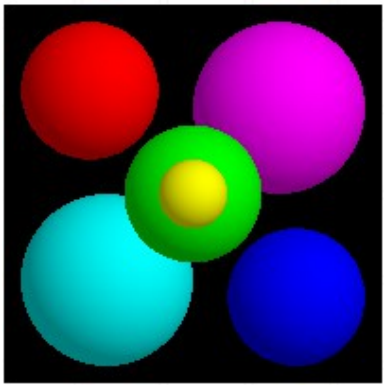
Homework #8

Ray Casting II



In this assignment, you will add new primitives (*planes* and *triangles*) and *affine transformations*. You will also implement a *perspective* camera, and two simple shading modes: *normal* visualization and *diffuse* shading. For the normal visualization, you will simply display the **absolute** value of the coordinates of the normal vector as an (*r*, *g*, *b*) color. For example, a normal pointing in the positive *or* negative z direction will be displayed as pure blue (0, 0, 1). You should use black as the color for the background (undefined normal).

You will also implement diffuse shading. Given the direction to the light L and the normal N we can compute the diffuse shading as a clamped dot product:

$$d = \begin{cases} L \cdot N & \text{if } L \cdot N > 0 \\ 0 & \text{otherwise} \end{cases}$$

If the visible object has color $c_{object} = (r, g, b)$, and the light source has color $c_{light} = (L_r, L_g, L_b)$, then the pixel color is $c_{pixel} = (rdL_r, gdL_g, bdL_b)$. Multiple light sources are handled by simply summing their contributions. We can also include an ambient light with color $c_{ambient}$, which can be very helpful in debugging. Without it, parts facing away from the light source appear completely black. Putting this all together, the formula is:

$$c_{pixel} = c_{ambient} * c_{object} + \sum_i max(L_i \cdot N, 0) * c_{light} * c_{object}$$

The object color c_{object} here acts as the diffuse and ambient coefficients from the lectures and earlier homeworks. Color vectors are multiplied term by term. Note that if the ambient light color is $(1,1,1)$ and the light source color is $(0,0,0)$, then you have the constant shading used in the previous homework, where the ambient color is applied to every pixel.

Tasks

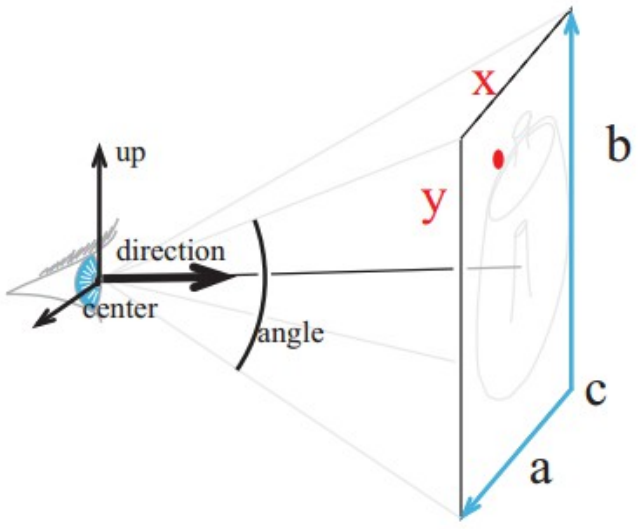
- Update the **Hit** data structure to store normals. Update your sphere intersection routine to pass the normal to the hit.
- Add simple *normal* and *diffuse* shading. At this point, they can be implemented in the main loop. Diffuse shading should include an ambient term (no specular yet!).
- Add a **perspective** camera class, and implement its 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.

Classes you need to write/update

- The `Hit` class has been modified to store the normal of the intersection point. Update your sphere intersection routine to pass the normal to the `Hit`.
- Implement the new rendering mode, ***normal*** visualization. Add code to parse an additional command line option `-normals <normal_file.tga>` to specify the output file for this visualization (see examples below).
- Add diffuse shading. We provide the pure virtual `Light` class and a simple **directional** light source. Scene lighting can be accessed with the `SceneParser::getLight()` and `SceneParser::getAmbientLight()` methods. Use the `Light` class method:

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

  to find the illumination at a particular location in space. `p` is the intersection point that you want to shade, and the function returns the *normalized* direction toward the light source in `dir` and the light color and intensity in `col`.



