RayX

Ray Tracer for Photo-Realistic Images and Videos

Tianlin Shi

Institute for Interdisciplinary Information Sciences

January 2014
Chapter 1

Core Implementation

1.1 Language

RayX is implemented in C++ 11, and should be compilable with gcc version > 4.8, or LLVM version > 3.0. It depends on no third-party libraries except for libpng and png++, which brings graphics I/O capabilities to our program. However, RayX and these libraries are not strongly coupled, as we would explain later.

Why C++ 11? Well, this 2011 version of C++ brings many features of "modern" programming languages. An interesting feature is functional (lambda expression). In RayX, functionals are used for two purposes. One is to reduce code duplication. For example, in the following code segment from RayXBounding.cpp, two local functions compute, project are defined and reused to compute bounding surface for each dimension: x, y, z.

```cpp
auto compute = [&] (int i) {
    dis_max = -DBL_MAX;
    for(Vector& vi : points) {
        for(Vector& vj : points) {
            if((vi-vj).norm() > dis_max) {
                dis_max = (vi-vj).norm();
                vi_max = vi;
                vj_max = vj;
            }
        }
        norm[i] = (vi_max-vj_max).normalize();
        df[i] = fmax(norm[i]*vi_max, norm[i]*vj_max);
        dn[i] = fmin(norm[i]*vi_max, norm[i]*vj_max);
    }
    auto project = [&] (int normi) {
        for(Vector& v : points) {
            v = v-norm(normi)*vn(norm[normi]);
        }
    }
    compute(0);
    project(0);
    compute(1);
    project(1);
    compute(2);
};
```

Another way functional is used in this project is for decoupling of modules. To illustrate the merit, let's consider texture mapping in the ray tracing program. The texture may be
generated in a diverse spectrum of ways: image, fractals, simulations, etc. The C++ 11 solution is pretty simple: define a functional describing the textures,

```cpp
using TextureMapFunc = std::function<ColorRGB(double, double)>;
```

and use the functional as interface during tracing:

```cpp
if(texture.textureMap != nullptr)
  info.color = texture.textureMap{ix, iy};
```

RayX is a multi-thread program and most functions are re-entrant. The combination of functionals and std::thread leads to powerful simplifications – unlike pthread, we now can define `inline` thread procedures.

```cpp
threads[t1,num_thread_unit+t1j] = std::thread([&int_id] { 
  int t1 = _id/num_thread_unit, t1j = _id%num_thread_unit;
  for(int i = t1; i < resol_height; i += num_thread_unit) {
    for(int j = t1j; j < resol_width; j += num_thread_unit) {
      // DO TRACING.
    }
  }, t1,num_thread_unit+t1j);
```

### 1.2 Basic Types

Two intensely used types in RayX is **Vector** and **ColorRGB**.

**Vector** is a C++ class supporting arithmetic operators `+,-,*,/`, reference `[]` and norm computation.

```cpp
/* 3-d vector */
class Vector {
public:
  /* values */
  double x, y, z;
  Vector operator*(const double num) const;
  Vector operator-(const Vector& v) const;
  ...
  Vector outprod(const Vector& v);
  ...
  double norm(int ell = 2);
};
```

**ColorRGB** extends the **Vector** by reloading its parameters.

```cpp
class ColorRGB : public Vector {
public:
  /* constructor */
```
ColorRGB(double r, double g, double b)  
: r(x), g(y), b(z) // reload x, y, z {  
   this->r = r; this->g = g; this->b = b;  
}  
/* RGB color representation */  
double &r, &g, &b;

RayX::Beam is a simple class of light beams defined by the equation:

\[ s = s_0 + tr, \quad t > 0 \quad (1.1) \]

namespace RayX {
    class Beam {
        public:
            Vector source, direction;
            ColorRGB color;
            Medium medium;
    };
}

It also specifies the medium it currently travels, so that its speed can be computed.

1.3 Geometric Objects

RayX::Object is the base class for all geometric objects, including simple geometric shapes such as sphere, plane, and complex meshes.

