주말에 레이 트레이싱

피터 셜리
스티브 홀라쉬와 트레버 데이비드 블랙 편집

버전 3.2.3, 2020-12-07

내용물

1 개요

2 이미지를 출력하세요
2.1 PPM 이미지 형식
2.2 이미지 파일 만들기
2.3 진행 지표 추가하기

3 vec3 클래스
3.1 변수와 방법
3.2 vec3 유틸리티 기능
3.3 컬러 유틸리티 기능

4개의 광선, 간단한 카메라, 그리고 배경
4.1 레이 클래스
4.2 장면으로 광선 보내기

5 구체 추가하기
5.1 광선 구체 교차로
5.2 우리의 첫 번째 광선 추적 이미지 만들기

6개의 표면 법랑과 여러 물체
6.1 표면 법으로 음영
6.2 레이스피어 교차로 코드 단순화
6.3 타격할 수 있는 물체에 대한 추상화
6.4 앞면 대 뒷면
6.5 히트블 오브젝트 목록
6.6 몇 가지 새로운 C++ 기능
6.7 공통 상수와 유틸리티 함수

7 안티앨리어싱
7.1 일부 난수 유틸리티
7.2 여러 샘플로 픽셀 생성

8개의 확산 재료
8.1 간단한 확산 재료
8.2 아동 광선의 수 제한
8.3 정확한 색 강도를 위한 감마 보정 사용
8.4 그림자 여드름 고치기
8.5 진정한 람베르티안 반사
8.6 대체 확산 제형

9 금속
9.1 재료에 대한 초록 수업
9.2 레이-객체 교차로를 설명하기 위한 데이터 구조
9.3 광 산란 및 반사율 모델링
9.4 미러링된 빛 반사
9.5 금속 구체가 있는 장면
9.6 퍼지 반사

10개의 유전체
10.1 굴절
10.2 스넬의 법칙
10.3 총 내부 반사
10.4 슐릭 근사치
10.5 중공 유리 구체 모델링

11 위치 가능한 카메라
11.1 카메라 보기 기하학
11.2 카메라 위치 지정 및 방향 지정

12 디포커스 블러
12.1 얇은 렌즈 근사치
12.2 샘플 광선 생성

13 다음은 어디야?
13.1 최종 렌더링
13.2 다음 단계

14개의 감사

15 이 책을 인용하기
15.1 기본 데이터
15.2 스니펫
15.2.1 마크다운

15.2.2 HTML
15.2.3 LaTeX와 BibTex
15.2.4 빕라텍스
15.2.5 IEEE
15.2.6 MLA:

1. 개요

나는 수년 동안 많은 그래픽 수업을 가르쳤다. 나는 종종 레이 트레이싱에서 그것들을 한다. 왜냐하면 당신은 모든 코드를 작성해야 하기 때문이다. 하지만 당신은 여전히 API 없이 멋진 이미지를 얻을 수 있다. 나는 가능한 한 빨리 당신을 멋진 프로그램으로 데려가기 위해 내 코스 노트를 사용법에 적용하기로 결정했다. 그것은 완전한 기능을 갖춘 레이 트레이서가 아닐 것이지만, 레이 트레이싱을 영화에서 필수품으로 만든 간접 조명을 가지고 있다. 다음 단계를 따르세요, 그리고 당신이 생산하는 레이 트레이서의 구조는 당신이 흥분하고 그것을 추구하고 싶다면 더 광범위한 레이 트레이서로 확장하는 데 좋을 것입니다.

누군가가 "레이 트레이싱"이라고 말할 때 그것은 많은 것을 의미할 수 있다. 내가 설명할 것은 기술적으로 경로 추적자이며, 상당히 일반적인 것이다. 코드는 꽤 간단할 거야 (컴퓨터가 일을 하게 하게 와! 나는 네가 만들 수 있는 이미지에 매우 만족할 거라고 생각해.

I'll take you through writing a ray tracer in the order I do it, along with some debugging tips. By the end, you will have a ray tracer that produces some great images. You should be able to do this in a weekend. If you take longer, don't worry about it. I use C++ as the driving language, but you don't need to. However, I suggest you do, because it's fast, portable, and most production movie and video game renderers are written in C++. Note that I avoid most "modern features" of C++, but inheritance and operator overloading are too useful for ray tracers to pass on. I do not provide the code online, but the code is real and I show all of it except for a few straightforward operators in the `vec3` class. I am a big believer in typing in code to learn it, but when code is available I use it, so I only practice what I preach when the code is not available. So don't ask!

나는 그 마지막 부분을 남겼어. 왜냐하면 내가 180을 한 것이 웃기기 때문이야. 몇몇 독자들은 우리가 코드를 비교할 때 도움이 된 미묘한 오류로 끝났다. 그러니 코드를 입력해 주세요, 하지만 제 코드를 보고 싶다면 다음과 같습니다:

https://github.com/RayTracing/raytracing.github.io/

나는 벡터에 조금 익숙하다고 가정한다(점 곱과 벡터 추가와 같은). 만약 당신이 그것을 모른다면, 약간의 검토를 하세요. 그 리뷰가 필요하거나 처음으로 배우고 싶다면, Marschner와 내 그래픽 텍스트, Foley, Van Dam *등*, 또는 McGuire의 그래픽 코덱스를 확인하세요.

만약 당신이 곤경에 처하거나, 누군가에게 보여주고 싶은 멋진 일을 한다면, ptrshrl@gmail.com으로 이메일을 보내주세요.

나는 이 책과 관련된 블로그 https://in1weekend.blogspot.com/에서 추가 읽기와 리소스 링크를 포함하여 책과 관련된 사이트를 유지할 것이다.

이 프로젝트에 도움을 주신 모든 분들께 감사드립니다. 당신은 이 책의 끝에 있는 인정 섹션에서 그것들을 찾을 수 있습니다.

계속하자!

2. 이미지를 출력하세요

2.1. PPM 이미지 포맷

렌더러를 시작할 때마다, 이미지를 볼 수 있는 방법이 필요합니다. 가장 간단한 방법은 그것을 파일에 쓰는 것이다. 문제는, 너무 많은 형식이 있다는 것이다. 그것들 중 다수는 복잡하다. 나는 항상 일반 텍스트 ppm 파일로 시작한다. 여기 위키피디아의 좋은 설명이 있습니다:

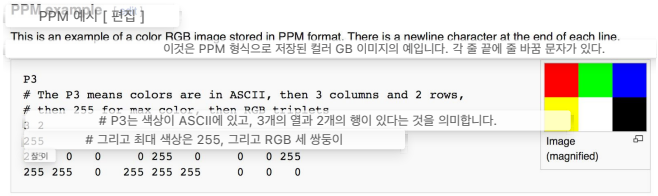


그림 1: *PPM 예시*

그런 것을 출력하기 위해 C++ 코드를 만들어 봅시다:

```
#include <iostream>

int main() {

    // Image

    const int image_width = 256;
    const int image_height = 256;

    // Render

    std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        for (int i = 0; i < image_width; ++i) {
            auto r = double(i) / (image_width-1);
            auto g = double(j) / (image_height-1);
            auto b = 0.25;

            int ir = static_cast<int>(255.999 * r);
            int ig = static_cast<int>(255.999 * g);
            int ib = static_cast<int>(255.999 * b);

            std::cout << ir << ' ' << ig << ' ' << ib << '\n';
        }
    }
}
```

Listing 1: `[main.cc]` *Creating your first image*

그 코드에는 몇 가지 주의해야 할 사항이 있습니다:

1. 픽셀은 왼쪽에서 오른쪽으로 픽셀이 있는 행으로 기록됩니다.
2. 행은 위에서 아래로 쓰여진다.
3. 관례에 따라, 각 빨간색/녹색/파란색 구성 요소는 0.0에서 1.0까지 다양합니다. 우리는 나중에 내부적으로 높은 다이내믹 레인지를 사용할 때 그것을 완화할 것이지만, 출력 전에 톤 맵을 0에서 1 범위로 매핑할 것이므로 이 코드는 변경되지 않을 것입니다.
4. 빨간색은 왼쪽에서 오른쪽으로 완전히 꺼짐(검은색)에서 완전히 켜짐(밝은 빨간색)으로, 녹색은 하단의 검은색에서 상단에서 완전히 켜짐으로 간다. 빨간색과 녹색은 함께 노란색을 만들기 때문에 우리는 오른쪽 상단 모서리가 노란색이 될 것으로 예상해야 한다.

2.2. 이미지 파일 만들기

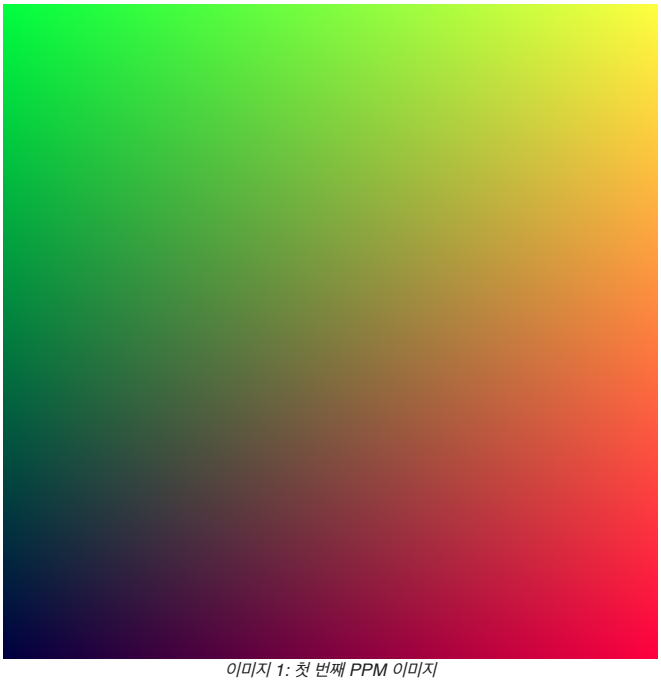
Because the file is written to the program output, you'll need to redirect it to an image file. Typically this is done from the command-line by using the `>` redirection operator, like so:

```
build\Release\inOneWeekend.exe > image.ppm
```

이것이 윈도우에서 사물이 어떻게 보일지이다. 맥이나 리눅스에서, 그것은 다음과 같이 보일 것이다:

```
build/inOneWeekend > image.ppm
```

출력 파일(Mac의 `ToyViewer`, 좋아하는 뷰어에서 시도하고 시청자가 지원하지 않는 경우 Google "ppm 뷰어"에서 시도하십시오)를 열면 다음 결과가 표시됩니다.



