6.837: Computer Graphics Fall 2010

Programming Assignment 4: Ray Casting

Due November 17th at 8:00pm.

In this assignment, you will implement a basic ray caster. This will be the basis of your final assignment, so proper code design is quite important. As seen in class, a ray caster sends a ray for each pixel and intersects it with all the objects in the scene. Your ray caster will support orthographic and perspective cameras as well as several primitives (spheres, planes, and triangles). You will also have to support several shading modes and visualizations (constant and diffuse shading, depth and normal visualization).

The remainder of this document is organized as follows:

1 Getting Started

Note that this assignment is a lot of work. Please start as early as possible. One significant way in which this assignment differs from previous assignments is that you start off with a lot less starter code.

Run the sample solution `a4soln` as follows:

```
./a4soln -input scene1_01.txt -size 200 200 -output output1_01.tga -depth 9 10 depth1_01.tga
```

This will generate an image named `output1_01.tga`. You may view this file using `xv output1_01.tga` (you may have to `add graphics` first). We'll describe the rest of the command-line parameters later. When your program is complete, you will be able to render this scene as well as well as the other test cases given below.

2 Summary of Requirements

This section summarizes the core requirements of this assignment. There are a lot of them and you should start early. Let's walk through them.

You will use object-oriented design to make your ray caster flexible and extendable. A generic `Object3D` class will serve as the parent class for all 3D primitives. You will derive subclasses (such as `Sphere`, `Plane`, `Triangle`, `Group`, and `Transform`) to implement specialized primitives. Similarly, this assignment requires the implementation of a general `Camera` class with orthographic camera and perspective camera subclasses.

You will use two shading models: objects with a constant color and diffuse objects. Diffuse shading is our first step toward modeling the interaction of light and materials. Given the direction to the light $\mathbf{L}$ and the normal $\mathbf{N}$ we can compute the diffuse shading as a clamped dot product:

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

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 $c_{pixel} = (rL_rd, gL_gd, bL_bd)$. 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 for debugging. Without it, parts facing away from the light source appear completely black. Putting this all together, the formula is:

$$c_{pixel} = c_{ambient} * c_{object} + \sum_i \left[clamp(\mathbf{L}_i \cdot \mathbf{N}) * c_{light} * c_{object} \right]$$

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 constant shading.

You will also implement two visualization modes. One mode will display the distance t of each pixel to the camera. The other mode is a visualization of the surface normal. 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).

Your code will be tested using a script on all the test cases below. Make sure that your program handles the exact same arguments as the examples below.

3 Starter Code

Compile the code with `make`. You can type `make clean; make` to rebuild everything from scratch. Note that, initially, the starter code won't even compile. Fear not, however, you still have some goodies to help you get started.

The `Image` class is used to initialize and edit the RGB values of images. Be careful—do not try to read or write to pixels outside the bounds of the image. The class also includes functions for loading and saving simple `.tga` image files. `.tga` files can be viewed with `xv`, Adobe Photoshop, and other imaging software.

For linear algebra, you should use the `vecmath` library that you are familiar with from previous assignments.

We provide you with a `Ray` class and a `Hit` class to manipulate camera rays and their intersection points, and a skeleton `Material` class. A `Ray` is represented by its origin and direction vectors. The `Hit` class stores information about the closest intersection point and normal, the value of the ray parameter t and a pointer to the `Material` of the object at the intersection. The `Hit` data structure must be initialized with a very large t value (try `FLT_MAX`). It is modified by the intersection computation to store the new closest t and the `Material` of intersected object.

Your program should take a number of command line arguments to specify the input file, output image size and output file. Make sure the examples below work, as this is how we will test your program. A simple scene file parser for this assignment is provided. Several constructors and the `Group::addObject` method you will write are called from the parser (and will be a source for many compilation errors initially). Look in the `scene_parser.cpp` file for details.

If you're interested, the scene description file grammar used in this assignment is included in the source file distribution. Finally, you will have to add any additional files to the provided `Makefile` as needed.

4 Implementation Notes

This is a very large assignment. We can't repeat this enough. Here's a suggested recipe to follow to get as far as possible, as quickly as possible.

