Chapter 1

Ray Tracing

1.1 Using the ray tracer package

The ray tracing software package implements an object-oriented approach to rendering and ray tracing. The object oriented design includes base classes for materials, for geometries, for lights and for textures. This allows considerable flexibility in adding new features, since it allows the system to be augmented with new features, such as new geometries without affecting the functionality of older code.

The material classes include the usual material properties such as ambient, diffuse and specular color, spotlight affects, and attenuation. In addition, the material classes include reflection and transmission color coefficients for use in recursive ray tracing.

The geometry classes describe the shape of viewable objects: currently the only supported geometric shapes are spheres, triangles and parallelograms. I hope to include more geometic shapes soon. The geometry classes are responsible for detecting intersections of rays against a particular geometric shape. The geometry classes also calculate normals and keep track of the material of an objects. In addition, they calculate u and v coordinates for texture mapping purposes,

The texture mapping classes are in essence implemented as "callback" routines. This means that it is easy to add algorithmic texture maps in addition to the more traditional bit-map (table lookup) texture maps. Currently the only supported texture maps are algorithmic and generate a checkerboard pattern of squares of two different materials.

In order to make the ray tracer software more modular, as well as easier to use, the software is split into three levels (in separate C++ projects). These levels are:

Top level: ray tracing. The top level routines implement the recursive ray tracing algorithm and the high level scene description. In the sample implementation, this is split into two parts. First, the program

`RayTraceData` makes function calls to set up the lights and the viewable geometries and their materials and textures. Second, the program `RayTrace` implements the high-level recursive ray-tracing algorithm.

These routines are intended to be easy to modify without have to understand the internal structure of the intermediate and low levels. Thus it would be relatively straightforward to modify `RayTrace` to incorporate distributed ray-tracing.

Imtermediate level: geometry and rendering. The intermediate level routines handle lights, geometries, materials and texture maps. In addition, they include code for local lighting calculation (only Phong lighting is supported at present). The lights include all the usual OpenGL style features such as ambient, diffuse and specular light components, attenuation, and spotlight effects. The material include the usual OpenGL material properties, including ambient, diffuse and specular colors and shininess, plus additional properties such as reflection and transmission coefficients. The geometries are implemented with a C++ base class called `VeiwableBase`: at present, derived classes include spheres, triangles and parallelograms (more geometry classe are planned to be implemented. These geometries include efficient routines for calculating intersections with rays, and also calculate normals and u, v coordinates for texture mapping. Texture maps are also C++ classes and are implemented somewhat like callback routines: a texture map is attached to a geometry, and when a geometry is intersected by a ray, the texture map routines are called. This allows a flexible framework for texture mapping that automatically allows texture maps that are procedurally calculated as needed.

Low level routines: linear algebra. These routines are in the project *VrMath* and include a number of linear algebra routines for 2-vectors, 3-vectors and 4-vectors. The intermediate and high level routines have been written so as to isolate the low-level routines, to allow modifications to the high-level routines to need very little knowledges of the low-level routines.

1.1.1 The high-level raytracing routines

The high-level ray tracing routines are illustrated by the example code in `RayTrace.cpp` and in `RayTraceData.cpp` and `RayTraceData.cpp`. We will discuss first the `RayTrace` routines, which control the recursive ray tracing procedures.

`RayTraceView:` This is the highest level routine which initiates the ray tracing procedure. `RayTraceView` loops over all the pixels in the view window. For each pixel, it calculates the view ray from the view position towards the center of the pixel and then calls the recursive routine `RayTrace` to calculate the color value to render the pixel. The `RayTrace` routine returns the value `curPixelColor` which is stored into the pixel array. After calculating all the pixel colors, the pixel array's `Draw` routine is called to store the pixel colors into the rear OpenGL rendering buffer; then the buffers are switched to show the raytraced scene.

`RayTrace`: This is the routine that recursively traces rays and combines color values. It is the heart of the recursive ray tracing algorithm. Its parameters consist of (a) a trace depth, (b) a starting position for the ray, (c) a unit vector giving the direction of the ray, (d) a 4-vector `VectorR4` in which a color value is returned, and (e) an avoidance number, which specifies what object the array originates from.

