersection point in 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 reflected color. Make sure that `traceRay` checks the appropriate stopping conditions. 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.

- Implement *transparency* effects by sending rays recursively in the refracted direction. If the material is transparent (`getTransparentColor()` > (0,0,0)), 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.

- Incorporate the updated code for point light sources and verify that your shading and shadowing implementation correctly renders these lights. The `PointLight` source method `getIllumination()` method handles the distance attenuation term needed for Phong shading i.e. the light intensity is attenuated by moving away from the light source.
- **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.

Utilities Provided

Parsing command line arguments & input files
The scene parser has been updated to include the extra information in the scene files e.g. point lights and phong materials. The `PhongMaterial` constructor you will write is called from the parser. Look in the `scene_parser.C` file for details.

Lights
`light.h` has been modified to include point light sources.

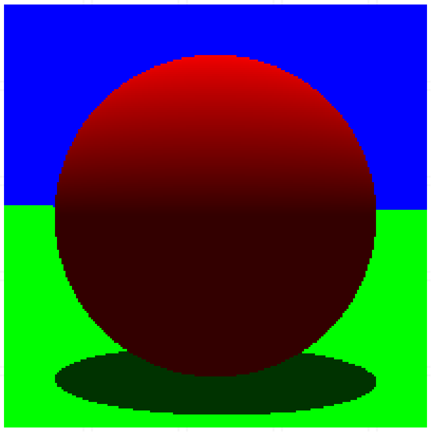
Other
You will need to update the `Makefile` to include the other classes you are writing.

Hints

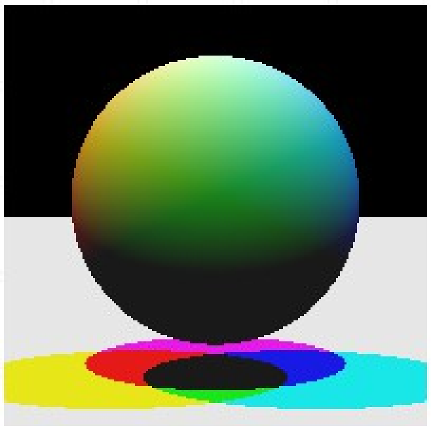
- You do not need to declare all methods in a class virtual, only the ones which subclasses will override.
- Print as much information as you need for debugging. When you get weird results, don't hesitate to use simple cases, and do the calculations manually to verify your results. Perhaps instead of 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.

Sample Results

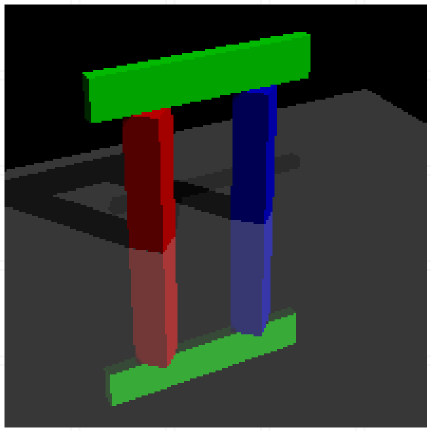
```
raytracer -input scene4_01_sphere_shadow.txt -size 200 200 -output output4_01.tga
   -shadows
```



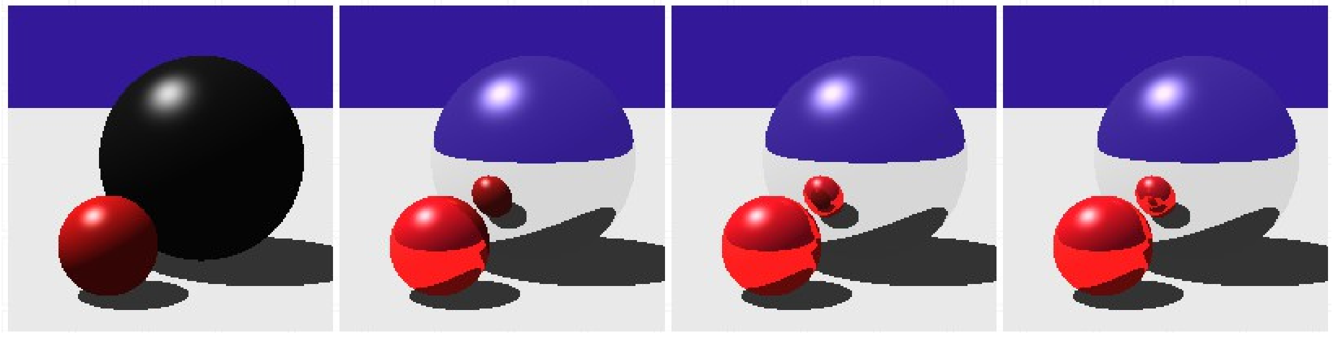
```
raytracer -input scene4_02_colored_shadows.txt -size 200 200 -output output4_02.tga
   -shadows
```