/* decide if the object intersects with the given beam */
virtual LocalInfo intersect(Beam& beam) = 0;

/* decide if the object intersect with plane, used in k-d tree */
/* 1 if all points on the pos side, -1 if on the neg side, 0 if intersect */
virtual int intersect(Vector norm, double bias);

/* apply a transform to the given object */
virtual void applyTransform(RayX::Vector translate, RayX::Vector scaling,
                            RayX::Vector rotation);

/* get which type this object belongs, since C++ does not support refraction */
virtual ObjectType getType();
Every simple geometric shape implements at least partially these interfaces. Some built-in simple geometric shapes include \texttt{RayX::Surface}, \texttt{RayX::Triangle}, \texttt{Ray::Sphere}, etc. In the next chapter, we introduce complex meshes, which is essentially based on the class \texttt{RayX::Triangle}.

### 1.4 Light Sources

The point light is defined by

```cpp
class Light {
    public:
        Light(Vector pos, ColorRGB I_m, double r0, double r1, double r2);
}
```

Given a location at distance $d$, its \textit{irradiance} can be computed by

```cpp
/* compute light intensity */
ColorRGB intensity(Vector p);
```

which is exactly based on the model

$$i_m = \frac{I_m}{r0 + r1d + r2d^2} \quad (1.2)$$

### 1.5 Cameras

The \texttt{RayX::Camera} takes a Tracer functional and shoots beam from a regular lattice to fill in a colorbuf.

```cpp
class Camera{
    public:
        /* constructor */
        Camera() {}
        ~Camera();
        Camera(Vector refpoint, Vector center, Vector oy, double pixel_spanx, double pixel_spany, int resol_width, int resol_height);
        /* snapshot the scene with the given tracer functional */
        template<typename Func>
```
void snapshot(Func tracer) {
    /* display the color buffer with the plot function. */
    * input: plot(x,y,red,green,blue). */
    template<typename Func>
    void display(Func plot) {
    }

private:
    /* camera buffer */
    ColorRGB** colorbuf;
}

For example, a class implementing the trace() function can call snapshot() like this:

camera->snapshot([&] (Beam& beam) {this->trace(beam);});

Once the snapshot finishes, one can output the image with display() function, which takes a customized display functional. In RayX, Func plot is implemented using libpng library.

camera->display([&] (int x, int y, double r, double g, double b) {return
    image[y][x] = png::rgb_pixel(int(r*255),int(g*255),int(b*255));});

However, it is flexible and free to replace other image generators without the need to rewrite this part of RayX.

### 1.6 Scene

A scene sets up pools for three categories of objects: geometric shapes, lamps and cameras, which are added through the register functions. Once all parts are set, the render() function is called, which calls the snapshot function of all cameras in the scene with the callback function Scene::trace.

class Scene {
public:
    /* add a camera */
    void registerCamera(Camera& camera);

    /* add an object */
    void registerObject(Object& object);

    /* add a light source */
void registerLight(Light& light);

/* render the scene for all cameras */
void render();

/* trace a beam */
LocalInfo trace(Beam& beam, unsigned int depth = 0);
}

The trace function looks like this.

LocalInfo info = intersect(beam);
if(!info.isnull) {
    info.norm = info.norm.normalize();
    /* incorporate diffuse light */
    for(Light* light : light_pool) {
        beam.color += model(light, beam, info);
    }
    if(depth < MAX_TRACE_DEPTH)
        /* trace reflection */
        ...
        /* trace refraction */
        ...
}
}

The model is a lambda expression for illumination. For example, we could define Phong model as follows:

auto phong = [] (Light* light, const Beam& beam, const LocalInfo& info) ->
    ColorRGB {
        /* Phong model */
    };

where the Phong model computes the ColorRGB at the intersection:

\[ I_p = k_a i_a + \sum_{m \in \text{lights}} \left( k_d (\ell_m \cdot n) i_m + k_s (r_m \cdot v)^\alpha i_m \right). \]  

(1.3)

Each \( i_m \) is computed via Equation (1.2), and \( i_a \) is the ambient light. \( k_a \) is the ambient coefficient, \( k_d \) is the diffusion coefficient and \( k_s \) is the specular coefficient. \( \ell_m \) is the direction
towards the m-th light source, \( n \) is the surface normal, \( r_m \) is the direction in which the light is reflected and \( v \) is the direction towards the viewer.