`RayTrace` begins by calling `FindIntersection` which calculates whether the ray intersects some viewable object (i.e., some geometry). If not, then the ray is presumed to have passed completely out of the scene and the default, background color is returned. Otherwise, an intersection point is returned. This intersection point is returned as an object of type `VisiblePoint`: the `VisiblePoint` class includes information about the position, the normal and the material properties of the intersected point. `RayTrace` calls the routine `CalcAllDirectIllum` to calculate the illumination of the intersected point with the local lighting model: the local lighting model incorporates the effect of global ambient lighting and of direct illumination from lights (the latter is presently completed according to the Phong lighting model). Then, if the trace depth has not been exhausted yet, `RayTrace` spawns reflection rays and transmission rays. Reflection rays have their direction computed by equation ??. `RayTrace` is called recursively with the reflection ray. Whatever color returned is filtered by the reflective color of the material on the intersection point and added to the color as already calculated according to the direct illumination. Finally, a transmission ray may be spawned. The direction of the reflection ray is calculated by a routine `CalcRefractDir` according to equation ??, and otherwise transmission rays are handled in the same manner as reflection rays.

When examining the routine `RayTrace`, you will note that it uses some C++ classes `VectorR3` and `VectorR4`. These are vectors of real numbers, and their members are accessed via `A.x`, `A.y`, `A.z`, and `A.w`. The main reason for their use in the high-level ray trace routines is that positions and directions in 3-space are conveniently stored in a single `VectorR3`, which greatly simplifies the interfaces for the ray trace routines: the alternative would be to pass arrays of floating point numbers, which would be somewhat less elegant (and certainly more prone to errors due to the lack of type-checking errors). If you are modifying only the high level ray tracing routines, you can probably avoid using very much of the `VectorR3` or `VectorR4` classes.

`SeekIntersection`: This routine loops through the array of viewable objects, checking each one for an intersection with a ray. The ray is specified in terms of its starting position, and a unit vector giving its direction. The very first thing that is done is that the starting position is moved a very small distance in the direction of the ray: this is to avoid have a repeated intersection with the same point due to roundoff errors.* `SeekIntersection` must check every viewable geometry for intersections with the ray, and returns the closest intersection found.

`CalcAllDirectIllum:` This routine takes as input a view position and a visible point. It is presumed that the visible point is in fact visible from the view position. This not includes the case where the view position is the position of the viewer, but also includes the case where the view position is a position with a traced ray has orginated from a reflection or refraction. The direct illumination of the point includes the following components: any emissive color of the visible point, color due to global ambient lights, and, for each light, the color due to direct illumination by the light. Before any illumination from a light is calculated, a shadow ray, or "shadow feeler", is traced from the visible point to the light position. If the shadow feeler is not intersected by any viewable object (as determined by the routine `ShadowFeeler`), then the light is presumed to be shining on the visible point, otherwise, the light is deemed to have its direct light completely blocked. In either case, the value of `percentLit` is set, as appropriate, to either 0 or 1, indicating the fraction of the light which is illuminating the visible point. Each light has its illumination calculated with `DirectIlluminateViewPos` whether or not it is shadowed, since even shadowed lights contribute ambient lighting.

`ShadowFeeler:` This routine works similarly to `SeekIntersection`. However, it does not need to return a visible point, and since it returns only `true` or `false` depending on whether an intersection is found, it can stop as soon as any intersection is found, without needing to continue searching for a closer intersection.

The program `RayTraceData` contains routines for describing the virtual scene to be rendered. First, there is the routine `SetUpMainView`, which describes information about the main view position, i.e., the position of the camera. This first creates a new `CameraView`, and sets its position, and its direction of view. The direction of view is specified by any non-zero vector (in this case as stored in an array of `double`'s. The camera view is conceptualized by envisioning the camera as pointing at the center of a view screen of pixels — the view screen is thought of as being positioned in a rectangular array in 3-space. The distance to the view screen is set by a a call to `SetScreenDistance`, and its width and height by a call to `SetScreenDimensions`. However, the routine `ResizeWindow` in `RayTrace` will resize the array of pixels as necessary to as to keep the entire view screen in view. In the aspect ratio of the OpenGL rendering window is different from the aspect ratio of the view screen, then the pixels are positioned so the entire view screen is area is rendered.

