Interactive Computer Graphics
Coursework: Ray Tracing

Important

- The Computer Graphics coursework MUST be submitted electronically via CATE. For the deadline of the coursework see CATE. The files you need to submit are described later in this document.
- Before starting the assignment please make sure you have read the description of the environment and data formats below.

Make sure that you give yourself enough time to do the coursework by starting it well in advance of the deadline. If you have questions about the coursework or need any clarifications then you should come to the tutorials!

Coursework

Your goal in this assignment is to implement a simple ray caster. As given in the notes, a ray caster projects a ray for each pixel in the viewing plane and finds its intersections (if any) with objects in the scene being viewed.

To keep the model as simple as possible, you will implement the ray caster using the following assumptions:

- The camera is orthographic.
- Only primary rays need to be considered (i.e. reflection and refraction can be ignored).
- The primitives being rendered will consist of spheres only.
- The shading model will be very basic and does not need to incorporate lighting: Spheres will be rendered using either
  - a constant colour, the colour of the sphere's material, or
  - a value derived from the distance each ray travels from the camera to the intersection.

Good code design is obviously important in your assignments and you will be provided with a number of classes that you can incorporate your code into.

A generic *Object3D* class is provided to serve as the parent class for 3D primitives. You will derive the *Sphere* subclass from Object3D. A class *Group* is also derived from Object3D. This class represents a group of different primitives. Part of the code for this class is provided and you will need to complete the implementation of some of its functions.

An abstract *Camera* class is provided as the parent class for different types of camera. From this, you will derive the *OrthographicCamera* subclass. We also provide you with a *Ray* class and a *Hit* class to manipulate camera rays and their intersection points.

A *SceneParser* class is provided that can parse a text file used to represent all the components needed to render a scene, i.e. the camera, group of objects and the colour of the background. The scene information is stored in a text file and the name of the text file is given to your program as a command line argument so that the scene can be parsed.

The main part of the program takes command line arguments to specify various choices:

- The name of an input file containing the scene details to be parsed (as described above).
- The size of the camera's viewing rectangle.
- The details needed for a solid colour rendering of the scene.
- The details needed for a depth map rendering of the scene (see below).

Code and Classes you will need to write

Sphere

Sphere is a subclass of the pure virtual class Object3D that stores a center and radius. The arguments of the Sphere constructor should be the center, radius, and color. The Sphere class implements the virtual intersect method of its parent class:

```
virtual bool intersect(const Ray &r, Hit &h);
```

This function aims to find the intersection along a Ray that is closest to the camera. If a ray intersects with a primitive a value of `true` is returned and the Hit object contains the relevant information on the intersection: The distance, represented by `t` and the colour of the primitive intersected.

If an intersection is found such that `t` is less than the value of the intersection currently stored in the Hit object, Hit needs to be updated. Note that both the value of `t` and the colour must be modified.

Group

A Group is a special subclass of Object3D that represents multiple 3D primitives. It stores an array of pointers to Object3D instances. Most of the code is provided for this class, but you will need to provide implementations for two routines:

- The *intersect* method which loops over all primitives in the group calling each one's intersection method in turn.
- The *addObject* method which adds a new primitive to those already stored.

OrthographicCamera

The generic Camera class is provided for you. This class is abstract, the only method it contains is a pure virtual one:

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

This method provides a ray for a given screen location. The screen location is provided as a two dimensional floating point vector (`Vec2f`) which represents *screen* coordinates. Each component the screen coordinates is scaled to a value between zero and one. The Ray object returned by the method contains the origin and direction of the ray in *world* coordinates. The way a ray is generated depends on the type of camera being used. You will write the OrthographicCamera subclass which generates rays that all have the same direction but the origin of each ray varies depending on the screen location specified.

In world coordinates, an orthographic camera is described by a point (the centre of the image plane), three orthogonal unit vectors (an orthonormal basis) and a viewport size. The viewport is a square within the plane where the image is rendered, see Figure 1.

The arguments of the orthographic camera constructor are 1. The center of the viewport. 2. The projection direction. 3. An up vector. 4. The viewport size.

Note that 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 projection direction. It must be modified to be orthonormal to the projection direction. The third basis vector, the horizontal vector of the image plane, can be calculated from the projection and up vectors by using vector algebra.