이미지 1: 첫 번째 PPM 이미지

만세! 이것은 "안녕하세요 세계" 그래픽입니다. 이미지가 그렇게 보이지 않는다면, 텍스트 편집기에서 출력 파일을 열고 어떻게 생겼는지 보세요. 그것은 다음과 같이 시작해야 한다:

```
P3
256 256
255
0 255 63
1 255 63
2 255 63
3 255 63
4 255 63
5 255 63
6 255 63
7 255 63
8 255 63
9 255 63
...
```

목록 2: *첫 번째 이미지 출력*

그렇지 않다면, 당신은 아마도 이미지 리더를 혼란스럽게 하는 몇 가지 줄 바꿈이나 이와 유사한 것을 가지고 있을 것입니다.

PPM보다 더 많은 이미지 유형을 생성하고 싶다면, 저는 https://github.com/nothings/stb의 GitHub에서 사용할 수 있는 헤더 전용 이미지 라이브러리인 `stb_image.h` 팬입니다.

2.3. 진행 지표 추가하기

계속하기 전에, 출력에 진행 지표를 추가해 봅시다. 이것은 긴 렌더링의 진행 상황을 추적하고, 인피니트루프나 다른 문제로 인해 멈춘 실행을 식별할 수 있는 편리한 방법입니다.

우리 프로그램은 이미지를 표준 출력 스트림(`std::cout`)으로 출력하므로 그대로 두고 대신 오류 출력 스트림(`std::cerr`)에 쓰세요.

```
    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            auto r = double(i) / (image_width-1);
            auto g = double(j) / (image_height-1);
            auto b = 0.25;

            int ir = static_cast<int>(255.999 * r);
            int ig = static_cast<int>(255.999 * g);
            int ib = static_cast<int>(255.999 * b);

            std::cout << ir << ' ' << ig << ' ' << ib << '\n';
        }
    }

    std::cerr << "\nDone.\n";
```

Listing 3: `[main.cc]` *Main render loop with progress reporting*

3. vec3 수업

Almost all graphics programs have some class(es) for storing geometric vectors and colors. In many systems these vectors are 4D (3D plus a homogeneous coordinate for geometry, and RGB plus an alpha transparency channel for colors). For our purposes, three coordinates suffices. We'll use the same class `vec3` for colors, locations, directions, offsets, whatever. Some people don't like this because it doesn't prevent you from doing something silly, like adding a color to a location. They have a good point, but we're going to always take the "less code" route when not obviously wrong. In spite of this, we do declare two aliases for `vec3`: `point3` and `color`. Since these two types are just aliases for `vec3`, you won't get warnings if you pass a `color` to a function expecting a `point3`, for example. We use them only to clarify intent and use.

3.1. 변수와 방법

Here's the top part of my `vec3` class:

```
#ifndef VEC3_H
#define VEC3_H

#include <cmath>
#include <iostream>

using std::sqrt;

class vec3 {
    public:
        vec3() : e{0,0,0} {}
        vec3(double e0, double e1, double e2) : e{e0, e1, e2} {}

        double x() const { return e[0]; }
        double y() const { return e[1]; }
        double z() const { return e[2]; }

        vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
        double operator[](int i) const { return e[i]; }
        double& operator[](int i) { return e[i]; }

        vec3& operator+=(const vec3 &v) {
            e[0] += v.e[0];
            e[1] += v.e[1];
            e[2] += v.e[2];
            return *this;
        }

        vec3& operator*=(const double t) {
            e[0] *= t;
            e[1] *= t;
            e[2] *= t;
            return *this;
        }

        vec3& operator/=(const double t) {
            return *this *= 1/t;
        }

        double length() const {
            return sqrt(length_squared());
        }

        double length_squared() const {
            return e[0]*e[0] + e[1]*e[1] + e[2]*e[2];
        }

    public:
        double e[3];
};

// Type aliases for vec3
using point3 = vec3;   // 3D point
using color = vec3;    // RGB color

#endif
```

Listing 4: `[vec3.h]` *vec3 class*

We use `double` here, but some ray tracers use `float`. Either one is fine — follow your own tastes.

3.2. vec3 유틸리티 기능

헤더 파일의 두 번째 부분에는 벡터 유틸리티 기능이 포함되어 있습니다:

```
// vec3 Utility Functions

inline std::ostream& operator<<(std::ostream &out, const vec3 &v) {
    return out << v.e[0] << ' ' << v.e[1] << ' ' << v.e[2];
}

inline vec3 operator+(const vec3 &u, const vec3 &v) {
    return vec3(u.e[0] + v.e[0], u.e[1] + v.e[1], u.e[2] + v.e[2]);
}

inline vec3 operator-(const vec3 &u, const vec3 &v) {
    return vec3(u.e[0] - v.e[0], u.e[1] - v.e[1], u.e[2] - v.e[2]);
}

inline vec3 operator*(const vec3 &u, const vec3 &v) {
    return vec3(u.e[0] * v.e[0], u.e[1] * v.e[1], u.e[2] * v.e[2]);
}

inline vec3 operator*(double t, const vec3 &v) {
    return vec3(t*v.e[0], t*v.e[1], t*v.e[2]);
}

inline vec3 operator*(const vec3 &v, double t) {
    return t * v;
}

inline vec3 operator/(vec3 v, double t) {
    return (1/t) * v;
}

inline double dot(const vec3 &u, const vec3 &v) {
    return u.e[0] * v.e[0]
         + u.e[1] * v.e[1]
         + u.e[2] * v.e[2];
}

inline vec3 cross(const vec3 &u, const vec3 &v) {
    return vec3(u.e[1] * v.e[2] - u.e[2] * v.e[1],
                u.e[2] * v.e[0] - u.e[0] * v.e[2],
                u.e[0] * v.e[1] - u.e[1] * v.e[0]);
}

inline vec3 unit_vector(vec3 v) {
    return v / v.length();
}
```

Listing 5: `[vec3.h]` *vec3 utility functions*

3.3. 컬러 유틸리티 기능

Using our new `vec3` class, we'll create a utility function to write a single pixel's color out to the standard output stream.

```
#ifndef COLOR_H
#define COLOR_H

#include "vec3.h"

#include <iostream>

void write_color(std::ostream &out, color pixel_color) {
    // Write the translated [0,255] value of each color component.
    out << static_cast<int>(255.999 * pixel_color.x()) << ' '
        << static_cast<int>(255.999 * pixel_color.y()) << ' '
        << static_cast<int>(255.999 * pixel_color.z()) << '\n';
}

#endif
```

Listing 6: `[color.h]` *color utility functions*

이제 우리는 이것을 사용하기 위해 메인을 바꿀 수 있습니다:

```
#include "color.h"
#include "vec3.h"

#include <iostream>

int main() {

    // Image

    const int image_width = 256;
    const int image_height = 256;

    // Render

    std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(double(i)/(image_width-1), double(j)/(image_height-1), 0.25);
            write_color(std::cout, pixel_color);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 7: `[main.cc]` *Final code for the first PPM image*

4. 광선, 간단한 카메라, 그리고 배경

4.1. 레이 클래스

모든 광선 추적자가 가지고 있는 한 가지는 광선 클래스와 광선을 따라 어떤 색이 보이는지에 대한 계산이다. 광선을 함수로 생각하자. $\mathbf{P}(t)=A+tb$. 여기$\mathbf{P}$3D의 선을 따라 3D 위치입니다.$\mathbf{A}$광선의 기원과$\mathbf{b}$광선 방향이야. 광선 매개 변수t실수입니다 (코드의 `double`). 다른 것을 꽂으세요.t, 리고$\mathbf{P}(t)$광선을 따라 지점을 움직인다. 부정적인 것을 추가하세요.t, 치와 당신은 3D 라인의 어느 곳이든 갈 수 있습니다. 긍정적인 것을 위해t, 당신은 앞에 있는 부분만 얻을 수 있습니다.$\mathbf{A}$그리고 이것은 종종 하프라인 또는 광선이라고 불리는 것이다.

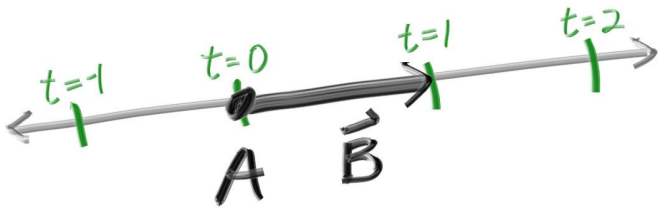


그림 2: *선형 보간*

기능$\mathbf{P}(t)$더 자세한 코드 형태로 나는 `ray::at(t)`라고 부른다:

```
#ifndef RAY_H
#define RAY_H

#include "vec3.h"

class ray {
    public:
        ray() {}
        ray(const point3& origin, const vec3& direction)
            : orig(origin), dir(direction)
        {}

        point3 origin() const  { return orig; }
        vec3 direction() const { return dir; }

        point3 at(double t) const {
            return orig + t*dir;
        }

    public:
        point3 orig;
        vec3 dir;
};

