6.837 Intro to Computer Graphics
Assignment 2: Transformations & Additional Primitives

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).

Diffuse shading is our first step toward modeling the interaction of light and materials. Given the direction to the light **L** and the normal **N** we can compute the diffuse shading as a clamped dot product:

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

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 $c_{pixel} = (rL_rd, gL_gd, bL_bd)$. 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:

$$c_{pixel} = c_{ambient} * c_{object} + \mathrm{SUM}_i\ [\ \mathrm{clamped}(\mathbf{L_i}\cdot\mathbf{N}) * c_{light_i} * c_{object}\ ]$$

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 assignment 1.

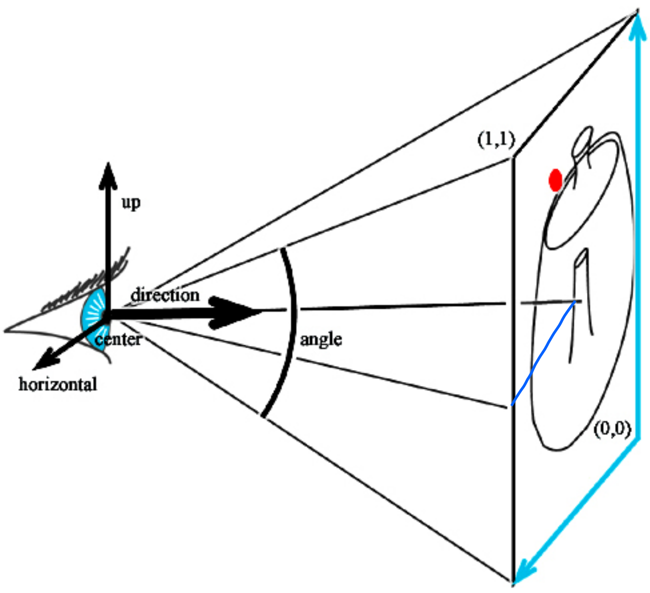
Tasks

- 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`.
    - hit.h
- 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`.
    - light.h
- In test scenes 5 & 7 below, we've asked you to render the "wrong" or "back" side of both a Sphere and a Triangle primitive. Add the `-shade_back` option to your raytracer. When this option is specified, we'd like you to treat both sides of your object surfaces in the same manner. This means you'll need to flip the normal when the eye is on the "wrong" side of the surface (when the dot product of the ray direction & the normal is positive). Do this normal flip just before you shade a pixel, not within the object intersection code. If the `-shade_back` flag is not specified, you should shade back-facing surfaces differently, to aid in debugging. Back-facing surfaces must be detected to implement refraction through translucent objects, and are often not rendered at all for efficiency in real-time applications. We'll see this again in upcoming lectures and assignments.
- 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 center, direction, 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);
  ```



$a^2=\tan^2/(1-\tan^2) *d^2$

Hint: In class, we often talk about a "virtual screen" in space. You can calculate the location and extents of this "virtual screen" using some simple trigonometry. You can then interpolate over points on the virtual screen in the same way you interpolated over points on the screen for the orthographic camera. Direction vectors can then be calculated by subtracting the camera center point from the screen point. Don't forget to normalize! In contrast, if you interpolate over the camera angle to obtain your direction vectors, your scene will look distorted - especially for large camera angles, which will give the appearance of a fisheye lens.

Note: the distance to the image plane and the size of the image plane are unnecessary. Why?

- 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** · **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);
  ```

  Use the method of your choice to implement the ray-triangle intersection: general polygon with in-polygon test, barycentric coordinates, etc. 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 lecture. 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.

- Extra credit: Implement two different ray-triangle intersection methods and compare; add cones or cylinders; implement Constructive Solid Geometry (CSG), IFS, or non-linear cameras. For CSG, you need to implement a new `intersectAll` method for your `Object3D` classes. This function returns *all* of the intersections of the ray with the object, not just the closest one. For additional primitives such as cones and cylinders, implement the simple case of axis-aligned primitives and use transformations.

Updated Files

- scene_parser.h
- scene_parser.C

If you're interested, here's the scene description file grammar used in this assignment.

You will need to edit the Makefile to include any .C files that you add to the project.

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.