- Write a virtual `Object3D` class (a virtual class in C++ is like an abstract class in Java). It only provides the specification for 3D primitives, and in particular the ability to be intersected with a ray via the virtual method: `virtual bool intersect( const Ray& r, Hit& h, float tmin ) = 0;`

  Since this method is pure virtual for the `Object3D` class, the prototype in the header file includes '`= 0`'. This '`= 0`' tells the compiler that `Object3D` won't implement the method, but that subclasses derived from `Object3D` must implement this routine. An `Object3D` stores a pointer to its `Material` type. For this assignment, materials are very simple and consist of a single color. Your `Object3D` class must have:

  - a default constructor and destructor
  - a pointer to a `Material` instance
  - a pure virtual intersection method

- Derive `Sphere`, a subclass of `Object3D`, that additionally stores a center point and a radius. The `Sphere` constructor will be given a center, a radius, and a pointer to a `Material` instance. The `Sphere` class implements the virtual `intersect` method mentioned above (but without the '`= 0`'): `virtual bool intersect(const Ray& r, Hit& h, float tmin);`

  With the `intersect` routine, we are looking for the closest intersection along a `Ray`, parameterized by t. `tmin` is used to restrict the range of intersection. If an intersection is found such that $t >$ `tmin` and t is less than the value of the intersection currently stored in the `Hit` data structure, `Hit` is updated as necessary. Note that if the new intersection is closer than the previous one, both t and `Material` must be modified. It is important that your intersection routine verifies that $t >=$ `tmin`. `tmin` depends on the type of camera (see below) and is not modified by the intersection routine.

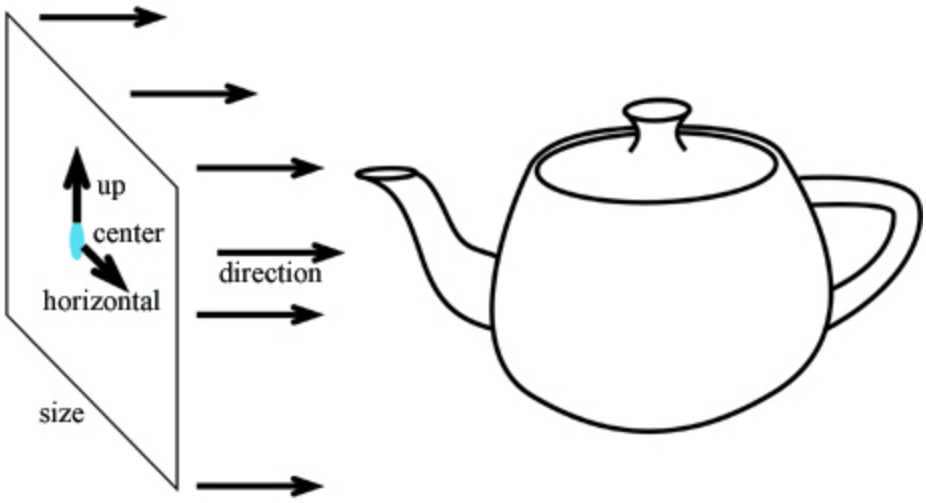
- Derive `Group`, also a subclass of `Object3D`, that stores an array of pointers to `Object3D` instances. For example, it will be used to store the entire 3D scene. You'll need to write the `intersect` method of `Group` which loops through all these instances, calling their intersection methods. The `Group` constructor should take as input the number of objects under the group. The group should include a method to add objects: `void addObject(int index, Object3D* obj);`

- Write a pure virtual `Camera` class (in Java parlance, an interface) and subclass `OrthographicCamera`. The `Camera` class has two pure virtual methods:

  ```
  virtual Ray generateRay( const Vector2f& point ) = 0;
  virtual float getTMin() const = 0;
  ```

  The first is used to generate rays for each screen-space coordinate, described as a `Vector2f`. The direction of the rays generated by an orthographic camera is always the same, but the origin varies. The `getTMin()` method will be useful when tracing rays through the scene. For an orthographic camera, rays always start at infinity, so `tmin` will be a large negative value (try `FLT_MIN`). However, you will also implement a perspective camera and the value of `tmin` will be zero to correctly clip objects behind the viewpoint.