#endif
```

Listing 8: `[ray.h]` *The ray class*

4.2. 장면으로 광선 보내기

Now we are ready to turn the corner and make a ray tracer. At the core, the ray tracer sends rays through pixels and computes the color seen in the direction of those rays. The involved steps are (1) calculate the ray from the eye to the pixel, (2) determine which objects the ray intersects, and (3) compute a color for that intersection point. When first developing a ray tracer, I always do a simple camera for getting the code up and running. I also make a simple `ray_color(ray)` function that returns the color of the background (a simple gradient).

나는 종종 디버깅을 위해 정사각형 이미지를 사용하는 데 어려움을 겪었다. 왜냐하면 나는 전치하기 때문이다.x그리고y너무 자주, 그래서 나는 정사각형이 아닌 이미지를 사용할 것이다. 지금은 16:9 종횡비를 사용할 것이다. 왜냐하면 그것은 매우 일반적이기 때문이다.

렌더링된 이미지의 픽셀 크기를 설정하는 것 외에도, 우리는 장면 광선을 통과할 수 있는 가상 뷰포트를 설정해야 합니다. 표준 정사각형 픽셀 간격의 경우, 뷰포트의 종횡비는 렌더링된 이미지와 동일해야 합니다. 우리는 높이의 뷰포트 두 유닛을 고를 것이다. 우리는 또한 투영면과 투영점 사이의 거리를 한 단위로 설정할 것이다. 이것은 "초점 길이"라고 불리며, 나중에 발표할 "초점 거리"와 혼동해서는 안 됩니다.

내가 "눈"(또는 카메라를 생각한다면 카메라 센터)을 놓을게$(0,0,0)$. 나는 y축이 올라가고, x축이 오른쪽으로 올라가게 할 것이다. 오른손잡이 좌표계의 관습을 존중하기 위해, 화면에는 음의 z축이 있다. 나는 왼쪽 상단 모서리에서 화면을 가로질러, 화면 측면을 따라 두 개의 오프셋 벡터를 사용하여 화면을 가로질러 광선 끝점을 이동할 것이다. 나는 그렇게 하지 않으면 더 간단하고 약간 더 빠른 코드를 만들 수 있다고 생각하기 때문에 광선 방향을 단위 길이 벡터로 만들지 않는다는 점에 유의하십시오.

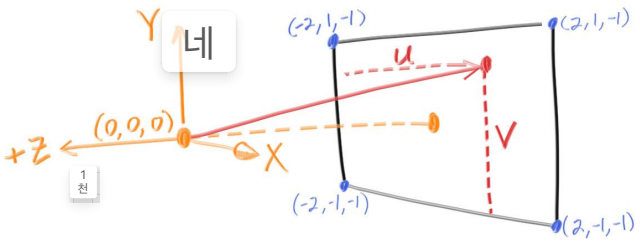


그림 3: *카메라 기하학*

코드에서, 레이 `r` 대략 픽셀 센터로 갑니다 (나중에 안티앨리어싱을 추가할 것이기 때문에 지금은 정확성에 대해 걱정하지 않을 것입니다):

```
#include "color.h"
#include "ray.h"
#include "vec3.h"

#include <iostream>

color ray_color(const ray& r) {
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}

int main() {

    // Image
    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);

    // Camera

    auto viewport_height = 2.0;
    auto viewport_width = aspect_ratio * viewport_height;
    auto focal_length = 1.0;

    auto origin = point3(0, 0, 0);
    auto horizontal = vec3(viewport_width, 0, 0);
    auto vertical = vec3(0, viewport_height, 0);
    auto lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0, focal_length);

    // Render

    std::cout << "P3\n" << image_width << " " << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            auto u = double(i) / (image_width-1);
            auto v = double(j) / (image_height-1);
            ray r(origin, lower_left_corner + u*horizontal + v*vertical - origin);
            color pixel_color = ray_color(r);
            write_color(std::cout, pixel_color);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 9: `[main.cc]` *Rendering a blue-to-white gradient*

The `ray_color(ray)` function linearly blends white and blue depending on the height of the 에광선 방향을 단위 길이로 확장한 후 좌표를 조정하세요 (그래서$-1.0 < y < 1.0$). 왜냐하면 우리는 보고 있기 때문이야에퀘터를 정규화한 후 높이, 수직 그라디언트 외에도 색상에 대한 수평 그라디언트를 알 수 있습니다.

그리고 나서 나는 그것을 스케일링하는 표준 그래픽 트릭을 했다.$0.0 \le t \le 1.0$. 언제$t = 1.0$ 나는 파란색을 원해. 언제$t = 0.0$ 난 흰색을 원해. 그 사이에, 나는 블렌드를 원해. 이것은 두 가지 사이에 "선형 혼합" 또는 "선형간풀" 또는 줄여서 "lerp"를 형성한다. lerp는 항상 형태이다.

$$\text{혼합값} = (1 - t) \cdot \text{startValue} + t \cdot \text{endValue},$$

와 함께t에서 1로 가고 있어. 우리의 경우 이것은 다음을 생산한다:

이미지 2: 광선 Y 좌표에 따른 파란색에서 흰색으로의 그라디언트

5. 구체 추가하기

레이 트레이서에 하나의 물체를 추가합시다. 광선이 구체에 부딪히는지 여부를 계산하는 것은 꽤 간단하기 때문에 사람들은 종종 광선 추적기에서 구체를 사용한다.

5.1. 레이-스피어 교차로

반지름의 원점을 중심으로 한 구의 방정식을 기억하세요.R 이다$x^2 + 예^2 + z^2 = R^2$. 다른 방법을 말해봐, 만약 주어진 지점이라면 (x,y,z) 구체 위에 있어, 그럼$x^2 + 예^2 + z^2 = R^2$. 만약 주어진 지점이라면(x,y,z) 구체 *안에* 있어, 그럼$x^2 + 예^2 + z^2 < R^2$, 그리고 주어진 지점이(x,y,z) 그럼, 구체 *밖에* 있어$x^2 + 예^2 + z^2 > R^2$.

구체 중심이 있으면 더 못생겼어$(C_x, C_예, C_z)$:

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

In graphics, you almost always want your formulas to be in terms of vectors so all the x/y/z stuff is under the hood in the `vec3` class. You might note that the vector from center $\mathbf{C} = (C_x, C_예, C_z)$ 가리키다$\mathbf{P} = (x,y,z)$ 이다$(\mathbf{P}-\mathbf{C})$, 그리고 그러므로

$$(\mathbf{P}-\mathbf{C})\cdot(\mathbf{P}-\mathbf{C}) = (x - C_x)^2 + (y - C_예)^2 + (z - C_z)^2$$

그래서 벡터 형태의 구의 방정식은 다음과 같다:

$$(\mathbf{P}-\mathbf{C})\cdot(\mathbf{P}-\mathbf{C}) = r^2$$

우리는 이것을 "어떤 지점이든"으로 읽을 수 있다.$\mathbf{P}$이 방정식을 만족시키는 것은 구체에 있다. 우리는 우리의 광선이 있는지 알고 싶어요. $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$ 어디든 구체를 치는 적이 있다. 만약 그것이 구체에 부딪히면, 몇 가지가 있다.t너는 것을 위해$\mathbf{P}(t)$구체 방정식을 만족시킨다. 그래서 우리는 아무거나 찾고 있어t에게 사실인 곳:

$$(\mathbf{P}(t)-\mathbf{C})\cdot(\mathbf{P}(t)-\mathbf{C}) = r^2$$

또는 광선의 전체 형태를 확장하는 것$\mathbf{P}(t)$:

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

벡터 대수학의 규칙은 우리가 여기서 원하는 전부이다. 우리가 그 방정식을 확장하고 모든 용어를 왼쪽으로 옮기면 다음을 얻을 수 있습니다:

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

벡터와r그 방정식은 모두 일정하고 알려져 있다. 알 수 없는 것은t,그리고 그 방정식은 당신이 고등학교 수학 수업에서 본 것처럼 이차입니다. 넌 해결할 수 있어t,그리고 긍정적(두 개의 실제 해결책을 의미함), 부정적(실제 해결책 없음을 의미함) 또는 0(한 개의 실제 해결책을 의미함)인 제곱근 부분이 있습니다. 그래픽에서, 대수학은 거의 항상 기하학과 매우 직접적으로 관련이 있다. 우리가 가진 것은:

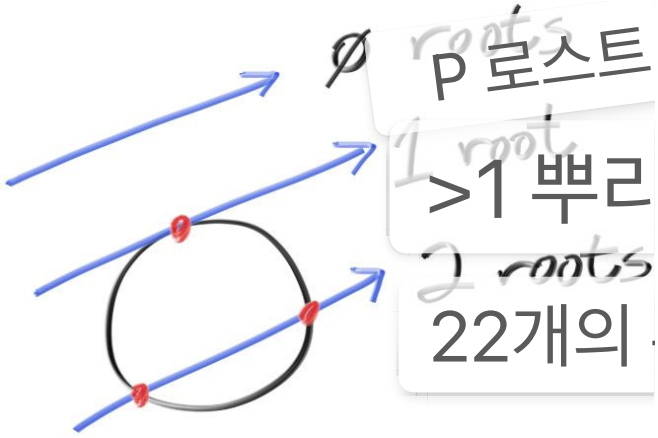


그림 4: *광선-구 교차 결과*

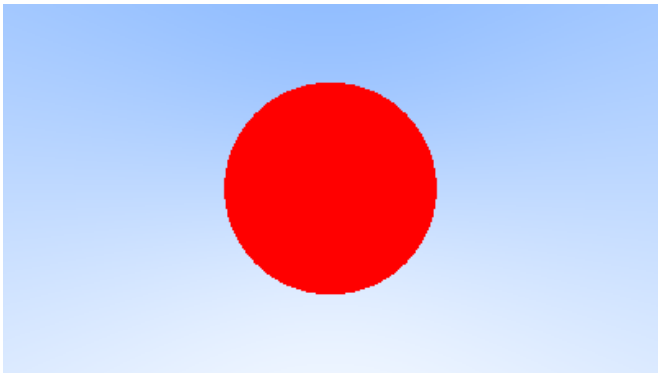
5.2. 우리의 첫 번째 광선 추적 이미지 만들기

우리가 그 수학을 가지고 그것을 프로그램에 하드 코딩한다면, 우리는 z축의 -1에 배치한 작은 구에 부딪히는 모든 픽셀을 빨간색으로 색칠하여 테스트할 수 있습니다.

```
bool hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = dot(r.direction(), r.direction());
    auto b = 2.0 * dot(oc, r.direction());
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;
    return (discriminant > 0);
}

color ray_color(const ray& r) {
    if (hit_sphere(point3(0,0,-1), 0.5, r))
        return color(1, 0, 0);
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 10: `[main.cc]` *Rendering a red sphere*

우리가 얻는 것은 이것입니다:



이미지 3: 단순한 붉은 구체

이제 이것은 음영과 반사 광선, 하나 이상의 물체와 같은 모든 종류의 것들이 부족하지만, 우리는 시작보다 절반에 가까워졌습니다! 알아야 할 한 가지는 광선이 구체에 부딪히는지 여부를 테스트했다는 것이다.$t< 0$ 해결책은 잘 작동해. 만약 당신이 구체 중심을 바꾸면 $z=+1$ 당신은 당신 뒤에 있는 것들을 보기 때문에 정확히 같은 사진을 얻을 것입니다. 이건 특징이 아니야! 우리는 그 문제들을 다음에 해결할 것이다.

6. 표면 법랑과 여러 물체

6.1. 표면 노멀을 사용한 음영

먼저, 우리가 그늘을 칠 수 있도록 표면을 정상으로 만들자. 이것은 교차점의 표면에 수직인 벡터이다. 정상을 위해 내려야 할 두 가지 디자인 결정이 있다. 첫 번째는 이 법전체가 단위 길이인지 여부이다. 그것은 음영에 편리하기 때문에 나는 그렇다고 말할 것이지만, 나는 코드에서 그것을 시행하지 않을 것이다. 이것은 미묘한 버그를 허용할 수 있으므로, 대부분의 디자인 결정과 마찬가지로 개인적인 취향이라는 것을 알아두세요. 구의 경우, 바깥쪽 법면은 중심을 뺀 히트 포인트의 방향에 있다:

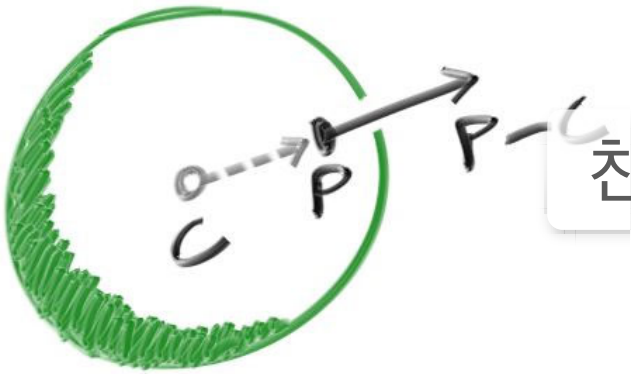


그림 5: *구 표면 정규 기하학*

지구에서, 이것은 지구의 중심에서 당신에게 이르는 벡터가 똑바로 뽀다는 것을 의미한다. 지금 그것을 코드에 던지고, 음영 처리하자. 우리는 아직 조명이나 아무것도 가지고 있지 않으니, 컬러 맵으로 정상을 시각화해 봅시다. 노멀을 시각화하는 데 사용되는 일반적인 트릭 (왜냐하면 가정하기 쉽고 다소 직관적이기 때문에)**n**단위 길이 벡터이므로 각 구성 요소는 -1과 1) 사이에 있으며, 각 구성 요소를 0에서 1까지의 간격에 매핑한 다음 x/y/z를 r/g/b에 매핑합니다. 정상을 위해, 우리는 우리가 쳤든 안 쳤든뿐만 아니라 히트 포인트가 필요하다. 우리는 장면에 하나의 구만 있고, 그것은 카메라 바로 앞에 있으므로, 우리는 음수에 대해 걱정하지 않을 것입니다.더[]직. 우리는 가장 가까운 히트 포인트를 가정할 거야 (가장 작은타) 코드의 이러한 변화는 우리가 계산하고 시각화할 수 있게 해준다.**n**:

```
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = dot(r.direction(), r.direction());
    auto b = 2.0 * dot(oc, r.direction());
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;
    if (discriminant < 0) {
        return -1.0;
    } else {
        return (-b - sqrt(discriminant) ) / (2.0*a);
    }
}

color ray_color(const ray& r) {
    auto t = hit_sphere(point3(0,0,-1), 0.5, r);
    if (t > 0.0) {
        vec3 N = unit_vector(r.at(t) - vec3(0,0,-1));
        return 0.5*color(N.x()+1, N.y()+1, N.z()+1);
    }
    vec3 unit_direction = unit_vector(r.direction());
    t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 11: `[main.cc]` *Rendering surface normals on a sphere*

그리고 그것은 이 그림을 산출한다:

이미지 4: 일반 벡터에 따라 색칠된 구

6.2. 레이-스피어 교차로 코드 단순화

광선-구 방정식을 다시 살펴봅시다:

```
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = dot(r.direction(), r.direction());
    auto b = 2.0 * dot(oc, r.direction());
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;

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

Listing 12: `[main.cc]` *Ray-sphere intersection code (before)*

먼저, 점선으로 점선된 벡터가 그 벡터의 제곱 길이와 같다는 것을 기억하세요.

둘째, b 방정식이 어떻게 2의 인수를 갖는지 주목하세요. 이차 방정식에 무슨 일이 일어나는지 생각해 보세요.$b = 2h$:

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

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

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

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

이러한 관찰을 사용하여, 우리는 이제 구 교차 코드를 이렇게 단순화할 수 있습니다:

```
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = r.direction().length_squared();
    auto half_b = dot(oc, r.direction());
    auto c = oc.length_squared() - radius*radius;
    auto discriminant = half_b*half_b - a*c;

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

Listing 13: `[main.cc]` *Ray-sphere intersection code (after)*

6.3. 타격할 수 있는 물체에 대한 추상화

이제, 몇몇 구체는 어때? 구체의 배열을 갖는 것은 유혹적이지만, 매우 깨끗한 해결책은 광선이 칠 수 있는 모든 것에 대해 "추상적인 클래스"를 만들고, 구체와 구체 목록을 모두 당신이 칠 수 있는 것으로 만드는 것이다. 그 클래스가 불려야 하는 것은 곤경의 일이다 - "객체"라고 부르는 것은 "객체 지향" 프로그래밍이 아니라면 좋을 것이다. "표면"은 종종 사용되며, 약점은 우리가 볼륨을 원할 수도 있다는 것이다. "히트테이블"은 그들을 하나로 묶는 멤버 기능을 강조한다. 나는 이것들 중 어느 것도 좋아하지 않지만, "히터블"으로 갈 것이다.

This `hittable` abstract class will have a hit function that takes in a ray. Most ray tracers have found it convenient to add a valid interval for hits t_{min} 에 t_{max}, 그래서 히트는 "카운트"만 t_{min} <t < t_{max} . 초기 광선에 대해 이것은 긍정적이다.티하지만 우리가 볼 수 있듯이, 코드의 몇 가지 세부 사항이 간격을 갖는 데 도움이 될 수 있습니다.t_{min} 에 t_{max}. 한 가지 디자인 질문은 우리가 무언가를 치면 정상을 계산하는 것과 같은 일을 할 것인지 여부이다. 우리는 검색을 할 때 더 가까운 무언가에 부딪히게 될 수도 있고, 가장 가까운 것의 정상만 필요할 것이다. 나는 간단한 해결책으로 가서 어떤 구조에 저장할 물건 묶음을 계산할 것이다. 여기 추상 수업이 있습니다:

```
#ifndef HITTABLE_H
#define HITTABLE_H

#include "ray.h"

struct hit_record {
    point3 p;
    vec3 normal;
    double t;
};

class hittable {
    public:
        virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec) const = 0;
};

#endif
```

Listing 14: `[hittable.h]` *The hittable class*

그리고 여기 구체가 있습니다:

```
#ifndef SPHERE_H
#define SPHERE_H

#include "hittable.h"
#include "vec3.h"

class sphere : public hittable {
    public:
        sphere() {}
        sphere(point3 cen, double r) : center(cen), radius(r) {};

        virtual bool hit(
            const ray& r, double t_min, double t_max, hit_record& rec) const override;

    public:
        point3 center;
        double radius;
};

bool sphere::hit(const ray& r, double t_min, double t_max, hit_record& rec) const {
    vec3 oc = r.origin() - center;
    auto a = r.direction().length_squared();
    auto half_b = dot(oc, r.direction());
    auto c = oc.length_squared() - radius*radius;

    auto discriminant = half_b*half_b - a*c;
    if (discriminant < 0) return false;
    auto sqrtd = sqrt(discriminant);

    // Find the nearest root that lies in the acceptable range.
    auto root = (-half_b - sqrtd) / a;
    if (root < t_min || t_max < root) {
        root = (-half_b + sqrtd) / a;
        if (root < t_min || t_max < root)
            return false;
    }

    rec.t = root;
    rec.p = r.at(rec.t);
    rec.normal = (rec.p - center) / radius;

    return true;
}

#endif
```

Listing 15: `[sphere.h]` *The sphere class*

6.4. 앞면 대 뒷면

정상에 대한 두 번째 설계 결정은 그들이 항상 지적해야 하는지 여부이다. 현재, 발견된 정상은 항상 교차점(정상점)의 중심 방향에 있을 것이다. 광선이 외부에서 구체와 교차하면, 정상은 광선을 가리킨다. 광선이 내부에서 구와 교차하면, 정상(항상 지적하는)은 광선을 가리킨다. 또는, 우리는 항상 광선에 대한 정상 지점을 가질 수 있다. 광선이 구체 밖에 있다면, 정상은 바깥쪽을 가리킬 것이지만, 광선이 구체 안에 있다면, 정상은 안쪽을 가리킬 것이다.