Input Files

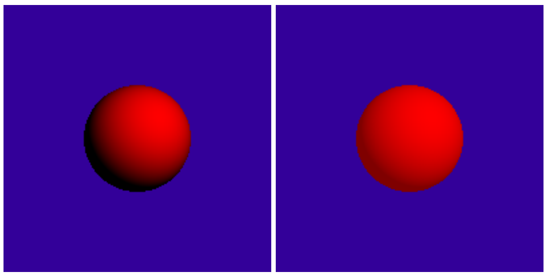
- scene2_01_diffuse.txt
- scene2_02_ambient.txt
- scene2_03_colored_lights.txt
- scene2_04_perspective.txt
- scene2_05_inside_sphere.txt
- scene2_06_plane.txt
- scene2_07_sphere_triangles.txt
- scene2_08_cube.txt
- scene2_09_bunny_200.txt
- scene2_10_bunny_1k.txt
- scene2_11_squashed_sphere.txt
- scene2_12_rotated_sphere.txt
- scene2_13_rotated_squashed_sphere.txt
- scene2_14_axes_cube.txt
- scene2_15_crazy_transforms.txt
- scene2_16_t_scale.txt (new! to test t scale)

Triangle Meshes (.obj format)

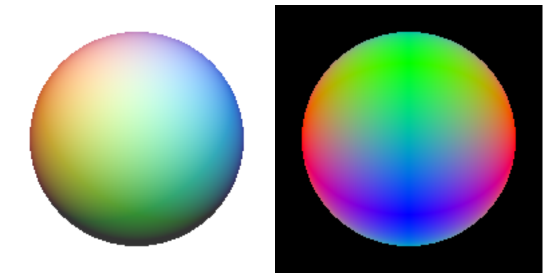
- cube.obj
- bunny_200.obj
- bunny_1k.obj

Sample Results

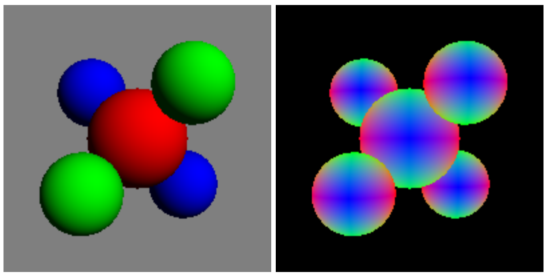
```
raytracer -input scene2_01_diffuse.txt -size 200 200 -output output2_01.tga
raytracer -input scene2_02_ambient.txt -size 200 200 -output output2_02.tga
```



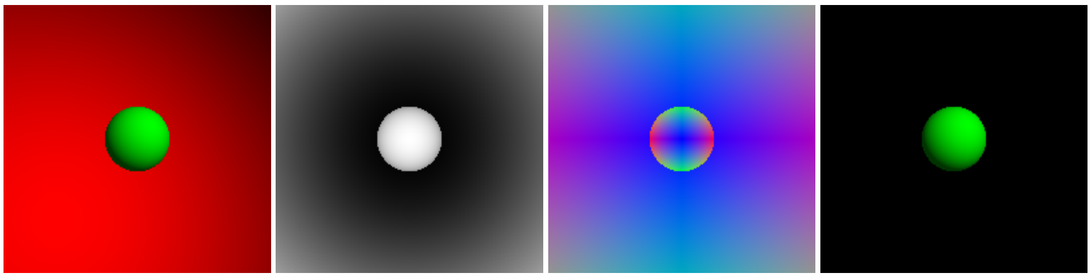
```
raytracer -input scene2_03_colored_lights.txt -size 200 200 -output output2_03.tga -normals normals2_03.tga
```



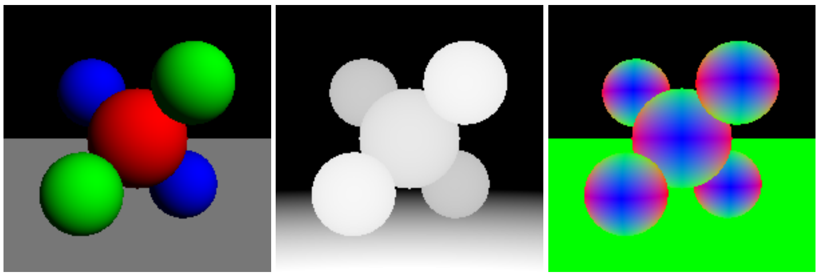
```
raytracer -input scene2_04_perspective.txt -size 200 200 -output output2_04.tga -normals normals2_04.tga
```



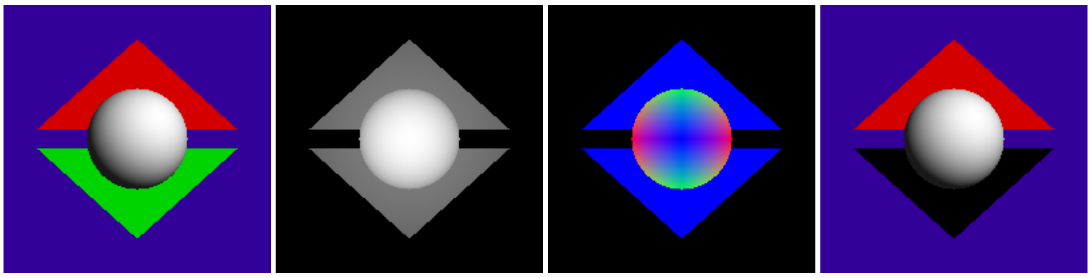
```
raytracer -input scene2_05_inside_sphere.txt -size 200 200 -output output2_05.tga -depth 9 11 depth2_05.tga -normals normals2_05.tga -shac
raytracer -input scene2_05_inside_sphere.txt -size 200 200 -output output2_05_no_back.tga
```



