RAY TRACING IN ONE WEEKEND

翻译

翻译 BY MARGOO

原英文版本：**VERSION 4.0.0-ALPHA.1, 2023-08-06**

Copyright 2018-2023 Peter Shirley. All rights reserved.

译者前文

Peter Shirley 教授的 Ray Tracing Weekend 系列书籍一直是我个人十分喜爱的一本入门书籍，在初次阅读完后就萌生了对其进行翻译的想法，鉴于原书是基于 STBImage 的代码，和国内大多使用 EasyX 进行教学的环境有一定出入，因此我对代码进行了一定的“本土化”，并且，我对原著中的图片也重绘翻译，对一些篇章进行了修改，希望可以有利于后来者吧。

本译文是译者在繁重的课业之余用碎片化的时间所译成的，必定存在许多瑕疵，希望各位读者不要吝啬建议，可以通过我在前文末留下的联系方式同我取得联系，共同交流。

对于阅读本译本所需要的知识，译者假设读者已经熟悉了如何使用 EasyX 并且已经配置好了相关的环境，本译文不再赘述相关环境的配置等问题。

本文有关的所有代码均可以在 https://github.com/FSMargoo/rtweekend-translate 中获取。

译者

2023/12/8

注：

一、Peter Shirley 教授是一个十分有意思的人，希望各位可以去他的个人主页看看这位有意思的大牛：https://www.petershirley.com/。

二、译者的邮箱：1683691371@qq.com。

1. 原文前文

我已经教授有关光线追踪的图形学课程很多年了，由于你会被要求去独立完成所有的代码，并且你可以不依靠任何 API[1] 就可以得到一些炫酷的光线追踪渲染图，于是我决定将我的课程笔记改编成一个能教你怎么去快速编写生成炫酷的渲染图的教程；当然，可以肯定的是你将不会写出一个具有一个完整功能的光线追踪渲染器，但起码这个渲染器有一个和电影一样炫酷的间接光照；并且，跟着书中的步骤，你自己所写出的一个光线追踪渲染器的基本框架将会让你兴奋地有足够的动力去完善它，并最终得到一个完整的光线追踪渲染器。

当有一个人提到"光线追踪"时，其可能代表很多东西；我将介绍的是一个被称作路线追踪[2]的技术，并且不会讨论过深；尽管代码十分简单，但我相信你肯对会对自己生成的渲染图而感到高兴的。

我将会带着你按照我的顺序来写一个光线追踪渲染器并且附带上一些调试技巧；在最后你将会写出一个能够生成一些炫酷渲染图的光线追踪渲染器，你肯定可以在一个周末内做完这些工作[3]，如果你用了更久的时间也不要紧，我是用 C++ 作为本书代码编写语言，但你没有必要，但是鉴于 C++ 的速度，兼容性，以及大部分渲染器都是用它写的，我依然建议你使用 C++，值得注意的是，我避免了大部分 C++ 的"现代特性"，但是继承和符号重载对一个光线追踪渲染器来说太过于好用以至于不可忽略。

"我不提供任何代码，但是这些代码都是真实的[4]，并且我会在书中展示一个个片段，

---

[1] 译者注：此处指的并不是广义上的 API，而特指已经封装好了光线追踪相关操作的 API。

[2] 译者注：此处英文原文为"Path Tracer"。

[3] 译者注：此处为 Ray Tracing in One Weekend 的前文，故作者说"可以在一个周末内做完这些工作"。

[4] 译者注：作者的意思应该是这些代码不是他随手对着编辑器随手敲敲的，而是他真的有去写代码，不提供代码的理由会在后文给出，并且作者最后还是给出了所有代码，在本译本中我将会提供译者所有使用 EasyX 实现的代码。

除非都是些显而易见的符号重载操作；我始终相信自己编写代码是学习中不可缺少的一环，但是我只会在代码不可用的时候才去实践我的这个信念；所以请不要再问我有关这方面的事情啦!”

作者后期补充：由于我觉得我一百八十度大转弯的态度很好笑所有我还是保留了这一发言，我发现一些读者所产生的问题都通过对比代码后得到了解决，因此公开源码还是有必要的，但是还是请你一定要自己去写代码，顺带一提本书的所有完整代码都可以在 https://github.com/RayTracing/raytracing.github.io/ 中找到。

这里再做一些关于本书实现代码的说明，代码理念优先考虑一下几点：

- 代码必须实现书中所提到的所有概念。
- 尽管使用 C++，但是为了尽可能地简单，代码风格和 C 代码非常相像，但是依然为了更简单的使用和理解而选择采用一些特性。
- 为了连续性，代码应该和原著的风格保持一直。
- 为了保持代码和书中所列出的代码一直，每行代码应该控制在 96 个字符以内。

综上所述，我所给出的代码给出了一个刚过及格线的实现，而至于剩下的优化，就留给读者自行去享受了。代码可以优化和现代化的地方太多了；只选取最简单的解决方案。

在本文中，假设你已经对一些线性代数内容（例如向量，点乘，向量加法）熟悉，如果你并不清楚这些内容，不如先做点复习，或者先从头开始学起，线上的学习方式，我推荐 Morgan McGuire 的 Graphics Codex（https://graphicscodex.com/），如果你想要阅读书本的话，我推荐 Peter Shirley 和 Steve Marschner 的 Fundamentals of Computer Graphics 或者是 J.D. Foley 和 Adny Van Dam 的 Fundamentals of Interactive Computer Graphics。

Peter 维护一个和本书有关的网站：https://in1weekend.blogspot.com/，可以提供一些进

一步阅读的资源链接。

如果你想与我们取得联系，请随意给我们发送邮箱：

- Peter Shirley：ptrshrl@gmail.com
- Steve Hollasch：steve@hollasch.net
- Trevor David Black：trevordblack@trevord.black

最后，如果你的实现遇到了一些问题，或者想要分享你的一些想法或者是成果，欢迎来到 GitHub 的讨论区（https://github.com/RayTracing/raytracing.github.io/discussions/）。

感谢所有在这个项目中帮助了我的人，你可以在本书最后的致谢部分找到他们的名字。

让我们开始吧。

2. 输出一张图片[5]

2.1 PPM 图片格式

不论你打算怎么实现一个渲染器，你总需要一个途径去看到你的渲染结果吧，最直接的方法就是把你的图片保存为一个文件，可问题是，那么多的图片格式，而且其中的很多格式都很复杂，所以我总是一开始选择 PPM（Portable Pixmap Format） 格式，一个直接使用普通文字的格式，下面是来自 Wikipedia 有关 PPM 格式的介绍[6]:

此处是一个将 RGB 图像存储在 PPM 格式中的例子，每行的末尾都有一个换行符:

```
1.  P3
2.  # P3 代表所有的颜色都是用 ASCII 表示的，接着表示散列两行
3.  # 255 表示最大值，接着是 RGB 三元组
4.
5.  3 2
6.  255
7.  255   0   0    0   255    0   0   0   255
8.  255 255   0  255   255  255   0   0     0
```

2.2 在 EasyX 中输出图片

与平时在 EasyX 中输出图片使用 loadimage 以及 putimage 不同的是，此处将通过直接读写屏幕缓冲区的方法来输出图片，具体的来说，就是通过 GetImageBuffer 函数获得屏幕缓冲区的指针，然后直接改写指针所指向内存的值，当然，最好还是提供一个设备相关的操作类来方便读写缓冲区（没有人会乐意对着一个指针如同闭眼开车一样操作的）。

类 Device 可以用于创建一个指定长宽的窗口，并且可以读写缓冲区，代码如下:

---

[5] 译者注：原文中介绍的是如何使用 C++ 创建一个 PPM 文件，但由于本译文需要将代码改写成 EasyX 版本的代码，所以本章有关图像显示的代码都会被改写成 EasyX 版本的代码，并且对本章的内容进行了部分调整。

[6] 译者注：原百科截图为英文，此处译者翻译成了中文。

```cpp
/**
 * \file device.h
 * \brief 封装了有关绘图设备相关的操作
 */

#pragma once

#include <graphics.h>
#include <iostream>

/**
 * \brief 绘图设备的封装类
 */
class Device {
public:
    Device(const int& WWidth, const int& WHeight)
        : Width(WWidth), Height(WHeight), MaximumPixel(Width * Height) {
        Handle = initgraph(Width, Height);
        if (Handle != nullptr) {
            Buffer = GetImageBuffer();

            BeginBatchDraw();
        } else {
            std::clog << "\rDevice Creating Failure : No widget has been
created."
                      << std::flush;

            exit(-1);
        }
    }

public:
    DWORD& At(const size_t& X, const size_t& Y) const {
        if (Y * Height + X >= MaximumPixel) {
            std::clog << "\rDevice Failure : Subscript out of range." <<
std::flush;
        }

        return Buffer[Y * Width + X];
    }

    static void Flush() {
        FlushBatchDraw();
    }
```

```
private:
    HWND   Handle;
    DWORD* Buffer;
    int    Width;
    int    Height;
    int    MaximumPixel;
};
```

代码 1：Device 类的代码[device.h]

于是现在可以使用 Device 类来创建第一个图像了：

```
#define UNICODE
#define _UNICODE

#include <conio.h>
#include <device.h>

int main() {
    constexpr int Width  = 640;
    constexpr int Height = 480;
    const Device  Graphics(Width, Height);

    for (size_t Y = 0; Y < Height; ++Y) {
        const double G = static_cast<double>(Y) / (Height - 1) * 255.999;
        for (size_t X = 0; X < Width; ++X) {
            const double R = static_cast<double>(X) / (Width - 1) * 255.999;

            Graphics.At(X, Y) = BGR(RGB(R, G, 0));
        }
    }

    Device::Flush();

    _getch();

    return 0;
}
```

代码 2：第一个图片

上面的代码会在屏幕中会渲染出一个由黑色渐变为黄色的渐变图片，具体效果如图：

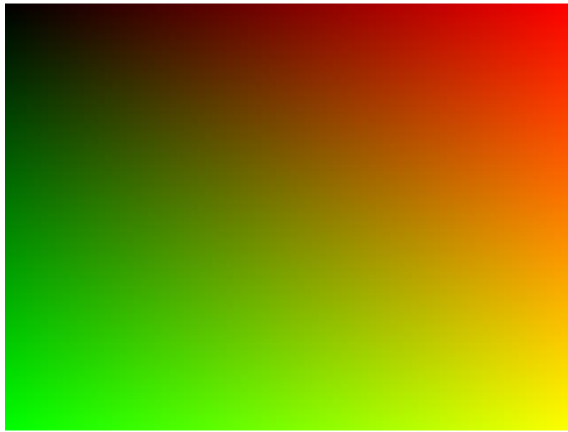

图 1：第一个图片

2.3 添加一个进度条

在接着开始之前，不妨先给程序添加上一个进度条，方便跟踪长时间渲染，并且有助于识别无限循环或者是其他可能会导致程序停摆的问题。

程序使用标准输出流（std::cout）来输出图像，所以不妨让使用日志输出流来取代标准输出流（std::clog）[7]：

```
for (size_t Y = 0; Y < Height; ++Y) {
    std::clog << "\r 还剩下: " << (Height - Y) << "条扫描线待渲染。 " <<
std::flush;
    const double G = static_cast<double>(Y) / (Height - 1) * 255.999;
    for (size_t X = 0; X < Width; ++X) {
        const double R = static_cast<double>(X) / (Width - 1) * 255.999;

        Graphics.At(X, Y) = BGR(RGB(R, G, 0));
    }
}

std::clog << "\r 渲染完毕。             \n";
```

代码 3：有渲染进度条的渲染循环[main.cpp]

[7] 译者注：原文中使用的是标准输出流来输出 PPM 格式的图像，但是文中由于需要使用 EasyX 改写代码并没有使用标准输出流，此处读者可以更换成标准输出流（std::cout），但译文为了上下文统一性依旧选择使用日志输出流（std::clog）

现在再运行代码，你将会看到一个能够提示剩余渲染线的进度条，希望这个进度条能快到你看都看不清；但别担心，在接下来一步一步扩充光线追踪渲染器以后你将会看到一个比现在慢得多的渲染进度条[8]。

3. Vec3 类

几乎所有的图形学程序都有一些类去存储向量和颜色；在许多的系统中这些向量是 4D 的（其表示 3D 的坐标附带上一个齐次坐标ω，或者表示 RGB 加上一个 Alpha 透明度值）；但对场景而言，只是用 3D 就足够了；使用一个类去表示颜色、坐标、方向、位移量等等；有些人并不喜欢这样做是因为这将不能阻止你去做一些傻事，例如用一个坐标减去一个颜色；这当然是个不错的观点，但总是在总体路线不错的情况下追求“极简化的代码”，尽管如此，依然选择定义两个别名 point3、color 给 vec3，尽管他们只是一个别名，并且依然没有任何阻止你去干蠢事的方法，但起码会让代码更具有可读性。

在头文件的上半部分定义 Vec3 类，而在下半部分则定义一些有用的向量操作函数：

```
/**
 * \file vec3.h
 * \brief 定义了向量有关的操作
 */

#pragma once

#include <cmath>
#include <iostream>

class Vec3 {
public:
    Vec3() : e{0, 0, 0} {
```

[8] 译者注：译者使用的环境为 CLion ; Nova（Preview），其 IDE 中自带了一个控制台以供输出，若读者使用了其他未自带控制台的 IDE 可能无法看到控制台输出（这是因为 EasyX 默认会关闭控制台），若读者有需要，可以在 Device 类中将 initgraph 改写为“initgraph(Width, Height, EW_SHOWCONSOLE)”。

```
    }
    Vec3(const double& e0, const double& e1, const double& e2) : e{e0, e1,
e2} {
    }

public:
    double X() const {
        return e[0];
    }
    double Y() const {
        return e[1];
    }
    double Z() const {
        return e[2];
    }

public:
    Vec3 operator-() const {
        return Vec3{-e[0], -e[1], -e[2]};
    }
    double operator[](const int& Index) const {
        return e[Index];
    }
    double& operator[](const int& Index) {
        return e[Index];
    }

    Vec3& operator+=(const Vec3& Vector) {
        e[0] += Vector.e[0];
        e[1] += Vector.e[1];
        e[2] += Vector.e[2];

        return *this;
    }
    Vec3& operator*=(const Vec3& Vector) {
        e[0] *= Vector.e[0];
        e[1] *= Vector.e[1];
        e[2] *= Vector.e[2];

        return *this;
    }
    Vec3& operator*=(const double& Value) {
        e[0] *= Value;
        e[1] *= Value;
```

```
        e[2] *= Value;

        return *this;
    }
    Vec3& operator/=(const double& Value) {
        return *this *= 1 / Value;
    }

public:
    /**
     * \brief 计算向量模长
     */
    double Length() const {
        return sqrt(LengthSquared());
    }
    /**
     * \brief 计算向量模长（未开方）
     */
    double LengthSquared() const {
        return std::pow(e[0], 2) + std::pow(e[1], 2) + std::pow(e[2], 2);
    }

public:
    double e[3];
};

/**
 * Point3 只是一个 Vec3 的别名，但可以提高代码可读性
 */
using Point3 = Vec3;

/**
* 向量操作有关函数
*/

inline std::ostream& operator<<(std::ostream& Output, const Vec3& Vector) {
    return Output << Vector.e[0] << ' ' << Vector.e[1] << ' ' << Vector.e[2];
}
inline Vec3 operator+(const Vec3& Left, const Vec3& Right) {
    return Vec3{ Left.e[0] + Right.e[0], Left.e[1] + Right.e[1], Left.e[2] +
Right.e[2] };
}
inline Vec3 operator-(const Vec3& Left, const Vec3& Right) {
```

```
    return Vec3{ Left.e[0] - Right.e[0], Left.e[1] - Right.e[1], Left.e[2] -
Right.e[2] };
}
inline Vec3 operator*(const Vec3& Left, const Vec3& Right) {
    return Vec3{ Left.e[0] * Right.e[0], Left.e[1] * Right.e[1], Left.e[2] *
Right.e[2] };
}
inline Vec3 operator*(const double& Value, const Vec3& Vector) {
    return Vec3{ Value * Vector.e[0], Value * Vector.e[1], Value *
Vector.e[2] };
}
inline Vec3 operator/(const Vec3& Left, const double& Value) {
    return (1 / Value) * Left;
}
/**
 * 向量点乘操作
 */
inline double Dot(const Vec3& w, const Vec3& v) {
    return w.e[0] * v.e[0] + w.e[1] * v.e[1] + w.e[2] * v.e[2];
}
/**
 * 向量叉乘操作
 */
inline Vec3 Cross(const Vec3& w, const Vec3& v) {
    return Vec3{ w.e[1] * v.e[2] - w.e[2] * v.e[1],
               w.e[2] * v.e[0] - w.e[0] * v.e[2],
               w.e[0] * v.e[1] - w.e[1] * v.e[0] };
}
/**
 * 归一化向量
 */
inline Vec3 UnitVector(const Vec3& Vector) {
    return Vector / Vector.Length();
}
```

代码 4：类 Vec3 的定义[vec3.h]

此处使用了 double 类型，但是一些渲染器使用了 float 类型，其拥有更高的精度和范围，但是其是 double 类型大小的两倍；大小的增加可能会显得非常重要如果你在一个被限制了内存的环境下编写代码（例如硬件着色器）；但不论是 double 还是 float 都是可以的，可以根据读者个人喜好调整。

利用新定义的 Vec3 类型，可以创建一个新的表示颜色的类 Color，只需要简单地定义一个 Vec3 的别名 Color，并且定义相关的颜色操作函数：

```
/**
 * \file color.h
 * \brief 定义颜色
 */

#pragma once

#include <vec3.h>
#include <graphics.h>

using Color = Vec3;

/**
 * 将颜色转换为对应的内存 DWORD 值
 */
DWORD ColorToDWORD(const Color& Value) {
    return BGR(RGB(Value.X(), Value.Y(), Value.Z()));
}
```

代码 5：类 Color 的定义[color.h]

用上这两个类，现在可以修改原本的渲染循环如下：

```
#define UNICODE
#define _UNICODE

#include <device.h>
#include <conio.h>
#include <color.h>
#include <vec3.h>

int main() {
    constexpr int Width  = 640;
    constexpr int Height = 480;
    const Device  Graphics(Width, Height);
```

[9] 译者注：原文中作者由于使用了 PPM 图片格式，需要特别地定义一些有关颜色操作的函数，由于此处代码使用了 EasyX，故不需要采用原文中的方法，因此译者对本节内容做了调整。

```
    Color WriteColor;

    for (size_t Y = 0; Y < Height; ++Y) {
        std::clog << "\r还剩下: " << (Height - Y) << "条扫描线待渲染。 " <<
std::flush;
        WriteColor.e[1] = static_cast<double>(Y) / (Height - 1) * 255.999;
        for (size_t X = 0; X < Width; ++X) {
            WriteColor.e[0] = static_cast<double>(X) / (Width - 1) * 255.999;

            Graphics.At(X, Y) = ColorToDWORD(WriteColor);
        }
    }

    std::clog << "\r渲染完毕。           \n";

    Device::Flush();

    _getch();

    return 0;
}
```

代码 6：输出图像的最终代码

并且你应该获得一个和原来一样的图像。

4. 光线、一个简单的相机还有背景

4.1 Ray 类

一个 Ray 类，和计算光线一路上应该是什么颜色的实现，是所有光线追踪渲染器都有的东西，现在让将光线考虑成一个函数 $\vec{P}(t)=\vec{A}+t\vec{b}$，其中 $\vec{P}$ 表示一个沿着 3D 直线的线段，$\vec{A}$ 代表光线，而 $\vec{b}$ 代表光线的方向，t 是一个实数（在代码中，其为 double 类型）；插入不同的 t 值将会使线段 $\vec{P}$ 沿着直线移动一段距离；如果插入一个负值的 t，你将可以取到该 3D 直线的任何地方，而对于正值的 t，你将只能得到 $\vec{A}$ 前面一部分，这就是常说的半线或者是射线。

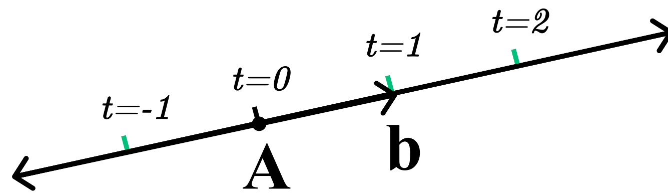

图 2：线性插值

可以将这些想法表示为一个 Ray 类，而函数 $\vec{P}(t)$ 则表示为一个成员函数称作

Ray::at(t)：

```
/**
 * \file ray.h
 * \brief 定义了光线相关的操作
 */

#pragma once

#include <vec3.h>

class Ray {
public:
    Ray() = default;
    Ray(const Point3& Orig, const Vec3& Direct) : Origin(Orig),
Direction(Direct) {  }

public:
    const Point3 OriginPoint() const {
        return Origin;
    }
    const Point3 RayDirection() const {
        return Direction;
    }
    Point3 At(const double& Value) const {
        return Origin + Value * Direction;
    }

private:
    Point3 Origin;
    Vec3   Direction;
};
```

代码 7：Ray 类的定义[ray.h]