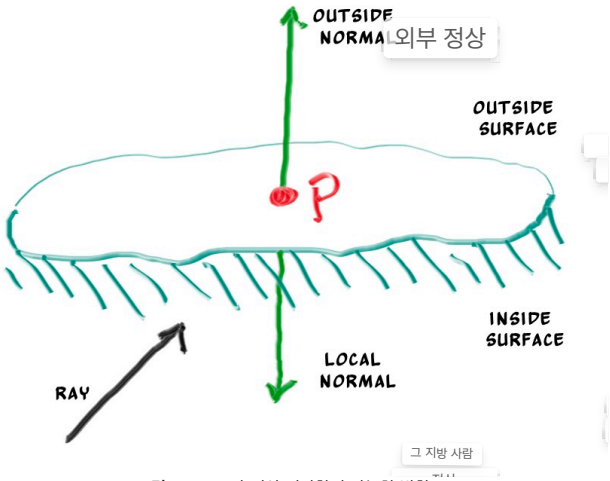


그림 6: *구 표면-정상 기하학의 가능한 방향*

우리는 결국 광선이 표면의 어느 쪽에서 오는지 결정하고 싶을 것이기 때문에 이러한 가능성 중 하나를 선택해야 한다. 이것은 양면 종이의 텍스트와 같이 각 측면에서 다르게 렌더링되는 물체나 유리 공과 같이 내부와 외부가 있는 물체에 중요합니다.

만약 우리가 법선을 항상 지적하기로 결정했다면, 우리는 그것을 색칠할 때 광선이 어느 쪽에 있는지 결정해야 할 것이다. 우리는 광선을 정상과 비교함으로써 이것을 알아낼 수 있다. 광선과 정상적인 면이 같은 방향에 있다면, 광선은 물체 안에 있고, 광선과 정상적인 면이 반대 방향이라면, 광선은 물체 외부에 있다. 이것은 두 벡터의 도트 곱을 취함으로써 결정될 수 있으며, 그들의 점이 양수라면, 광선은 구체 안에 있다.

```
if (dot(ray_direction, outward_normal) > 0.0) {
    // ray is inside the sphere
    ...
} else {
    // ray is outside the sphere
    ...
}
```

목록 16: *광선과 정상 비교하기*

만약 우리가 법선들이 항상 광선을 가리키도록 하기로 결정한다면, 우리는 광선이 표면의 어느 쪽에 있는지 결정하기 위해 점 생성물을 사용할 수 없을 것이다. 대신, 우리는 그 정보를 저장해야 할 것이다:

```
bool front_face;
if (dot(ray_direction, outward_normal) > 0.0) {
    // ray is inside the sphere
    normal = -outward_normal;
    front_face = false;
} else {
    // ray is outside the sphere
    normal = outward_normal;
    front_face = true;
}
```

목록 17: *표면의 측면을 기억하기*

우리는 정상이 항상 표면에서 "외부"를 가리키거나, 항상 입사 광선을 가리키도록 설정할 수 있다. 이 결정은 기하학 교차 시 또는 채색 시 표면의 측면을 결정하고 싶은지 여부에 따라 결정됩니다. 이 책에서 우리는 기하학 유형보다 더 많은 재료 유형을 가지고 있으므로, 우리는 더 적은 작업을 하고 기하학 시간에 결정을 내릴 것이다. 이것은 단순히 선호도의 문제이며, 문헌에서 두 가지 구현을 모두 볼 수 있습니다.

We add the `front_face` bool to the `hit_record` struct. We'll also add a function to solve this calculation for us.

```
struct hit_record {
    point3 p;
    vec3 normal;
    double t;
    bool front_face;

    inline void set_face_normal(const ray& r, const vec3& outward_normal) {
        front_face = dot(r.direction(), outward_normal) < 0;
        normal = front_face ? outward_normal :-outward_normal;
    }
};
```

Listing 18: `[hittable.h]` *Adding front-face tracking to hit_record*

그리고 나서 우리는 수업에 표면 측면 결정을 추가합니다:

```
bool sphere::hit(const ray& r, double t_min, double t_max, hit_record& rec) const {
    ...

    rec.t = root;
    rec.p = r.at(rec.t);
    vec3 outward_normal = (rec.p - center) / radius;
    rec.set_face_normal(r, outward_normal);

    return true;
}
```

Listing 19: `[sphere.h]` *The sphere class with normal determination*

6.5. 히트블 오브젝트 목록

We have a generic object called a `hittable` that the ray can intersect with. We now add a class that stores a list of `hittable`s:

```
#ifndef HITTABLE_LIST_H
#define HITTABLE_LIST_H

#include "hittable.h"

#include <memory>
#include <vector>

using std::shared_ptr;
using std::make_shared;

class hittable_list : public hittable {
    public:
        hittable_list() {}
        hittable_list(shared_ptr<hittable> object) { add(object); }

        void clear() { objects.clear(); }
        void add(shared_ptr<hittable> object) { objects.push_back(object); }

        virtual bool hit(
            const ray& r, double t_min, double t_max, hit_record& rec) const override;

    public:
        std::vector<shared_ptr<hittable>> objects;
};

bool hittable_list::hit(const ray& r, double t_min, double t_max, hit_record& rec) const {
    hit_record temp_rec;
    bool hit_anything = false;
    auto closest_so_far = t_max;

    for (const auto& object : objects) {
        if (object->hit(r, t_min, closest_so_far, temp_rec)) {
            hit_anything = true;
            closest_so_far = temp_rec.t;
            rec = temp_rec;
        }
    }

    return hit_anything;
}

#endif
```

Listing 20: `[hittable_list.h]` *The hittable_list class*

6.6. 몇 가지 새로운 C++ 기능

The `hittable_list` class code uses two C++ features that may trip you up if you're not normally a C++ programmer: `vector` and `shared_ptr`.

`shared_ptr<type>`참조 계산 의미와 함께 할당된 유형에 대한 포인터입니다. 다른 공유 포인터(보통 간단한 할당)에 값을 할당할 때마다 참조 수가 증가합니다. 공유 포인터가 범위를 벗어나면서(블록이나 함수의 끝과 같이), 참조 수는 감소한다. 카운트가 0이 되면, 객체는 삭제됩니다.

일반적으로, 공유 포인터는 다음과 같이 새로 할당된 객체로 먼저 초기화됩니다:

```
shared_ptr<double> double_ptr = make_shared<double>(0.37);
shared_ptr<vec3>   vec3_ptr   = make_shared<vec3>(1.414214, 2.718281, 1.618034);
shared_ptr<sphere> sphere_ptr = make_shared<sphere>(point3(0,0,0), 1.0);
```

목록 21: *shared_ptr를 사용한 할당 예시*

`make_shared<thing>(thing_constructor_params ...)`생성자 매개 변수를 사용하여 typething의 새로운 인스턴스를 할당합니다. 그것은 `shared_ptr<thing>`을 반환합니다.

Since the type can be automatically deduced by the return type of `make_shared<type>(...)`, the above lines can be more simply expressed using C++'s `auto` type specifier:

```
auto double_ptr = make_shared<double>(0.37);
auto vec3_ptr   = make_shared<vec3>(1.414214, 2.718281, 1.618034);
auto sphere_ptr = make_shared<sphere>(point3(0,0,0), 1.0);
```

목록 22: *자동 유형으로 shared_ptr를 사용한 할당 예시*

우리는 코드에서 공유 포인터를 사용할 것입니다. 왜냐하면 여러 지오메트리가 공통 인스턴스(예를 들어, 모두 동일한 텍스처 맵 재료를 사용하는 구체 무리)를 공유할 수 있고, 메모리 관리를 자동으로 쉽게 추론할 수 있기 때문입니다.

`std::shared_ptr` is included with the `<memory>` header.

The second C++ feature you may be unfamiliar with is `std::vector`. This is a generic array-like collection of an arbitrary type. Above, we use a collection of pointers to `hittable`. `std::vector` automatically grows as more values are added: `objects.push_back(object)` adds a value to the end of the `std::vector` member variable `objects`.

`std::vector` is included with the `<vector>` header.

Finally, the `using` statements in listing 20 tell the compiler that we'll be getting `shared_ptr` and `make_shared` from the `std` library, so we don't need to prefex these with `std::` every time we reference them.

6.7. 공통 상수와 유틸리티 함수

우리는 그들 자신의 헤더 파일에서 편리하게 정의하는 수학 상수가 필요하다. 지금은 무한대만 필요하지만, 나중에 필요할 파이에 대한 우리 자신의 정의를 거기에 던질 것이다. 파이의 표준 휴대용 정의가 없기 때문에, 우리는 그것에 대해 우리 자신의 상수를 정의합니다. 우리는 일반적인 메인 헤더 파일인 `rtweekend.h` 일반적인 유용한 상수와 미래의 유틸리티 함수를 던질 것입니다.

```
#ifndef RTWEEKEND_H
#define RTWEEKEND_H

#include <cmath>
#include <limits>
#include <memory>


// Usings

using std::shared_ptr;
using std::make_shared;
using std::sqrt;

// Constants

const double infinity = std::numeric_limits<double>::infinity();
const double pi = 3.1415926535897932385;

// Utility Functions

inline double degrees_to_radians(double degrees) {
    return degrees * pi / 180.0;
}

// Common Headers

#include "ray.h"
#include "vec3.h"

#endif
```

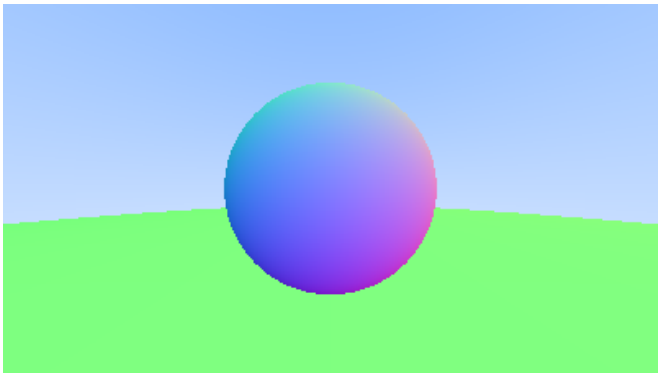
Listing 23: `[rtweekend.h]` *The rtweekend.h common header*