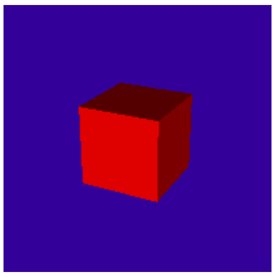
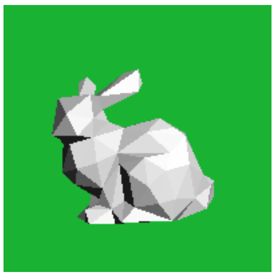
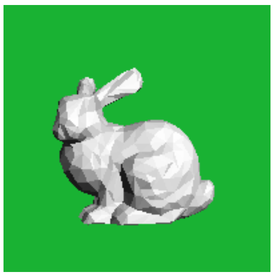
```
raytracer -input scene2_06_plane.txt -size 200 200 -output output2_06.tga -depth 8 20 depth2_06.tga -normals normals2_06.tga
```



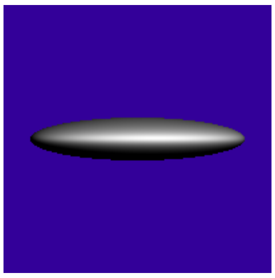
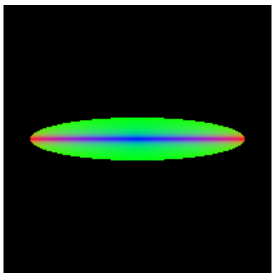
```
raytracer -input scene2_07_sphere_triangles.txt -size 200 200 -output output2_07.tga -depth 9 11 depth2_07.tga -normals normals2_07.tga -s
raytracer -input scene2_07_sphere_triangles.txt -size 200 200 -output output2_07_no_back.tga
```



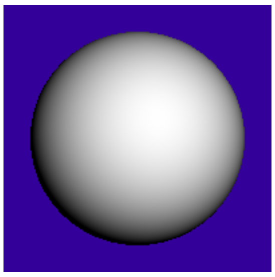
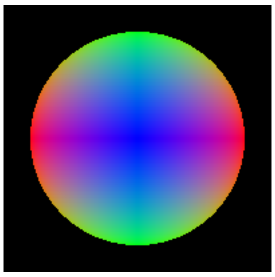
```
raytracer -input scene2_08_cube.txt -size 200 200 -output output2_08.tga
raytracer -input scene2_09_bunny_200.txt -size 200 200 -output output2_09.tga
raytracer -input scene2_10_bunny_1k.txt -size 200 200 -output output2_10.tga
```

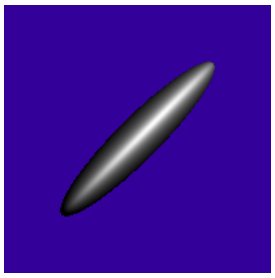
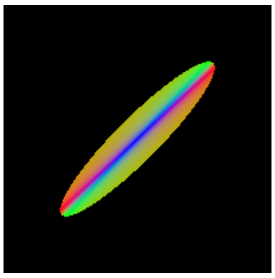
```
raytracer -input scene2_11_squashed_sphere.txt -size 200 200 -output output2_11.tga -normals normals2_11.tga
```

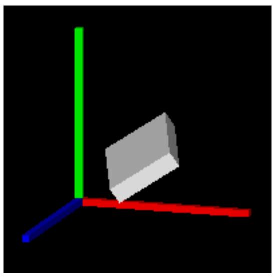
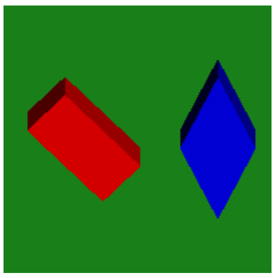
```
raytracer -input scene2_12_rotated_sphere.txt -size 200 200 -output output2_12.tga -normals normals2_12.tga
```

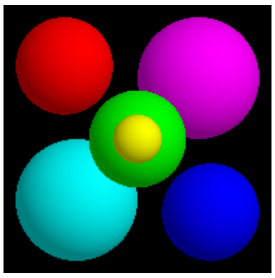
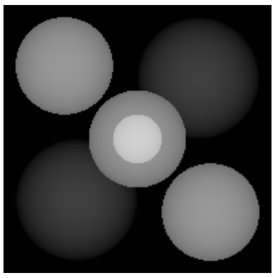
```
raytracer -input scene2_13_rotated_squashed_sphere.txt -size 200 200 -output output2_13.tga -normals normals2_13.tga
```

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

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

See the main Assignments Page for submission information.