现在已经准备好去制作一个渲染器了；作为渲染器的核心功能即通过每个像素点发射射线并且计算每个像素点的颜色；基本步骤如下：

1. 计算从“眼睛”穿过像素的光线。
2. 确定那些物体将会和光线相交。
3. 计算最近交叉点的颜色。

最开始开发一个路径追踪渲染器，为了先把代码跑起来，我总是先做一个非常简单的相机。

我总是因为在方形图像中颠倒 x 和 y 过于频繁而导致陷入一些调试困难，所以将会使用一个非方形图像；因为一个方形图像的长宽相等，所以其纵横比是 1:1，想要一个非方形图像，所以选择一个常见的纵横比—16:9，一个 16:9 的纵横比意味着这个图像的宽和高的比例为 16:9；换而言之，如果给定一个横纵比为 16:9 的图像，那么：

$$\frac{\text{width}}{\text{height}}=\frac{16}{9}\approx 1.7778$$

再举一个更实际的例子，一个宽 800 像素高 400 像素的图像的横纵比为 2:1。

一个图像的横纵比可以根据其宽和高来确定，但是，由于只考虑横纵比，所以给定图形的宽和横纵比去计算其高度会显得更容易一点；于是乎可以通过修改图像的宽来缩放图像，并且不用担心这个操作会破坏横纵比，只需要去确保当求解出图像的高度时，其结果起码大于 1。

除了设置渲染图片的尺寸以外，也需要设置一个虚拟的*视口*（*viewport*）来传递场景的光线；视口是一个在 3D 光线中包含了图片像素网格的虚拟矩形；如果像素的水平间距和垂直间距相同，那么一个包含了该像素网格的视口也会有和渲染图像一致的横纵比；两个

相邻的像素之间的距离称作像素间距，以长方形像素为标准。

作为开始，随便选择一个视口高度 2.0，并缩放视口的宽度以获得所需要的横纵比；相关的代码片段大概如下：

```
/*
 * 横纵比
 */
auto AspectRatio = 16.0 / 9.0;
int  Width       = 400;

/*
 * 求解图像高度，并确保其起码大于 1
 */
int Height = static_cast<int>(Width / AspectRatio);
Height     = (Height < 1) ? 1 : Height;

/*
 * 视口的宽可以小于 1
 */
auto          ViewportHeight = 2.0;
auto          ViewportWidth  = ViewportHeight * (static_cast<double>(Width) / Height);
```

代码 8：渲染图设置

如果你在思考为什么不直接使用 AspectRatio 来计算 ViewportWidth，这是因为 AspectRatio 的值其实是理想比例值，其可能不是 Width 和 Height 的实际比值；如果 Height 可以是实值（不仅仅是整数），那么使用 AspectRatio 是完全可行的，但 Width 和 Height 的实际比例可能会被两个因素影响——首先，IntegerHeight 是向下取整后最接近的整数，其可能增加比例值，第二，由于并不允许 IntegerHeight 小于 1，这也可能会导致实际比例的变化。

注意 AspectRatio 只是一个理想比例，只是用基于整数的图像宽高比例尽可能地逼近；为了让视口比例和图像比例完全匹配，使用计算出的图像长宽比来确定最终的视口宽度。

接下来将会定义一个摄像机中心：一个位于 3D 空间中发射所有光线的点（这也通常被成为*眼点——eye point*）。从摄像机中心到视口的向量将会与视口正交。将最一开始视口和相机中心点之间的距离设为一个单位，这个距离通常被称为*焦距*（*focal length*）。

最开始先简单地将摄像机中心设置位于$(0,0,0)$，还将 y 轴向上、 x 轴向右、负 z 轴指向观察方向（这通常被称为右手坐标系[10]——*right-handed coordinates*）。

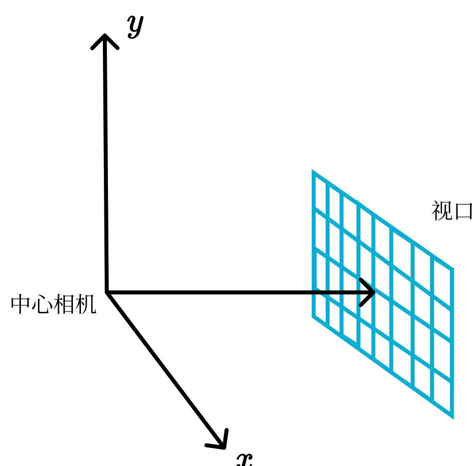

图 3：相机的几何特征

现在就是不可避免地棘手部分了，虽然 3D 空间具有上的规定，但这与图像坐标相冲突，图像的坐标系是左上角为原点，然后一直向下到图像右下角的最后一个像素；这意味着图像坐标 y 轴是反转的： y 沿着图像递增。

当扫描图像时，总会从最左边的像素$(0,0)$开始，然后从左到右逐个扫描，然后再从上到下逐行扫描，为了辅助检索视口中的像素网格，使用一个从左边缘到右边缘的向量$\overrightarrow{V_u}$，和一个从上边缘到下边缘的向量$\overrightarrow{V_v}$。

像素网格将由视口边缘缩进半个像素间距。这样，视口区域就被均匀地划分多个长×宽的区域，像素网格和视口将会长成这样：

---

[10] 译者注：右手坐标系是在空间中规定直角坐标系的方法之一。此坐标系中 x 轴，y 轴和 z 轴的正方向是如下规定的：把右手放在原点的位置，使大拇指，食指和中指互成直角，把大拇指指向 x 轴的正方向，食指指向 y 轴的正方向时，中指所指的方向就是 z 轴的正方向。

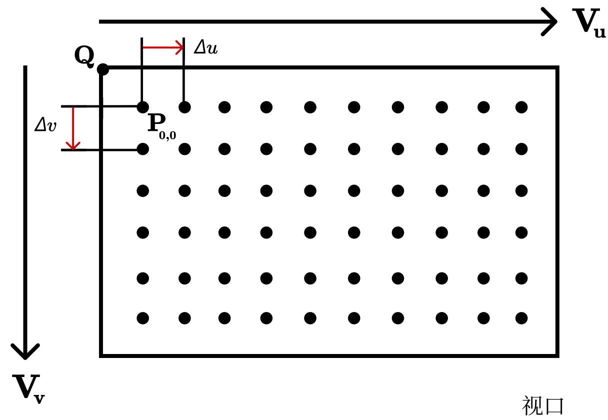

图 4：视口和像素网格

本图展示了一个视口和分辨率为 10×6 的像素网格，该视口的左上角 $\vec{Q}$、像素点 $\overrightarrow{P_{0,0}}$ 以及视口向量 $\vec{V_u}$ 和 $\vec{V_v}$，还有他们的像素增量 $\overrightarrow{\Delta u}$ 和 $\overrightarrow{\Delta v}$。

综上所述，如下是摄像机的代码实现；我们定义一个函数 RayColor(const Ray& Light) 返回一个给定场景光线的颜色，但是现在我们先让他默认返回黑色：

```
#define UNICODE
#define _UNICODE

#include <color.h>
#include <conio.h>
#include <device.h>
#include <ray.h>
#include <vec3.h>

Color RayColor(const Ray& Light) {
    return Color{ 0, 0, 0 };
}

int main() {
    // 图像部分

    const auto AspectRatio = 16.0 / 9.0;
    const auto ImageWidth  = 640;

    // 计算图像的高，并且确保其起码为一

    auto ImageHeight = static_cast<int>(ImageWidth / AspectRatio);
```

```
    ImageHeight      = (ImageHeight < 1) ? 1 : ImageHeight;

    // 相机部分

    const auto FocalLength    = 1.0;
    const auto ViewportHeight = 2.0;
    const auto ViewportWidth = ViewportHeight *
(static_cast<double>(ImageWidth) / ImageHeight);
    const auto CameraCenter  = Point3(0, 0, 0);

    // 计算横纵向量 u 和 v

    const auto ViewportU = Vec3(ViewportWidth, 0, 0);
    const auto ViewportV = Vec3(0, -ViewportHeight, 0);

    // 由像素间距计算横纵增量向量

    const auto PixelDeltaU = ViewportU / ImageWidth;
    const auto PixelDeltaV = ViewportV / ImageHeight;

    // 计算左上角像素的位置

    const auto ViewportUpperLeft =
        CameraCenter - Vec3(0, 0, FocalLength) - ViewportU / 2 - ViewportV /
2;
    const auto  Pixel100Loc = ViewportUpperLeft + 0.5 * (PixelDeltaU +
PixelDeltaV);
    const Device Graphics(ImageWidth, ImageHeight);

    // 渲染部分

    Color WriteColor;

    for (size_t Y = 0; Y < ImageHeight; ++Y) {
        std::clog << "\r还剩下: " << (ImageHeight - Y) << "条扫描线待渲染。 " <<
std::flush;
        const auto HeightVec = static_cast<double>(Y) * PixelDeltaV;
        for (size_t X = 0; X < ImageWidth; ++X) {
            auto PixelCenter = Pixel100Loc + (static_cast<double>(X) *
PixelDeltaU) + HeightVec;
            const auto RayDirection = PixelCenter - CameraCenter;
            Ray        Light(CameraCenter, RayDirection);

            WriteColor = RayColor(Light);
```

```
            Graphics.At(X, Y) = ColorToDWORD(WriteColor);
        }
    }

    std::clog << "\r渲染完毕。           \n";

    Device::Flush();

    _getch();

    return 0;
}
```

代码 9：创建场景光线[main.cpp]

注意上面的代码，我并没有把 RayDirection 进行归一化操作，这是因为我认为不去这么做可以更简单而且写代码能稍微快点。

现在来补充 RayColor 函数来实现一个简单的渐变；将光线方向向量归一化后，该函数将根据 y 坐标线性插值生成一个白色到蓝色的渐变（因此 $y \in (-1.0, 1.0)$），在两者之间求得一个过度值便就构成了“线性插值”或“线性混合”，这通常成为两个值之间的线性插值运算（*lerp*），通常一个线性插值运算是这样组成的，其中 a 将会从 0 到 1 线性增加：

$$Result = (1-a) \cdot Value_{start} + a \cdot Value_{end}$$ [11]

将所有得结合起来，就可以修改 RayColor 函数如下：

```
Color RayColor(const Ray& Light) {
    Vec3 UnitDirection = UnitVector(Light.RayDirection());
    const auto a = 0.5 * (UnitDirection.Y() + 1.0);
    auto Result = (1.0 - a) * Color(1.0, 1.0, 1.0) + a * Color(0.5, 0.7, 
1.0);
    Result *= 255.0;

    return Result;
}
```

代码 10：修改过后的 RayColor 函数

[11] 译者注：该公式的中文形式是：$插值结果 = (1-a) \cdot 起始值 + a \cdot 终止值$。

如果代码无误，最终将会生成如下图片：

图 5：根据 y 坐标插值得到的蓝白渐变

5. 添加一个球体

让我们添加一个对象到我们的路径追踪渲染器中；人们通常选择球体因为其与光线相交得计算相对简单。

5.1 光线与球体的相交

一个位于原点、半径为 r 的球体方程是一个很重要的数学方程：

$$x^2+y^2+z^2=r^2$$

由此可知，如果给定一个点 (x,y,z)，若 $x^2+y^2+z^2=r^2$，那么该点位于圆上，若 $x^2+y^2+z^2<r^2$，那么该点位于圆内，若 $x^2+y^2+z^2>r^2$，则该点位于圆外。

如果我们希望球体能够位于任意中心点 (C_x,C_y,C_z)，则方程可以变成下面的形式：

$$(x-C_x)^2+(y-C_y)^2+(z-C_z)^2=r^2$$

在图形学中，总是希望公式能以向量表示，这样所有类似 (x,y,z) 这样的东西都可以被 Vec3 类来表示；你可能已经注意到了一个由中心 $\vec{C}=(C_x,C_y,C_z)$ 到点 $\vec{P}=(x,y,z)$ 的向量可以表示为 $(\vec{P}-\vec{C})$，于是利用向量点乘的定义有：

$$\left(\vec{P}-\vec{C}\right)\cdot\left(\vec{P}-\vec{C}\right)=\left(x-C_x\right)^2+\left(y-C_y\right)^2+\left(z-C_z\right)^2$$

故可以用向量重写上文的球体方程:

$$\left(\vec{P}-\vec{C}\right)\cdot\left(\vec{P}-\vec{C}\right)=r^2$$

该式可以读作:“任何满足该等式的点$\vec{P}$都在球体上”; 欲求光线$\vec{P}(t)=\vec{A}+t\vec{b}$是否与球体相交, 先考虑若其与球体相交, 则因存在一个t使得$\vec{P}(t)$满足该等式, 于是可先求得t:

$$\left(\vec{P}(t)-\vec{C}\right)\cdot\left(\vec{P}(t)-\vec{C}\right)=r^2$$

代入$\vec{P}(t)$展开得:

$$\left[\left(\vec{A}+t\vec{b}\right)-\vec{C}\right]\cdot\left[\left(\vec{A}+t\vec{b}\right)-\vec{C}\right]=r^2$$

该式中左边有三个向量，右边有三个向量，如果求出完整的点积，那么将会得到九个向量，尽管你确实可以直接写出来这些向量，但其实没有必要那么麻烦；最终目标是解出t，所以应该先讨论t的存在性:

$$\left[t\vec{b}+\left(\vec{A}-\vec{C}\right)\right]\cdot\left[t\vec{b}+\left(\vec{A}-\vec{C}\right)\right]=r^2$$

现根据向量代数的规则来分布点乘如下:

$$t^2\vec{b}\cdot\vec{b}+2t\vec{b}\cdot\left(\vec{A}-\vec{C}\right)+\left(\vec{A}-\vec{C}\right)\cdot\left(\vec{A}-\vec{C}\right)=r^2$$

移项得:

$$t^2\vec{b}\cdot\vec{b}+2t\vec{b}\cdot\left(\vec{A}-\vec{C}\right)+\left(\vec{A}-\vec{C}\right)\cdot\left(\vec{A}-\vec{C}\right)-r^2=0$$

这个方程让人点摸不着头脑，但注意到r和一切向量都是确定且已知的；此外，向量可以通过点乘转换成数量，因此，唯一的未知数只有t，并且注意到式中存在t^2，故这是一个关于t的二次方程，可以非常简单地使用二次函数的求根公式来求出t:

$$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

而对于求根公式中的变量，有:

$$a=\vec{b}\cdot\vec{b}$$

$$b = 2\vec{b} \cdot \left(\vec{A} - \vec{C}\right)$$

$$c = \left(\vec{A} - \vec{C}\right) \cdot \left(\vec{A} - \vec{C}\right) - r^2$$

由此可以解得t，但是求根公式的根号下的式子可以是一个正数（意味着两个实根）、负数（意味着没有实根）、0（意味着只有一个实根）；在图形学中，代数几乎总是与几何有非常直接的联系；对于根的不同存在情况对应下图中三个不同的几何意义：

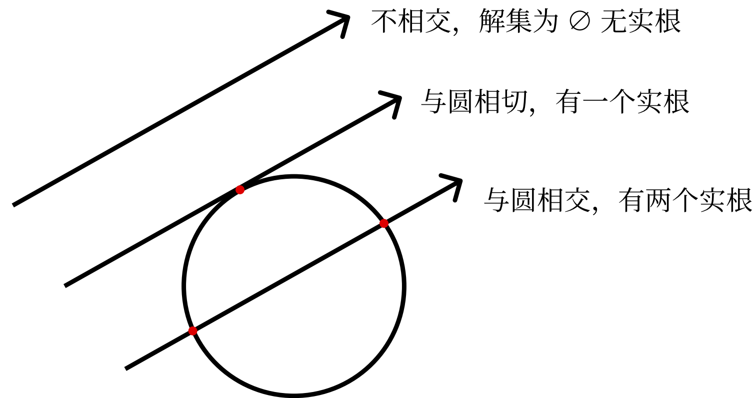

图 6：光线与球体相交的几种可能示意图

5.2 创建第一个光线追踪图片

如果将数学公式硬编码到的程序中，可以通过在 z 轴 -1 处放置一个小球体，然后将与其相交的任何像素着色为红色来测试代码：

```
bool HitSphere(const Point3& Center, double Radius, const Ray& Light) {
    Vec3       Origin = Light.OriginPoint() - Center;
    const auto a      = Dot(Light.RayDirection(), Light.RayDirection());
    const auto b      = 2.0 * Dot(Origin, Light.RayDirection());
    const auto c      = Dot(Origin, Origin) - Radius * Radius;
    const auto Delta  = b * b - 4 * a * c;

    return (Delta >= 0);
}

Color RayColor(const Ray& Light) {
    if (HitSphere(Point3(0, 0, -1), 0.5, Light)) {
        auto Result = Color(1, 0, 0);
```

```
        // 注意 EasyX 中的颜色采用 [0, 255] 的 RGB 制，应把原 [0, 1] 表示颜色的方法映
射到 [0, 255] 上
        Result *= 255;

        return Result;
    }

    Vec3       UnitDirection = UnitVector(Light.RayDirection());
    const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
    auto       Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);
    Result *= 255.0;

    return Result;
}
```

代码 11：渲染一个红色球体[main.cpp]

如果代码无误，那么理应得到如下的图像：

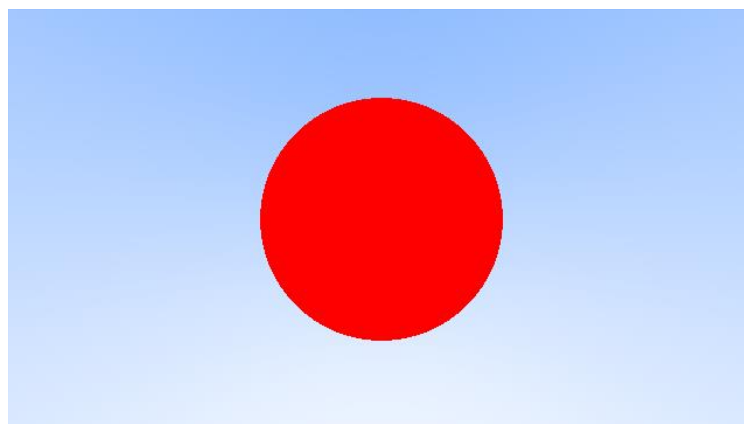

图 7：一个简单的红色球体

现在这个球体还缺少很多东西，例如着色、反光，以及多个对象，但是没关系，现在已经比一开始要更接近一半了；值得注意的是代码中通过求解二次方程的根存在情况来求解光线是否与球体相交，但是为负值的解 t 却也能正常工作，如果将球体的中心点调整至 $z=1$ 将会得到一个完全相同的图像，这是因为这么做不能区分相机前的物体和相机后的物体；这可不是一个特性，下一章中就会将这个问题解决。

6. 表面法线与多个对象

6.1 表面法线着色

首先，先求得一个能够帮助着色的表面法线；表面法线是一个垂直于交点处的向量。

法向量是否具有任意长度，或者是否归一化为单位长度是在程序中设计法向量的关键。

非必要情况下，放弃由于平方根操作而带来高昂运算成本的向量归一化操作十分诱人；然而，在实践中，有三个需要注意的地方：

1) 如果确实需要一个归一化的法向量，你可能只需要计算一次归一化的法向量，而不是每次都在需要的地方再次计算。
2) 在一些地方，的确需要归一化的法向量。
3) 如果需要一个法向量作为单位长度，那么通常可以通过了解特定的几何形状，在其构造函数或是 hit 函数中生成一个有效的向量。

对于第三点，举个例子，球体的法线可以简单地有单位长度除以球体半径求得，从而完全避免平方根操作。

综上所述，最终采用所有法向量均为单位长度的策略。

考虑一个球体，外法线的方向是光线命中点减去中心的方向：

图 8：球体法向量几何示意图

在地球上，这意味着从地心到你的向量是竖直向上的；接下来在回归代码，为小球添加着色；目前代码中并没有任何光线或者其他东西，所以先简单的将法向量可视化为颜色图；一个常用的可视化法向量的方法（假设 $\vec{n}$ 是单位向量很容易且直观其取值位于 -1 到 1 间）是将每个分量映射到 $[0,1]$ 上，由此将 (x,y,z) 映射到 (r,g,b) 中；正常情况下，不

仅需要判断光线是否与球体相交，还需要具体的相交的点（这是目前正在计算的所有内容）；由于场景中只有一个球体，并且其直接位于相机前，所以目前为止还不需要担心负值 t 的问题；只要假设最近的命中点（t 的最小值）就是所需要的命中点，于是可以修改代码如下并可视化 $\vec{n}$：

```
double HitSphere(const Point3& Center, double Radius, const Ray& Light) {
    Vec3       Origin = Light.OriginPoint() - Center;
    const auto a      = Dot(Light.RayDirection(), Light.RayDirection());
    const auto b      = 2.0 * Dot(Origin, Light.RayDirection());
    const auto c      = Dot(Origin, Origin) - Radius * Radius;
    const auto Delta  = b * b - 4 * a * c;

    if (Delta < 0) {
        return -1.0;
    } else {
        return (-b - sqrt(Delta)) / (2.0 * a);
    }
}

Color RayColor(const Ray& Light) {
    const auto t = HitSphere(Point3(0, 0, -1), 0.5, Light);
    // 计算法线
    if (t > 0.0) {
        Vec3 Normal = UnitVector(Light.At(t) - Vec3(0, 0, -1));
        return 255 * 0.5 * Color(Normal.X() + 1, Normal.Y() + 1, Normal.Z() +
1);
    }

    Vec3       UnitDirection = UnitVector(Light.RayDirection());
    const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
    auto       Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);

    return 255.0 * Result;
}
```

代码 12：在球体上渲染表面法线

如果代码正确，那么程序最终会产生这样的图像：

图 9：一个由表面法线着色的球体

6.2 简化球体与光线相交的代码

先来重温光线与球体相交的代码：

```
double HitSphere(const Point3& Center, double Radius, const Ray& Light) {
    Vec3       Origin = Light.OriginPoint() - Center;
    const auto a      = Dot(Light.RayDirection(), Light.RayDirection());
    const auto b      = 2.0 * Dot(Origin, Light.RayDirection());
    const auto c      = Dot(Origin, Origin) - Radius * Radius;
    const auto Delta  = b * b - 4 * a * c;

    if (Delta < 0) {
        return -1.0;
    } else {
        return (-b - sqrt(Delta)) / (2.0 * a);
    }
}
```