There is no use of a near clipping plane or far clipping plane. However, the variable `MAX_DIST` should be set to be at least as large as the diameter of the scene, since rays are traced only to that distance.

*This needs to be reworked a little bit, so this method of avoiding repeated interserctions may change very soon.

Second there is a call to `SetUpMaterials`. This creates an array of pointers to materials. Each material is created with the C++ operator `new`. It then has its material properties set. These material properties include its ambient color, its diffuse color, its specular color, it reflection color and its transmissive color. They also include its shininess and its index of refraction. If the reflection color is set to zero (but by default it is not zero), then the object does not generate reflection rays. If the transmission color is zero (and zero is its default value), then the object does not generate refracted rays. The index of refraction is used only for refracted rays of course. Good values for the index of refraction would be numbers like 1.33, which is approximately the index of refraction of water, or 1.5, which is approximately the index of refraction for glass.

The color values for materials are specified by given three or four floating point numbers for the red, green, blue and possibly alpha values of the color. (The alpha values are just ignored for ray tracing purposes, but were included in my software in case future versions of the code want to exploit them.) The color values can be specified by either giving an array of floating point numbers, very similar to the convention used by OpenGL, or alternatively they can be specified by passing in a `VectorR3` or `VectorR4` object. This means that there a wide variety of interfaces supported to set color values, and you may use whatever one seems most convenient.

For example, here are the eight possible ways to set the ambient color of a material `Mat` (the alpha value defaults to 1.0):

```
M.SetColorAmbient(0.2,0.3,0.4);

M.SetColorAmbient(0.2,0.3,0.4,1.0);

M.SetColorAmbient(0.2,0.3,0.4);

double c[3]={0.2,0.3,0.4};
M.SetColor3Ambient(&c[0]);

double c[4]={0.2,0.3,0.4,1.0};
M.SetColor4Ambient(&c[0]);

float c[3]={0.2,0.3,0.4};
M.SetColor3Ambient(&c[0]);

float c[4]={0.2,0.3,0.4,1.0};
M.SetColor4Ambient(&c[0]);

M.SetColorAmbient( VectorR4(0.2,0.3,0.4,1.0) );
```

You can probably get by just copying what is done in `RayTraceData.cpp`, but for more information you will need to check the header files, such as `Material.h` to learn exactly what interfaces are available.

The next routine that is called is `SetUpLights`. It defines one or more lights, and sets their properties. Their properties include the usual ones such as position, ambient color, diffuse color, and specular color. It is also possible to set attentuation constants and spotlight effects of exactly the same kind as are supported by OpenGL. In addition, the light can be made directional instead of positional.

The next routine that is called is `SetUpViewableObjects`. There are three kinds of viewable objects currently supported, triangles, parallograms and spheres.

`ViewableSphere` class: You may set the center position and the radius of the sphere. In addition, you may attach a material to the sphere.†

`ViewableTriangle` class: The triangles are specified by giving their three vertices, in counterclockwise order. You may attach separate materials to the front and back faces of the triangle.

`ViewableParallelogram` class: A parallelogram is specified by three of its vertices (the A, B and C vertices). The fourth, D vertex is computed from the other three. You may attach separate materials to the front and back of the parallelogram.

Every viewable object also supports texture maps, which are applied by called `TextureMap`. You may also have separate texture maps for the front and back surfaces of the geometry. Triangles and parallelograms have default texture coordinates, or you may give (bilinearly interpolated) texture coordinates for triangles and parallelograms by calling the routines `TextureCoordX(u,v)`, where "X" is a valid vertex designation A, B, C or D. An example of texture coordinates for triangles can be found in `RayTraceData`. Examples of texture coordinates for parallelograms and spheres can be found in `RayTraceData2` as soon as I make it available.

†Currently you may only attach a single material, but I plan to change this so that you may have different materials for the inside and outside surfaces of the sphere. This would, for example, let you suppress internal reflections by attaching a different material to the inside than to the outside of the sphere.