- Add a `PerspectiveCamera` class that derives from `Camera`. Choose your favorite internal

camera representation. Similar to an orthographic camera, the scene parser provides you with the camera center, main direction (looking into the scene), and up vectors. But for a perspective camera, the field of view is specified with an angle (as shown in the diagram).

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

You should compute the extent of the field of view (i.e. left, right, top, bottom) of the viewing space from the angle using simple trigonometry. In particular, assuming the viewing plane is at distance 1 from the camera, then the length of the side of the viewing rectangle (square in that case) is $s = \tan\frac{\theta}{2}$ where θ is the field of view. In that case, the extent of the viewing space is $t = r = s$ and $l = b = -s$.

- Implement `Plane`, an infinite plane primitive derived from `Object3D`. Use the representation of your choice, but the constructor is assumed to be:

```
Plane(Vec3f &normal, float d, Material *m);
```

`d` is the offset from the origin, meaning that the plane equation is $p \cdot n$ = `d`. You can also implement other constructors (e.g. using 3 points). Implement `intersect`, and remember that you also need to update the normal stored by Hit, in addition to the intersection distance `t` and color.

- Implement a triangle primitive which also derives from `Object3D`. The constructor takes 3 vertices:

```
Triangle(Vec3f &a, Vec3f &b, Vec3f &c, Material *m);
```

Implement the intersection using the method explained in the lectures. We can compute the normal by taking the cross-product of two edges, but note that the normal direction for a triangle is ambiguous. We'll use the usual convention that **counter-clockwise** vertex ordering indicates the outward-facing side. If your renderings look incorrect, just flip the cross-product to match the convention.

- Derive a subclass `Transform` from `Object3D`. Similar to a `Group`, a `Transform` will store a pointer to an `Object3D` (but only one, not an array). The constructor of a `Transform` takes a 4x4 matrix as input and a pointer to the `Object3D` modified by the transformation:

```
Transform(Matrix &m, Object3D *o);
```

The `intersect` routine will first transform the ray, then delegate to the `intersect` routine of the contained object. Make sure to correctly transform the resulting normal according to the rule seen in lectures. You may choose to normalize the direction of the transformed ray or leave it un-normalized. If you decide not to normalize the direction, you might need to update some of your intersection code. Make sure to transform the ray direction without the translation part (this is automatically the case if you represent direction vectors with **0** in their fourth coordinate).

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. lights and diffuse materials.`

Other

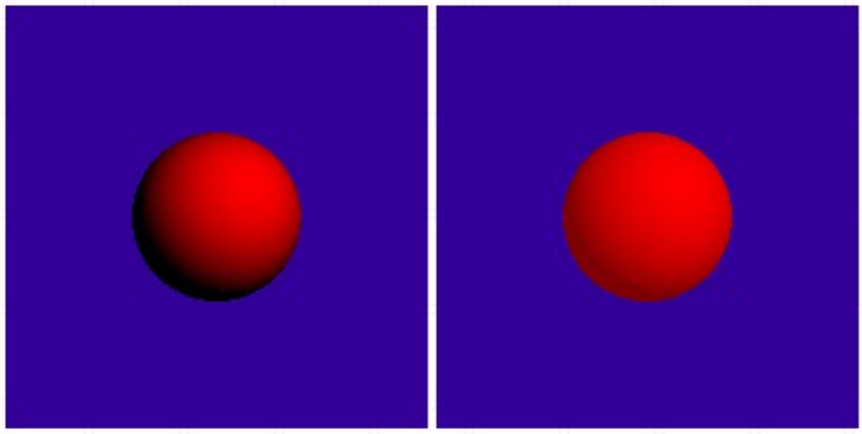
You will need to update the `Makefile` to include the other classes you are writing.