代码 13：光线与球体相交的代码[main.cpp]

首先，回想一下，一个向量点乘于自己的结果等于该向量长度的平方。

其次，注意 b 有一个系数 2；考虑当 $b = 2h$ 时，有：

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$= \frac{-2h \pm \sqrt{(2h)^2 - 4ac}}{2a}$$

$$= \frac{-2h \pm 2\sqrt{h^2 - ac}}{2a}$$

$$= \frac{-h \pm \sqrt{h^2 - ac}}{a}$$

于是现在便就可以把光线和球体相交的代码简化为：

```
double HitSphere(const Point3& Center, double Radius, const Ray& Light) {
    Vec3       Origin = Light.OriginPoint() - Center;
    const auto a      = Dot(Light.RayDirection(), Light.RayDirection());
    const auto HalfB  = Dot(Origin, Light.RayDirection());
    const auto c      = Origin.LengthSquared() - Radius * Radius;
    const auto Delta  = HalfB * HalfB - a * c;

    if (Delta < 0) {
        return -1.0;
    } else {
        return (-HalfB - sqrt(Delta)) / a;
    }
}
```

代码 14：修改后的光线球体相交代码

6.3 可命中对象的抽象

现在，如何创建多个球体呢？虽然直接创建一个球体的数组很省事，但是一个简洁的解决方案是为所有光线可能命中的物体创建一个抽象类，然后让并使球体和球体数组都成为可以被击中的东西；如果不是使用面向对象编程，称其为“对象（*object*）”或许更好，然而，人们也会通常使用“表面（*surface*）”，但如果我们需要体积（例如云雾）的话这种叫法就会显得逊色，“可命中的（*hittable*）”将会强调他们的成员函数，在这其中我一个都不喜欢，所以我会选择使用“hittable”这个名字。

类 Hittable 将会有一个接受光线为参数的函数 hit；大部分光线追踪器都会为了方便而参加一个从t_{min}到t_{max}的有效间距，所以只有当$t_{min} < t < t_{max}$时，光线命中才会被计算；对于初始光线而言，t将总会是一个正值，但加入一个有效区间对于代码的实现还是很有益的；但有一个设计问题就是，在物体被命中后是否都要去计算法向量，其实并不用，只有离原点最近的交点才需要去求法向量，而其他的则会被遮盖；这里将提供一个简单的解决方案并且把一堆将要计算的数据存在一个结构体了；代码如下：

```
/**
 * \file Hittable.h
 * \brief 有关求交的类定义
 */

#pragma once

#include <ray.h>

class HitRecord {
public:
    HitRecord() = default;

public:
    Point3 Point;
    Vec3   NormalVector;
    double TValue{};
};

class Hittable {
public:
    virtual ~Hittable()                          = default;
    virtual bool Hit(const Ray& Light, const double& RayTMin, const double&
RayTMax,
                   HitRecord& Record) const = 0;
};
```

代码 15：Hittable 类[hittable.h]

接着是球体的 Sphere 类：

```
/**
```

```
 * \file sphere.h
 * \brief 球体相关定义代码
 */

#pragma once

#include <hittable.h>

class Sphere : public Hittable {
public:
    Sphere(Point3 ICenter, const double& IRadius) : Center(ICenter),
Radius(IRadius) {
    }
    bool Hit(const Ray& Light, const double& RayTMin, const double& RayTMax,
             HitRecord& Record) const override {
        Vec3 Origin = Light.OriginPoint() - Center;

        // 求判别式

        const auto A     = Light.RayDirection().LengthSquared();
        const auto HalfB = Dot(Origin, Light.RayDirection());
        const auto C     = Origin.LengthSquared() - Radius * Radius;
        const auto Delta = HalfB * HalfB - A * C;
        if (Delta < 0) {
            return false;
        }

        const auto SqrtDelta = sqrt(Delta);
        // 这里只采用最靠近原点（即最小的根）
        auto Root = (-HalfB - SqrtDelta) / A;
        // 如果超出范围，则选用另一个根
        if (Root <= RayTMin || RayTMax <= Root) {
            // 如果依然 OOR 则没有交点
            Root = (-HalfB + SqrtDelta) / A;
            if (Root <= RayTMin || RayTMax <= Root) {
                return false;
            }
        }

        // 求解出交点并记录在 Record 中
        Record.TValue       = Root;
        Record.Point        = Light.At(Root);
        Record.NormalVector = (Record.Point - Center) / Radius;
```

```
        return true;
    }

private:
    Point3 Center;
    double Radius;
};
```

代码 16：Sphere 类[sphere.h]

6.4 正反面

第二个设计决策就是，对于法向量，是否应该总是向外，现在所有的求法线的方法只是单纯地将求出中心到交点地方向（这意味着法向量的方向始终向外）；如果光线从外部与球体相交，则法线的方向与光线相反；如果光线从内部射出，则法向量总是向外；或者可以让法线始终与射线方向相反；如果光线在球体外部，则法向量将指向外，但如果光线在球体内部，则法向量向内：

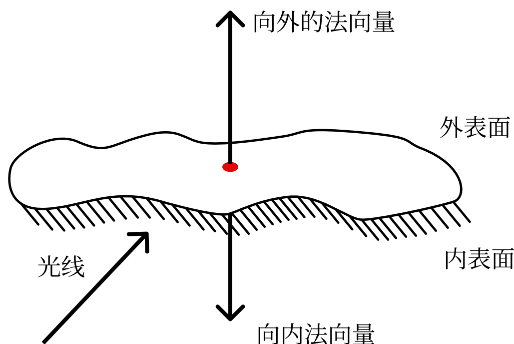

图 10：球体表面法向量的可能方向几何示意图

由于我们最终想要确定光线来自表面的哪一侧，所以最终要选择其中一种可能，这对于每一面渲染不同的对象（例如双面纸张上的文本）或者具有内部外部的对象（例如玻璃球）非常重要。

如果选择让法向量始终向外，那么在着色时，就需要确定光线位于哪一侧；这可以通

过对比光线和法向量来确定，如果光线和法向量方向相同，则光线朝向物体之内，如果光线和法向量方向相反，则光线朝向物体之外；这可以由两个向量的点积决定，如果它们的点积为正，则射线位于球体内部。

```
if (Dot(RayDirection, OutwardNormal) > 0.0) {
    // 光线在球体之内
    ...
} else {
    // 光线在球体之外
    ...
}
```

代码 17：光线和法向量的对比

如果选择让法向量始终指向光线方向，则不需要使用点积去判断光线位于哪一侧；取而代之的是，将会需要储存更多信息：

```
bool FrontFace;
if (Dot(RayDirection, OutwardNormal) > 0.0) {
    // 光线在球体之内
    Normal      = -OutwardNormal;
    FrontFace   = false;
} else {
    // 光线在球体之外
    Normal      = OutwardNormal;
    FrontFace   = true;
}
```

代码 18：存储光线位于何侧

可以通过设置让法向量始终相对于表面向外，或者总是始终指向入射光线；这取决于你是要在几何体求交时还是在着色时确定内外侧，由于在本书中将会介绍比几何类型还多的材质类型，故考虑最为简单的在几何体求交时确定内外侧；这只是一个个人偏好，在别的文章中你也可以看到这两种实现。

接着将 FrontFace 加入 HitRecord 类中，同样地添加一个能够用于求解计算的函数 SetFaceNormal；为了方便起见，假设传递给新 SetFaceNormal 函数的向量是其归一化后的向量；参数总是可以被显式地归一化，在和几何有关的代码这样做会更有效，尤其是当在

特定已知几何图形下时。

```
class HitRecord {
public:
    HitRecord() = default;

    void SetFaceNormal(const Ray& Light, const Vec3& OutwardNormal) {
        // 设置 HitRecord 法向量
        // 注意：参数'OutwardNormal'已经假设拥有单位长度
        FrontFace    = Dot(Light.RayDirection(), OutwardNormal) < 0;
        NormalVector = FrontFace ? OutwardNormal : -OutwardNormal;
    }

public:
    Point3 Point;
    Vec3   NormalVector;
    double TValue{};
    bool   FrontFace;
};
```

代码 19：添加 FrontFace 变量到 HitRecord 中[hittable.h]

于是便可以在 Sphere 类中加入表面侧判断：

```
class Sphere : public Hittable {
public:
    ...
    bool Hit(const Ray& Light, const double& RayTMin, const double& RayTMax,
             HitRecord& Record) const override {
        ...

        // 求解出交点并记录在 Record 中
        Record.TValue      = Root;
        Record.Point       = Light.At(Root);
        Vec3 OutwardNormal = (Record.Point - Center) / Radius;
        Record.SetFaceNormal(Light, OutwardNormal);

        return true;
    }
    ...
};
```

代码 20：拥有了表面侧判断的 Sphere 类[sphere.h]

6.5 创建一个 Hittable 对象的列表

现在已经有了一个名叫 Hittable 的封装了与光线相交相关抽象接口的抽象类；现在还需要加入一个能储存这些 Hittable 对象的列表：

```
/**
 * \file HittableList.h
 * \brief 一个 Hittable 类的列表
 */

#pragma once

#include <hittable.h>

#include <memory>
#include <vector>

class HittableList : public Hittable {
public:
    HittableList() = default;
    explicit HittableList(const std::shared_ptr<Hittable>& Object) {
        Add(Object);
    }

public:
    void Clear() {
        Objects.clear();
    }
    void Add(const std::shared_ptr<Hittable>& Object) {
        Objects.push_back(Object);
    }
    bool Hit(const Ray& Light, const double& RayTMin, const double& RayTMax,
             HitRecord& Record) const override {
        HitRecord Temp;
        auto      HitFlag = false;
        auto      Closet  = RayTMax;

        for (const auto& Object : Objects) {
            if (Object->Hit(Light, RayTMin, Closet, Temp)) {
                HitFlag = true;
                Closet  = Temp.TValue;
                Record  = Temp;
            }
```

```
        }

        return HitFlag;
    }

public:
    std::vector<std::shared_ptr<Hittable>> Objects;
};
```

代码 21：HittableList 类[hittableList.h]

6.6 一些 C++ 新特性介绍

有关 HittableList 类的代码中使用了两个 C++ 特性，如果你不是一个 C++ 使用者，你可能会感到陌生，即 vector 和 shared_ptr。

shared_ptr<Type>是指向某个已分配类型的指针，具有引用计数语义；每次将其值分配给另一个共享指针对象时（通常只是一个简单的赋值），引用计数就会增加一；当共享指针对象离开某个域（例如代码块的结尾或者一个函数）时，引用计数就会减一；当引用计数变为零时，对象就会被安全的释放。

通常，共享指针首先使用新分配的对象进行初始化，例如：

```
shared_ptr<double> DoublePointer = make_shared<double>(0.37);
shared_ptr<Sphere> SpherePointer = make_shared<Sphere>(Point3(0,0,0), 1.0);
```

代码 22：一个使用 shared_ptr 来分配内存的示例

其中 make_shared<Type>(Para …) 分配了一个类型为 Type，使用 Para 作为构造函数参数的实例，并返回一个 shared_ptr<type>。

由于 make_shared<Type>(…) 的返回类型可以被自动推导，故可以使用 C++ 中的 auto 关键字来让代码更简洁：

```
auto DoublePointer = make_shared<double>(0.37);
auto SpherePointer = make_shared<Sphere>(Point3(0,0,0), 1.0);
```

代码 23：一个使用 auto 关键字和 shared_ptr 来分配内存的示例

之所以选用 shared_pointer<Type>，是因为其允许多个几何体共享同一个实例（例如：一堆都使用相同颜色材质的球体），且其可以自动地进行内存管理并且更易于理解实现原理。

std::shared_ptr 被定义在 <memory> 头文件中。

第二个你可能感到陌生的 C++ 特性即为 std::vector，一个通用的很像任意类型数组的集合；上文中便是定义了一个 Hittable 对象的集合；std::vector 随着添加更多值而自动增长：Objects.push_back(Object) 在类型为 std::vector 的成员变量 Objects 的末端添加了一个对象。

最后，using 语句[12]告诉编译器从 std 库中将 shared_ptr 和 make_shared 取出，以便不用每次引用其都需要加上 std:: 前缀。

6.7 常用的数学常量和功能性函数

代码中经常需要一些数学常量，为了方便将其全部定义在一个新的头文件中；目前为止只需要无穷，但也需要在此处定义 π，因为其将会在后期代码中被使用；rtweekend.h 作为主头文件将会定义一些常用的数学常量和功能性函数：

```
/**
 * \file rtweekend.h
 * \brief 常用数学常量和功能性函数定义
 */

#pragma once

#include <cmath>
#include <limits>
```

[12] 译者注：译者个人不提倡在全局域中使用 using 语句，这种行为可能会造成定义域污染；尽管在当前代码环境下无关紧要，但是译者依然移除了有关 using 语句的代码，并使用 std:: 前缀代替，原文中使用了 using 语句做了导出即“using std::shared_ptr;”和“using std::make_shared;”，此处对相关介绍内容不做移除。

```
#include <memory>
#include <numbers>

// 数学常量

constexpr double Infinity = std::numeric_limits<double>::infinity();
constexpr double Pi       = std::numbers::pi;

// 功能性函数

inline double DegreeToRad(const double& Degrees) {
    return Degrees * Pi / 180.f;
}

#include <ray.h>
#include <vec3.h>
```

代码 24：rtweekend.h 通用头文件[rtweekend.h]

以及新的 main 函数：

```
#define UNICODE
#define _UNICODE

#include <rtweekend.h>

#include <color.h>
#include <device.h>
#include <hittable.h>
#include <hittableList.h>
#include <sphere.h>

#include <conio.h>

Color RayColor(const Ray& Light, const Hittable& World) {
    HitRecord Record;
    if (World.Hit(Light, 0, Infinity, Record)) {
        return 255.f * 0.5 * (Record.NormalVector + Color(1, 1, 1));
    }

    Vec3       UnitDirection = UnitVector(Light.RayDirection());
    const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
    auto       Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);
```

```
    return 255.0 * Result;
}

int main() {
    // 图像部分

    const auto AspectRatio = 16.0 / 9.0;
    const auto ImageWidth  = 640;

    // 计算图像的高，并且确保其起码为一

    auto ImageHeight = static_cast<int>(ImageWidth / AspectRatio);
    ImageHeight      = (ImageHeight < 1) ? 1 : ImageHeight;

    // 相机部分

    const auto FocalLength    = 1.0;
    const auto ViewportHeight = 2.0;
    const auto ViewportWidth = ViewportHeight *
(static_cast<double>(ImageWidth) / ImageHeight);
    const auto CameraCenter  = Point3(0, 0, 0);

    // 计算横纵向量 u 和 v

    const auto ViewportU = Vec3(ViewportWidth, 0, 0);
    const auto ViewportV = Vec3(0, -ViewportHeight, 0);

    // 由像素间距计算横纵增量向量

    const auto PixelDeltaU = ViewportU / ImageWidth;
    const auto PixelDeltaV = ViewportV / ImageHeight;

    // 计算左上角像素的位置

    const auto ViewportUpperLeft =
        CameraCenter - Vec3(0, 0, FocalLength) - ViewportU / 2 - ViewportV /
2;
    const auto  Pixel100Loc = ViewportUpperLeft + 0.5 * (PixelDeltaU +
PixelDeltaV);
    const Device Graphics(ImageWidth, ImageHeight);

    // 渲染光线

    HittableList World;
```

```
    auto GroundSphere    = std::make_shared<Sphere>(Point3(0, 0, -1), 0.5);
    auto FakeGround      = std::make_shared<Sphere>(Point3(0, -100.5, -1),
100);

    World.Add(GroundSphere);
    World.Add(FakeGround);

    // 渲染部分

    Color WriteColor;

    for (size_t Y = 0; Y < ImageHeight; ++Y) {
        std::clog << "\r 还剩下: " << (ImageHeight - Y) << "条扫描线待渲染。 " <<
std::flush;
        const auto HeightVec = static_cast<double>(Y) * PixelDeltaV;
        for (size_t X = 0; X < ImageWidth; ++X) {
            auto PixelCenter = Pixel100Loc + (static_cast<double>(X) *
PixelDeltaU) + HeightVec;
            const auto RayDirection = PixelCenter - CameraCenter;
            Ray        Light(CameraCenter, RayDirection);

            WriteColor = RayColor(Light, World);

            Graphics.At(X, Y) = ColorToDWORD(WriteColor);
        }
    }

    std::clog << "\r 渲染完毕。            \n";

    Device::Flush();

    _getch();

    return 0;
}
```

代码 25: 应用了 Hittable 新的 main 函数[main.cpp]

产生的图片实际上只是球体位置及其表面法线的可视化图像；但这是通常检查几何模型的任何缺陷或查看特定特征的好方法。

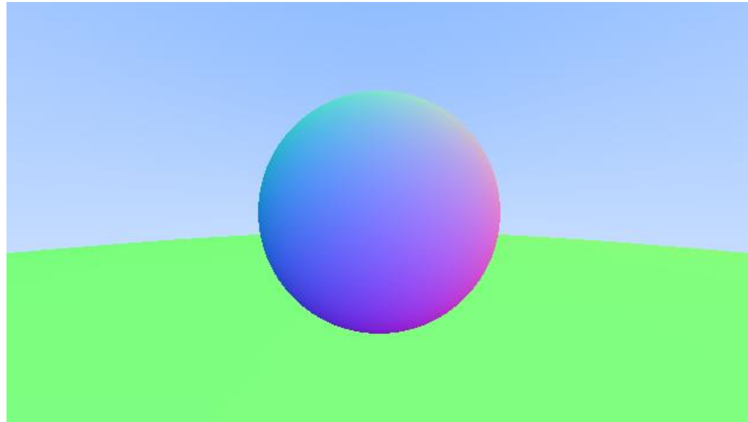

图 11：带有地面的法线着色渲染结果

6.8 区间类

在继续之前，需要先实现一个区间类来管理具有做大最小值的实数区间，这在以后的工作中也会被经常用到：

```
/**
 * \file interval.h
 * \brief 一个区间类
 */

#pragma once

#include <rtweekend.h>

class Interval {
public:
    // 默认一个空区间
    Interval() : Minimum(+Infinity), Maximum(-Infinity) {
    }
    Interval(const double& Min, const double& Max) : Minimum(Min),
Maximum(Max) {
    }

public:
    // 闭区间
    bool Contains(const double& Value) const {
        return Value >= Minimum && Value <= Maximum;
    }
```

```
    // 开区间
    bool Surrounds(const double& Value) const {
        return Value > Minimum && Value < Maximum;
    }

public:
    static const Interval Empty;
    static const Interval Universe;

public:
    double Minimum;
    double Maximum;
};

const static Interval Empty(+Infinity, -Infinity);
const static Interval Universe(-Infinity, +Infinity);
```

代码 26：区间类[interval.h]

```
// 常用头文件

#include <ray.h>
#include <vec3.h>
#include <interval.h>
```

代码 27：导入新的区间类[rtweekend.h]

```
class Hittable {
public:
    ...
    virtual bool Hit(const Ray& Light, const Interval& RayRange,
                     HitRecord& Record) const = 0;
};
```

代码 28：Hittable::Hit() 使用区间改进[hittable.h]

```
class HittableList : public Hittable {
public:
    ...
    bool Hit(const Ray& Light, const Interval& RayRange, HitRecord& Record)
const override {
        HitRecord Temp;
        auto      HitFlag = false;
        auto      Closet   = RayRange.Maximum;

        for (const auto& Object : Objects) {
```

```
            if (Object->Hit(Light, Interval(RayRange.Minimum, Closet), Temp))
{
                HitFlag = true;
                Closet  = Temp.TValue;
                Record  = Temp;
            }
        }

        return HitFlag;
    }
    ...
};
```

代码 29：HittableList::Hit 使用区间改进[hittable_list.h]

```
class Sphere : public Hittable {
public:
    ...
    bool Hit(const Ray& Light, const Interval& RayRange,
             HitRecord& Record) const override {
        ...

        const auto SqrtDelta = sqrt(Delta);
        // 这里只采用最靠近原点（即最小的根）
        auto Root = (-HalfB - SqrtDelta) / A;
        // 如果超出范围，则选用另一个根
        if (!RayRange.Surrounds(Root)) {
            // 如果依然 OOR 则没有交点
            Root = (-HalfB + SqrtDelta) / A;
            if (!RayRange.Surrounds(Root)) {
                return false;
            }
        }
        ...
    }
    ...
};
```

代码 30：使用区间改进的 Sphere 类[sphere.h]

```
...
Color RayColor(const Ray& Light, const Hittable& World) {
    HitRecord Record;
    if (World.Hit(Light, Interval(0, Infinity), Record)) {
        return 255.f * 0.5 * (Record.NormalVector + Color(1, 1,
1));
```

```
    }

    Vec3       UnitDirection = UnitVector(Light.RayDirection());
    const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
    auto       Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a
* Color(0.5, 0.7, 1.0);

    return 255.0 * Result;
}
...
```

代码 31：使用区间改进的新 main 函数[main.cpp]

7. 将相机的代码整合成类

在继续以前，先将我们的相机和场景渲染代码合并到一个新类：相机类；相机类将负责两项重要工作：

1. 构造并向场景中发射光线
2. 使用这些光线计算的结果来构建渲染图

在此次重构中，将整合 RayColor 函数以及主程序的图像、相机和渲染部分代码；新的相机类将包含两个公共方法 Initialize 和 Render，以及两个私有辅助方法 GetRay 和 RayColor。