- An orthographic camera is described by an orthonormal basis (one point and three vectors) and an image size (one float). The constructor takes as input the center of the image, the direction vector, an up vector, and the image size. The input direction might not be a unit vector and must be



normalized. The input up vector might not be a unit vector or perpendicular to the direction. It must be modified to be orthonormal to the direction. The third basis vector, the horizontal vector of the image plane, is deduced from the direction and the up vector (hint: remember your linear algebra and cross products). The origin of the rays generated by the camera for the screen coordinates, which vary from $(-1,-1) \rightarrow (1,1)$, should vary from: $\mathbf{center} - (size * \mathbf{up})/2 - (size * \mathbf{horizontal})/2 \rightarrow \mathbf{center} + (size * \mathbf{up})/2 + (size * \mathbf{horizontal})/2$

The camera does not know about screen resolution. Image resolution should be handled in your main loop. For non-square image ratios, just crop the screen coordinates accordingly.

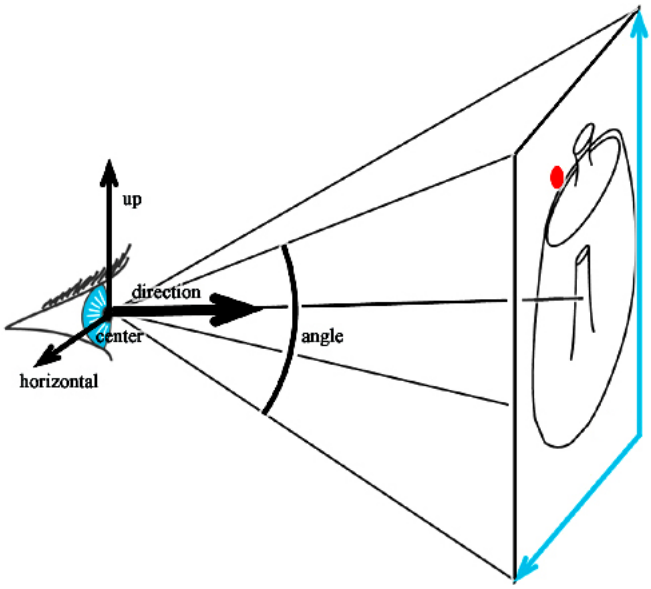
- Use the input file parsing code provided to load the camera, background color and objects of the scene.
- Write a `main` function that reads the scene (using the parsing code provided), loops over the pixels in the image plane, generates a ray using your camera class, intersects it with the high-level `Group` that stores the objects of the scene, and writes the color of the closest intersected object.
- Implement a second rendering style to visualize the depth t of objects in the scene. Two input depth values specify the range of depth values which should be mapped to shades of gray in the visualization. Depth values outside this range should be clamped.
- Update your sphere intersection routine to pass the correct 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` method:

  ```
  void getIllumination( const Vector3f& p, Vector3f& dir, Vector3f& 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 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( const Vector3f& center, const Vector3f& direction, const Vector3f& up, float angle );`

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( const Vector3f& normal, float d, Material* m );`