The screen coordinates passed to the camera's generateRay method vary in the range $(0, 0) \rightarrow (1, 1)$. The origin of rays that start in the orthographic camera's viewport will vary depending on which screen location (pixel) they

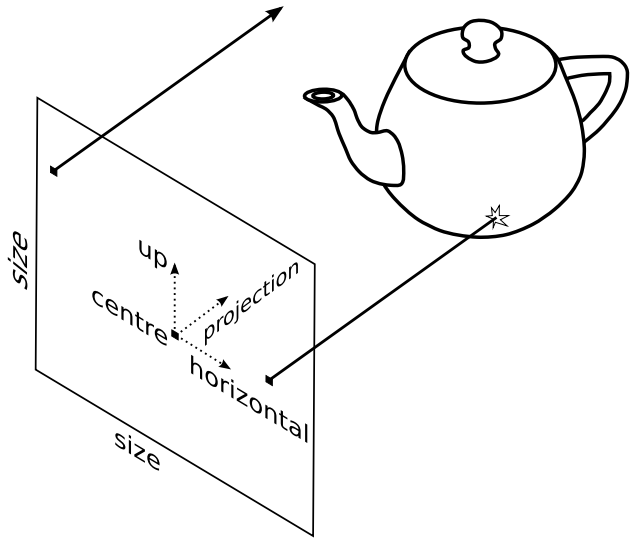

Figure 1: The model for an orthographic camera. Two examples of rays are illustrated, one hits the object in the scene and the other misses.

start from. In world coordinates, the origins of the rays should vary between:

$$\mathbf{c} - \frac{s}{2}\mathbf{u} - \frac{s}{2}\mathbf{h} \rightarrow \mathbf{c} + \frac{s}{2}\mathbf{u} + \frac{s}{2}\mathbf{h}$$

where s is the viewport size, $\mathbf{c}$ is the position vector of the centre point, $\mathbf{u}$ is the up vector and $\mathbf{h}$ is the horizontal vector.

The camera does not know about screen resolution. Image resolution should be handled in your top-level code.

The top-level rendering code

The file `raycast.cc` is provided for you and this contains the `main` function which reads input arguments, parses the scene and writes the output images.

You will write two top-level rendering functions:

```
void renderRGBImage(SceneParser &scene, Image &image)

void renderDepthImage(SceneParser &scene, Image &image)
```

Each function takes the scene that is parsed from the command line and renders it to the image according to the viewpoint of the camera and the details of the primitives. You will need to carry out two types of rendering: flat colour rendering and depth map rendering.

To render a flat colour image, you will need to determine if a ray intersects a primitive. A ray may intersect more than one primitive and may intersect a single primitive more than once.

You need to identify the intersection that is the nearest to the camera and which primitive this corresponds to. The colour for the pixel where the ray starts should then simply be the colour of the primitive. If a ray does not intersect any primitive, then the background colour should be assigned to the pixel.

To render a depth image, you again need to determine which intersection point along a ray is closest to the camera. This time, however, the colour of the primitive intersected is ignored and the distance from the camera to the intersection is used to obtain a pixel value instead. If a ray does not intersect any primitive at all, then the distance that should theoretically be recorded is infinity. Because this is impractical, we define two planes that are used to restrict the range of depth values that are assigned to the pixels. The planes are both parallel to the image plane and are referred to as the 'near' and 'far' planes.

If a ray's intersection is closer to the camera than the near plane, then a value of one should be assigned to the pixel. If an intersection is further than the far plane or the ray makes no intersection (i.e. intersects at infinity) then a value of zero should be assigned. For intersections between the near and far planes, the value assigned to the pixel should vary linearly between one (near) and zero (far). When assigning a depth value to a pixel in an RGB image, assign the same value to each of the red, green and blue components - this will give a greyscale image.

Code provided for you

ray.h : The Ray class represents rays from the camera by their 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. When initialising a Hit object, the colour should be set to the background color and a very large t value (*hint*: c++ has standard macros for infinity).

If a Ray and Hit object are passed to a primitive's intersect function, and the Ray intersects the primitive at a closer location than currently stored, the Hit object is modified to store the new closest t and the visible colour of the intersected primitive.
image.h : Code for reading and writing images, accessing and modifying pixels etc.
vectors.h : Code for manipulating and performing algebra on 2-, 3- and 4-D vectors, e.g. scalar products, cross products etc.
scene_parser.h, scene_parser.cc : Code to parse the scenes represented in text files which are given as command line arguments (see below).
group.h : Header file for a collection of primitives.

raycast.h : General header file to carry out all necessary `#include` commands.