그리고 새로운 메인:

```
#include "rtweekend.h"

#include "color.h"
#include "hittable_list.h"
#include "sphere.h"

#include <iostream>
color ray_color(const ray& r, const hittable& world) {
    hit_record rec;
    if (world.hit(r, 0, infinity, rec)) {
        return 0.5 * (rec.normal + color(1,1,1));
    }
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}

int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);

    // World
    hittable_list world;
    world.add(make_shared<sphere>(point3(0,0,-1), 0.5));
    world.add(make_shared<sphere>(point3(0,-100.5,-1), 100));

    // Camera

    auto viewport_height = 2.0;
    auto viewport_width = aspect_ratio * viewport_height;
    auto focal_length = 1.0;

    auto origin = point3(0, 0, 0);
    auto horizontal = vec3(viewport_width, 0, 0);
    auto vertical = vec3(0, viewport_height, 0);
    auto lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0, focal_length);

    // Render

    std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            auto u = double(i) / (image_width-1);
            auto v = double(j) / (image_height-1);
            ray r(origin, lower_left_corner + u*horizontal + v*vertical);
            color pixel_color = ray_color(r, world);
            write_color(std::cout, pixel_color);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 24: `[main.cc]` *The new main with hittables*

이것은 실제로 구체가 표면 정상과 함께 어디에 있는지를 시각화하는 그림을 산출한다. 이것은 종종 당신의 모델에서 결함과 특성을 볼 수 있는 좋은 방법입니다.



이미지 5: 접지와 함께 노멀 컬러 구의 결과 렌더링

7. 안티앨리어싱

실제 카메라가 사진을 찍을 때, 가장자리 픽셀은 일부 전경과 배경이 혼합되어 있기 때문에 보통 가장자리를 따라 재기가 없습니다. 우리는 각 픽셀 안에 많은 샘플을 평균화함으로써 같은 효과를 얻을 수 있다. 우리는 계층화에 신경 쓰지 않을 것이다. 이것은 논란의 여지가 있지만, 내 프로그램에서는 보통이다. 일부 레이 트레이서의 경우 매우 중요하지만, 우리가 쓰고 있는 일반적인 종류의 것은 그것으로부터 많은 이익을 얻지 못하고 코드를 더 못생기게 만든다. 우리는 나중에 더 멋진 카메라를 만들 수 있도록 카메라 수업을 조금 추상화한다.

7.1. 일부 난수 유틸리티

우리가 필요한 한 가지는 실제 난수를 반환하는 난수 생성기이다. 우리는 관습에 따라 범위에서 임의의 실제를 반환하는 표준 난수를 반환하는 함수가 필요합니다.$0 \le r < 1$. 1 이전의 "보다 덜"은 우리가 때때로 그것을 이용하기 때문에 중요하다.

A simple approach to this is to use the `rand()` function that can be found in `<cstdlib>`. This function returns a random integer in the range 0 and `RAND_MAX`. Hence we can get a real random number as desired with the following code snippet, added to `rtweekend.h`:

```
#include <cstdlib>
...

inline double random_double() {
    // Returns a random real in [0,1).
    return rand() / (RAND_MAX + 1.0);
}

inline double random_double(double min, double max) {
    // Returns a random real in [min,max).
    return min + (max-min)*random_double();
}
```

Listing 25: `[rtweekend.h]` *random_double() functions*

C++ did not traditionally have a standard random number generator, but newer versions of C++ have addressed this issue with the `<random>` header (if imperfectly according to some experts). If you want to use this, you can obtain a random number with the conditions we need as follows:

```
#include <random>

inline double random_double() {
    static std::uniform_real_distribution<double> distribution(0.0, 1.0);
    static std::mt19937 generator;
    return distribution(generator);
}
```

Listing 26: `[rtweekend.h]` *random_double(), alternate implemenation*

7.2. 여러 샘플로 픽셀 생성하기

주어진 픽셀에 대해 우리는 그 픽셀 내에 여러 개의 샘플을 가지고 있으며 각 샘플을 통해 광선을 보냅니다. 그런 다음 이 광선의 색은 평균화된다:

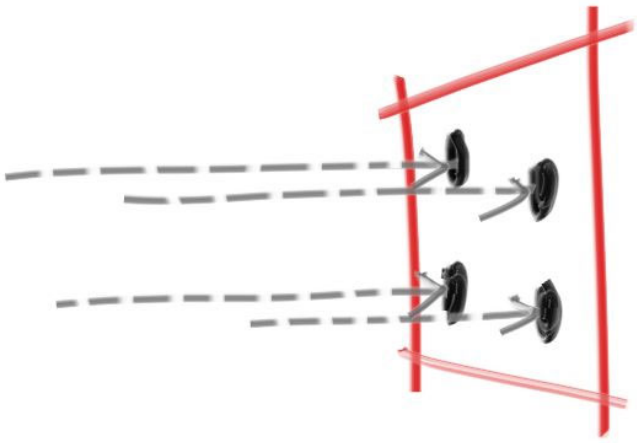


그림 7: *픽셀 샘플*

Now's a good time to create a `camera` class to manage our virtual camera and the related tasks of scene scampling. The following class implements a simple camera using the axis-aligned camera from before:

```
#ifndef CAMERA_H
#define CAMERA_H

#include "rtweekend.h"

class camera {
    public:
        camera() {
            auto aspect_ratio = 16.0 / 9.0;
            auto viewport_height = 2.0;
            auto viewport_width = aspect_ratio * viewport_height;
            auto focal_length = 1.0;

            origin = point3(0, 0, 0);
            horizontal = vec3(viewport_width, 0.0, 0.0);
            vertical = vec3(0.0, viewport_height, 0.0);
            lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0, focal_length);
        }

        ray get_ray(double u, double v) const {
            return ray(origin, lower_left_corner + u*horizontal + v*vertical - origin);
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};
#endif
```

Listing 27: `[camera.h]` *The camera class*

To handle the multi-sampled color computation, we'll update the `write_color()` function. Rather than adding in a fractional contribution each time we accumulate more light to the color, just add the full color each iteration, and then perform a single divide at the end (by the number of samples) when writing out the color. In addition, we'll add a handy utility function to the `rtweekend.h` utility header: `clamp(x,min,max)`, which clamps the value `x` to the range [min,max]:

```
inline double clamp(double x, double min, double max) {
    if (x < min) return min;
    if (x > max) return max;
    return x;
}
```

Listing 28: `[rtweekend.h]` *The clamp() utility function*

```
void write_color(std::ostream &out, color pixel_color, int samples_per_pixel) {
    auto r = pixel_color.x();
    auto g = pixel_color.y();
    auto b = pixel_color.z();

    // Divide the color by the number of samples.
    auto scale = 1.0 / samples_per_pixel;
    r *= scale;
    g *= scale;
    b *= scale;

    // Write the translated [0,255] value of each color component.
    out << static_cast<int>(256 * clamp(r, 0.0, 0.999)) << ' '
        << static_cast<int>(256 * clamp(g, 0.0, 0.999)) << ' '
        << static_cast<int>(256 * clamp(b, 0.0, 0.999)) << '\n';
}
```

Listing 29: `[color.h]` *The multi-sample write_color() function*

메인도 바뀌었다:

```
#include "camera.h"

...

int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 100;

    // World

    hittable_list world;
    world.add(make_shared<sphere>(point3(0,0,-1), 0.5));
    world.add(make_shared<sphere>(point3(0,-100.5,-1), 100));

    // Camera
    camera cam;

    // Render

    std::cout << "P3\n" << image_width << " " << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0, 0, 0);
            for (int s = 0; s < samples_per_pixel; ++s) {
                auto u = (i + random_double()) / (image_width-1);
                auto v = (j + random_double()) / (image_height-1);
                ray r = cam.get_ray(u, v);
                pixel_color += ray_color(r, world);
            }
            write_color(std::cout, pixel_color, samples_per_pixel);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 30: `[main.cc]` *Rendering with multi-sampled pixels*

생성된 이미지를 확대하면, 우리는 가장자리 픽셀의 차이를 볼 수 있다.

이미지 6: 안티앨리어싱 전후

8. 확산 재료

이제 우리는 픽셀당 물체와 여러 광선을 가지고 있으므로, 사실적으로 보이는 재료를 만들 수 있습니다. 우리는 확산(매트) 재료로 시작할 것이다. 한 가지 질문은 우리가 기하학과 재료를 혼합하고 일치시키는지(그래서 우리는 재료를 여러 구체에 할당할 수 있거나 그 반대의 경우도 마찬가지) 또는 기하학과 재료가 단단히 묶여 있는지 여부입니다(기하학과 재료가 연결된 절차적 물체에 유용할 수 있음). 우리는 대부분의 렌더러에서 흔히 볼 수 있는 별도의로 갈 것이지만, 그 제한에 유의해야 합니다.