Hints

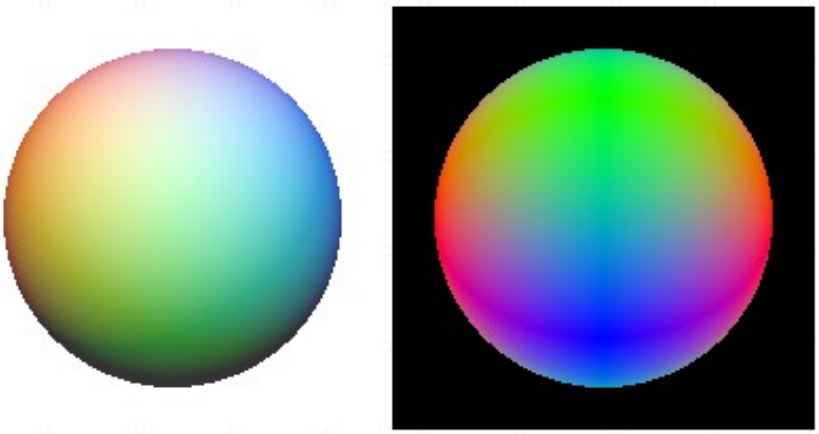
- Parse the arguments of the program in a separate function. It will make your code easier to read.
- Implement the normal visualization and diffuse shading before the transformations.
- Use the various rendering modes (normal, diffuse, distance) to debug your code.

Sample Results

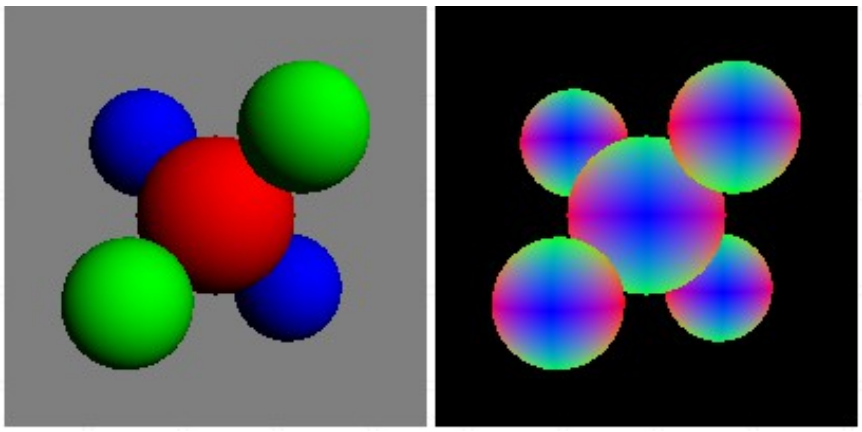
```
raycast -input scene2_01_diffuse.txt -size 200 200 -output output2_01.tga
raycast -input scene2_02_ambient.txt -size 200 200 -output output2_02.tga
```



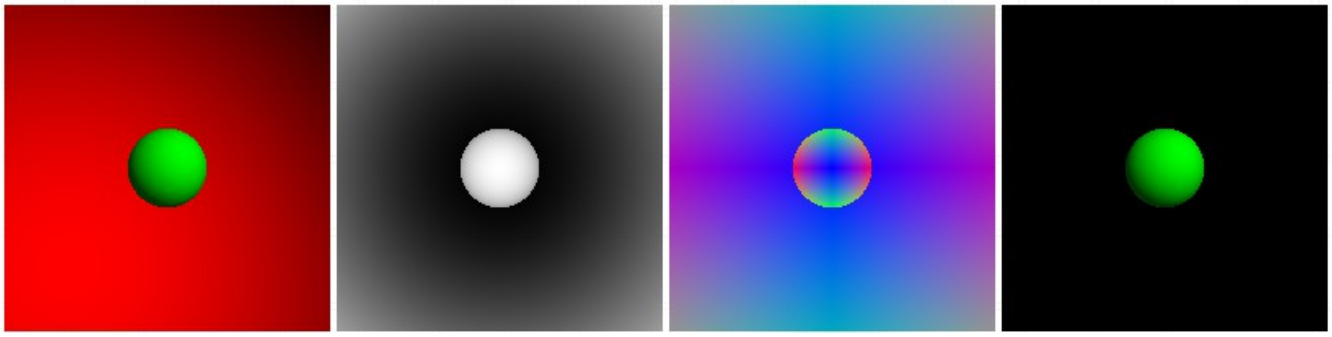
```
raycast -input scene2_03_colored_lights.txt -size 200 200 -output output2_03.tga
   -normals normals2_03.tga
```



