6.837 Intro to Computer Graphics Assignment 1: Ray Casting

In this assignment, you will implement a basic ray caster. This will be the basis of many future assignments, 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. You will implement a ray caster for an orthographic camera (parallel rays) for sphere primitives. You will use a very basic shading model: the objects have a constant color. You will also implement a visualization mode to display the distance *t* of each pixel to the camera.

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`, to implement specialized primitives. In later assignments, you will extend the set of primitives with planes and polygons. Similarly, this assignment requires the implementation of a general `Camera` class and an `OrthographicCamera` subclass. In the next assignment, you will also derive a general perspective camera.

We provide you with a `Ray` class and a `Hit` class to manipulate camera rays and their intersection points, and a skeleton `Material` class.

Tasks

- Write a pure virtual `Object3D` class. 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;`'. 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, and
  - 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 the center, radius, and 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 the objects:

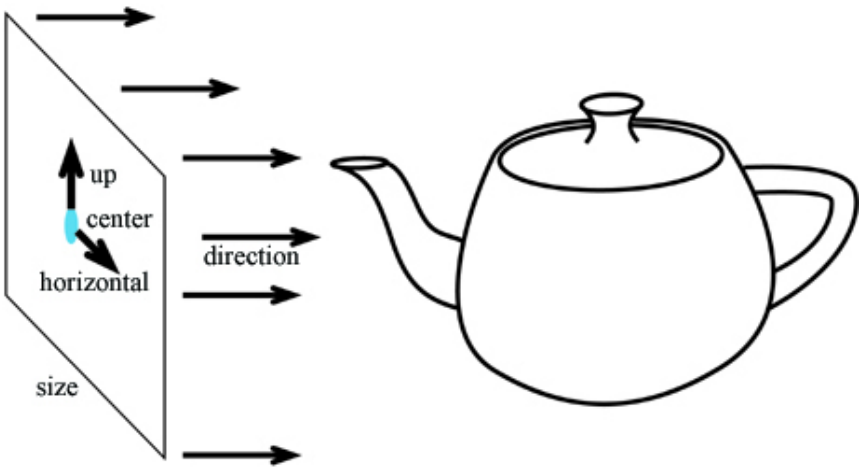
  ```
  void addObject(int index, Object3D *obj);
  ```

- Write a pure virtual `Camera` class and subclass `OrthographicCamera`. The `Camera` class has two pure virtual methods:

  ```
  virtual Ray generateRay(Vec2f point) = 0;
  virtual float getTMin() const = 0;
  ```