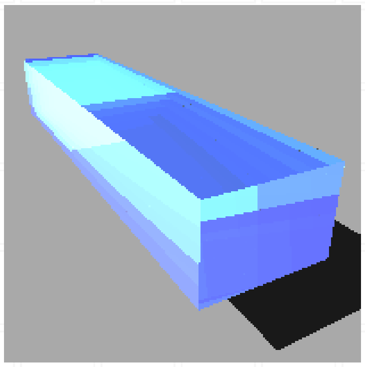
```
raytracer -input scene4_03_mirrored_floor.txt -size 200 200 -output output4_03.tga
   -shadows -bounces 1 -weight 0.01
```



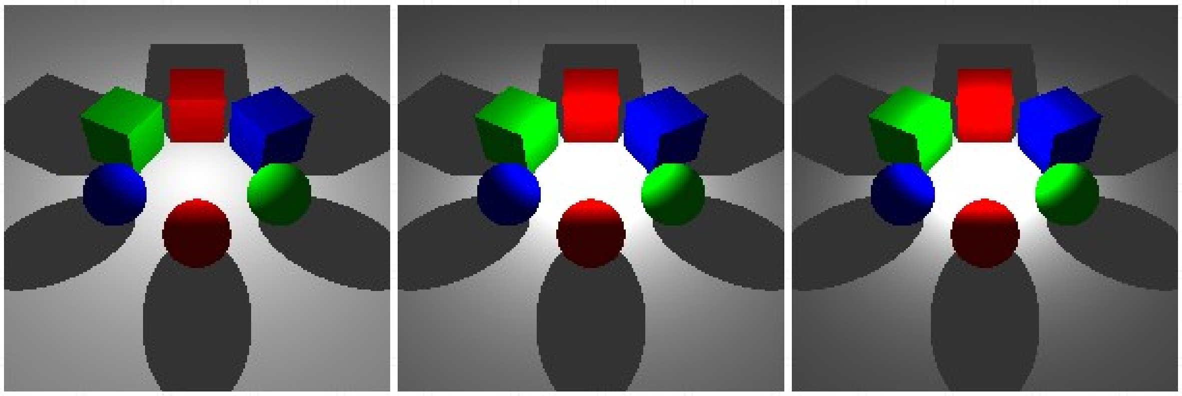
```
raytracer -input scene4_04_reflective_sphere.txt -size 200 200 -output
   output4_04a.tga -shadows -bounces 0 -weight 0.01
raytracer -input scene4_04_reflective_sphere.txt -size 200 200 -output
   output4_04b.tga -shadows -bounces 1 -weight 0.01
raytracer -input scene4_04_reflective_sphere.txt -size 200 200 -output
   output4_04c.tga -shadows -bounces 2 -weight 0.01
```