One more thing to notice about the tracer algorithm is the intersection with objects. Besides calling the objects' `intersect(Beam& beam)` function, it should commit to the convention of choosing the normal \( n \) that faces towards the viewers (there are always two normals to choose for any surface).

```cpp
/* append norm */
if(norm*beam.direction > 0) // flip to visual direction.
   info.norm = norm.neg();
else
   info.norm = norm;
```

### 1.7 Floating Point Issue

RayX uses `double` for all computations. Single-precision `float` could be problematic. For example, in refraction tracing and the sequential transversal algorithm of k-d tree, a small threshold is needed as a trick. The threshold is often set to be \( 1 \times 10^{-8} \) or \( 1 \times 10^{-16} \). Figure 1.1 shows some artifacts of the rendered scene using `float` precision.

### 1.8 Other Issues

It is important to consider the material's absorption of light during refraction. The example in Figure 1.2 shows rendering a glass with or without absorption. We see that with absorption considered, the material looks more realistic.

### 1.9 Results

Figure 2.5 shows some examples of this "bare bone" tracer.
Core Implementation

Figure 1.1: Artifacts From Single-Precision Computation

(a) Rendered using float

(a) Rendered using double
Figure 1.2: Left: without material absorption. Right: with material absorption.

Figure 1.3: Some examples
Chapter 2

Creating Complex Objects

2.1 Interface with Blender

Blender is a free and open-source 3D computer graphics software product used for creating animated films, visual effects, art, etc. RayX has an interface with Blender software, which is the .ObjX file format. The .ObjX file specifies a scene: the cameras, the lights and the objects.

We wrote a python script to export the scene from Blender and to generate the .Objx file.

```python
from math import *
import os
import bpy

working_path = "\Volumes\Universe\Courses\ComputerGraphics\RayX\RayX\Bin\Debug\modal/"
output = open(working_path + "default.obj", 'w')

# output material
for mat in bpy.data.materials:
    output.write(\nm
    output.write(\n    output.write(\n    output.write(\n    output.write(\n    output.write(\n    if len(mat.texture_slots) > 0 and hasattr(mat.texture_slots[0].texture, 'image')): output.write(\n
# output objects.
ucount = 0
```

FIGURE 2.1: Python script to export .ObjX from Blender
2.2 The .ObjX File Format

The Wavefront .obj file is an open and simple format for designating geometric shapes and textures. It represents object 3D geometry alone, including position of vertices, vertex normals, and the faces that make each polygon defined as a list of vertices, and texture vertices. An .obj file may look like this:

```
# List of Vertices, with (x,y,z[,w]) coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# Texture coordinates, in (u,v [,w]) coordinates, these will vary between 0 and 1, w is optional and default to 0.
vt 0.500 1 [0]
vt ...
...
# Normals in (x,y,z) form; normals might not be unit.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in ( u [,v] [,w] ) form; free form geometry statement ( see below )
vp 0.310000 3.210000 2.100000
vp ...
...
```
Creating Complex Objects

For further flexible description of the entire scenes and multiple objects involving meshes, cameras and light sources while maintaining the simplicity, we extend the format to .ObjX files, which is compatible with the .obj format. Below we include a list of specifications for this file format.

### 2.2.1 Objects

The .ObjX file format specifies not only how each object is shaped, but also its inherent properties such as position, rotation, scaling. We use "o" to mark the object specification and [dot] to indicate property affiliation, such as follows:

```
   o [name]
   o.location [x] [y] [z]
   o.scale [sx] [sy] [sz]
   o.rotation [α] [β] [γ]
   o.color [r] [g] [b]
```

The rotation angle uses Euler angle XYZ.