最终，相机将会遵循所能想到的最简单的使用方式：只提供一个无参数的默认构造函数，通过简单的复制来修改相机的公共变量，最后使用 Initialize 函数初始化所有内容；这种模式取代了所有者调用带有大量参数的构造函数或定义并调用大量 getter、setter 的做法；选择这种模式可以让代码只去设置其真正所需要的属性；最后，可以选择手动调用 Initialize 函数，或者让相机在 Render 函数中自动调用此函数；本文将采用第二种方法。

在 main 中创建了一个相机并设置其默认值后，其将调用 Render 方法；Render 方法将会初始化相机并启动渲染循环。

这是新相机类的大概结构:

```
#include <rtweekend.h>

#include <color.h>
#include <device.h>
#include <hittable.h>

class Camera {
  public:
    // 相机的公共参数

    void Render(const Hittable& World) {
        ...
    }

  private:
    // 相机的私有变量

    void Initialize() {
        ...
    }

    Color RayColor(const Ray& Light, const Hittable& World) const {
        ...
    }
};
```

代码 32: 相机类的大概架构[camera.h]

接下来, 先从 main.cpp 中把 RayColor 方法搬运过来:

```
class Camera {
    ...
private:
    ...

    Color RayColor(const Ray& Light, const Hittable& World) const {
        HitRecord Record;
        if (World.Hit(Light, Interval(0, Infinity), Record)) {
            return 255.f * 0.5 * (Record.NormalVector + Color(1, 1, 1));
        }

        Vec3       UnitDirection = UnitVector(Light.RayDirection());
        const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
```

```
        auto        Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);

        return 255.0 * Result;
    }
};
```

代码 33：Camera::RayColor 函数[camera.h]

现在需要将几乎所有东西都从 main 函数中移动到新的相机类里，main 函数中只保留的就是有关场景的构造代码；这里是的新迁移的代码：

```
class Camera {
public:
    void Render(const Hittable& World) {
        Initialize();

        // 渲染部分

        Color WriteColor;

        for (size_t Y = 0; Y < ImageHeight; ++Y) {
            std::clog << "\r还剩下: " << (ImageHeight - Y) << "条扫描线待渲染。 "
<< std::flush;
            const auto HeightVec = static_cast<double>(Y) * PixelDeltaV;
            for (size_t X = 0; X < ImageWidth; ++X) {
                auto PixelCenter =
                    Pixel100Loc + (static_cast<double>(X) * PixelDeltaU) +
HeightVec;
                const auto RayDirection = PixelCenter - CameraCenter;
                Ray        Light(CameraCenter, RayDirection);

                WriteColor = RayColor(Light, World);

                Graphics->At(X, Y) = ColorToDWORD(WriteColor);
            }
        }

        std::clog << "\r渲染完毕。           \n";

        Device::Flush();
    }

public:
```

```
    double AspectRatio  = 16.0 / 9.0;
    int    ImageWidth   = 640;

private:
    void Initialize() {
        // 计算图像的高，并且确保其起码为一

        ImageHeight = static_cast<int>(ImageWidth / AspectRatio);
        ImageHeight = (ImageHeight < 1) ? 1 : ImageHeight;

        // 相机部分

        const auto FocalLength    = 1.0;
        const auto ViewportHeight = 2.0;
        const auto ViewportWidth =
            ViewportHeight * (static_cast<double>(ImageWidth) / ImageHeight);
        CameraCenter = Point3(0, 0, 0);

        // 计算横纵向量 u 和 v

        const auto ViewportU = Vec3(ViewportWidth, 0, 0);
        const auto ViewportV = Vec3(0, -ViewportHeight, 0);

        // 由像素间距计算横纵增量向量

        PixelDeltaU = ViewportU / ImageWidth;
        PixelDeltaV = ViewportV / ImageHeight;

        // 计算左上角像素的位置

        const auto ViewportUpperLeft =
            CameraCenter - Vec3(0, 0, FocalLength) - ViewportU / 2 -
ViewportV / 2;
        Pixel100Loc = ViewportUpperLeft + 0.5 * (PixelDeltaU + PixelDeltaV);
        Graphics    = std::make_shared<Device>(ImageWidth, ImageHeight);
    }

    Color RayColor(const Ray& Light, const Hittable& World) const {
        ...
    }

private:
    std::shared_ptr<Device> Graphics;
```

```
    int     ImageHeight;
    Point3 CameraCenter;

    Vec3 PixelDeltaU;
    Vec3 PixelDeltaV;
    Vec3 Pixel100Loc;
};
```

代码 34：相机类[camera.h]

接着就是 main 函数的修改了：

```
#include <rtweekend.h>

#include <camera.h>
#include <color.h>
#include <device.h>
#include <hittableList.h>
#include <sphere.h>

#include <conio.h>

int main() {
    // 渲染场景

    HittableList World;

    auto GroundSphere = std::make_shared<Sphere>(Point3(0, 0, -1), 0.5);
    auto FakeGround   = std::make_shared<Sphere>(Point3(0, -100.5, -1), 100);

    World.Add(GroundSphere);
    World.Add(FakeGround);

    Camera WorldCamera;

    WorldCamera.AspectRatio = 16.f / 9.f;
    WorldCamera.ImageWidth  = 640;

    WorldCamera.Render(World);

    _getch();

    return 0;
}
```

代码 35：使用 Camera 类的新 main 函数[main.cc]

如果代码无误，运行程序应该得到和以前一模一样的渲染结果。

8. 抗锯齿

如果你将目前得到的渲染放大，可能会注意到渲染图像中边缘是粗糙的阶梯形状；这种阶梯状边缘一般被成为“锯齿（*aliasing or jaggies*）”；当一个真正的相机在拍摄图片时，其边缘通常没有锯齿，因为边缘像素是其和前景或背景融合而成的；与渲染图像不同；真实世界的图像是平滑连续的；换句话说，真实世界具有无限的分辨率；而在渲染图像中，可以通过对每个像素点大量采样取均值以获得相同的效果。

当一束光线穿过每个像素的中心时，这个过程就通常称为像素采样；像素采样过程可以通过渲染远处的小棋盘来说明：设想一个由 8×8 的黑白格子组成的棋盘，但只有四条光线照射到棋盘上，那么所有四条光线可能只与白色格子相交，或只与黑色格子相交，或与一些奇怪的组合相交；在现实世界中，当我们用眼睛观察远处的棋盘时，会觉得它是灰色的，而不是黑白分明的点；这是因为我们的眼睛在做光线追踪器做的事情：混合在渲染图像特定（离散）区域上的（连续）光线函数。

显然通过反复对像素中心的相同光线重采样并没有用，每次都只会获得同样的结果；取而代之，应该采样的是对像素周围的光线采样，并混合这些采样结果去接近真的连续结果；所以如何混合这些光线呢？

这里选取简单的模式：以原像素为中心，向四个相邻像素中的每一个延伸一半的正方形区域采样；这并不是最佳方法，但却是最直接的方法。(如需深入了解这，可以参阅这篇文章—“*A Pixel is Not a Little Square*”[13])。

[13] 译者注：文章链接：https://www.researchgate.net/publication/244986797

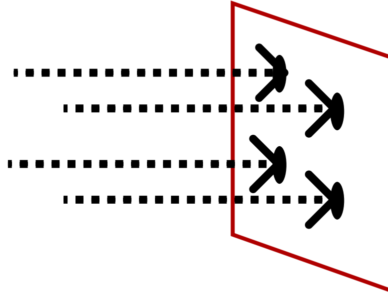

图 12：像素采样示意图

8.1 一些随机数方法

现在需要一个能够生成一个随机实数的随机数生成器；该函数必须返回一个规范的随机数，按照惯例，应有 $0 \le n < 1$，而 1 之前的小于号非常重要篇，因为有时将会用到它。

一个简单的实现方法就是使用 <cstdlib> 中的 rand 函数，其返回一个随机整数介于 0 到 RAND_MAX；因此可以在 rtweekend.h 中添加一下获取一个随机数的代码：

```
#pragma once

#include <cmath>
#include <limits>
#include <memory>
#include <numbers>
#include <cstdlib>

...

inline double RandomDouble() {
    // 返回一个随机数位于 [0, 1) 中
    return rand() / (RAND_MAX + 1.0);
}
inline double RandomDouble(const double& Min, const double& Max) {
    // 返回一个随机数位于 [Min, Max) 中
    return Min + RandomDouble() * (Max - Min);
}

...
```

代码 36：RandomDouble 函数[rtweekend.h]

传统 C++ 并没有一个标准随机数生成器，但较新版本的 C++ 已经通过 <random> 标头解决了这个问题（但一些专家说其并不完美）；如果你想使用新版 C++ 的随机数生成器，你可以取得一个随机数就像这样：

```
#include <random>

inline double RandomDouble() {
    static std::uniform_real_distribution<double>  Distribution(0.0, 1.0);
    static std::mt19937                             Generator;
    return Distribution(Generator);
}
```

代码 37：RandomDouble 函数，使用 C++ 新特性替代

8.2 使用多重采样生成一个像素

对于一个由多重采样组成的像素，其生成过程为从像素周围的区域中选择样本，并对所得的光（颜色）值进行平均。

首先，需要先将 WriteColor 函数进行升级，以考虑到使用的样本数量：需要找出所有样本的平均值；为了完成这项任务，在写入颜色前，需要将每次迭代的全部颜色相加，最后进行一次除法运算（除以采样数量）；为了确保最终结果的颜色成分保持在适当的$[0,1]$范围内，将添加并使用一个小的辅助函数：interval::clamp。

```
class Interval {
public:
    ...
    double Clamp(const double& X) const {
        if (X < Minimum) {
            return Minimum;
        }else if (X > Maximum) {
            return Maximum;
        }
        else {
            return X;
        }
    }
```

```
    ...
};
```

代码 38：Interval::Clamp 函数[interval.h]

这里是升级后添加了多重采样的 WriteColor 函数[14]：

```
/**
 * 将颜色转换为对应的内存 DWORD 值
 */
DWORD ColorToDWORD(const Color& Value, const int& SamplesPerPixel) {
    auto R = Value.X();
    auto G = Value.Y();
    auto B = Value.Z();

    // 将颜色除以采样数
    auto Scale = 1.f / SamplesPerPixel;
    R *= Scale;
    G *= Scale;
    B *= Scale;

    static const Interval Intensity(0, 255);
    return BGR(RGB(Intensity.Clamp(R), Intensity.Clamp(G),
Intensity.Clamp(B)));
}
```

代码 39：多重采样的 WriteColor 函数

现在来更新相机类来定义且使用一个新的 Camera::GetRay 函数，其将针对每个像素生成不同的采样；该函数将使用一个帮助函数 PixelSampleSquare，其生成一个随机采样点位于以中心点为原点的单位正方形内；然后，将随机样本从这个理想方格转换回当前采样的特定像素：

```
class Camera {
public:
    void Render(const Hittable& World) {
        Initialize();

        // 渲染部分

```

[14] 译者注：由于本译本采用了 EasyX 对代码进行重写，故此处应该修改的其实是 EasyX 版的 WriteColor 函数 ColorToDWORD，而原文中为 WriteColor 即将颜色写入 PPM 图像。

```
        for (size_t Y = 0; Y < ImageHeight; ++Y) {
            std::clog << "\r还剩下: " << (ImageHeight - Y) << "条扫描线待渲染。 "
<< std::flush;
            const auto HeightVec = static_cast<double>(Y) * PixelDeltaV;
            for (size_t X = 0; X < ImageWidth; ++X) {
                Color WriteColor;
                for (int Sample = 0; Sample < SamplesPerPixel; ++Sample) {
                    Ray Light = GetRay(X, Y, HeightVec);
                    WriteColor += RayColor(Light, World);
                }

                Graphics->At(X, Y) = ColorToDWORD(WriteColor,
SamplesPerPixel);
            }
        }

        std::clog << "\r渲染完毕。           \n";

        Device::Flush();
    }

public:
    double AspectRatio     = 16.0 / 9.0; // 图像纵高比
    int    ImageWidth      = 640;        // 图像像素宽
    int    SamplesPerPixel = 10;         // 采样率

private:
    ...

    Ray GetRay(const size_t& X, const size_t& Y, const Vec3& HeightVec) const
{
        // 从位于 (X, Y) 的像素获得获取一个随机采样光线
        auto PixelCenter = Pixel100Loc + (static_cast<double>(X) *
PixelDeltaU) + HeightVec;
        auto PixelSample = PixelCenter + PixelSampleSquare();

        const auto RayOrigin    = CameraCenter;
        const auto RayDirection = PixelSample - RayOrigin;

        return Ray{RayOrigin, RayDirection};
    }
    Vec3 PixelSampleSquare() const {
        // 返回一个位于像素为中心的正方形内的随机采样点
        auto PointX = -0.5 + RandomDouble();
```

```
        auto PointY = -0.5 + RandomDouble();

        return (PointX * PixelDeltaU) + (PointY * PixelDeltaV);
    }

    ...
};
```

代码 40：有了 SPP（Samples-Per-Pixel）的相机[camera.h][15]

main 函数同样需要更新设置相机参数的代码：

```
int main() {
    ...

    Camera WorldCamera;

    WorldCamera.AspectRatio = 16.f / 9.f;
    WorldCamera.ImageWidth  = 640;
    WorldCamera.SamplesPerPixel = 100;

    WorldCamera.Render(World);

    ...
}
```

代码 41：设计新的 SPP 参数[main.cpp]

现在再运行代码，放大图片，可以发现边缘变得更柔和了：

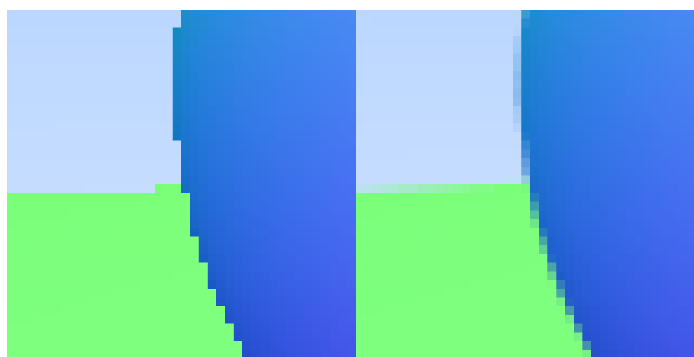

图 13：抗锯齿前和抗锯齿后

---

[15] 译者注：原著中注："除了上文中的 pixel_sample_square 函数，如果你想尝试非矩形形状的像素，你也可以在本书的 GitHub 仓库中找到 pixel_sample_disk 函数，但是本书中将不会是用，pixel_sample_disk 依赖于后文将会定义的 random_in_unit_disk"，但本书在 EasyX 版源码不再额外提供与原书中对应非矩形像素的 PixelSampleDisk 函数，读者若有需要自行查找。

9. 漫反射材质

现在一切的准备工作已经做好了，可以开始添加一下逼真的材质了；将从漫反射材质开始（通常也称为哑光）；一个问题是将材质和几何体独立开来（以便我们可以将材质分配给多个球体，反之亦然），还是将材质和几何体绑定（这对于几何体和材质相连的程序对象可能很有用）；本文将采用大多数渲染器采用的分离方式，但请注意还有其他方法。

9.1 一个简单的漫反射材质

一个不发光的漫反射对象只会吸收周围环境的颜色，但它们会用自己固有的颜色来调节周围环境的颜色；经由漫反射反射的光方向是随机的，所以如果三条光线射入两个漫反射表面之间的缝隙中，它们各自会有不同的随机行为：

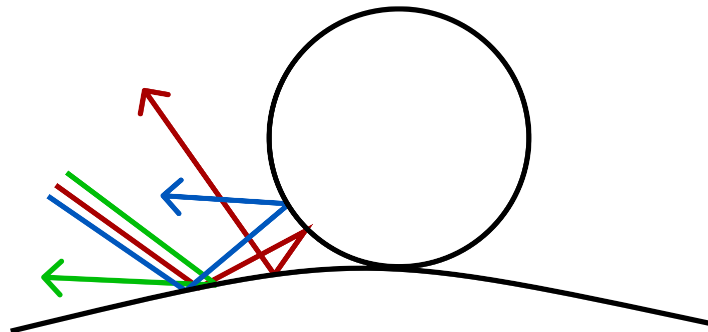

图 14：漫反射材的光线反射示意图

光线也可能会被吸收而不是反弹；约黑暗的表面，光线更容易被吸收（这也是为什么表面是黑色）；事实上，任何能随机调整方向的算法都能产生看起来像哑光的表面；先从最直观的开始吧：一个向所有反向随机反射光的表面；对于这种材质，一束与该表面产生交点的光线从表面向任意方向反射的可能性是均等的。

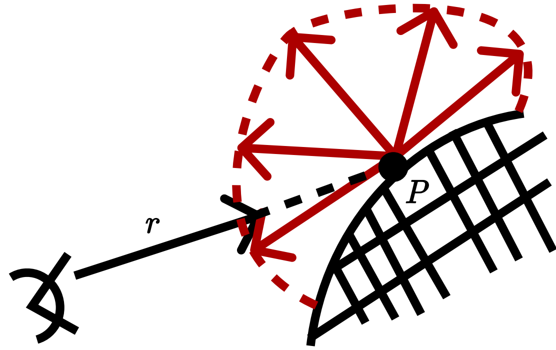

图 15：水平线上的等距离反射

这种非常直观的材料是最简单的一种漫反射，事实上，许多最早的光线追踪论文都使用了这种漫反射方法（在一种更精确的方法被发现之前，后文也将采用这种方法）；现在还没有一个随机反射光线的方法，所以需要在向量头中添加一些方法；第一件事情就是让其能够生成一些任意的随机向量：

```
class Vec3 {
    ...
public:
    ...

    double LengthSquared() const {
        return std::pow(e[0], 2) + std::pow(e[1], 2) + std::pow(e[2], 2);
    }
    static Vec3 Random() {
        return Vec3(RandomDouble(), RandomDouble(), RandomDouble());
    }
    static Vec3 Random(const double& Min, const double& Max) {
        return Vec3(RandomDouble(Min, Max), RandomDouble(Min, Max),
RandomDouble(Min, Max));
    }

    ...
};
```

代码 42：Vec3 的随机函数[vec3.h]

接着需要去弄清楚如何操控一个随机向量，以便只得到半球表面上的结果；有一些分析方法可以做到这一点，但实际上理解起来出奇地复杂，实践起来也相当复杂；于是使用通常

最简单的方法来取代：剔除法；提处法的工作原理是反复产生随机样本，直到产生一个符合所需的标准样本；换句话说，不断剔除样本，直到找到一个好样本。

有许多不相上下的生成半球体上的随机向量方法，但就目的而言，将使用最简单的方法，即：

1. 生成一个在单位球体中的随机向量
2. 对向量进行归一化操作
3. 反转归一化后的向量如果其位于错误的半圆上

首先，将采用剔除法去生成一个在单位球体内的随机向量；选取一个单位立方体中的随机点，其中 $x, y, z \in [-1, 1]$，并把在单位球体外的向量剔除：

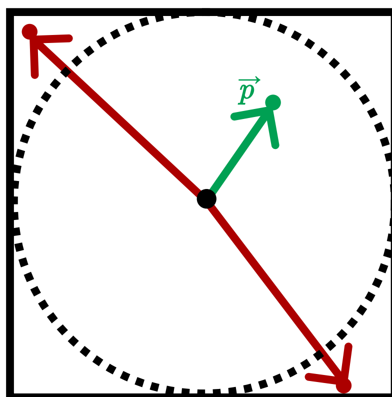

图 16：两个被剔除的向量和一个符合要求的向量

```
...

// 归一化向量
inline Vec3 UnitVector(const Vec3& Vector) {
    return Vector / Vector.Length();
}
// 生成一个位于单位球体中的向量
inline Vec3 RandomInUnitSphere() {
    while (true) {
        auto Vector = Vec3::Random();
        if (Vector.LengthSquared() < 1) {
            return Vector;
        }
    }
```

```
}
```

代码 43：RandomInUnitSphere 函数[vec3.h]

当拥有了一个位于单位球体中的随机向量，还需要对其进行归一化以获得一个在单位球体上向量。

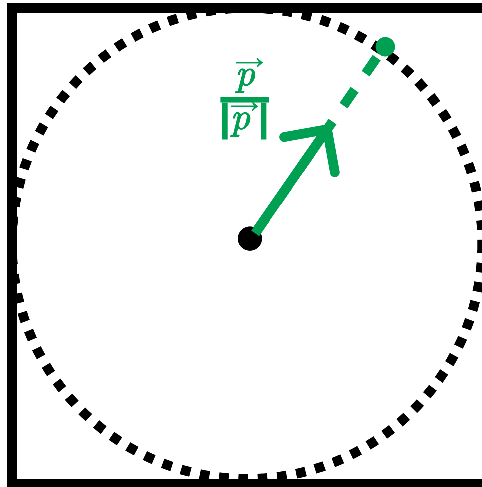

图 17：被保留的向量进行归一化处理，生成单位矢量

```
// 生成一个随机单位向量
inline Vec3 RandomUnitVector() {
    return UnitVector(RandomInUnitSphere());
}
```

代码 44：生成在点位球体上的随机向量[vec3.h]

