6.837 Introduction to Computer Graphics
Assignment 1: Ray Casting
Due Wednesday September 17, 2003 at 11:59pm

In this assignment, you will implement a basic ray caster. This will be the basis of all the following 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. As an alternative, you will also 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.

1 Tasks

- Write a pure virtual `Object3D` class (see specifications below).
- Derive `Sphere`, a subclass of `Object3D`, and implement the intersection of a sphere with a ray.
- Derive `Group`, also a subclass of `Object3D`, that stores an array of pointers to `Object3D` instances. Write the intersection routine.
- Write a pure virtual `Camera` class and subclass `OrthographicCamera`. Write the corresponding ray generation method for the subclass.
- 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 of objects in the scene.
- Extra credit: Write both the geometric and algebraic sphere intersection methods, add cylinders and cones, fog based on distance to the image plane, etc.
- Provide a `README.txt` file that discusses any problems you encountered, how long it took to complete the assignment, and any extra credit work that you did.

2 Classes you need to write

2.1 Object3D

This class is pure virtual. 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 (see description below). For this assignment, an `Object3D` will store its color as a `Vec3f`. Later in the semester, we will instead store a pointer to more complex materials. For now, your `Object3D` class must have:

- a default constructor and destructor,
- a color field, and
- a pure virtual intersection method.

2.2 Sphere

`Sphere` is a subclass of `Object3D` that additionally stores a center point and a radius. For this assignment, the `Sphere` constructor will be given the center, radius, and color. 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 $t \geq$ `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 `color` must be modified.

For an orthographic camera, rays always start at infinity, so `tmin` will just be set to a large negative value. However, in the next assignment you will implement a perspective camera and it will be important that your intersection routine verifies that $t \geq$ `tmin`. `tmin` is not modified by the intersection routine.

2.3 Group

A `Group` is a special subclass of `Object3D` that gathers multiple 3D primitives. For example, it will be used to store the entire 3D scene. It stores an array of pointers to `Object3D` instances. The `intersect` method of `Group` 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);
```

2.4 Camera and OrthographicCamera

Write both a pure virtual generic `Camera` class and an `OrthographicCamera` subclass. A camera must be able to generate a ray for each screen-space coordinate, described as a `Vec2f`:

```
Ray generateRay(Vec2f point);
```

The direction of the rays generated by an orthographic camera is always the same, but the origin varies.

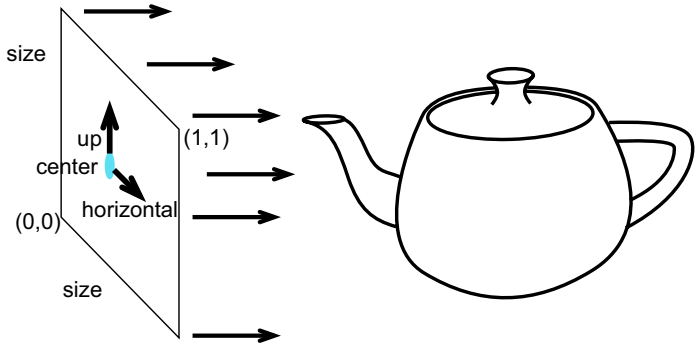

Figure 1: Orthographic camera.

An orthographic camera is described by an orthonormal basis (one point and

three vectors) and an image size (one floating point), as illustrated in Figure 1. The constructor takes as input the center of the image, the projection direction, an up vector, and the image size. The input projection 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 $(0,0) \rightarrow (1,1)$, should vary from:

$$center - \frac{size}{2} \cdot up - \frac{size}{2} \cdot horizontal \rightarrow center + \frac{size}{2} \cdot up + \frac{size}{2} \cdot horizontal$$

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.

3 Utilities Provided

Ray.h

A simple `Ray` class. A ray is represented by its origin and direction vector.

Hit.h

The `Hit` class stores information about the closest intersection point. It stores the value of the ray parameter t and the visible color of the object at the intersection. The `Hit` data structure must be initialized with the background color and a very large t value. It is modified by the intersection computation to store the new closest t and the new visible color of intersected object.

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

Updated classes from assignment 0.

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

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
```

A second rendering mode is to visualize the t value of the closest intersection for each ray. For example, the following command line renders the same image as above, except the depth values $8.5 \rightarrow 10.5$ are mapped to

grayscale values from white → black. Depth values outside this range are simply clamped.

```
raycast -input scene.txt -size 100 100 -depth 8.5 10.5 depth.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

- Use a small image size for faster debugging. 64x64 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).