As in .obj file, we use \( v \ [vx] \ [vy] \ [vz] \) for vertices and \( f \ [v_1] \ [v_2] \ [v_3] \ldots \ [v_k] \) for faces. An example is like

```
o Cone
o.location 1.415693 -0.169025 -2.710694
o.scale 1.000000 1.000000 1.000000
o.rotation_euler 0.000000 0.000000 0.000000
o.color 0.000000 0.146236 1.000000 1.000000
v 0.000000 0.000000 -1.000000
v 0.723607 -0.525725 -0.447220
...
f 106 118 119
```
The example shows three objects: a cone, a plane and a cube, as shown in the Figure 2.3.
2.2.2 The Camera

The objects in Figure 2.3 looks rotated becomes we specified a camera not aligned to the horizon. The scene/camera is defined by

\[
\begin{align*}
\text{s} & \quad \text{[name]} \\
\text{s.resolution}_\text{x} & \quad \text{[width]} \\
\text{s.resolution}_\text{y} & \quad \text{[height]} \\
\text{s.width} & \quad \text{[scene width]} \\
\text{s.height} & \quad \text{[scene height]} \\
\text{s.angle} & \quad \text{[angle]} \\
\text{s.location} & \quad \text{[x]} \quad \text{[y]} \quad \text{[z]} \\
\text{s.rotation}_\text{euler} & \quad \text{[} \alpha \text{]} \quad \text{[} \beta \text{]} \quad \text{[} \gamma \text{]} \\
\end{align*}
\]

Notice the depth of field “s.angle” and scene width/height naturally defines the distance between the lens and the CCD:

\[
\text{distance(lens, CCD)} = \sqrt{s\text{.width}^2 + s\text{.height}^2} \cdot \frac{\tan(s\text{.angle}/2)}
\]

The camera parameter in Figure 2.3 is

\[
\begin{align*}
\text{s. Scene} & \\
\text{s.resolution}_\text{x} & 1280 \\
\text{s.resolution}_\text{y} & 720 \\
\text{s.width} & 50.0 \\
\text{s.height} & 25.0 \\
\text{s.angle} & 1.5707969665527344 \\
\text{s.location} & -1.133831 \quad 5.953196 \quad 0.481656 \\
\text{s.rotation}_\text{euler} & -2.262752 \quad 3.136321 \quad 0.051288 \\
\end{align*}
\]

2.2.3 Lamp

\[
\begin{align*}
lamp & \quad \text{[name]} \\
lamp\text{.color} & \quad \text{[red]} \quad \text{[green]} \quad \text{[blue]} \\
lamp\text{.energy} & \quad \text{[intensity]} \\
lamp\text{.falloff}_\text{type} & \quad \text{[FALL_OFF_TYPE]} \\
lamp\text{.distance} & \quad \text{[distance]} \\
lamp\text{.location} & \quad \text{[x]} \quad \text{[y]} \quad \text{[z]} \\
\end{align*}
\]
Here is an example.

```plaintext
lamp Lamp
lamp.color 1.000000 1.000000 1.000000
lamp.energy 1.000000
lamp.falloff_type INVERSE_SQUARE
lamp.distance 29.999982833862305
lamp.location 4.076245 -3.868158 5.903862
```

### 2.2.4 Material

To make objects close to those in the real world, it is the art of design to pick the right material. In the basic Phong model (3.3), we have specified parameters $k_a$, $k_s$, $k_d$. In .ObjX file, materials are defined as

```plaintext
newmtl [name]
  Kd [red] [green] [blue]
  Ks [red] [green] [blue]
  Alpha [float]
  Mirror [red] [green] [blue]
  mtl.image [path]
```

The mtl.image species a texture mapping for the object, if possible. For example,

```plaintext
newmtl Material.001
  Kd 0.800000 0.800000 0.800000
  Ks 0.370629 0.370629 0.370629
  Alpha 1.000000
  Mirror 0.000000 0.000000 0.000000
  mtl.image /Volumes/Universe/RayX/Bin/Debug(bitmap/spotlight_soil.png
```

As illustration, we map the surface of mars to a ball.
2.3 Back to simple shapes

Since the solid objects in Blender are represented with either meshes or NURBS, problems arise if we still want to include simple geometric shapes. The issue is resolved if we designate a special property for these objects.