现在，已经有了一个在单位球体上的随机向量，可以通过将其与表面法线比较来确定它是否在正确的半球上：

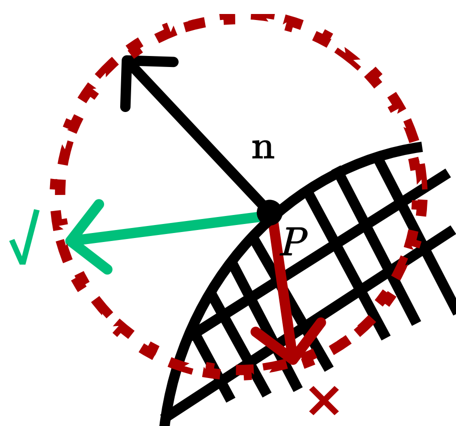

图 18：利用表面法线判断所需半球

可以利用表面法线与随机矢量的点积来确定它是否位于正确的半球；如果点积为正，则说明该向量位于正确的半球，如果点积为负，则需要反转矢量。

```
...

// 生成一个随机单位向量
inline Vec3 RandomUnitVector() {
    return UnitVector(RandomInUnitSphere());
}
// 判断随机所在半球
inline Vec3 RandomOnHemisphere(const Vec3& Normal) {
    Vec3 OnUnitSphere = RandomInUnitSphere();
    if (Dot(OnUnitSphere, Normal) > 0.f) {
        return OnUnitSphere;
    }
    else {
        // 如果不在同一个平面则反转变量
        return -OnUnitSphere;
    }
}
```

代码 45：RandomInHemisphere 函数[vec3.h]

如果一个光线被一个材质反射并 100% 地保留颜色，则称这种材质为白色的；如果一个光线被一个材质反射没有保留颜色，则称这个材料为黑色的；作为新漫反射材质的首次演示，将把 RayColor 函数设置为返回反弹颜色的 50%。这样应该会得到漂亮的灰色。

```
...

class Camera {
public:
    void Render(const Hittable& World) {
        Initialize();

        // 渲染部分

        for (size_t Y = 0; Y < ImageHeight; ++Y) {
            std::clog << "\r还剩下: " << (ImageHeight - Y) << "条扫描线待渲染。 "
<< std::flush;
            const auto HeightVec = static_cast<double>(Y) * PixelDeltaV;
            for (size_t X = 0; X < ImageWidth; ++X) {
                Color WriteColor;
```

```
                for (int Sample = 0; Sample < SamplesPerPixel; ++Sample) {
                    Ray Light = GetRay(X, Y, HeightVec);
                    WriteColor += 255.f * RayColor(Light, World);
                }

                Graphics->At(X, Y) = ColorToDWORD(WriteColor,
SamplesPerPixel);
            }
        }

        std::clog << "\r渲染完毕。           \n";

        Device::Flush();
    }

    ...

private:
    ...

    Color RayColor(const Ray& Light, const Hittable& World) const {
        HitRecord Record;
        if (World.Hit(Light, Interval(0, Infinity), Record)) {
            Vec3 Direction = RandomOnHemisphere(Record.NormalVector);
            return 0.5 * RayColor(Ray(Record.Point, Direction), World);
        }

        Vec3       UnitDirection = UnitVector(Light.RayDirection());
        const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
        auto       Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);

        return Result;
    }

    ...
};
```

代码 46：RayColor 使用随机光线方向[camera.h]

而最后确实得到了很漂亮的灰色球体：

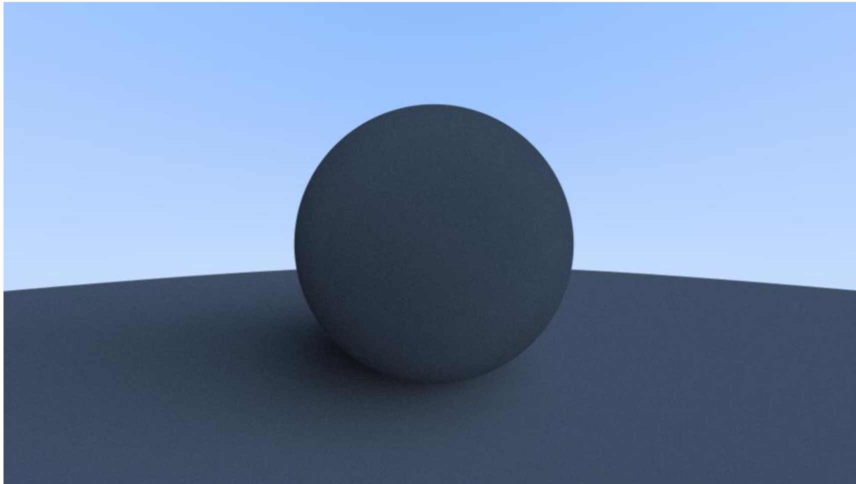

图 19：第一次渲染漫反射球体

9.2 限制子光线数量

这里有一个潜在的问题：请注意，RayColor 函数是递归的，那么什么时候其才会停止递归呢？答案是当其光线不再与任何物体相交时；在某些情况下这将带来很久的耗时，长的足够使栈溢出；为了防止这种情况，应该限制最大递归深度，在最大深时返回没有光照贡献：

```
class Camera {
public:
    void Render(const Hittable& World) {
        Initialize();

        // 渲染部分

        for (size_t Y = 0; Y < ImageHeight; ++Y) {
            std::clog << "\r还剩下: " << (ImageHeight - Y) << "条扫描线待渲染。 "
<< std::flush;
            const auto HeightVec = static_cast<double>(Y) * PixelDeltaV;
            for (size_t X = 0; X < ImageWidth; ++X) {
                Color WriteColor;
                for (int Sample = 0; Sample < SamplesPerPixel; ++Sample) {
                    Ray Light = GetRay(X, Y, HeightVec);
                    WriteColor += 255.f * RayColor(Light, MaxDepth, World);
                }
```

```
                Graphics->At(X, Y) = ColorToDWORD(WriteColor,
SamplesPerPixel);
            }
        }

        std::clog << "\r渲染完毕。           \n";

        Device::Flush();
    }

public:
    double AspectRatio     = 16.0 / 9.0; // 图像纵高比
    int    ImageWidth      = 640;        // 图像像素宽
    int    SamplesPerPixel = 10;         // 采样率
    int    MaxDepth        = 10;         // 最大光反射数量

    ...
};
```

代码 47：Camera::RayColor 函数使用光线迭代深度限制[camera.h]

main 函数也应该更新以使用新添加的深度限制：

```
int main() {
    ...

    Camera WorldCamera;

    WorldCamera.AspectRatio     = 16.f / 9.f;
    WorldCamera.ImageWidth      = 400;
    WorldCamera.SamplesPerPixel = 200;
    WorldCamera.MaxDepth        = 50;

    WorldCamera.Render(World);

    ...
}
```

代码 48：使用光线迭代深度限制[main.cpp]

对于这个非常简单的场景，应该得到基本一致的渲染结果：

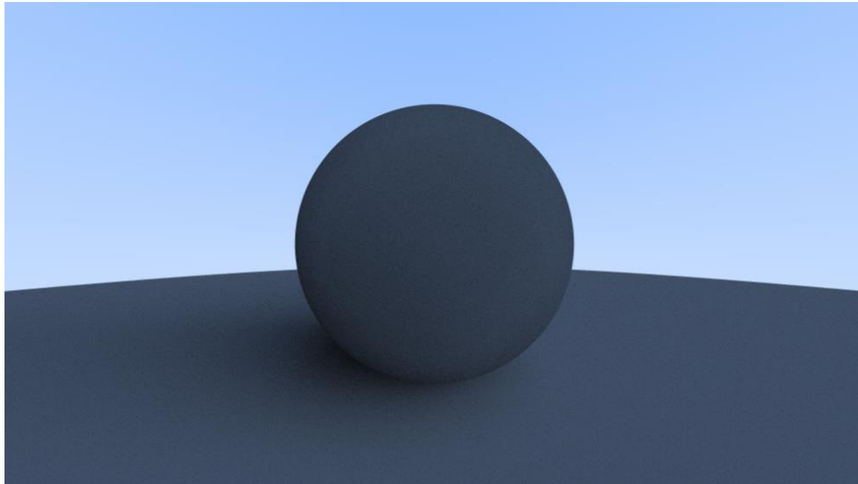

图 20：带有反射限制的第二次渲染漫反射材质

9.3 修复暗疮

同样还有一个需要去修复的神奇问题就是：当一束光线与一个表面交叉时，其将尝试去准确计算交叉点；不幸的是，这种计算很容易受到浮点舍入误差的影响，从而导致交叉点点稍有偏差；这意味着下一束光线的原点将会随机地从表面散落，不可能与表面完全齐平；其有可能在表面之上，也有在能在表面之下；如果光线的原点就在表面之下，其有可能与平面再次相交，这意味着其将找到最近的平面位于$t = 0.000000001$处或任何命中函数给出的浮点近似值；解决这个问题的最简单方法就是忽略与计算交点非常接近的交叉：

```
class Camera {
    ...
private:
    ...
    Color RayColor(const Ray& Light, int Depth, const Hittable& World) const
    {
        HitRecord Record;
        if (Depth <= 0) {
            return Color{0, 0, 0};
        }

        if (World.Hit(Light, Interval(0.001, Infinity), Record)) {
```

```
            Vec3 Direction = RandomOnHemisphere(Record.NormalVector);
            return 0.5 * RayColor(Ray(Record.Point, Direction), Depth - 1,
World);
        }

        Vec3        UnitDirection = UnitVector(Light.RayDirection());
        const auto a              = 0.5 * (UnitDirection.Y() + 1.0);
        auto        Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);

        return Result;
    }

    ...
};
```

代码 49：带有忽略的反射光线计算

这样就能摆脱暗疮问题，是的，它真的叫这个名字；这就是渲染结果：

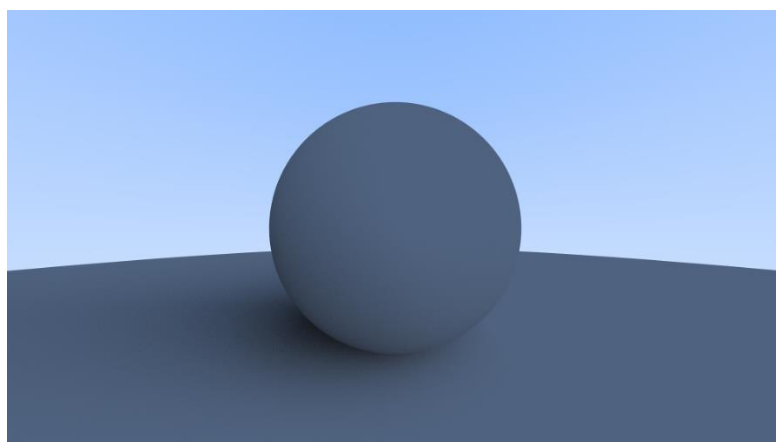

图 21：没有暗疮问题的漫反射球体

9.4 真正的朗伯反射

将反射光线均匀地散射到半球上会产生一个很好的柔和漫反射模型，但绝对可以做得更好；一个更准确的代表就是朗伯分布（*Lambertian distribution*）；朗伯分布对反射光线的散射与 $\cos(\Phi)$ 成正比，其中 Φ 是反射光与水平法线的夹角；这意味着一束反射光线最有可能向表面法线附近的方向散射，而不太可能偏离正常的方向散射；与之前的均匀散射相

比，这种非均匀朗伯分布能更好地模拟现实世界中的材料反射。

可以通过添加一个随机单位向量于法向量上来创建这种分布；在曲面上的交点就是击中点$\vec{P}$，以及平面法线$\vec{n}$；在交点上，曲面正好有两条边，因此只能有两个唯一的单位球体与任意交点相切（曲面的每条边上都有一个唯一的球体）；这两个单位球体将会被其半径长替代，对于一个单位球体来说，半径正好是 1。

其中一个球体将会被其表面法线的方向（$\vec{n}$）取代，另一个球体将会被相反的方向所取代（$-\vec{n}$）；这样，就有了两个单位大小的球体，它们在交点刚好接触表面；由此，其中一个球体的中心将位于$\left(\vec{P}+\vec{n}\right)$而另一个球体的中心将位于$\left(\vec{P}-\vec{n}\right)$；位于$\left(\vec{P}-\vec{n}\right)$的球体被认为是在表面里面的，相对地，位于$\left(\vec{P}+\vec{n}\right)$的球体被认为是在表面外面的。

要选择与射线原点位于曲面同一侧的切线单位球面；选择一个位于单位半径球体上的随机点$\vec{S}$并从其交点$\vec{P}$发射一条光线到随机点$\vec{S}$（即为向量$\left(\vec{S}-\vec{P}\right)$）。

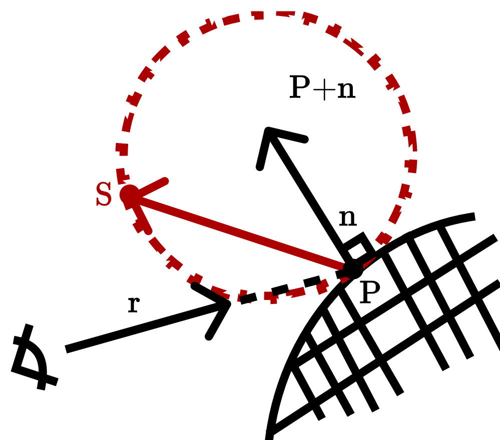

图 22：根据朗伯分布随机生成向量

实际上，需要改动的代码很少：

```
class Camera {
    ...

    Color RayColor(const Ray& Light, int Depth, const Hittable& World) const
{
        HitRecord Record;
```

```
        if (Depth <= 0) {
            return Color{0, 0, 0};
        }

        if (World.Hit(Light, Interval(0.001, Infinity), Record)) {
            Vec3 Direction = Record.NormalVector + RandomUnitVector();
            return 0.5 * RayColor(Ray(Record.Point, Direction), Depth - 1,
World);
        }

        Vec3       UnitDirection = UnitVector(Light.RayDirection());
        const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
        auto       Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);

        return Result;
    }

    ...
};
```

代码 50：替换了漫反射材质的 RayColor 函数[camera.h]

修改后代码得到了很相似的渲染结果：

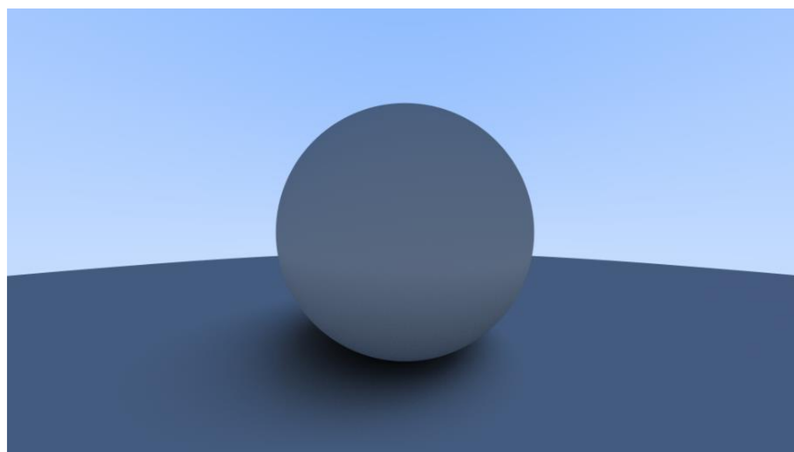

图 23：朗伯球体的正确渲染

由于代码中的两个球体场景非常简单，所以很难说出这两种漫反射方法之间的区别，但你应该能注意到两种重要的视觉差异：

1. 阴影更加明显
2. 变化后，两个球体都被天空染成蓝色

所有的这些变化都是因为光线的散射不够均匀，换句话说就是，更多射线向法线散射；这意味着对于漫反射物体而言，他们会显得更暗因为其反射更少的光线于相机中；而对于阴影，直射光较多，因此球体下方的区域较暗。

日常生活中，并不是很多物体都具有完美的漫反射效果，因此对这些物体在光线下的行为的视觉直觉可能并不完善；在本书的学习过程中，随着场景变得越来越复杂，我们鼓励你在本书介绍的不同漫反射渲染方式之间进行切换；大多数场景都包含大量漫反射材质；通过了解不同漫反射方法对场景照明的影响，可以获得宝贵的见解。

9.5 使用 Gamma 矫正以获得精确的色彩强度

现在注意球体下的阴影；画面非常暗，但球体只吸收每次反弹的一半能量，所以它们的反射率是 50%；球体应该看起来很亮（在现实世界中，应该是亮灰色）但是看起来会更暗一些；如果对漫反射材质的整个亮度范围进行调整，就能更清楚地看到这一点；首先将 RayColor 函数的反射率从 0.5（50%）设置为 0.1（10%）：

```
class Camera {
public:
    ...

    Color RayColor(const Ray& Light, int Depth, const Hittable& World) const
{
        HitRecord Record;
        if (Depth <= 0) {
            return Color{0, 0, 0};
        }

        if (World.Hit(Light, Interval(0.001, Infinity), Record)) {
            Vec3 Direction = Record.NormalVector + RandomUnitVector();
            return 0.1 * RayColor(Ray(Record.Point, Direction), Depth - 1,
World);
        }

        Vec3       UnitDirection = UnitVector(Light.RayDirection());
```

```
        const auto a            = 0.5 * (UnitDirection.Y() + 1.0);
        auto       Result       = (1.0 - a) * Color(1.0, 1.0, 1.0) + a *
Color(0.5, 0.7, 1.0);

        return Result;
    }

    ...
};
```

代码 51：10% 反射率的 RayColor 函数[camera.h]

以新的 10% 反射率进行渲染；然后将反射率设置为 30%，并再次渲染；依次重复 50%、70% 和 90%；你可以在自己选择的图片编辑器中将这些图片从左到右叠加，这样就能很直观地显示出所选色域的亮度增加情况：

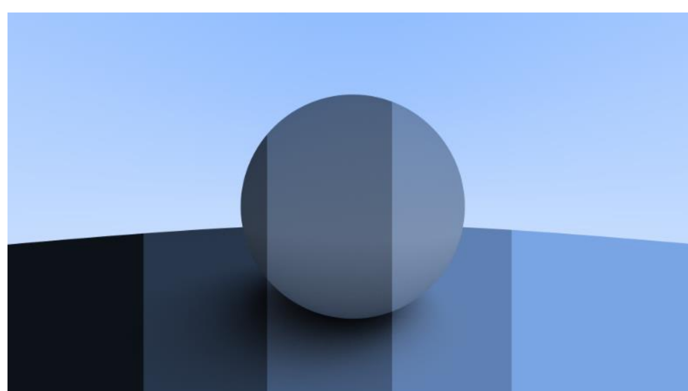

图 24：目前为止渲染器的色域

如果你看的仔细点，或者使用一个取色器，你会发现 50% 反射率（图 24 中间）的渲染结果颜色太深，介于白色和黑色（中灰色）之间；确实如此，70% 的反射率反而更接近中灰色；之所以会这样是因为几乎所有的计算机程序在写入图像文件之前都假设图像是“伽玛校正的”；这意味着，0 到 1 的数值在以字节形式存储之前会进行一些转换；写入数据而不进行转换的图像被称为线性空间图像，相对地，被转换后的图像被称为伽马空间（*Gamma Space*）；你所使用的图像查看器很可能预期的是伽马空间中的图像，而渲染器提供给它的却是线性空间中的图像；这就是图像看起来不准确的原因。

以伽马空间存储图像有很多很好处，但是对于目前的目的而言，只需要注意有这个东

西即可；现在需要修改代码将把数据转换到伽玛空间，以便图像查看器能更准确地显示图像；作为一个简单的近似，可以选择“Gamma 2”作为变换方法，也就是从伽马空间到线性空间时使用的指数；由于是需要将线性空间转换到伽马空间，这意味着需要将“Gamma 2”反转，这意味着指数将取倒数：$\frac{1}{gamma}$，即平方根。

```
// 线性空间到伽马空间的变化
inline double LinearToGamma(const double& LinearComponent) {
    return sqrt(LinearComponent);
}
// 将颜色转换为对应的内存 DWORD 值
DWORD ColorToDWORD(const Color& Value, const int& SamplesPerPixel) {
    auto R = Value.X();
    auto G = Value.Y();
    auto B = Value.Z();

    // 将颜色除以采样数
    auto Scale = 1.f / SamplesPerPixel;
    R *= Scale;
    G *= Scale;
    B *= Scale;

    // 进行线性空间到 Gamma 空间的变换
    R = LinearToGamma(R);
    G = LinearToGamma(G);
    B = LinearToGamma(B);

    // 将颜色分量映射到 RGB 空间上
    static const Interval Intensity(0.f, 0.999);
    return BGR(RGB(Intensity.Clamp(R) * 255, Intensity.Clamp(G) * 255,
Intensity.Clamp(B) * 255));
}
```

代码 52：有了 Gamma 矫正的 WriteColor 函数[color.h]

由于此处使用了线性空间到伽马空间的映射，也需要对 Camera 类的代码进行一些修改，使其传入 ColorToDWORD 的参数为颜色分量而不是 RGB 值[16]：

[16] 译者注：此处原著并没有，为译者自行根据 EasyX 版本的代码修改而成。

```
class Camera {
    ...

    void Render(const Hittable& World) {
        Initialize();

        // 渲染部分

        for (size_t Y = 0; Y < ImageHeight; ++Y) {
            std::clog << "\r还剩下: " << (ImageHeight - Y) << "条扫描线待渲染。 "
<< std::flush;
            const auto HeightVec = static_cast<double>(Y) * PixelDeltaV;
            for (size_t X = 0; X < ImageWidth; ++X) {
                Color WriteColor;
                for (int Sample = 0; Sample < SamplesPerPixel; ++Sample) {
                    Ray Light = GetRay(X, Y, HeightVec);
                    WriteColor += RayColor(Light, MaxDepth, World);
                }

                Graphics->At(X, Y) = ColorToDWORD(WriteColor,
SamplesPerPixel);
            }
        }

        std::clog << "\r渲染完毕。            \n";

        Device::Flush();
    }

    ...
}
```