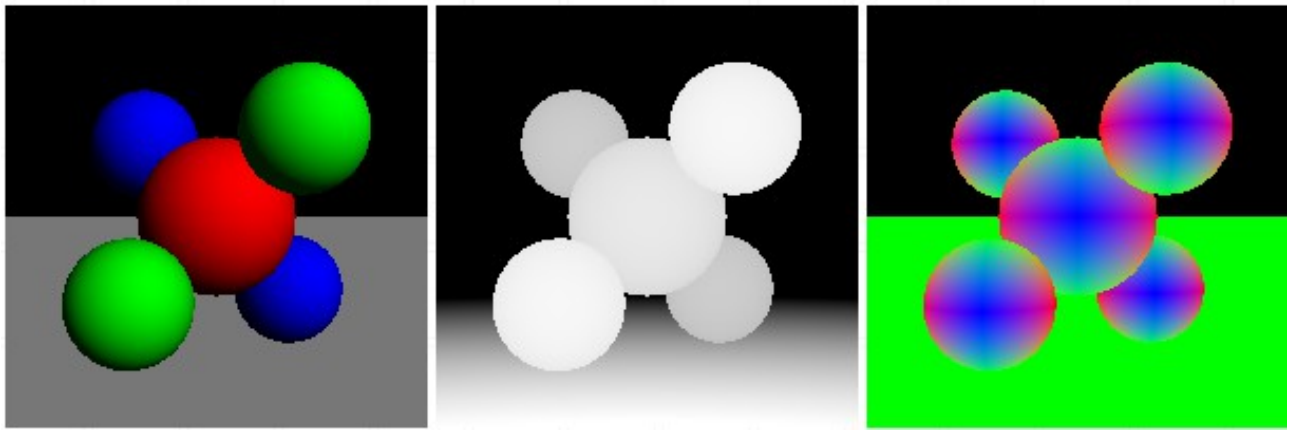
```
raycast -input scene2_04_perspective.txt -size 200 200 -output  output2_04.tga
   -normals  normals2_04.tga
```



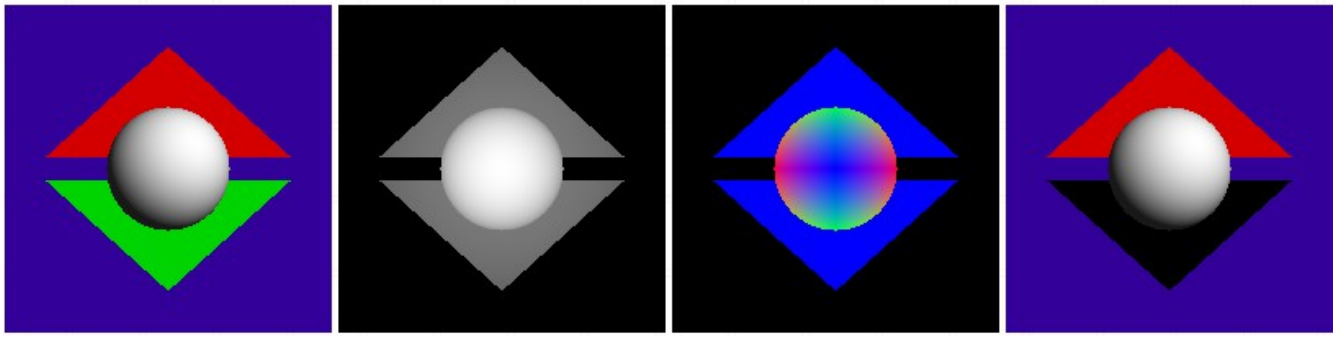
```
raycast -input scene2_05_inside_sphere.txt -size 200 200 -output output2_05.tga
   -depth 9 11 depth2_05.tga -normals normals2_05.tga -shade_back
raycast -input scene2_05_inside_sphere.txt -size 200 200 -output
   output2_05_no_back.tga
```



```
raycast -input scene2_06_plane.txt -size 200 200 -output output2_06.tga -depth 8
   20 depth2_06.tga -normals normals2_06.tga
```