  The first is used to generate rays for each screen-space coordinate, described as a `Vec2f`. 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. However, in the next assignment you will 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 floating point). 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 vector algebra and cross products). The origin of the rays generated by the camera for the screen coordinates, which vary from `(0,0) -> (1,1)`, should vary from:

```
center - (size*up)/2 - (size*horizontal)/2  ->  center + (size*up)/2 + (size*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 `OrthographicCamera` 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 are simply clamped.
- Extra credit: Write both the geometric and algebraic sphere intersection methods, add cylinders and cones, fog based on distance to the image plane, etc.

Ray, Hit & Material Classes

We provide the `Ray`, `Hit` & `Material` classes. A `Ray` is represented by its origin and direction vectors. The `Hit` class stores information about the closest intersection point, 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. It is modified by the intersection computation to store the new closest *t* and the `Material` of intersected object. For this assignment a `Material` stores just the diffuse color of the object. You will extend this class in future assignments.

- ray.h
- hit.h
- material.h

Parsing command line arguments & input files

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. 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.

- parse_code.txt
- scene_parser.h
- scene_parser.C

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

Hints

- Use a small image size for faster debugging. `64 x 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 pre-conditions, array indices, etc. See `assert.h`.
- 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 (for extra credit).

Additional References

- http://www.irtc.org/
- http://www.acm.org/tog/resources/RTNews/html/
- http://www.povray.org/
- http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm
- http://www.siggraph.org/education/materials/HyperGraph/raytrace/rt_java/raytrace.html

Input Files

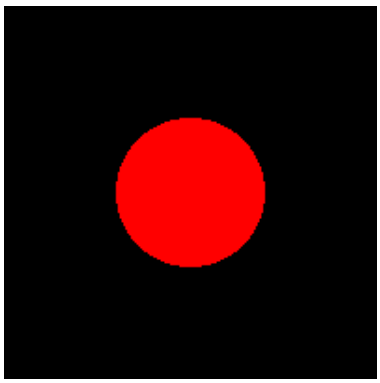
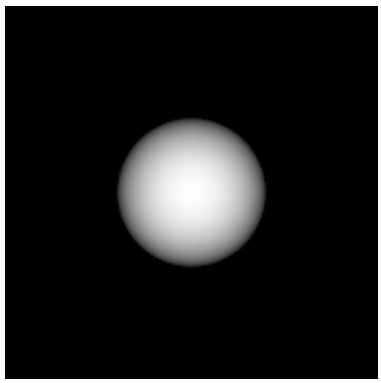
- scene1_01.txt
- scene1_02.txt
- scene1_03.txt
- scene1_04.txt
- scene1_05.txt
- scene1_06.txt
- scene1_07.txt

Makefile for g++ on LINUX

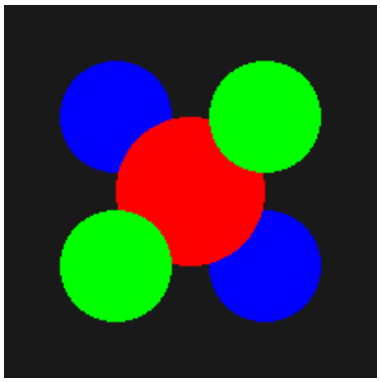
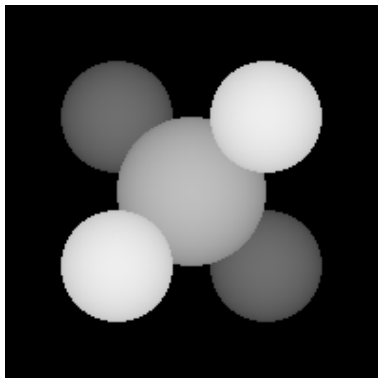
- Makefile

Sample Results

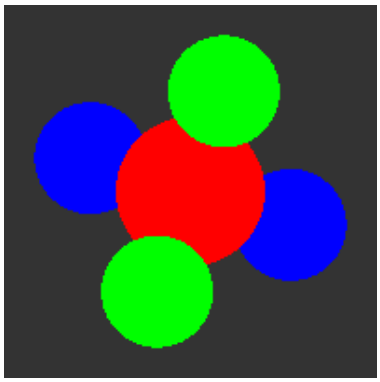
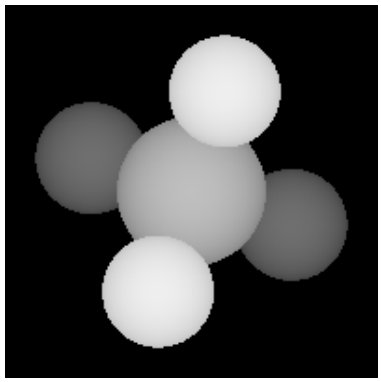
```
raytracer -input scene1_01.txt -size 200 200 -output output1_01.tga -depth 9 10 depth1_01.tga
```

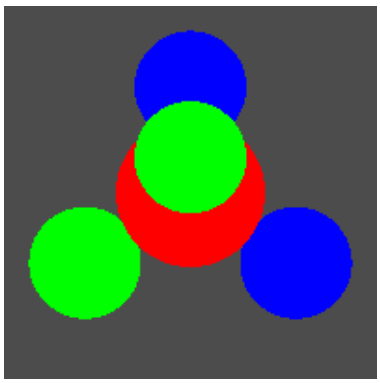
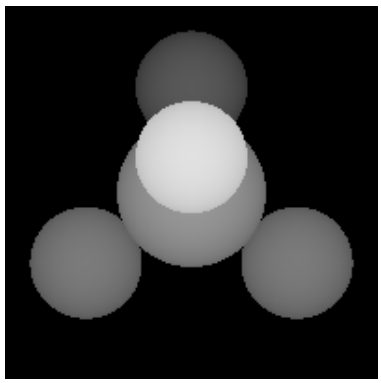
```
raytracer -input scene1_02.txt -size 200 200 -output output1_02.tga -depth 8 12 depth1_02.tga
```

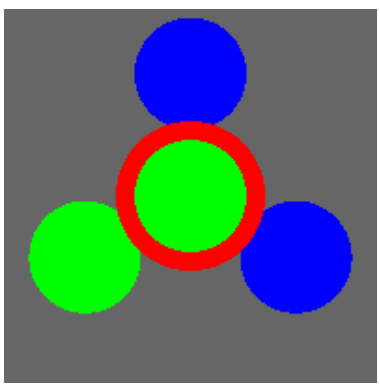
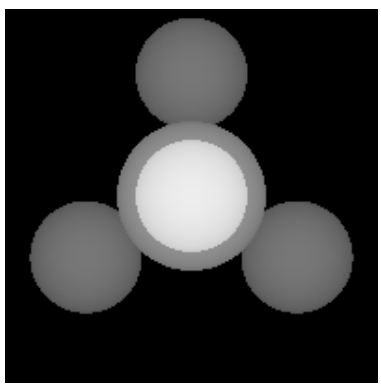
```
raytracer -input scene1_03.txt -size 200 200 -output output1_03.tga -depth 8 12 depth1_03.tga
```

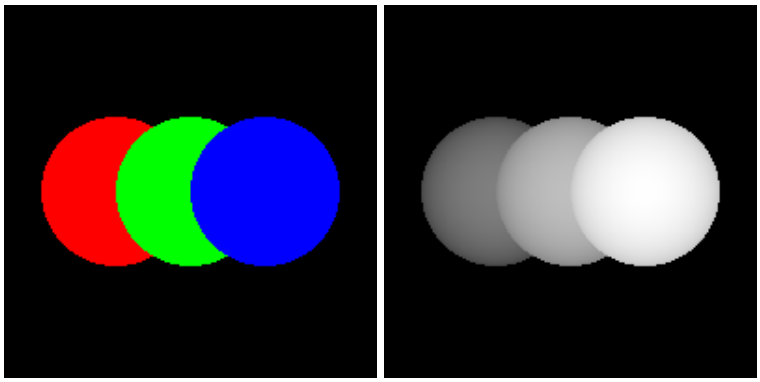
```
raytracer -input scene1_04.txt -size 200 200 -output output1_04.tga -depth 12 17 depth1_04.tga
```

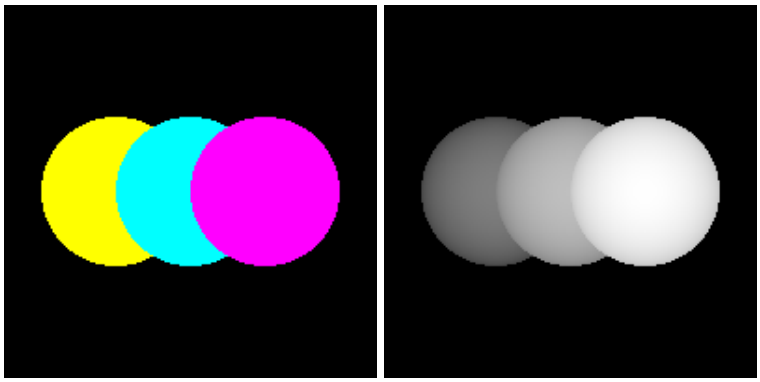
```
raytracer -input scene1_05.txt -size 200 200 -output output1_05.tga -depth 14.5 19.5 depth1_05.tga
```

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



```
raytracer -input scene1_07.txt -size 200 200 -output output1_07.tga -depth -2 2 depth1_07.tga
```



See the main Assignments Page for submission information.