代码 53：修改了传参的 Camera::Render 函数[camera.h]

有了伽马矫正后，现在，从黑暗到明亮的斜坡更加一致了[17]：

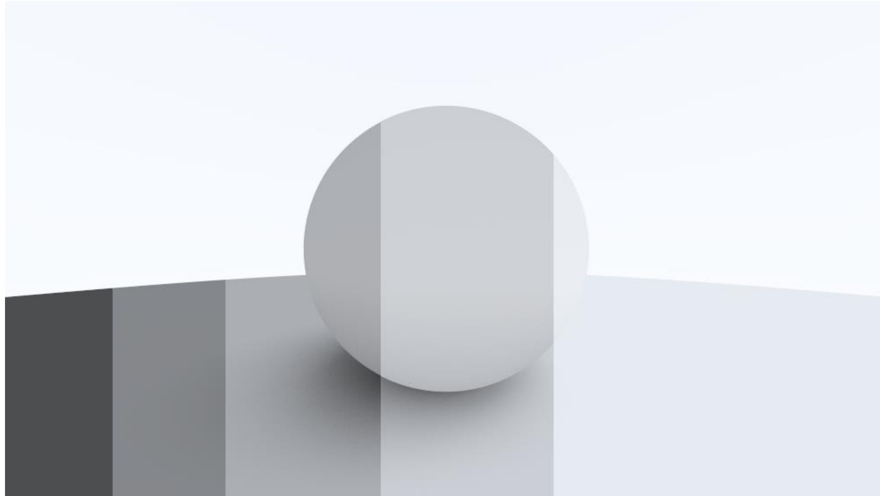

图 25：伽马修正后的渲染结果

10. 金属材料

10.1 对于材质的抽象类

如果希望不同的无图都有不同的材质，又需要做出一个程序设计上的抉择：可以设计一种带有大量参数的通用材料类型，这样任何单独的材料类型都可以忽略对其没有影响的参数，这种方法听着不错；或者，也可以用一个抽象的材料类来封装独特的行为；我喜欢后一种方法，对于我们的程序来说，材料需要做两件事：

1. 产生散射光线（或者说它吸收了入射光线）
2. 如果是散射，计算射线应该衰减多少

抽象类的代码如下：

```
/**
 * \file material.h
 * \brief 材质抽象类
 */

#pragma once
```

[17] 译者注：为了使效果更明显，此处译者修改了天空的颜色为几乎纯白。

```
#include <color.h>
#include <rtweekend.h>

class HitRecord;

class Material {
public:
    virtual ~Material() = default;
    virtual bool Scatter(const Ray& LightIn, const HitRecord& Recrod, Color&
Attenuation, Ray& Scattered) const = 0;
};
```

代码 54：材质抽象类[material.h]

10.2 描述射线与物体交点的数据结构

之所以使用 HitRecord 类是为了避免给函数传入一堆参数且也可以方便随时添加所需要的新信息；您可以使用参数来代替其类型，这只是个人喜好问题；命中物和材质需要能够在代码中引用对方的类型，因此会造成引用循环；在 C++ 中，需要添加一行预定义代码 class Material; 以告诉编译器 Material 类将会在稍后被定义；由于只是需要定义一个指向 Material 类的指针，编译器并不需要知道 Material 类的细节，这样子就可以解决循环引用的问题：

```
class Material;

class HitRecord {
public:
    HitRecord() = default;

    void SetFaceNormal(const Ray& Light, const Vec3& OutwardNormal) {
        // 设置 HitRecord 法向量
        // 注意：参数'OutwardNormal'已经假设拥有单位长度
        FrontFace    = Dot(Light.RayDirection(), OutwardNormal) < 0;
        NormalVector = FrontFace ? OutwardNormal : -OutwardNormal;
    }

public:
```

```
    Point3                   Point;
    Vec3                     NormalVector;
    double                   TValue{};
    bool                     FrontFace;
    std::shared_ptr<Material> OMaterial;
};
```

代码 55：带有 Material 指针的 HitRecord[hittable.h]

HitRecord 只是一种将一堆参数塞进一个类的方法以便将它们作为一组数据传递；当一束光线与表面相交（例如某个球体），HitRecord 中的 Material 指针将会被设置为指向已经在 main 函数中设置好的球体的材质；当 RayColor 获得 HitRecord 时，其可以调用材质指针的成员函数来计算散射的射线（如果有的话）。

为此，Sphere 类应当在 HitRecord 中记录下自身的材质。

```
/**
 * \file sphere.h
 * \brief 球体相关定义代码
 */

#pragma once

#include <hittable.h>

class Sphere : public Hittable {
public:
    Sphere(Point3 ICenter, const double& IRadius, std::shared_ptr<Material>
IMaterial) : Center(ICenter), Radius(IRadius), OMaterial(IMaterial) {
    }
    bool Hit(const Ray& Light, const Interval& RayRange, HitRecord& Record)
const override {
        ...

        // 求解出交点并记录在 Record 中
        Record.TValue      = Root;
        Record.Point       = Light.At(Root);
        Record.OMaterial   = OMaterial;
        Vec3 OutwardNormal = (Record.Point - Center) / Radius;
        Record.SetFaceNormal(Light, OutwardNormal);

        return true;
```

```
    }

private:
    Point3                      Center;
    double                      Radius;
    std::shared_ptr<Material> OMaterial;
};
```

代码 56：添加了材料信息的 Sphere 类[sphere.h]

10.3 为光散射和反射建模

对于现已经掌握的朗伯（漫反射）情况，它要么总是散射并按其反射率 R 衰减，要么散射（概率为 $1-R$）而不衰减（未散射的光线只是被材质吸收），也可能是这两个情况的混合；为了追求简单，本文将选择始终散射，因此朗伯材质就成了这个简单的类：

```
#include <color.h>
#include <rtweekend.h>
#include <hittableList.h>

class HitRecord;

class Material {
public:
    virtual ~Material()                             = default;
    virtual bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                         Ray& Scattered) const = 0;
};

// 朗伯漫反射材质
class Lambertian : public Material {
public:
    Lambertian(const Color& IAlbedo) : Albedo(IAlbedo) {
    }
    bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                 Ray& Scattered) const override {
        auto ScatterDirection = Record.NormalVector + RandomUnitVector();
        Scattered             = Ray(Record.Point, ScatterDirection);
        Attenuation           = Albedo;
```

```
        return true;
    }

private:
    Color Albedo;
};
```

代码 57：新的朗伯材质类

请注意，散射还有第三种方法：可以以某种固定概率 p 衰减并使衰减为 $\frac{albedo}{p}$，具体使用何种方法根据个人喜好即可。

如果仔细阅读上文中的代码，你可能会注意到可以小问题；如果随机单位向量刚好与法向量相反，其和将为零，这将导致散射方向向量为零；这会导致后面出现糟糕的情况（无穷大和 NaN），因此我们需要在将其传递之前拦截掉这种情况。

```
class Vec3 {
public:
    ...
    bool NearZero() const {
        auto Judge = 1e-8;
        return (fabs(e[0]) < Judge) && (fabs(e[1]) < Judge) && (fabs(e[2]) <
Judge);
    }
    ...
};
```

代码 58：Vec3::NearZero 方法[vec3.h]

```
class Lambertian : public Material {
public:
    Lambertian(const Color& IAlbedo) : Albedo(IAlbedo) {
    }
    bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                Ray& Scattered) const override {
        auto ScatterDirection = Record.NormalVector + RandomUnitVector();
        if (ScatterDirection.NearZero()) {
            ScatterDirection = Record.NormalVector;
        }
```

```
        Scattered              = Ray(Record.Point, ScatterDirection);
        Attenuation            = Albedo;

        return true;
    }

private:
    Color Albedo;
};
```

代码 59：修改后的朗伯反射[material.h]

10.4 镜面反射

对于抛光金属，光线不会随机散射；关键问题是：金属材质是如何镜面反射光线的？在这里，需要借助向量相关数学知识进行推导：

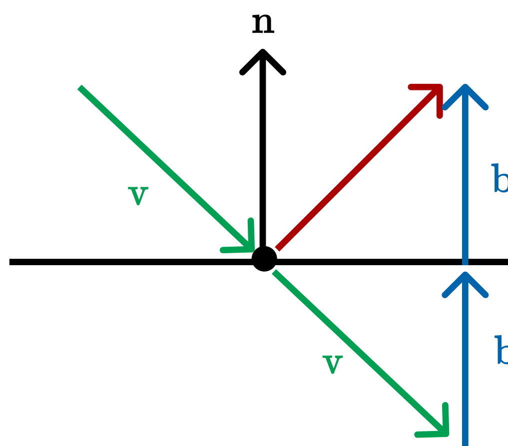

图 26：光线反射示意图

图中红色的即为反射光线$\vec{v}+2\vec{b}$；在本书代码设计中，$\vec{n}$必是单位向量，但$\vec{v}$有可能不是；$\vec{b}$的长度应为$\vec{v}\cdot\vec{n}$；由于$\vec{v}$指向内，需要对其取反，得出：

```
...
inline Vec3 RandomOnHemisphere(const Vec3& Normal) {
    ...
}
Vec3 Reflect(const Vec3& Vector, const Vec3& Normal) {
    return Vector - 2.f * Dot(Vector, Normal) * Normal;
}
```

代码 60：Vec3 的反射函数[vec3.h]

而金属材料的反射只需要套用 Reflect 函数即可：

```
...

class Metal : public Material {
public:
    Metal(const Color& IAlbedo) : Albedo(IAlbedo) {
    }
    bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                 Ray& Scattered) const override {
        Vec3 Reflected = Reflect(UnitVector(LightIn.RayDirection()),
Record.NormalVector);
        Scattered      = Ray(Record.Point, Reflected);
        Attenuation    = Albedo;

        return true;
    }

private:
    Color Albedo;
};
```

代码 61：带有反射函数的金属材料[material.h]

现在还需要修改 RayColor 函数：

```
#include <rtweekend.h>

#include <color.h>
#include <device.h>
#include <hittable.h>
#include <material.h>

class Camera {
    ...

private:
    ...

    Color RayColor(const Ray& Light, int Depth, const Hittable& World) const
{
        HitRecord Record;
```

```
        if (Depth <= 0) {
            return Color{0, 0, 0};
        }

        if (World.Hit(Light, Interval(0.001, Infinity), Record)) {
            Ray   Scattered;
            Color Attenuation;
            if (Record.OMaterial->Scatter(Light, Record, Attenuation, 
Scattered)) {
                return Attenuation * RayColor(Scattered, Depth - 1, World);
            }
            return Color{0, 0, 0};
        }

        Vec3       UnitDirection = UnitVector(Light.RayDirection());
        const auto a             = 0.5 * (UnitDirection.Y() + 1.0);
        auto       Result        = (1.0 - a) * Color(1.0, 1.0, 1.0) + a * 
Color(0.5, 0.7, 1.0);

        return Result;
    }

    ...
};
```

代码 62：带有反射的 RayColor 函数[camera.h]

10.5 带有金属球体的场景

现在来加入一些金属球体于场景中：

```
int main() {
    // 渲染场景

    HittableList World;

    auto MaterialGround = std::make_shared<Lambertian>(Color(0.8, 0.8, 0.0));
    auto MaterialCenter = std::make_shared<Lambertian>(Color(0.7, 0.3, 0.3));
    auto MaterialLeft   = std::make_shared<Metal>(Color(0.8, 0.8, 0.8));
    auto MaterialRight  = std::make_shared<Metal>(Color(0.8, 0.6, 0.2));

    World.Add(make_shared<Sphere>(Point3( 0.0, -100.5, -1.0), 100.0, 
MaterialGround));
```

```
    World.Add(make_shared<Sphere>(Point3( 0.0,    0.0, -1.0),   0.5, MaterialCenter));
    World.Add(make_shared<Sphere>(Point3(-1.0,    0.0, -1.0),   0.5, MaterialLeft));
    World.Add(make_shared<Sphere>(Point3( 1.0,    0.0, -1.0),   0.5, MaterialRight));

    Camera WorldCamera;

    WorldCamera.AspectRatio     = 16.f / 9.f;
    WorldCamera.ImageWidth      = 400;
    WorldCamera.SamplesPerPixel = 100;
    WorldCamera.MaxDepth        = 50;

    WorldCamera.Render(World);

    _getch();

    return 0;
}
```

代码 63：带有金属球体的场景[main.cpp]

渲染结果如下：

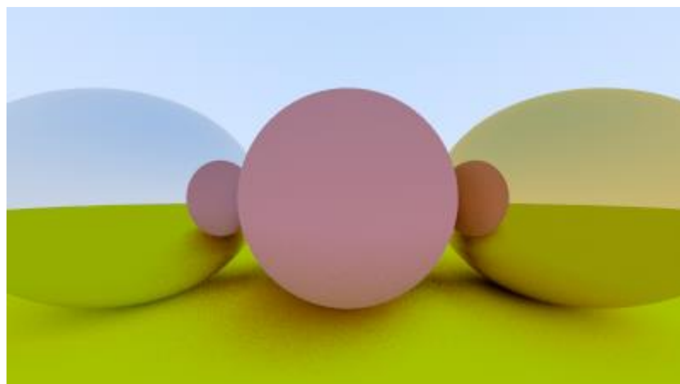

图 27：闪耀的金属球体

对于金属的模糊反射,可以通过一个小球体来为射线选择一个新的端点来随机调整反射方向;代码将从以原始端点为中心的球面上随机选择一个点,并按模糊因子缩放。

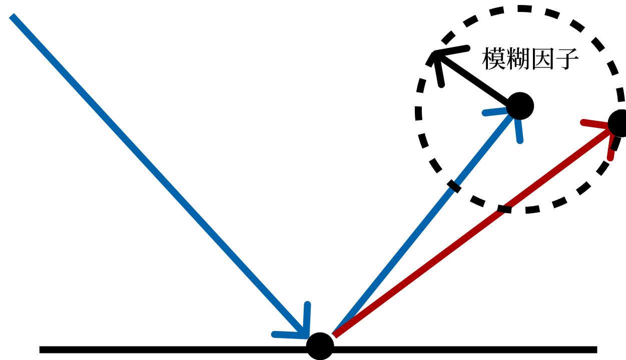

图 28:生成模糊反射光线

球体越大,反射就越模糊;这就需要添加一个模糊参数,该参数为球体的半径(因此零值不会产生扰动);问题是,对于大球体或掠过光线,可能会在表面以下散射,这时可以让表面吸收这些散射。

```
class Metal : public Material {
public:
    Metal(const Color& IAlbedo, const double& IFuzz) : Albedo(IAlbedo),
Fuzz(IFuzz < 1 ? IFuzz : 1) {
    }
    bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                 Ray& Scattered) const override {
        Vec3 Reflected = Reflect(UnitVector(LightIn.RayDirection()),
Record.NormalVector);
        Scattered     = Ray(Record.Point, Reflected + Fuzz *
RandomUnitVector());
        Attenuation    = Albedo;

        return (Dot(Scattered.RayDirection(), Record.NormalVector) > 0);
    }

private:
    Color Albedo;
    double Fuzz;
```

```
};
```

代码 64：带有模糊参数的金属球体[material.cpp]

同时也需要在 main 函数中为金属球加上模糊参数：

```
int main() {
    ...

    auto MaterialGround = std::make_shared<Lambertian>(Color(0.8, 0.8, 0.0));
    auto MaterialCenter = std::make_shared<Lambertian>(Color(0.7, 0.3, 0.3));
    auto MaterialLeft   = std::make_shared<Metal>(Color(0.8, 0.8, 0.8), 0.3);
    auto MaterialRight  = std::make_shared<Metal>(Color(0.8, 0.6, 0.2), 1.f);

    ...
}
```

代码 65：添加了模糊参数的金属球体[main.cpp]

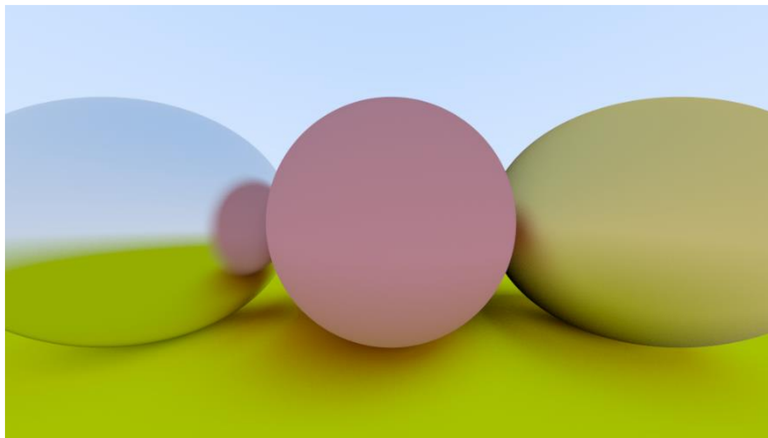

图 29：模糊金属

11. 电介质

水、玻璃和钻石等透明材料都是电介质；当一束光线击中他们的时候，其会被分为反射光线和折射（透明）光线；而渲染器将在反射和折射之间随机选择，每次相交只产生一条散射光线。

11.1 折射光

最难以调试的部分便是折射光；如果有折射光线的话，我总是通常会先让所有光线发生

折射；对于这个项目，我试着在场景中放两个玻璃球，结果是这样的（我还没有告诉你们怎么做是对怎么做是错，但很快就会告诉你们了）：

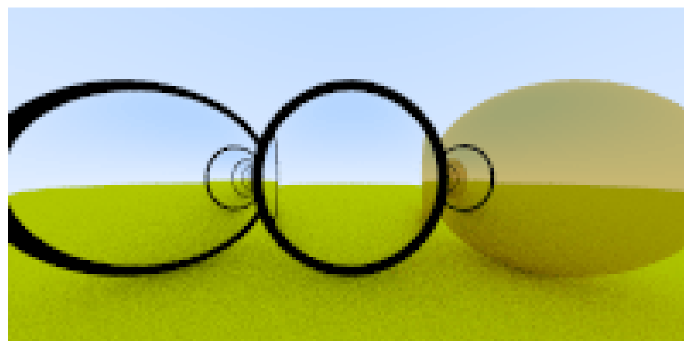

图 30：第一个玻璃球

图中的玻璃球渲染正确吗？玻璃球在现实生活中看起来确实很奇怪；但这个渲染结果是错的，玻璃球中的世界应该是颠倒的，也不应该有奇怪的黑色东西；我把射线从图像中间直接打印出来，结果明显不对。

11.2 斯涅尔定律

折射光线可以由斯涅尔定律（*Snell's Law*）描述：

$$\eta \cdot \sin\theta = \eta' \cdot \sin\theta'$$

其中θ和θ'是来自法向量的夹角，η和η'（读作“eta”和“eta prime”）为折射率（例如空气为 1.0，玻璃为 1.3 到 1.7，钻石为 2.4）；几何解释如下：

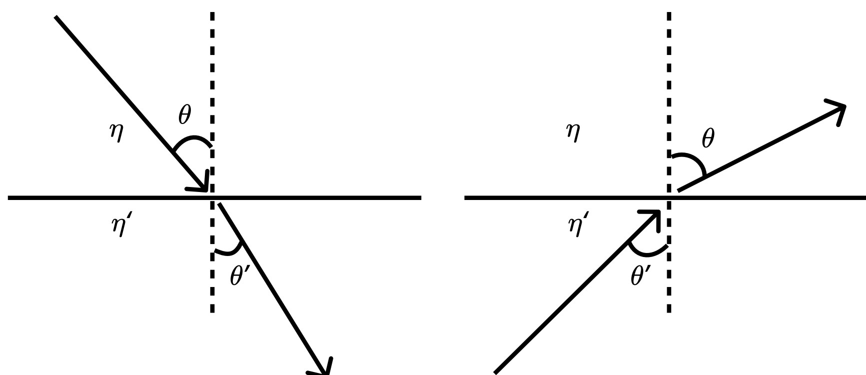

图 31：光线折射

为了确定折射光线的方向，需要解出$\sin\theta'$:

$$\sin\theta' = \frac{\eta}{\eta'} \cdot sin\theta$$

在折射面上有一条折射光线$\vec{R}'$以及法线$\vec{n}'$，其夹角为θ'；可以将$\vec{R}'$分解为两个部分，其中一个垂直于$\vec{n}'$，一个平行于$\vec{n}'$：

$$\vec{R}' = \vec{R}'_{\perp} + \vec{R}'_{\parallel}$$

随后解出$\vec{R}'_{\perp}$和$\vec{R}'_{\parallel}$:

$$\vec{R}'_{\perp} = \frac{\eta}{\eta'}\left(\vec{R} + \cos\theta \cdot \vec{n}\right)$$

$$\vec{R}'_{\parallel} = -\sqrt{1 - |\vec{R}'_{\perp}|^2} \cdot \vec{n}$$

此处读者可自行证明。

除了$\cos\theta$之外，我们知道右侧每一项的值；而两个向量的点积可以用它们之间夹角的余弦来表示:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|\cos\theta$$

