6.837 Introduction to Computer Graphics
Assignment 2:
Transformations and additional primitives
Due Wednesday September 24, 2003 at 11:59pm

In this assignment, you will add new primitives, planes and triangles, as well as affine transformations. You will also implement a perspective camera, and add two simple shading modes: diffuse shading and normal shading.

At this point, we will leave the shading computation in the main loop. In the next assignment, we will develop a special class for this computation. For normal shading, you will simply display the coordinates of the normal vector as an (r, g, b) color. For example a normal pointing in the z direction will be displayed as pure blue $(0, 0, 1)$. You should use black as the color for the background (undefined normal).

Diffuse shading is our first step to model the interaction of light and materials. The scene parser now provides you with alight source. Given the direction to the light $\vec{L}$ and the normal $\vec{N}$, we can compute the diffuse shading as a clamped dot product

$$d = \begin{cases} I \ \vec{L}.\vec{N} & if \qquad \qquad \vec{L}.\vec{N} > 0, \\ 0 & otherwise \end{cases}$$

If the visible object has color (r, g, b), and the light source has color (L_r, L_g, L_b), then the color of the pixel is (rL_rd, gL_gd, bL_bd).

The scene parser reads an array of light sources (see `scene_parser.h`. For this assignment, you can consider that there is only one light (index 0). But you can also decide to do it the clean way and write a loop on all light sources and add their contributions (i.e. add all the (rL_rd, gL_gd, bL_bd)). See below how to access the light source.

1 Tasks

- Update the `Hit` data structure to store normals. Update your sphere intersection routine to pass the normal to the hit.

- Add simple normal and diffuse shading. At this point, they can be implemented in the main loop. Diffuse shading should include an ambient term

(see below in the light section.)

- Add a perspective camera class, and implement the ray-generation method.

- Implement an infinite plane primitive. It should be a subclass of `Object3D` and implement the `intersect` method, including normal computation.

- Implement a triangle primitive and the corresponding ray-triangle intersection.

- Derive a subclass `Transformation` from `Object3D`. This class stores a 4x4 matrix and a pointer to an `Object3D` that undergoes the transformation. Implement the ray and normal transformation for proper intersection.

- Do 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 credits: Implement two ray-triangle intersection methods and compare; cones; cylinders; Constructive Solid Geometry (CSG); instantiation; IFS; non-linear cameras.

2 Classes you need to write

2.1 PerspectiveCamera

The `PerspectiveCamera` class derives from `Camera`. Choose your favorite internal camera representation. The file format provides you with the viewpoint (center), the main direction (looking towards the scene), and the field of view (see Fig. 1). See `scene_parser.C` line for the specification of the constructor.
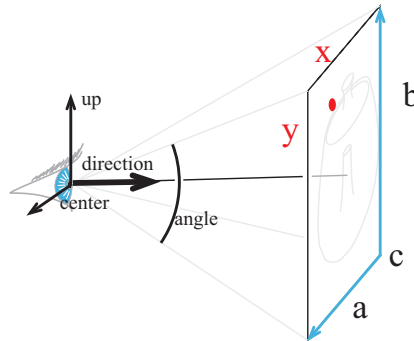

Figure 1: Perspective camera.

The Constructor is:

```
PerspectiveCamera(Vec3f &center, Vec3f &direction, Vec3f &up, float
angle);
```

2.2 Plane

`Plane` derives from `Object3D`. Use the representation of your choice, but the constructor is assumed to be:

`Plane(Vec3f &normal, float d, Vec3f &color);`

`d` is the offset from the origin, meaning that the plane equation is $\vec{P}.\vec{normal} = d$. You can also implement other constructors (e.g. using 3 points.).

Implement `intersect`, and remember that you now also need to update the normal stored by `hit`, in addition to the intersection distance t and color.

2.3 Triangle

Triangles also derive from `Object3D`. The constructor takes 3 vertices (Vec3f):

`Triangle(Vec3f &a, Vec3f &b, Vec3f &c, Vec3f &color);`

Use the method of your choice to compute the ray-triangle intersection (general polygon with in-polygon test,; barycentric coordinates; or Plücker space.)