  d is the offset from the origin, meaning that the plane equation is $\mathbf{P} \cdot \mathbf{n} = d$. 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( const Vector3f& a, const Vector3f& b, const Vector3f& 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.

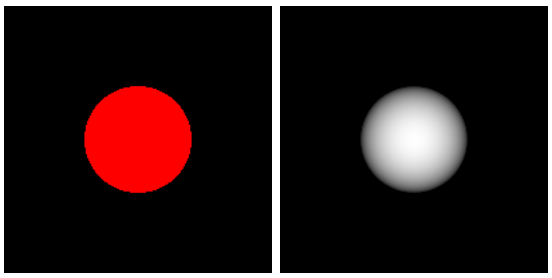
- Derive a subclass `Transform` from `Object3D`. Similar to a `Group`, a `Transform` will store a pointer to an `Object3D` (but only one, not an array). The constructor of a `Transform` takes a 4×4 matrix as input and a pointer to the `Object3D` modified by the transformation: `Transform( const Matrix4f& 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.

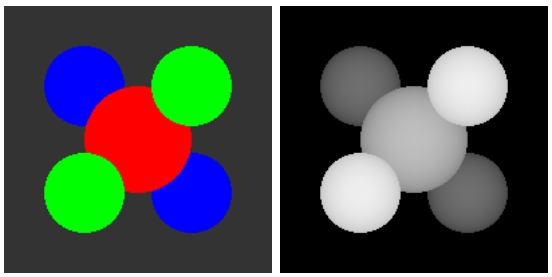
5 Test Cases

Your assignment will be graded by running a script that runs these examples below. Make sure your ray caster produces the same output if you want to receive full credit. You can use the UNIX command `cmp` to compare two files (for example, your output and the sample solution).

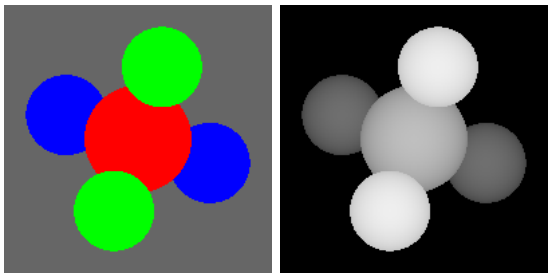
```
./a4 -input scene1_01.txt -size 200 200 -output output1_01.tga -depth 9 10 depth1_01.tga
```



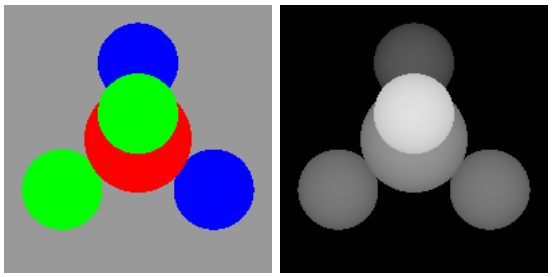
```
./a4 -input scene1_02.txt -size 200 200 -output output1_02.tga -depth 8 12 depth1_02.tga
```



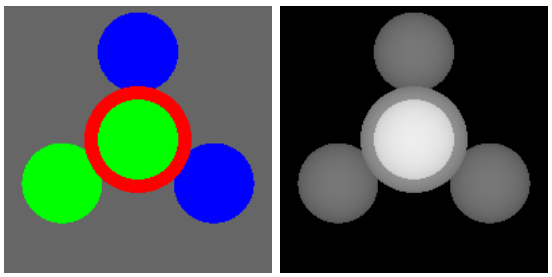
```
./a4 -input scene1_03.txt -size 200 200 -output output1_03.tga -depth 8 12 depth1_03.tga
```



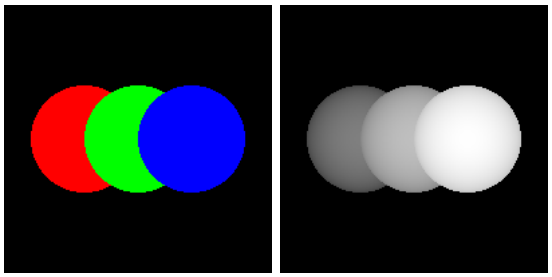
```
./a4 -input scene1_04.txt -size 200 200 -output output1_04.tga -depth 12 17 depth1_04.tga
```



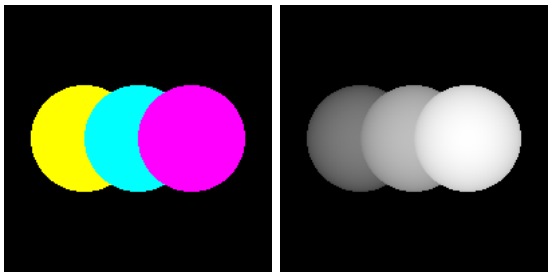
```
./a4 -input scene1_05.txt -size 200 200 -output output1_05.tga \
-depth 14.5 19.5 depth1_05.tga
```



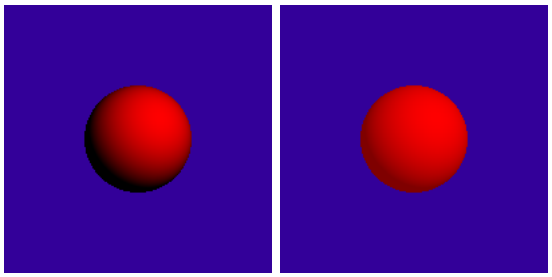
```
./a4 -input scene1_06.txt -size 200 200 -output output1_06.tga -depth 3 7 depth1_06.tga
```



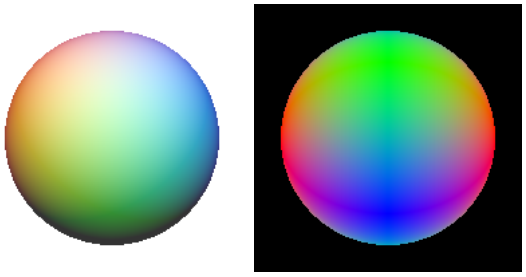
```
./a4 -input scene1_07.txt -size 200 200 -output output1_07.tga \
-depth -2 2 depth1_07.tga
```



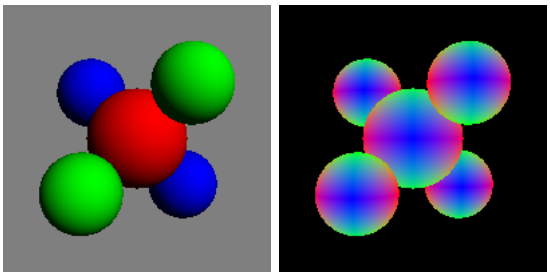
```
./a4 -input scene2_01_diffuse.txt -size 200 200 -output output2_01.tga
./a4 -input scene2_02_ambient.txt -size 200 200 -output output2_02.tga
```



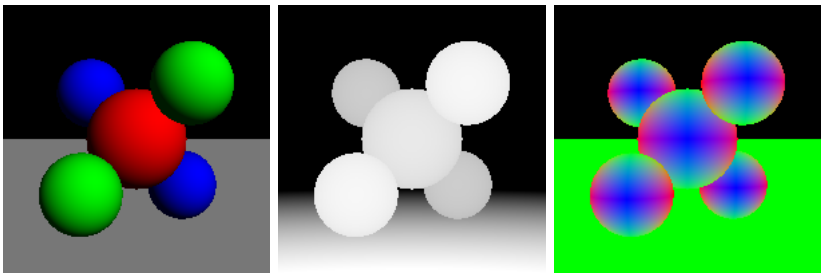
```
./a4 -input scene2_03_colored_lights.txt -size 200 200 -output output2_03.tga \
-normals normals2_03.tga
```