如果令$\vec{a}$与$\vec{b}$为单位向量，则有:

$$\vec{a} \cdot \vec{b} = \cos\theta$$

整理得:

$$\vec{R}'_{\perp} = \frac{\eta}{\eta'}\left[\vec{R} + \left(-\vec{R} \cdot n\right) \cdot \vec{n}\right]$$

$$\vec{R}'_{\parallel} = -\sqrt{1 - |\vec{R}'_{\perp}|^2} \cdot \vec{n}$$

综上所述，便可以写一个计算$\vec{R}'$的函数:

```
    ...
Vec3 Reflect(const Vec3& Vector, const Vec3& Normal) {
    ...
}
```

```
Vec3 Refract(const Vec3& UnitVector, const Vec3& Normal, const double& Frac)
{
    auto CosTheta      = fmin(Dot(-UnitVector, Normal), 1.f);
    Vec3 ROutPerp      = Frac *  (UnitVector + CosTheta * Normal);
    Vec3 ROutParallel  = -sqrt(fabs(1.f - ROutPerp.LengthSquared())) * Normal;

    return ROutPerp + ROutParallel;
}
```

代码 66：计算折射的函数[vec3.h]

于是一个总是发生折射的电解质材质代码为：

```
class Dielectric : public Material {
public:
    Dielectric(const double& Refraction) : IndexRefraction(Refraction) {

    }
    bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                Ray& Scattered) const override {
        Attenuation = Color{ 1.f, 1.f, 1.f };
        double RefractionRatio = Record.FrontFace ? (1.f / IndexRefraction) :
IndexRefraction;

        Vec3 UnitDirection = UnitVector(LightIn.RayDirection());
        Vec3 Refracted = Refract(UnitDirection, Record.NormalVector,
RefractionRatio);

        Scattered = Ray(Record.Point, Refracted);

        return true;
    }

private:
    double IndexRefraction;
};
```

代码 67：总是发生折射的电介质材质[material.h]

然后再修改 main 函数的代码，让中间和左边的球体材质变为玻璃球材质：

```
auto MaterialGround = std::make_shared<Lambertian>(Color(0.8, 0.8, 0.0));
auto MaterialCenter = std::make_shared<Dielectric>(1.5);
auto MaterialLeft   = std::make_shared<Dielectric>(1.5);
auto MaterialRight  = std::make_shared<Metal>(Color(0.8, 0.6, 0.2), 1.f);
```

代码 68：改变球体材质[main.cpp]

最终渲染结果如下：

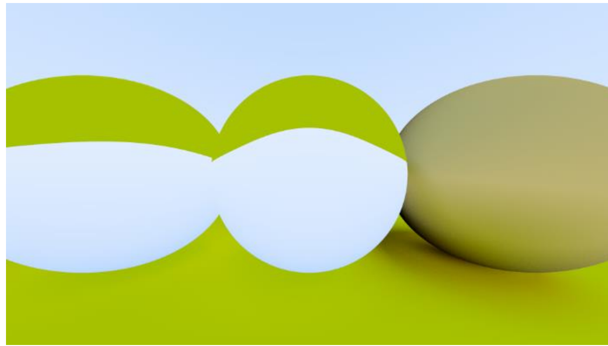

图 32：总是发生折射的玻璃球

11.3 全内反射

上节中的玻璃球起来肯定不对；一个麻烦的问题是，当光线处于折射率较高的材质中时，根据斯涅尔定律所列的方程并没有解，因此不可能发生折射；现在回过头去看斯涅尔定律和推导出的$\sin\theta'$：

$$\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

如果光线是由玻璃射向空气的话（即$\eta = 1.5$，$\eta' = 1.0$）：

$$\sin\theta' = \frac{1.5}{1.0} \cdot \sin\theta$$

$\sin\theta'$的值不可能大于 1，所以，如果有：

$$\frac{1.5}{1.0} \cdot \sin\theta > 1.0$$

等式两边的相等关系被打破，因此不可能存在解；如果不存在解，玻璃就不能折射，因

此必须反射光线:

```
if (RefractionRatio * SinTheta > 1.f) {
    // 必须反射
} else {
    // 可以折射
}
```

代码 69：判断光线是否可以折射

在这里，所有的光线都被反射，由于在实际中这通常是发生再固体物体内部，因此被称为“全内反射（*Total Internal Reflection*）”；这就是为什么有时当你浸入水中时，水气边界会像一面完美的镜子。

可以利用三角函数的性质求出$\sin\theta$：

$$\sin\theta = \sqrt{1-\cos^2\theta}$$

$$\cos\theta = \vec{R}\cdot\vec{n}$$

```
double CosTheta = fmin(Dot(-UnitDirection, Record.NormalVector), 1.f);
double SinTheta = sqrt(1.f - CosTheta * CosTheta);

if (RefractionRatio * SinTheta > 1.f) {
    // 必须反射
} else {
    // 可以折射
}
```

代码 70：判断光线是否可以折射[material.h]

而在可能的情况下，总是会发生折射的电介质材质:

```
class Dielectric : public Material {
public:
    Dielectric(const double& Refraction) : IndexRefraction(Refraction) {
    }
    bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                 Ray& Scattered) const override {
        Attenuation           = Color{1.f, 1.f, 1.f};
        double RefractionRatio = Record.FrontFace ? (1.f / IndexRefraction) :
IndexRefraction;

```

```
        Vec3 UnitDirection = UnitVector(LightIn.RayDirection());

        double CosTheta = fmin(Dot(-UnitDirection, Record.NormalVector),
1.f);
        double SinTheta = sqrt(1.f - CosTheta * CosTheta);
        Vec3 Direction;

        if (RefractionRatio * SinTheta > 1.f) {
            Direction = Reflect(UnitDirection, Record.NormalVector);
        } else {
            Direction = Refract(UnitDirection, Record.NormalVector,
RefractionRatio);
        }

        Scattered = Ray(Record.Point, Direction);

        return true;
    }

private:
    double IndexRefraction;
};
```

代码 71：带有反射的电介质材料[material.h]

衰减始终为 1，玻璃表面什么也不吸收；现在来用这些参数来试试：

```
auto MaterialGround = std::make_shared<Lambertian>(Color(0.8, 0.8, 0.0));
auto MaterialCenter = std::make_shared<Lambertian>(Color(0.1, 0.2, 0.5));
auto MaterialLeft   = std::make_shared<Dielectric>(1.5);
auto MaterialRight  = std::make_shared<Metal>(Color(0.8, 0.6, 0.2), 0.f);
```

代码 72：带有电介质球体的场景[main.cpp]

最终渲染结果如下：

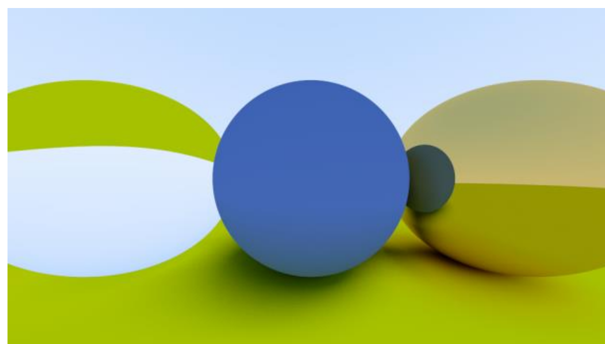

图 33：有时折射的玻璃

11.3 施利克近似法

真正的玻璃具有随角度变化的反射率—从某个角度观察窗户，它就会变成一面镜子；描述这个过程有一个很难看的大方程，但克里斯托弗施利克（Christophe Schlick）提出的一个惊人精确的多项式近似值经常被用来取代这个大方程：

$$R(\theta)=R_0+(1-R_0)(1-\cos\theta)^5$$

这样就得到了全玻璃材料：

```
class Dielectric : public Material {
public:
    Dielectric(const double& Refraction) : IndexRefraction(Refraction) {
    }
    bool Scatter(const Ray& LightIn, const HitRecord& Record, Color&
Attenuation,
                 Ray& Scattered) const override {
        Attenuation           = Color{1.f, 1.f, 1.f};
        double RefractionRatio = Record.FrontFace ? (1.f / IndexRefraction) :
IndexRefraction;

        Vec3 UnitDirection = UnitVector(LightIn.RayDirection());

        double CosTheta = fmin(Dot(-UnitDirection, Record.NormalVector),
1.f);
        double SinTheta = sqrt(1.f - CosTheta * CosTheta);
        Vec3 Direction;

        if (RefractionRatio * SinTheta > 1.f || Reflectance(CosTheta,
RefractionRatio) > RandomDouble()) {
            Direction = Reflect(UnitDirection, Record.NormalVector);
        } else {
            Direction = Refract(UnitDirection, Record.NormalVector,
RefractionRatio);
        }

        Scattered = Ray(Record.Point, Direction);

        return true;
    }

private:
```

```cpp
    // Schlick 近似法
    static double Reflectance(const double& Cosine, const double& RefIndex) {
        auto R0 = (1 - RefIndex) / (1 + RefIndex);
        R0 *= R0;

        return R0 + (1 - R0) * pow((1 - Cosine), 5);
    }

private:
    double IndexRefraction;
};
```

代码 73：完整的玻璃材质[material.h]

11.5 空心玻璃球

电介质球体的一个有趣而简单的技巧是：如果使用负半径，几何形状不受影响，但表面法线指向内侧，这可以用作制作空心玻璃球：

```cpp
World.Add(make_shared<Sphere>(Point3(0.0, -100.5, -1.0), 100.0,
MaterialGround));
World.Add(make_shared<Sphere>(Point3(0.0, 0.0, -1.0), 0.5, MaterialCenter));
World.Add(make_shared<Sphere>(Point3(-1.0, 0.0, -1.0), 0.5, MaterialLeft));
World.Add(make_shared<Sphere>(Point3(-1.0, 0.0, -1.0), -0.4, MaterialLeft));
World.Add(make_shared<Sphere>(Point3(1.0, 0.0, -1.0), 0.5, MaterialRight));
```

代码 74：有空心玻璃球的场景[main.cpp]

渲染结果如下：

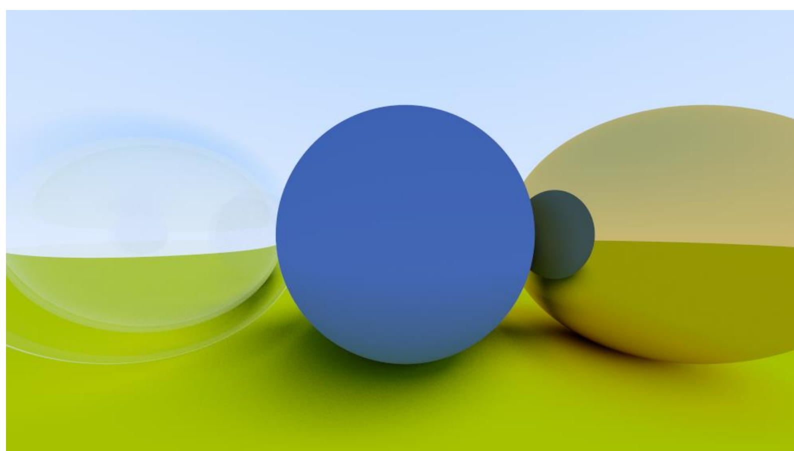

图 34：空心玻璃球

12. 可定位摄像机

摄像机，就像电介质一样，调试起来很痛苦；所以我总是循序渐进地开发；首先来看看可调整的视场（fov）；这是视觉边缘到渲染图像边缘角度；由于图像不是正方形，因此水平和垂直的视场角是不同的；我总是使用垂直视角，也通常以度为单位指定视角，然后在构造函数中改为弧度，这仅仅只是个人喜好问题。

12.1 摄像机视角几何

首先，保持光线从原点出发向平面 $z=-1$；当然也可以是平面 $z=2$ 或者其他的平面，不论怎样，只要我们使 h 成为到该距离的比例：

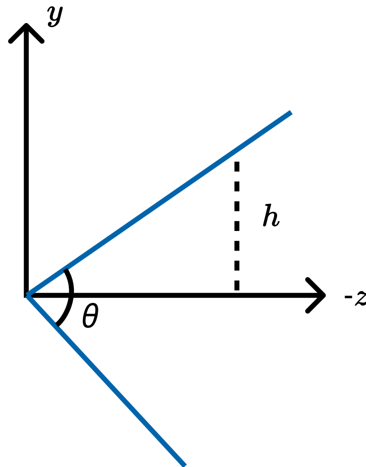

图 35：相机观察几何图形（从侧面）

于是就有 $h=\tan\left(\frac{\theta}{2}\right)$，相机的代码可以改写成：

```
class Camera {
    ...

public:
    double AspectRatio     = 16.0 / 9.0; // 图像纵高比
    int    ImageWidth      = 640;        // 图像像素宽
    int    SamplesPerPixel = 10;         // 采样率
    int    MaxDepth        = 10;         // 最大光反射数量

    double VFov = 90.f; // 垂直视角（视野）
```

```
private:
    void Initialize() {
        // 计算图像的高，并且确保其起码为一

        ImageHeight = static_cast<int>(ImageWidth / AspectRatio);
        ImageHeight = (ImageHeight < 1) ? 1 : ImageHeight;

        // 相机部分

        const auto FocalLength    = 1.0;
        const auto Theta          = DegreeToRad(VFov);
        const auto H              = tan(Theta / 2);
        const auto ViewportHeight = 2 * H * FocalLength;
        const auto ViewportWidth =
            ViewportHeight * (static_cast<double>(ImageWidth) / ImageHeight);
        CameraCenter = Point3(0, 0, 0);

        // 计算横纵向量 u 和 v

        const auto ViewportU = Vec3(ViewportWidth, 0, 0);
        const auto ViewportV = Vec3(0, -ViewportHeight, 0);

        // 由像素间距计算横纵增量向量

        PixelDeltaU = ViewportU / ImageWidth;
        PixelDeltaV = ViewportV / ImageHeight;

        // 计算左上角像素的位置

        const auto ViewportUpperLeft =
            CameraCenter - Vec3(0, 0, FocalLength) - ViewportU / 2 -
ViewportV / 2;
        Pixel100Loc = ViewportUpperLeft + 0.5 * (PixelDeltaU + PixelDeltaV);
        Graphics    = std::make_shared<Device>(ImageWidth, ImageHeight);
    }
}
```

代码 75：带有可调节垂直视野的相机

我们将使用一个简单的场景，即两个相碰的球体，以 90° 的视角来测试这些变化：

```
int main() {
    // 渲染场景

    HittableList World;
```

```
    auto R = cos(Pi / 4);

    auto MaterialLeft  = std::make_shared<Lambertian>(Color{0, 0, 1});
    auto MaterialRight = std::make_shared<Lambertian>(Color{1, 0, 0});

    World.Add(std::make_shared<Sphere>(Point3(-R, 0, -1), R, MaterialLeft));
    World.Add(std::make_shared<Sphere>(Point3(R, 0, -1), R, MaterialRight));

    Camera WorldCamera;

    WorldCamera.AspectRatio     = 16.f / 9.f;
    WorldCamera.ImageWidth      = 400;
    WorldCamera.SamplesPerPixel = 100;
    WorldCamera.MaxDepth        = 50;

    WorldCamera.VFov = 90;

    WorldCamera.Render(World);

    _getch();

    return 0;
}
```

代码 76：广角相机场景[main.cpp]

渲染结果如下：

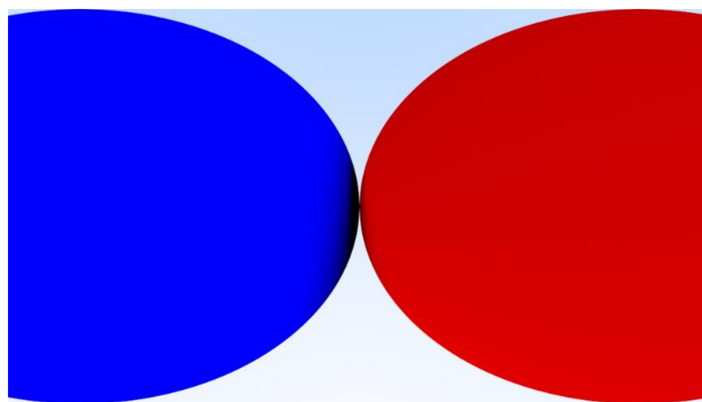

图 36：广角视野

要获得任意视角，首先要命名一些关键点；把放置摄像机的位置称为 *lookfrom*，把观察的点称为 *lookat*(稍后，如果你愿意，可以定义一个观察方向而不是观察点)。

还需要一种方法来指定摄像机的移动或侧向倾斜：即围绕 *lookat-lookfrom* 轴的旋转；带入自己想想，即使保持 *lookfrom* 和 *lookat* 轴不变，你的头部仍然可以围绕鼻子旋转；所以，需要的是一种为摄像机指定“向上”矢量的方法。

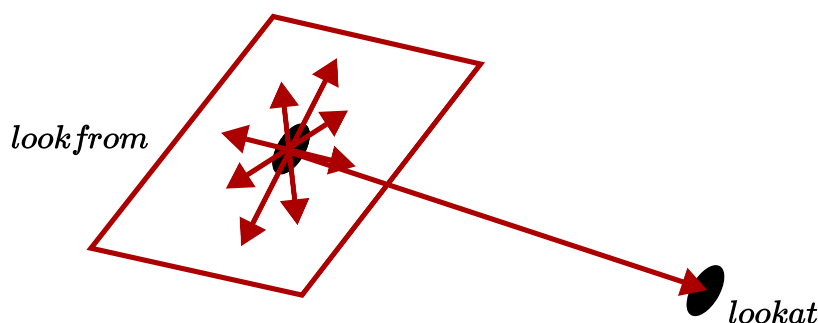

图 37：摄像机视角方向

相机可以被指定任何向上的矢量，只要其不与视图方向平行即可；将这个向上矢量投影到与视角方向正交的平面上，就得到了一个与摄像机相关的向上矢量；我习惯将其命名为“视图向上”（*vup*）向量；经过一些交叉积和矢量归一化处理后，我们现在有了一个完整的正交基(u, v, w)来描述摄像机的方向；其中u将是指向摄像机右侧的单位向量，v是指向摄像机上方的单位向量，w是与视线方向相反的单位向量（因为本文使用右手坐标），以及位于原点的相机中心。

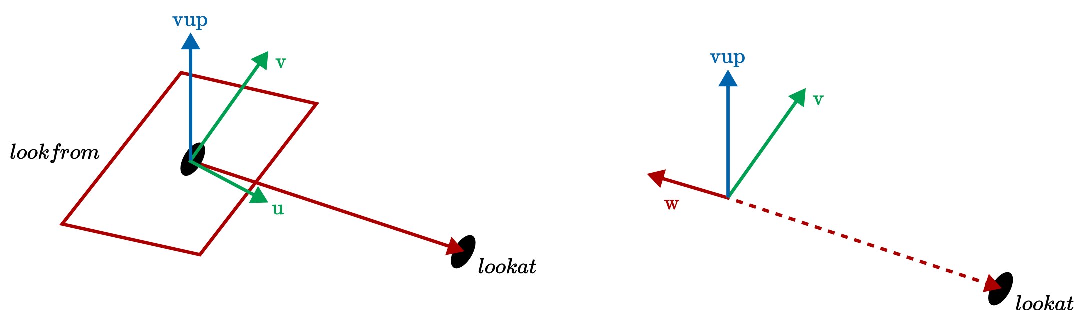

图 38：摄像机视角朝上

和之前一样，当固定摄像机朝向$-Z$时，任意视角摄像机朝向$-w$；请记住可以（不是必需的）使用$(0,1,0)$来指定 vup；这样做很方便，而且在你决定尝试疯狂的摄像机角度之前，可以很自然地保持摄像机水平：

```
class Camera {
    ...

public:
    ...

    Point3 LookFrom = Point3(0, 0, -1);
    Point3 LookAt   = Point3(0, 0, 0);
    Vec3   VUP      = Vec3(0, 1, 0);
    double VFov     = 90.f; // 垂直视角（视野）

private:
    void Initialize() {
        // 计算图像的高，并且确保其起码为一

        ImageHeight = static_cast<int>(ImageWidth / AspectRatio);
        ImageHeight = (ImageHeight < 1) ? 1 : ImageHeight;

        CameraCenter = LookFrom;

        // 相机部分

        const auto FocalLength    = (LookFrom - LookAt).Length();
        const auto Theta          = DegreeToRad(VFov);
        const auto H              = tan(Theta / 2);
        const auto ViewportHeight = 2 * H * FocalLength;
        const auto ViewportWidth =
            ViewportHeight * (static_cast<double>(ImageWidth) / ImageHeight);

        W = UnitVector(LookFrom - LookAt);
        U = UnitVector(Cross(VUP, W));
        V = Cross(W, U);

        // 计算横纵向量 u 和 v

        const auto ViewportU = ViewportWidth * U;
        const auto ViewportV = ViewportHeight * -V;
```

```
        // 由像素间距计算横纵增量向量

        PixelDeltaU = ViewportU / ImageWidth;
        PixelDeltaV = ViewportV / ImageHeight;

        // 计算左上角像素的位置

        const auto ViewportUpperLeft =
            CameraCenter - (FocalLength * W) - ViewportU / 2 - ViewportV / 2;
        Pixel100Loc = ViewportUpperLeft + 0.5 * (PixelDeltaU + PixelDeltaV);
        Graphics    = std::make_shared<Device>(ImageWidth, ImageHeight);
    }

public:
    ...

    Vec3 U;
    Vec3 V;
    Vec3 W;
};
```

代码 77：可定位和定向的摄像头[camera.h]