2.4 Transformation

We propose that you implement `Transformations` as a simple `Object3D` subclass. Similar to a `Group`, a `Transformation` will store a pointer to an `Object3D` (but only one, not an array). The intersection routine will first transform the ray, then delegate to the `intersect` routine of the child object, and finally transform the normal according to the rule seen in class: the origin undergoes the inverse transformation and the direction undergoes only the linear part of the inverse transform (i.e. the direction does not undergo translation).

You must decide whether you want to keep the direction of the ray normalized or not, as discussed in class. If you decide to use non-normalized rays, you might need to update some of your intersection code.

The constructor of a `Transformation` takes a 4x4 matrix as input and a pointer to an `Object3d`:

`Transform(Matrix &m, Object_3D *o);`

3 Utilities Provided

Ray.h

Unchanged.

Hit.h

Hit must be modified to store the normal of the intersection point. We provide you with the updated file.

Images and Linear Algebra (`image.h`, `vectors.h`, `matrix.h`, and `matrix.C`)

The matrix classes have been updated to better handle transformations. Be sure to download the file again.

Light (`light.h`) Light source description. The abstract class `Light` provides a virtual function `getIllumination` that computes the normalized illumination direction and the intensity and color at a given location in space:

```
void getIllumination (const Vec3f &p, Vec3f &dir, Vec3f &col);
```

where p is the intersection point that you want to shade, and the function passes the direction towards the light source in `dir` and the light color and intensity in `col` (i.e. the components of `col` can be greater than 1 because they also encode the light intensity.)

The scene parser now reads light sources. You can access the number of light sources through `getNumLights ()` and pointers to the light sources using `Light* getLight(int i);`. Although for this assignment you can assume that there will be only one light source, it is easy to treat the case of multiple light sources by looping through them.

You must add an *ambient* term to your shading, because otherwise parts facing away from the light source appear completely black. The parser provides an ambient color. Simply multiply the ambient color by the color of the object.

Note that if the ambient color is $(1, 1, 1)$ and the light source color is $(0, 0, 0)$, then you have the constant shading used in assignment 1.

Given the material color $color(P)$ at the intersection point P; the normal $\vec{n}$; the direction $\vec{dir_i}(P)$ and color $L_i(P)$ of the light source returned by `getIllumination` for light index i, then the color of the pixel is:

$$pixel(P) = ambient * color(P) + \sum_i clamped(\vec{dir_i}(P).\vec{n}) \ \ color(P) * L_i(P)$$

where color vectors are multiplied term by term. Remember that the dot product is clamped to positive values.

Parsing command line arguments & input files (`parse.C`, `scene_parser.h`, and `scene_parser.C`)

These files have been updated to parse the new primitives that you must implement.

Your program should take a number of command line arguments to specify the input file, output image size and output file. Make sure the following example works, as this is how we will test your program:

```
raycast -input scene.txt -size 100 100 -output image.tga
```

The new rendering mode by default should perform diffuse shading instant of the constant color per object. You must add a new option -normal to perform shading according to the normal (see above).

```
raycast -input scene.txt -size 100 100 -normal normal.tga
```

A simple scene file parser that is adequate for this assignment is provided. The `OrthographicCamera`, `Group` and `Sphere` constructors and the `Group::addObject` method you will write are called from the parser. Look in the `scene_parser.C` file for details.

Other

A simple `Makefile` for use with `g++` and Linux is provided.

4 Hints

- Parse the arguments of the program in a separate function outside the main function. It will make your code easier to read.

- Implement the normal and diffuse shading before the transformations. It will help you assess whether your normals are correct. Implement normal shading first (this is a general advice, always test sub-expressions when you can.)

- Use the various rendering modes (normal, diffuse, distance) to debug your code.

- For CSG (extra credits), you need to implement a new `intersectAll` method for your `Object3D` classes. This function needs to return *all* the intersection of the ray with the object, and not only the first one. This is because the first intersection of a simple object might be outside the intersection of the two objects, for example. We advise you to use the interval solution to perform the CSG along the ray. Fun CSG operations are intersection and difference.

- For additional primitives such as cones and cylinder, implement the simple case of axis-aligned primitives and use transformations.