```
./a4 -input scene2_04_perspective.txt -size 200 200 -output output2_04.tga \
-normals normals2_04.tga
```

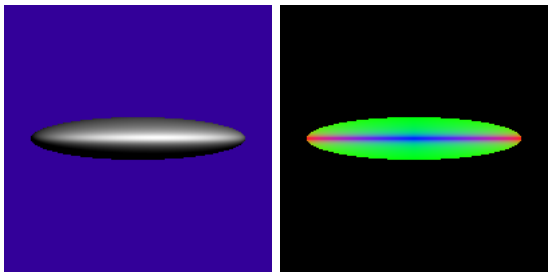


```
./a4 -input scene2_06_plane.txt -size 200 200 -output output2_06.tga \
-depth 8 20 depth2_06.tga -normals normals2_06.tga
```

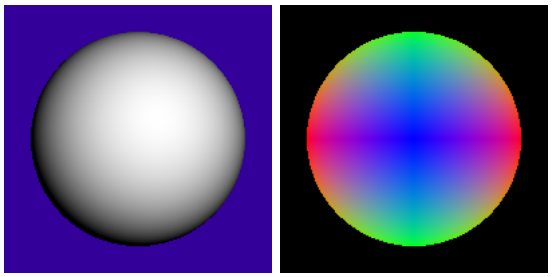