By "fitting" the meshes, we could create the simple objects.

```cpp
if(special_flag == "is_surface") {
    Surface* surface = new Surface();
    surface->buildFromMeshes(*obj);
    surface->applyTransform(obj_center, obj_scale, obj_rotation);
    delete obj;
    objx = surface;
}
```
In this chapter, we introduced two major improvements over the "bare bone" tracer.

3.1 K-D Tree Acceleration

Inspired by the review article [1], RayX::KDTree build the K-D using medium heuristics. For each node, we add objects that intersect with the bounding box. To check intersection, we use both the sequential and transversal algorithm [1].

```cpp
void KDTree::build(KDTreeNode* node) {
    int depth = node->m_depth;
    if(node->satisfy_terminate()) return;
    SplitPlane plane = node->findSplitPlane();
    KDTreeNode* left = new KDTreeNode(depth+1);
    KDTreeNode* right = new KDTreeNode(depth+1);
    node->subdivide(plane, left, right);
    this->build(left);
    this->build(right);
}

LocalInfo KDTree::intersect(Beam& beam) {
    double entry_distance, exit_distance;
    LocalInfo info;
    if(m_root->m_bbox.intersect(beam, entry_distance, exit_distance) ==
   true) {
        exit_distance = fmax(0, entry_distance);
        for(;;) {
```
KDTreeNode* leaf = this->locate_leaf(m_root, beam.source+beam.direction*(exit_distance+ZERO_THRESHOLD));
if(leaf == nullptr) break;
if(leaf->intersect(beam, info, entry_distance, exit_distance)) {
    return info;
}
info.isnull = true;
return info;

With the KDTree, we can now render much more complicated meshes with speed hundreds of times faster.

![Example Car](image)

**Figure 3.1:** Example Car.
3.2 Smooth Shading

For complex objects modeled with meshes, the rendered ones usually do not look smooth. Instead of increasing the number of meshes, smooth shading offers a way to solve this problem by computing a pseudo-normal at the intersection point.

\[ n(v) = \frac{1}{|\text{adj}(v)|} \sum_{f \in \text{adj}(v)} n(f) \] (3.1)

Suppose the barycentric coordinate of the intersection point is \( p = (\alpha, \beta, \gamma) \), then its pseudo-normal is computed as
Despite the simplicity of this method, some technique issues arise during practice. Recall that in our illumination model, we use $n \cdot v$, the dot product between the normal and the viewer direction to determine if a side of the surface can be seen. However, if we use smooth shading, the surface normal would change. As shown in Figure 3.4(a), the other half of the cylinder is not "seen" due to change of surface normals.

To fix this, we use the original surface normal for intersection while the pseudo-normal for shading. In this case, the angle between pseudo-normal and light source direction may be greater than 90 degrees. We modify the Phong model as follows

$$I_p = k_a i_a + \sum_{m \in \text{lights}} \left( k_d \max(0, \ell_m \cdot n) i_m + k_s (r_m \cdot v)^\alpha i_m \right).$$  \hfill (3.3)
Figure 3.5: Change of surface normal can lead to misestimate of illumination. (a). not with smoothshading. (b). with the wrong shading.

```cpp
info.norm = info.norm.neg();
```

Furthermore, since the .ObjX file does not include surface normal information, we are not able to infer and choose normals so that they face "outwards" the object (see Figure 3.5). Therefore, while averaging the surface normals to compute the vertex normal, we enforce a continuity constraint.

```cpp
void Vertex::pushNorm(const RayX::Vector &_norm) {
    if(norm_count >= 1 && norm*_norm < 0) // force norm continuity.
        norm = norm+_norm.neg();
    else
        norm = norm+_norm;
    norm_count++;
}
```

With all these trick, we are able to present an ideal smooth shading method.
FIGURE 3.6: Car with smooth shading
Bibliography