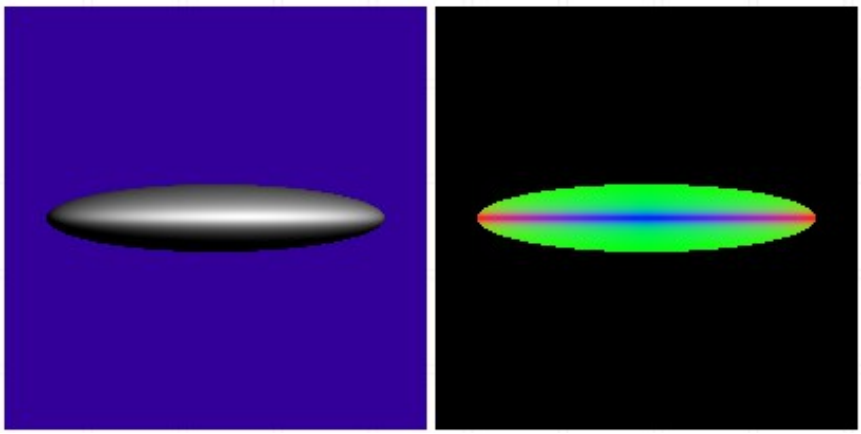


```
raycast -input scene2_07_sphere_triangles.txt -size 200 200 -output
   output2_07.tga -depth 9 11 depth2_07.tga -normals normals2_07.tga -shade_back
raycast -input scene2_07_sphere_triangles.txt -size 200 200 -output
output2_07_no_back.tga
```

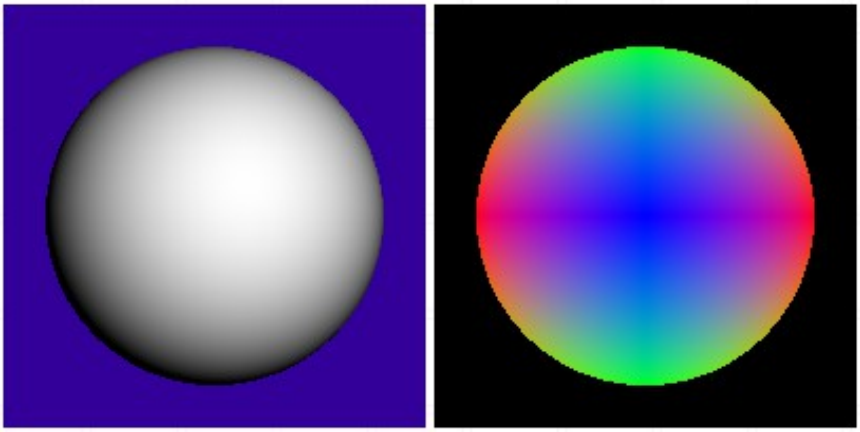


```
raycast -input scene2_08_cube.txt -size 200 200 -output output2_08.tga
raycast -input scene2_09_bunny_200.txt -size 200 200 -output output2_09.tga
raycast -input scene2_10_bunny_1k.txt -size 200 200 -output output2_10.tga
```