```
./a4 -input scene2_08_cube.txt -size 200 200 -output output2_08.tga
./a4 -input scene2_09_bunny_200.txt -size 200 200 -output output2_09.tga
./a4 -input scene2_10_bunny_1k.txt -size 200 200 -output output2_10.tga
```

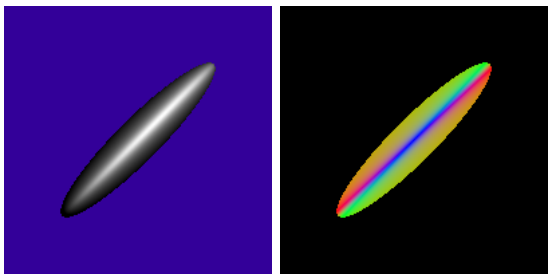
```
./a4 -input scene2_11_squashed_sphere.txt -size 200 200 -output output2_11.tga \
-normals normals2_11.tga
```



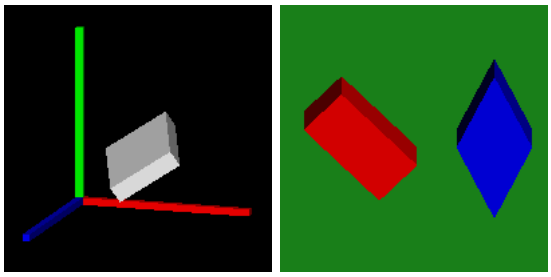
```
./a4 -input scene2_12_rotated_sphere.txt -size 200 200 -output output2_12.tga \
-normals normals2_12.tga
```



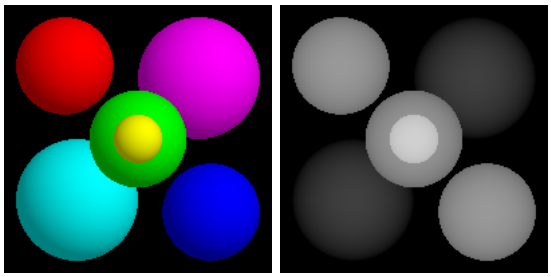
```
./a4 -input scene2_13_rotated_squashed_sphere.txt -size 200 200 -output output2_13.tga \
-normals normals2_13.tga
```



```
./a4 -input scene2_14_axes_cube.txt -size 200 200 -output output2_14.tga
./a4 -input scene2_15_crazy_transforms.txt -size 200 200 -output output2_15.tga
```



```
./a4 -input scene2_16_t_scale.txt -size 200 200 -output output2_16.tga \
-depth 2 7 depth2_16.tga
```



6 Hints

- Use a small image size for faster debugging. 64×64 pixels is usually enough to realize that something might be wrong.
- As usual, don't hesitate to print as much information as needed for debugging, such as the direction vector of the rays, the hit values, etc.
- Use `assert()` to check function preconditions, array indices, etc. See `cassert`.
- The "very large" negative and positive values for t used in the `Hit` class and the intersect routine can simply be initialized with large values relative to the camera position and scene dimensions. However, to be more correct, you can use the positive and negative values for infinity from the IEEE floating point standard.
- 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

7 Extra Credit

Note that there isn't much extra credit for this assignment. That's because we want you to focus on a good design so that your code will survive not only this assignment but the next one as well. The following extra credit ideas will only merit a few points.

7.1 Easy

- Add simple fog to your ray tracer by attenuating rays according to their length. Allow the color of the fog to be specified by the user in the scene file.
- Add other types of simple primitives to your ray tracer, and extend the file format and parser accordingly. For instance, how about a cylinder or cone? These can make your scenes much more interesting.
- Add a new oblique camera type (or some other weird camera). In a standard camera, the projection window is centered on the z-axis of the camera. By sliding this projection window around, you can get some cool effects.

7.2 Medium

- Implement a torus or higher order implicit surfaces by solving for t with a numerical root finder.

8 Submission Instructions

You are to write a `README.txt` (or optionally a PDF) that answers the following questions:

- How do you compile your code? Provide instructions for Athena Linux. You will not need to provide instructions on how to run your code, because it must run with the exact command line given earlier in this document.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.
- Got any comments about this assignment that you'd like to share?

Submit your assignment on Stellar by **November 17th by 8:00pm**. Please submit a single archive (`.zip` or `.tar.gz`) containing:

- Your source code.
- A compiled executable named `a4`.
- Any additional files that are necessary.
- The `README` file.