切换回之前的场景，并使用新的视角：

```
int main() {
    // 渲染场景

    HittableList World;

    auto MaterialGround = std::make_shared<Lambertian>(Color(0.8, 0.8, 0.0));
    auto MaterialCenter = std::make_shared<Lambertian>(Color(0.1, 0.2, 0.5));
    auto MaterialLeft   = std::make_shared<Dielectric>(1.5);
    auto MaterialRight  = std::make_shared<Metal>(Color(0.8, 0.6, 0.2), 0.f);

    World.Add(make_shared<Sphere>(Point3(0.0, -100.5, -1.0), 100.0,
MaterialGround));
    World.Add(make_shared<Sphere>(Point3(0.0, 0.0, -1.0), 0.5,
MaterialCenter));
    World.Add(make_shared<Sphere>(Point3(-1.0, 0.0, -1.0), 0.5,
MaterialLeft));
    World.Add(make_shared<Sphere>(Point3(-1.0, 0.0, -1.0), -0.4,
MaterialLeft));
    World.Add(make_shared<Sphere>(Point3(1.0, 0.0, -1.0), 0.5,
MaterialRight));
```

```
    Camera WorldCamera;

    WorldCamera.AspectRatio     = 16.f / 9.f;
    WorldCamera.ImageWidth      = 300;
    WorldCamera.SamplesPerPixel = 50;
    WorldCamera.MaxDepth        = 50;

    WorldCamera.VFov     = 90;
    WorldCamera.LookFrom = Point3(-2, 2, 1);
    WorldCamera.LookAt   = Point3(0, 0, -1);
    WorldCamera.VUP      = Vec3(0, 1, 0);

    WorldCamera.Render(World);

    _getch();

    return 0;
}
```

代码 78：另一种视角的场景[main.cpp]

渲染结果如下：

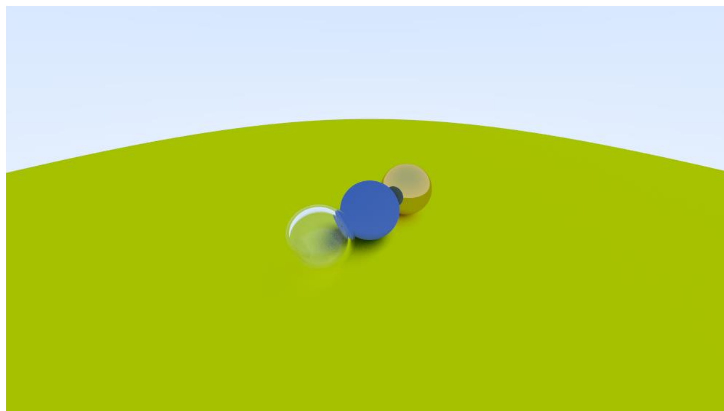

图 39：远眺视角

还可以更改视场角：

```
WorldCamera.VFov     = 20;
```

代码 79：更改视场角[main.cpp]

得到渲染结果如下：

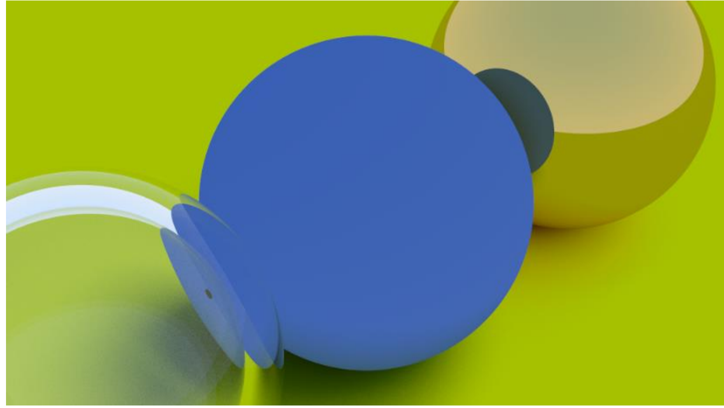

图 40：视场角放大

13. 离焦模糊

现在是最后一项功能：离焦模糊。摄影师称之为景深，但是光线追踪中使用虚焦模糊这一术语。

真实相机中之所以会出现虚焦模糊，是因为它们需要一个大洞（而不仅仅是针孔）来收集光线；一个大洞会使所有物体失焦，但如果在胶片或传感器前放置一个镜头，那么在一定距离内所有物体都会对焦，并且离这个距离越远越模糊；可以这样来理解镜头：从对焦距离上的特定点射来的所有光线，只要打到镜头上，都会被折回到图像传感器上的一个点。

通常将相机中心与所有物体都完全对焦的平面之间的距离称为焦距；但值得注意的是，对焦距离通常与焦距不同—焦距是相机中心与图像平面之间的距离；不过，在当前的模型中，这两个值是相同的，因为目前将把像素网格放在对焦平面上，也就是离相机中心的对焦距离。

在实体相机中，对焦距离由镜头和胶片或传感器之间的距离控制；这就是为什么当你改变对焦内容时，你会看到镜头相对于相机移动的原因（这也可能发生在你的手机相机中，但传感器会移动）；光圈是一个孔，可以有效控制镜头的大小；对于真实相机来说，如果你需要更多的光线，就需要把光圈开得更大，这样远离对焦距离的物体就会更模糊。对于虚拟相机来说，可以拥有一个完美的传感器，永远不需要更多的光线，因此只在需要离焦模糊时才使用光圈。

真正的相机有一个复杂的镜头；对于渲染器的代码，可以模拟这样的顺序：传感器、镜头、光圈；随后，可以计算出射线的发送位置，并在计算后翻转图像（图像在胶片上的投影是颠倒的）；然而，图形学中通常使用薄透镜近似：

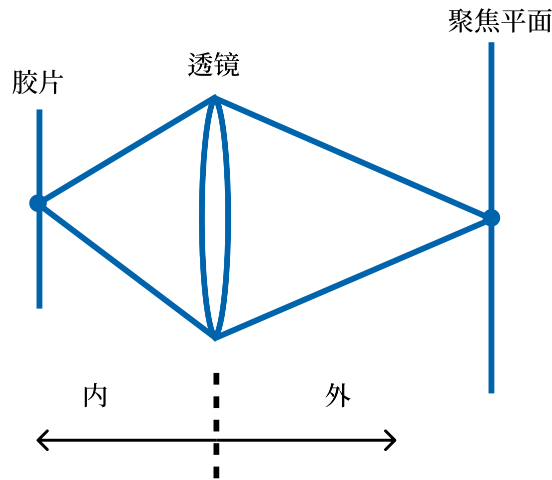

图 41：相机的薄透镜模型

渲染器不需要模拟摄像机内部的任何情况—为了渲染摄像机外的图像，这将是不必要的复杂性；相反，我通常会从一个无限细的圆形“镜头”发出光线，然后将光线射向焦点平面上（离镜头 FocalLength 远）的相关像素，在三维世界中，该平面上的所有物体都是完全对焦的。

在实际中，通过将视口放在这个平面上来实现这一点。

综上所述：

1. 对焦平面与摄像机视角方向正交
2. 对焦距离是摄像机中心与对焦平面之间的距离
3. 视口位于焦点平面上，以摄像机视图方向矢量为中心
4. 像素位置网格位于视口中（位于三维世界中）
5. 从当前像素位置周围区域随机选择图像样本位置

6. 摄像机从镜头上的随机点发射光线，穿过当前图像样本位置

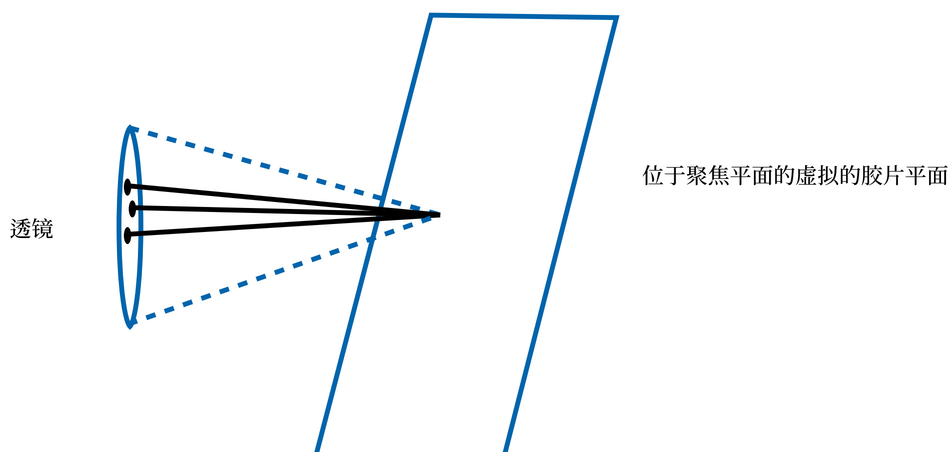

图 42：相机对焦平面

13.2 生成样本射线

在没有散焦模糊的情况下，所有的场景光线都来自摄像机中心（或 *lookfrom*）；为了实现散焦模糊，需要构建一个以摄像机中心为圆心的圆盘；半径越大，散焦模糊的程度就越高；可以把最初的摄像机想象成一个半径为零（完全不模糊）的离焦圆盘，因此所有光线都来自圆盘中心（*lookfrom*）。

那么，散焦盘应该有多大？由于这个圆盘的大小控制着离焦模糊的程度，因此它应该是摄像机类的一个参数；可以将圆盘的半径作为摄像机参数，但模糊程度会因投影距离的不同而变化；一个稍微简单的参数是指定锥体的角度，顶点位于视口中心，底点（散焦盘）位于摄像机中心；这样在改变给定镜头的对焦距离时，效果会更加一致。

渲染器将从离焦盘中随机选择点，需要一个函数来完成这一操作：RandomInUnitDisk 函；该函数与 RandomInUnitSphere 函数工作原理完全相同，不过是两个维度区别：

```
inline Vec3 UnitVector(const Vec3& Vector) {
    return Vector / Vector.Length();
}
// 生成一个离焦盘上的随机点
inline Vec3 RandomInUnitDisk() {
```

```
    while (true) {
        auto Point = Vec3(RandomDouble(-1, 1), RandomDouble(-1, 1), 0);
        if (Point.LengthSquared() < 1) {
            return Point;
        }
    }
}
```

代码 80：生成一个离焦盘上的随机点[vec3.h]

现在，来更新摄像机，让光线从散焦盘发出：

```
class Camera {
public:
    ...

public:
    ...

    double DefocusAngle  = 0;
    double FocusDistance = 10;

private:
    void Initialize() {
        // 计算图像的高，并且确保其起码为一

        ImageHeight = static_cast<int>(ImageWidth / AspectRatio);
        ImageHeight = (ImageHeight < 1) ? 1 : ImageHeight;

        CameraCenter = LookFrom;

        // 相机部分

        const auto Theta          = DegreeToRad(VFov);
        const auto H              = tan(Theta / 2);
        const auto ViewportHeight = 2 * H * FocusDistance;
        const auto ViewportWidth =
            ViewportHeight * (static_cast<double>(ImageWidth) / ImageHeight);

        W = UnitVector(LookFrom - LookAt);
        U = UnitVector(Cross(VUP, W));
        V = Cross(W, U);

        // 计算横纵向量 u 和 v
```

```
        const auto ViewportU = ViewportWidth * U;
        const auto ViewportV = ViewportHeight * -V;

        // 由像素间距计算横纵增量向量

        PixelDeltaU = ViewportU / ImageWidth;
        PixelDeltaV = ViewportV / ImageHeight;

        // 计算左上角像素的位置

        const auto ViewportUpperLeft =
            CameraCenter - (FocusDistance * W) - ViewportU / 2 - ViewportV /
2;
        Pixel100Loc = ViewportUpperLeft + 0.5 * (PixelDeltaU + PixelDeltaV);
        Graphics    = std::make_shared<Device>(ImageWidth, ImageHeight);

        auto DefocusRadius = FocusDistance * tan(DegreeToRad(DefocusAngle /
2));
        DefocusDiskU = DefocusRadius * U;
        DefocusDiskV = DefocusRadius * V;
    }

    Ray GetRay(const size_t& X, const size_t& Y, const Vec3& HeightVec) const
{
        // 从位于 (X, Y) 的像素获得获取一个随机采样光线
        auto PixelCenter = Pixel100Loc + (static_cast<double>(X) *
PixelDeltaU) + HeightVec;
        auto PixelSample = PixelCenter + PixelSampleSquare();

        const auto RayOrigin    = (DefocusAngle <= 0) ? CameraCenter :
DefocusDiskSample();
        const auto RayDirection = PixelSample - RayOrigin;

        return Ray{RayOrigin, RayDirection};
    }
    Point3 DefocusDiskSample() const {
        auto Point = RandomInUnitDisk();
        return CameraCenter + (Point[0] * DefocusDiskU) + (Point[1] *
DefocusDiskV);
    }

public:
    std::shared_ptr<Device> Graphics;
```

```
    int     ImageHeight;
    Point3 CameraCenter;

    Vec3 PixelDeltaU;
    Vec3 PixelDeltaV;
    Vec3 Pixel100Loc;

    Vec3 U;
    Vec3 V;
    Vec3 W;
    Vec3 DefocusDiskU;
    Vec3 DefocusDiskV;
};
```

代码 81：可调节景深（*dof*）的相机[camera.h]

在 main 函数中设置使用大光圈：

```
int main() {
    ...

    Camera WorldCamera;

    WorldCamera.AspectRatio     = 16.f / 9.f;
    WorldCamera.ImageWidth      = 400;
    WorldCamera.SamplesPerPixel = 100;
    WorldCamera.MaxDepth        = 50;

    WorldCamera.VFov     = 20;
    WorldCamera.LookFrom = Point3(-2, 2, 1);
    WorldCamera.LookAt   = Point3(0, 0, -1);
    WorldCamera.VUP      = Vec3(0, 1, 0);

    WorldCamera.DefocusAngle  = 10.f;
    WorldCamera.FocusDistance = 3.4;

    WorldCamera.Render(World);

    ...
}
```

代码 82：带景深的场景摄像机[main.cpp]

渲染结果如下：

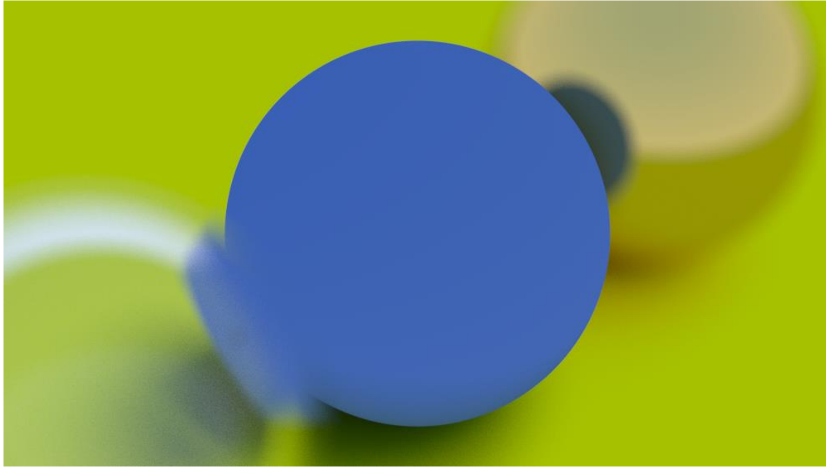

图 43：有景深的场景

14. 接下来做什么?

14.1 最后的渲染

让我们来一起把本书封面（一堆随机的球体）来渲染出来:

```
int main() {
    // 渲染场景

    HittableList World;

    auto GroundMaterial = std::make_shared<Lambertian>(Color(0.5, 0.5, 0.5));
    World.Add(std::make_shared<Sphere>(Point3(0, -1000, 0), 1000,
GroundMaterial));

    for (int a = -11; a < 11; a++) {
        for (int b = -11; b < 11; b++) {
            auto  ChooseMat = RandomDouble();
            Point3 Center(a + 0.9 * RandomDouble(), 0.2, b + 0.9 *
RandomDouble());

            if ((Center - Point3(4, 0.2, 0)).Length() > 0.9) {
                std::shared_ptr<Material> SphereMaterial;

                if (ChooseMat < 0.8) {
                    auto Albedo     = Color::Random() * Color::Random();
                    SphereMaterial = std::make_shared<Lambertian>(Albedo);
                    World.Add(std::make_shared<Sphere>(Center, 0.2,
SphereMaterial));
                } else if (ChooseMat < 0.95) {
                    auto Albedo     = Color::Random(0.5, 1);
                    auto Fuzz       = RandomDouble(0, 0.5);
                    SphereMaterial = std::make_shared<Metal>(Albedo, Fuzz);
                    World.Add(std::make_shared<Sphere>(Center, 0.2,
SphereMaterial));
                } else {
                    SphereMaterial = std::make_shared<Dielectric>(1.5);
                    World.Add(std::make_shared<Sphere>(Center, 0.2,
SphereMaterial));
                }
            }
        }
    }
```

```
    auto Material1 = std::make_shared<Dielectric>(1.5);
    World.Add(std::make_shared<Sphere>(Point3(0, 1, 0), 1.0, Material1));

    auto Material2 = std::make_shared<Lambertian>(Color(0.4, 0.2, 0.1));
    World.Add(std::make_shared<Sphere>(Point3(-4, 1, 0), 1.0, Material2));

    auto Material3 = std::make_shared<Metal>(Color(0.7, 0.6, 0.5), 0.0);
    World.Add(std::make_shared<Sphere>(Point3(4, 1, 0), 1.0, Material3));

    Camera WorldCamera;

    WorldCamera.AspectRatio     = 16.f / 9.f;
    WorldCamera.ImageWidth      = 1500;
    WorldCamera.SamplesPerPixel = 500;
    WorldCamera.MaxDepth        = 50;

    WorldCamera.VFov     = 20;
    WorldCamera.LookFrom = Point3(13, 2, 3);
    WorldCamera.LookAt   = Point3(0, 0, 0);
    WorldCamera.VUP      = Vec3(0, 1, 0);

    WorldCamera.DefocusAngle  = 0.6;
    WorldCamera.FocusDistance = 10.f;

    WorldCamera.Render(World);

    _getch();

    return 0;
}
```

代码 83：最后的场景[main.cpp]

注意，上述代码与示例代码略有不同，上述代码的采样率设置为 500，以获得质量较高的图像，于是渲染时间会比较长；示例代码中使用的值为 100，以便在开发和验证过程中保持合理的运行时间。

这是最终的渲染效果：

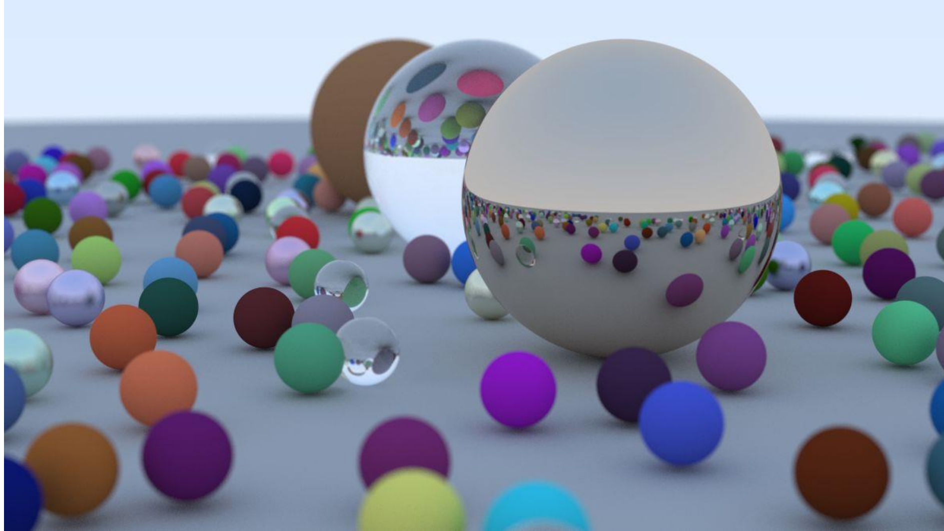

图 44：最终的场景

你可能会注意到一件有趣的事情，那就是玻璃球并没有阴影，这让它们看起来像是漂浮在空中；这并不是一个错误，在现实生活中你很少看到玻璃球，它们在现实中看起来也有点奇怪，而且在阴天的时候确实会漂浮起来；玻璃球下大球体上的一个点仍然有很多光线照射到它，因为天空是重新排序的，而不是被遮挡住了。

原文致谢内容

Original Manuscript Help

- Dave Hart
- Jean Buckley

Web Release

- Berna Kabadayı
- Lorenzo Mancini
- Lori Whippler Hollasch
- Ronald Wotzlaw

Corrections and Improvements

- Aaryaman Vasishta
- Andrew Kensler
- Antonio Gamiz
- Apoorva Joshi
- Aras Pranckevičius
- Becker
- Ben Kerl
- Benjamin Summerton
- Bennett Hardwick
- Dan Drummond
- David Chambers
- David Hart
- Dmitry Lomov
- Eric Haines
- Fabio Sancinetti
- Filipe Scur
- Frank He
- Gerrit Wessendorf
- Grue Debry
- Gustaf Waldemarson
- Ingo Wald
- Jason Stone
- JC-ProgJava
- Jean Buckley
- Joey Cho
- John Kilpatrick
- Kaan Eraslan
- Lorenzo Mancini
- Manas Kale

- Marcus Ottosson
- Mark Craig
- Matthew Heimlich
- Nakata Daisuke
- Paul Melis
- Phil Cristensen
- LollipopFt
- Ronald Wotzlaw
- Shaun P. Lee
- Shota Kawajiri
- Tatsuya Ogawa
- Thiago Ize
- Vahan Sosoyan
- Yann Herklotz
- ZeHao Chen

Special Thanks
Thanks to the team at Limnu for help on the figures.

These books are entirely written in Morgan McGuire's fantastic and free Markdeep library. To see what this looks like, view the page source from your browser.

Thanks to Helen Hu for graciously donating her https://github.com/RayTracing/ GitHub organization to this project.