Computer Graphics
Second Coursework: Ray Tracing

February 22, 2009

Important Notes

- 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 at the end of this document.
- Before starting the assignment please make sure you have read the description of the environment and data formats below.

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, or
  - the distance of each ray's intersection from the camera.

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 represent all the components needed to render a scene, i.e. the camera, group of objects and the colour of the background. This scene information can be stored in a text file which is passed to your program as a command line argument so that the scene can be parsed.

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

- The name of an input file containing the scene details to be parsed.
- The size of the camera's viewing rectangle.
- The name of an output image containing a solid colour rendering.
- Details for rendering a depth map of the scene:
  - Distances of the near and far clipping planes - two planes parallel to the viewing plane. The depths of intersections within the clipping planes should be scaled so that they vary from 1 (near) to 0 (far). Depth values outside this range are simply clamped.
  - The name of an output image containing the depth map rendering.

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, float tmin);
```

This function aims to find the intersection along a Ray that is closest to the camera and still within the clipping planes. 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. The argument `tmin` is used to specify the distance of the near clipping plane.

If an intersection is found such that `t` > `tmin` and `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 or the 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`) with each component scaled to be between zero and one. The returned Ray object 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 according to the screen location specified.

In world coordinates, an orthographic camera is described by a point and three orthogonal unit vectors (an orthonormal basis) along with a viewport size. The viewport is a square in the image plane where the image will be rendered. This is illustrated in Figure 1

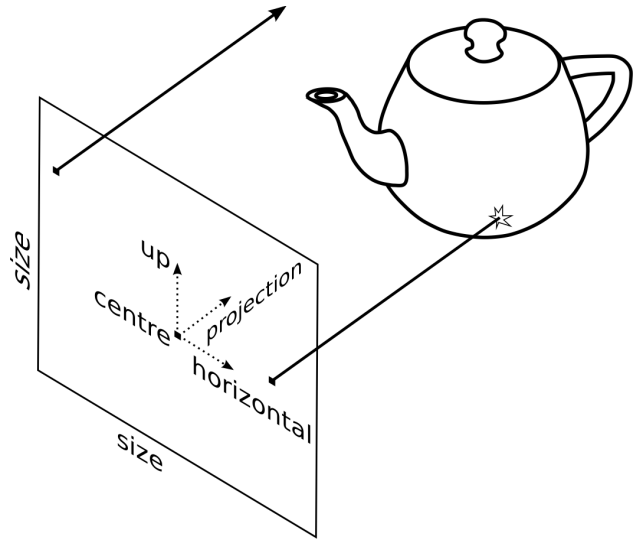

Figure 1: The model for an orthographic camera. Two examples of rays are illustrated, one hits the object in the scene and the other misses.

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 direction and the up vector using vector algebra.

The screen coordinates passed to the camera's generateRay method vary in the range $(0, 0) \rightarrow (1, 1)$. The world coordinates for the the origins of the corresponding rays should vary in the range:

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

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 as input the scene that is parsed from the input argument and renders to the image according to the viewpoint of the camera and the primitives. The first function renders the scene using a flat colour while the second renders according to the depth of intersections from the camera as described above. As described earlier, the depth values should vary from one (near) to 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, 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.

matrix.h, matrix.cc

Code for manipulating and performing algebra on matrices, e.g. addition, scaling, multiplying by a vector 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 within them.

Environment

The programming environment for the exercises is available on the Linux machines of the department. 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 in your home directory and place all the source provided and the Makefile into it.

When you have provided the required source code, you can compile the program by going to the new directory and typing make. If you have questions about the coursework come to the tutorials!

Data formats

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.

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
    }
}
```

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
```

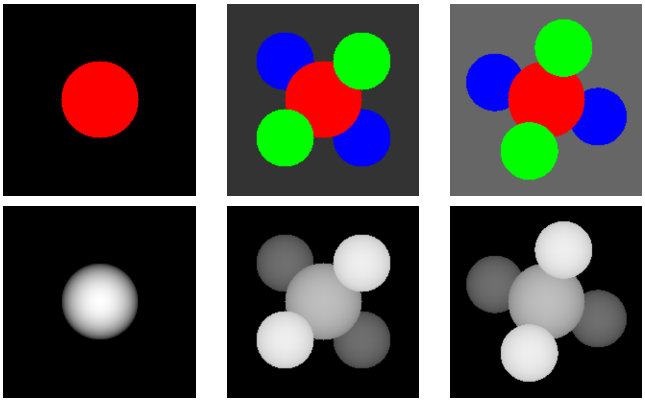

Figure 2: Rendered images for the test scenes provided. Left: scene 1. Middle: scene 2. Right: scene 3. Top row: rendered colour images. Bottom row: rendered depth images. See text.

```
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 the resulting RGB and depth renderings to CATE. You also need to submit the source code that you used in a zip or tar file.