8.1. 간단한 확산 물질

빛을 방출하지 않는 확산 물체는 단지 주변 환경의 색을 취하지만, 그들 자신의 본질적인 색으로 그것을 조절한다. 확산된 표면에서 반사되는 빛은 방향이 무작위화된다. 그래서, 우리가 두 개의 확산된 표면 사이의 균열에 세 개의 광선을 보내면 그들은 각각 다른 무작위 행동을 할 것이다:

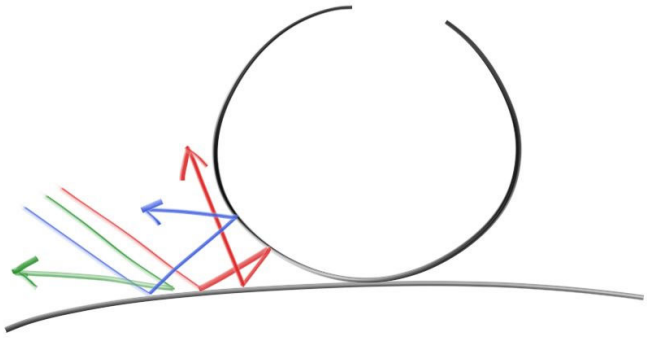


그림 8: *광선이 튕겨*

그들은 또한 반영되기보다는 흡수될 수 있다. 표면이 어두울수록, 흡수 가능성이 더 높다. (그게 어두워진 이유야!) 방향을 무작위화하는 모든 알고리즘은 무광택으로 보이는 표면을 생성할 것이다. 이것을 하는 가장 간단한 방법 중 하나는 이상적인 확산 표면을 위해 정확히 올바른 것으로 밝혀졌다. (나는 수학적으로 이상적인 램버트에 가까운 게으른 해킹으로 그것을 하곤 했다.)

(독자 바실렌 치조프는 게으른 해킹이 실제로 게으른 해킹일 뿐이며 부정확하다는 것을 증명했다. 이상적인 람베르티안의 올바른 표현은 더 많은 일이 아니며, 장의 끝에 제시된다.)

히트 포인트에 접하는 두 개의 단위 반경 구체가 있다.**p**표면의. 이 두 구체는 중심을 가지고 있다.$(\mathbf{P}+\mathbf{n})$ 그리고$(\mathbf{P}-\mathbf{n})$, 어디**n**표면의 정상이다. 중심이 있는 구체$(\mathbf{P}-\mathbf{n})$ 중심이 있는 구체는 표면 *내부로* 간주된다.$(\mathbf{P}+\mathbf{n})$ 표면 *밖에서* 여겨진다. 광선 원점과 같은 쪽에 있는 접선 단위 반경 구를 선택하세요. 임의의 지점을 고르세요.**S**이 단위 반경 구 안에 들어가서 히트 포인트에서 광선을 보내세요.**p**임의의 지점까지**S**(이건 벡터야$(\mathbf{S}-\mathbf{P})$):

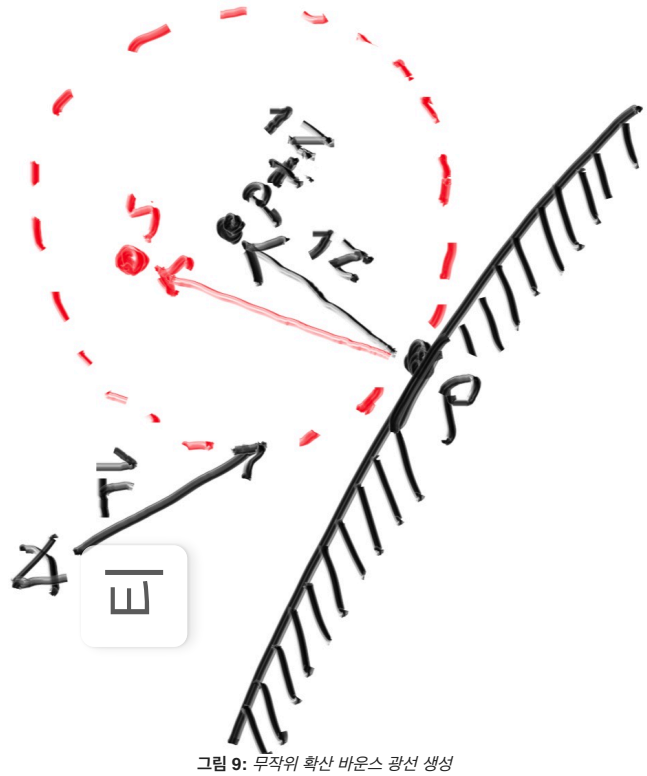


그림 9: *무작위 확산 바운스 광선 생성*

우리는 단위 반경 구에서 임의의 점을 선택할 방법이 필요하다. 우리는 보통 가장 쉬운 알고리즘인 거절 방법을 사용할 것이다. 먼저, x, y, z가 모두 -1에서 +1인 단위 큐브에서 임의의 점을 선택하세요. 이 점을 거부하고 그 점이 구체 밖에 있다면 다시 시도하세요.

```
class vec3 {
  public:
    ...
    inline static vec3 random() {
        return vec3(random_double(), random_double(), random_double());
    }

    inline static vec3 random(double min, double max) {
        return vec3(random_double(min,max), random_double(min,max), random_double(min,max));
    }
```

Listing 31: `[vec3.h]` *vec3 random utility functions*

```
vec3 random_in_unit_sphere() {
    while (true) {
        auto p = vec3::random(-1,1);
        if (p.length_squared() >= 1) continue;
        return p;
    }
}
```

Listing 32: `[vec3.h]` *The random_in_unit_sphere() function*

Then update the `ray_color()` function to use the new random direction generator:

```
color ray_color(const ray& r, const hittable& world) {
    hit_record rec;

    if (world.hit(r, 0, infinity, rec)) {
        point3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 33: `[main.cc]` *ray_color() using a random ray direction*

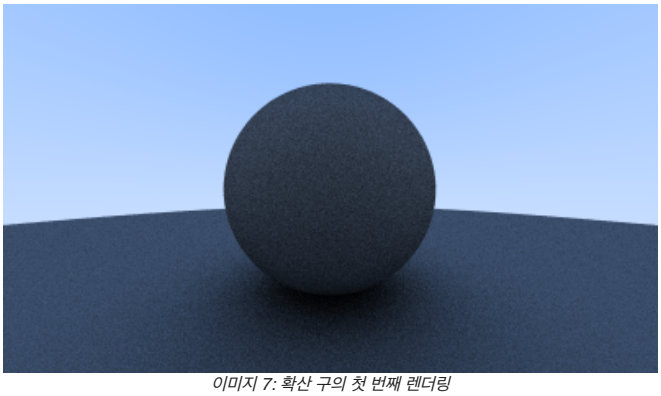
8.2. 어린이 광선의 수를 제한하기

There's one potential problem lurking here. Notice that the `ray_color` function is recursive. When will it stop recursing? When it fails to hit anything. In some cases, however, that may be a long time — long enough to blow the stack. To guard against that, let's limit the maximum recursion depth, returning no light contribution at the maximum depth:

```
color ray_color(const ray& r, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    if (world.hit(r, 0, infinity, rec)) {
        point3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world, depth-1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}

...

int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 100;
    const int max_depth = 50;
    ...

    // Render

    std::cout << "P3\n" << image_width << " " << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0, 0, 0);
            for (int s = 0; s < samples_per_pixel; ++s) {
                auto u = (i + random_double()) / (image_width-1);
                auto v = (j + random_double()) / (image_height-1);
                ray r = cam.get_ray(u, v);
                pixel_color += ray_color(r, world, max_depth);
            }
            write_color(std::cout, pixel_color, samples_per_pixel);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 34: `[main.cc]` *ray_color() with depth limiting*

이것은 우리에게 다음을 준다:



이미지 7: 확산 구의 첫 번째 렌더링

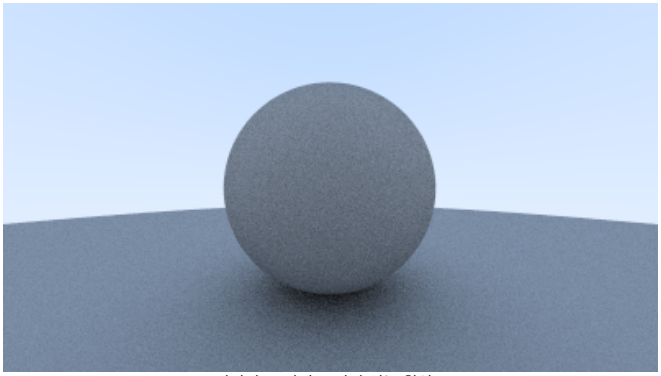
8.3. 정확한 색 강도를 위한 감마 보정 사용

구 아래의 그림자에 주목하세요. 이 그림은 매우 어둡지만, 우리의 구체는 각 바운스마다 에너지의 절반만 흡수하기 때문에 50% 반사체이다. 만약 당신이 그림자를 볼 수 없다면, 걱정하지 마세요, 우리는 지금 그것을 고칠 것입니다. 이 구체들은 꽤 가벼워 보여야 한다 (실생활에서는 밝은 회색). 그 이유는 거의 모든 이미지 시청자가 이미지가 "감마 수정"이라고 가정하기 때문입니다. 즉, 0에서 1 값은 바이트로 저장되기 전에 약간의 변환이 있다는 것을 의미합니다. 그것에 대한 많은 좋은 이유가 있지만, 우리의 목적을 위해 우리는 그것을 알아야 한다. 첫 번째 근사치로, 우리는 색상을 전력으로 높이는 것을 의미하는 "감마 2"를 사용할 수 있습니다.$1/\text{감마}$, 또는 우리의 간단한 경우 ½, 그것은 단지 제곱근이다:

```
void write_color(std::ostream &out, color pixel_color, int samples_per_pixel) {
    auto r = pixel_color.x();
    auto g = pixel_color.y();
    auto b = pixel_color.z();

    // Divide the color by the number of samples and gamma-correct for gamma=2.0.
    auto scale = 1.0 / samples_per_pixel;
    r = sqrt(scale * r);
    g = sqrt(scale * g);
    b = sqrt(scale * b);

    // Write the translated [0,255] value of each color component.
    out << static_cast<int>(256 * clamp(r, 0.0, 0.999)) << ' '
        << static_cast<int>(256 * clamp(g, 0.0, 0.999)) << ' '
        << static_cast<int>(256 * clamp(b, 0.0, 0.999)) << '\n';
}
```

Listing 35: `[color.h]` *write_color(), with gamma correction*

그것은 우리가 원하는 대로 밝은 회색을 산출한다:



이미지 8: 감마 보정이 있는 확산 구

8.4. 그림자 여드름 고치기

거기에는 미묘한 버그도 있다. 반사된 광선 중 일부는 정확히 반사되지 않은 물체에 부딪혔다.$t = 0$, 하지만 대신$t = -0.0000001$ 또는 $t = 0.00000001$ 또는 구 인터섹터가 우리에게 주는 부동 소수점 근사치가 무엇이든. 그래서 우리는 0에 가까운 히트를 무시해야 한다:

```
if (world.hit(r, 0.001, infinity, rec)) {
```

Listing 36: `[main.cc]` *Calculating reflected ray origins with tolerance*

이것은 그림자 여드름 문제를 제거한다. 응 그건 정말 그렇게 불려.