The files

- `sphere.h` and `sphere.cc`
- `raycast.cc`
- `orthographic_camera.h` and `orthographic_camera.cc`
- `group.cc`

are incomplete and you can add your code to them.

Environment

The programming environment for the exercises is available on the Linux machines of the department and most Macintosh machines. You can also use the Windows machines in the lab but we recommend using Linux. This exercise should be programmed in C++.

Create a new directory and place all the source provided and the Makefile into it. To compile the program, go to the new directory and type `make` at the command line. This should compile the basic program for you although there will be warnings as the code is incomplete.

Data formats

Part of the input to the raycast executable is the description of the scene. This is given in the form of a text file. An example is given in Scene 1. The example begins by describing the parameters for an orthographic camera, then the background colour and finally a group consisting of two spheres. The first sphere is red and the second is green. All coordinates given in a scene file are given as world coordinates. The code provided for the assignment can parse such a text file for you.

The output from your program will be images in the PPM (Portable Pix Map) format. On the lab machines, the 'eye of Gnome' (eog) viewer is available on Linux for viewing the image files. Just type `eog myImage.ppm` at the command line.

Testing your code

You can test your code with scene files that are provided for you. You are also given the output images that your program should generate for each scene. You can compare your output with the output that should be given to check if your code is working correctly.

The command lines for rendering each scene are shown below and the corresponding output images are shown in Figure 2:

```
raycast -input scene1.txt -size 200 200 -output scene1.ppm -depth 9 10 depth1.ppm

raycast -input scene2.txt -size 200 200 -output scene2.ppm -depth 8 12 depth2.ppm

raycast -input scene3.txt -size 200 200 -output scene3.ppm -depth  8 12 depth3.ppm
```

What you need to submit

As well as the scenes and output provided for testing, two further scenes are also provided. When you are happy that your code is working, use it to generate the output files with the following command lines:

```
raycast -input scene4.txt -size 200 200 -output scene4.ppm -depth 13 16 depth4.ppm
raycast -input scene5.txt -size 300 300 -output scene5.ppm -depth 1 7 depth5.ppm
```

You will need to submit a zip/tar file containing the resulting RGB and depth renderings to CATE. You also need to submit the source code that you used. Include these in the zip or tar file in a folder named 'source_code'.

As an optional extra, make up your own scene and add it to the zip/tar file. Call the relevant files: scene6.txt, scene6.ppm and depth6.ppm.

Scene 1 An example of a scene description that can be read from a text file.

```
OrthographicCamera {
    center 0 0 10
    direction 0 0 -1
    up 0 1 0
    size 5
}

Background { color 0.2 0.2 0.2 }

Group {
    num_objects 2

    Material { diffuseColor 1 0 0 }
    Sphere {
        center 0 0 0
        radius 1
    }
    Material { diffuseColor 0 1 0 }
    Sphere {
        center 1 1 1
        radius 0.75
    }
}
```

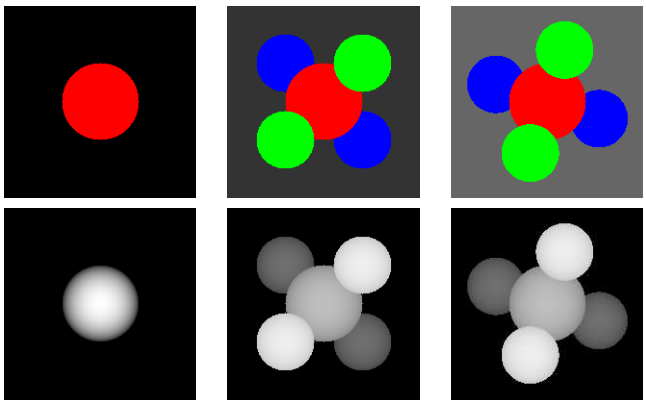

Figure 2: Rendered images for the test scenes provided. Left: scene 1. Middle: scene 2. Right: scene 3. Top row: rendered colour images. Bottom row: rendered depth images. See text.