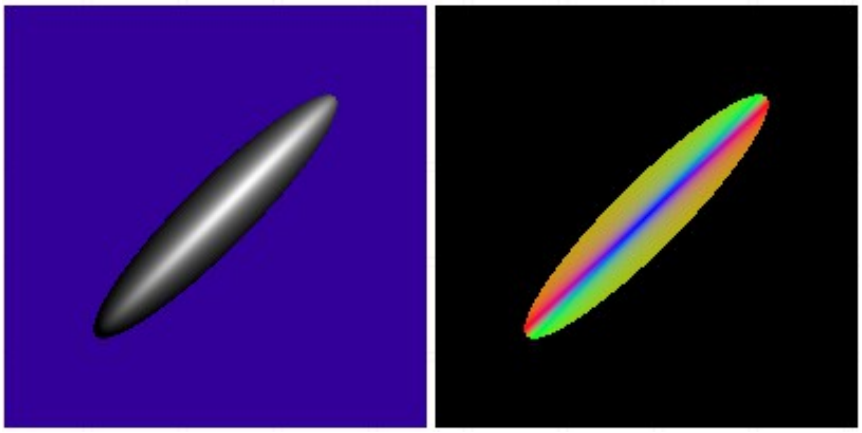
```
raycast -input scene2_11_squashed_sphere.txt -size 200 200 -output output2_11.tga
   -normals normals2_11.tga
```



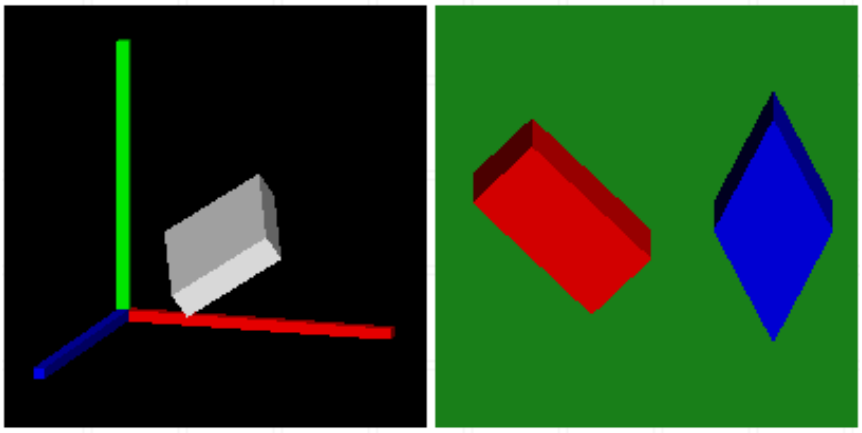
```
raycast -input scene2_12_rotated_sphere.txt -size 200 200 -output output2_12.tga
   -normals normals2_12.tga
```



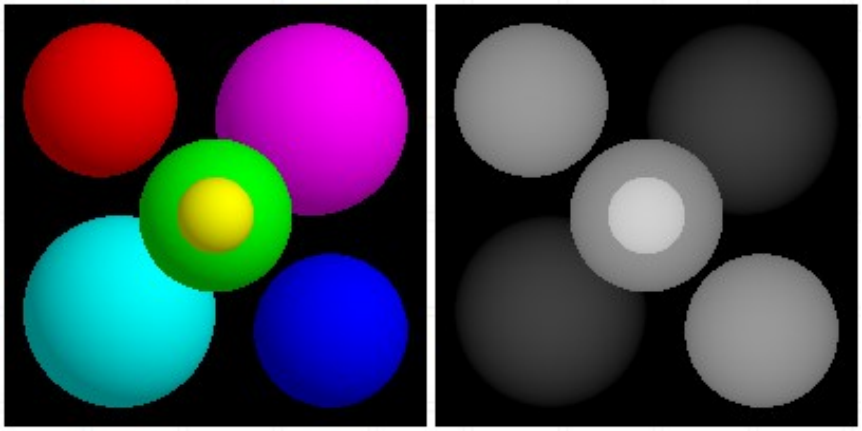
```
raycast -input scene2_13_rotated_squashed_sphere.txt -size 200 200 -output
   output2_13.tga -normals normals2_13.tga
```



```
raycast -input scene2_14_axes_cube.txt -size 200 200 -output output2_14.tga
raycast -input scene2_15_crazy_transforms.txt -size 200 200 -output
   output2_15.tga
```



```
raycast -input scene2_16_t_scale.txt -size 200 200 -output output2_16.tga -depth
    2 7 depth2_16.tga
```



Acknowledgment

This homework is adapted from class 6.837 at MIT.