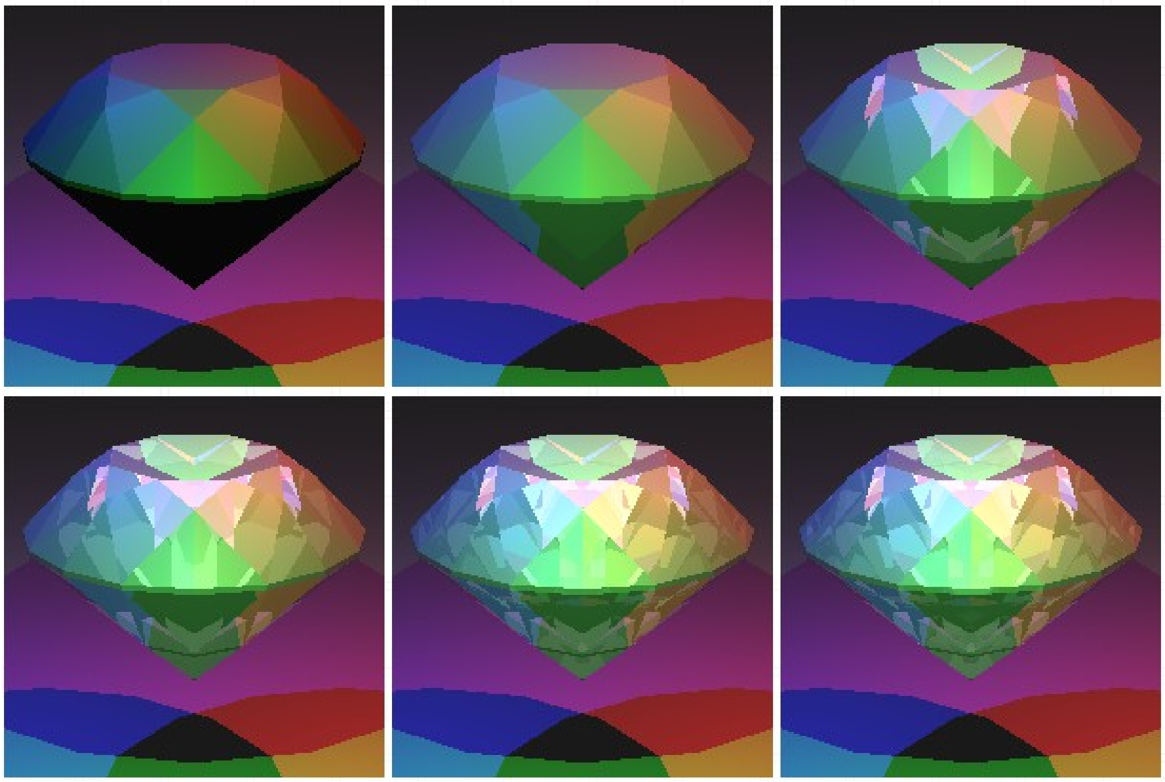
```
raytracer -input scene4_04_reflective_sphere.txt -size 200 200 -output
   output4_04d.tga -shadows -bounces 3 -weight 0.01
raytracer -input scene4_05_transparent_bar.txt -size 200 200 -output output4_05.tga
   -shadows -bounces 10 -weight 0.01
```



```
raytracer -input scene4_11_point_light_circle.txt -size 200 200 -output
   output4_11.tga -shadows
raytracer -input scene4_12_point_light_circle_d_attenuation.txt -size 200 200
   -output output4_12.tga -shadows
raytracer -input scene4_13_point_light_circle_d2_attenuation.txt -size 200 200 -
   output output4_13.tga -shadows
```



```
raytracer -input scene4_14_faceted_gem.txt -size 200 200 -output output4_14a.tga
   -shadows -shade_back -bounces 0 -weight 0.01
raytracer -input scene4_14_faceted_gem.txt -size 200 200 -output output4_14b.tga
   -shadows -shade_back -bounces 1 -weight 0.01
raytracer -input scene4_14_faceted_gem.txt -size 200 200 -output output4_14c.tga
   -shadows -shade_back -bounces 2 -weight 0.01
raytracer -input scene4_14_faceted_gem.txt -size 200 200 -output output4_14d.tga
   -shadows -shade_back -bounces 3 -weight 0.01
raytracer -input scene4_14_faceted_gem.txt -size 200 200 -output output4_14e.tga
   -shadows -shade_back -bounces 4 -weight 0.01
raytracer -input scene4_14_faceted_gem.txt -size 200 200 -output output4_14f.tga
-shadows -shade_back -bounces 5 -weight 0.01
```



Acknowledgment

This homework is adapted from class 6.837 at MIT.