8.5. 진정한 람베르티안 반사

여기에 제시된 거부 방법은 표면 법선을 따라 오프셋된 단위 볼에서 임의의 점을 생성합니다. 이것은 정상에 가까운 높은 확률로 반구의 방향을 고르는 것과 방목 각도에서 광선을 산란할 확률이 낮은 것에 해당한다. 이 분포는$\cos(\varphi)$어디에φ정상으로부터의 각도이다. 이것은 얕은 각도에 도달하는 빛이 더 넓은 지역에 퍼지기 때문에, 따라서 최종 색상에 대한 기여도가 낮기 때문에 유용하다.

그러나, 우리는 분포를 가진 Lambertian 분포에 관심이 있다.$\cos(\varphi)^3$진정한 람베르티안은 정상에 가까운 광선 산란에 대한 확률이 더 높지만, 분포는 더 균일하다. 이것은 단위 구의 표면에서 임의의 점을 선택하고 표면을 따라 오프셋함으로써 달성된다. 단위 구에서 임의의 점을 선택하는 것은 단위 구에서 임의의 점을 선택한 다음 정규화함으로써 달성할 수 있다.

```
inline vec3 random_in_unit_sphere() {
    ...
}
vec3 random_unit_vector() {
    return unit_vector(random_in_unit_sphere());
}
```

Listing 37: `[vec3.h]` *The random_unit_vector() function*

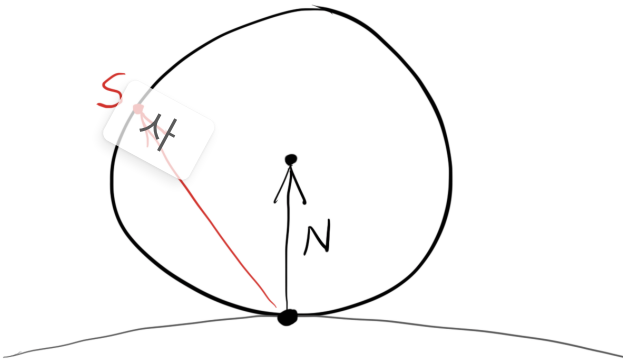


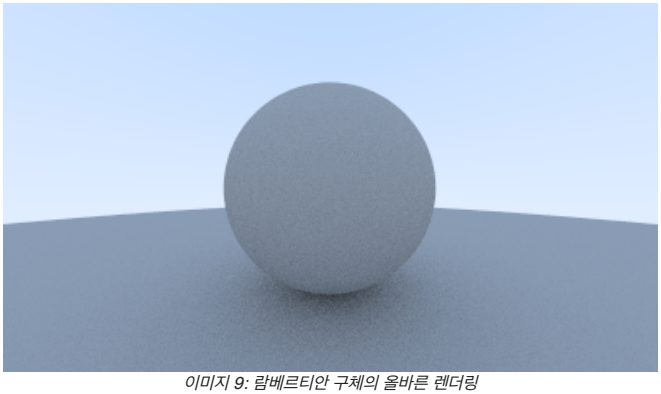
그림 10: *무작위 단위 벡터 생성*

This `random_unit_vector()` is a drop-in replacement for the existing `random_in_unit_sphere()` function.

```
color ray_color(const ray& r, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    if (world.hit(r, 0.001, infinity, rec)) {
        point3 target = rec.p + rec.normal + random_unit_vector();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world, depth-1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 38: `[main.cc]` *ray_color() with replacement diffuse*

렌더링 후 우리는 비슷한 이미지를 얻는다:



이미지 9: 람베르티안 구체의 올바른 렌더링

우리의 두 영역의 장면이 매우 간단하다는 점을 감안할 때, 이 두 가지 확산 방법의 차이를 구별하기는 어렵지만, 두 가지 중요한 시각적 차이를 알아차릴 수 있어야 합니다.

1. 변화 후에 그림자는 덜 두드러진다.
2. 두 구체 모두 변화 후 외관이 더 가볍다.

이 두 가지 변화는 광선의 더 균일한 산란으로 인한 것이며, 더 적은 광선이 정상을 향해 산란하기 때문이다. 이것은 확산된 물체의 경우, 더 많은 빛이 카메라를 향해 튕기기 때문에 *더 가벼워* 보일 것이라는 것을 의미한다. 그림자의 경우, 더 적은 빛이 똑바로 반사되므로, 더 작은 구 바로 아래에 있는 더 큰 구의 부분이 더 밝다.

8.6. 대체 확산 제형

이 책에 제시된 초기 해킹은 이상적인 람베르티안 확산의 잘못된 근사치로 판명되기까지 오랜 시간이 지속되었다. 오류가 그렇게 오랫동안 지속된 가장 큰 이유는 다음과 같이 어려울 수 있기 때문입니다:

1. 확률 분포가 틀렸다는 것을 수학적으로 증명한다.
2. 왜 그런지 직관적으로 설명하세요코스(φ是배는 바람직하다 (그리고 그것이 어떻게 생겼는지)

일반적인 일상적인 물체는 많지 않기 때문에, 이러한 물체가 빛 아래에서 어떻게 행동하는지에 대한 우리의 시각적 직관은 제대로 형성되지 않을 수 있다.

학습을 위해, 우리는 직관적이고 이해하기 쉬운 확산 방법을 포함하고 있습니다. 위의 두 가지 방법에 대해 우리는 임의의 벡터를 가지고 있었는데, 처음에는 무작위 길이이고 그 다음에는 단위 길이로, 히트 포인트에서 법선으로 오프셋되었습니다. 왜 벡터가 정상에 의해 대체되어야 하는지 즉시 명확하지 않을 수 있다.

보다 직관적인 접근 방식은 정상에서 각도에 의존하지 않고 히트포인트에서 떨어진 모든 각도에 대해 균일한 산란 방향을 갖는 것이다. 많은 첫 번째 레이 트레이싱 종이는 이 확산 방법을 사용했다(람베르티안 확산을 채택하기 전에).

```
vec3 random_in_hemisphere(const vec3& normal) {
    vec3 in_unit_sphere = random_in_unit_sphere();
    if (dot(in_unit_sphere, normal) > 0.0) // In the same hemisphere as the normal
        return in_unit_sphere;
    else
        return -in_unit_sphere;
}
```

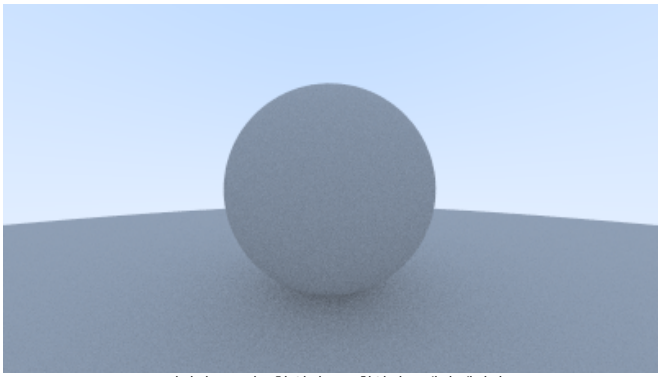
Listing 39: `[vec3.h]` *The random_in_hemisphere(normal) function*

Plugging the new formula into the `ray_color()` function:

```
color ray_color(const ray& r, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    if (world.hit(r, 0.001, infinity, rec)) {
        point3 target = rec.p + random_in_hemisphere(rec.normal);
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world, depth-1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 40: `[main.cc]` *ray_color() with hemispherical scattering*

우리에게 다음과 같은 이미지를 준다:



이미지 10: 반구형 산란으로 확산된 구체의 렌더링

그 책의 과정에서 장면은 더 복잡해질 것이다. 여기에 제시된 다양한 확산 렌더러 간에 전환하는 것이 좋습니다. 대부분의 관심 장면은 불균형한 양의 확산 물질을 포함할 것이다. 다양한 확산 방법이 장면의 조명에 미치는 영향을 이해함으로써 귀중한 통찰력을 얻을 수 있습니다.

9. 금속

9.1. 재료에 대한 추상 수업

만약 우리가 다른 물체가 다른 재료를 갖기를 원한다면, 우리는 디자인 결정을 내린다. 우리는 많은 매개 변수와 다른 재료 유형을 가진 보편적인 재료를 가질 수 있다. 이건 나쁜 접근 방식이 아니야. 또는 우리는 행동을 캡슐화하는 추상적인 재료 클래스를 가질 수 있다. 나는 후자의 접근 방식의 팬이다. 우리 프로그램을 위해 자료는 두 가지를 해야 합니다:

1. 흩어진 광선을 생성하세요 (또는 입사 광선을 흡수했다고 말하세요).
2. 흩어져 있다면, 광선이 얼마나 감쇠되어야 하는지 말하세요.

이것은 추상적인 수업을 제안한다:

```
#ifndef MATERIAL_H
#define MATERIAL_H

#include "rtweekend.h"

struct hit_record;

class material {
    public:
        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const = 0;
};

#endif
```

Listing 41: `[material.h]` *The material class*

9.2. 레이-객체 교차로를 설명하는 데이터 구조

`hit_record` 우리가 원하는 모든 정보를 채울 수 있도록 많은 논쟁을 피하기 위한 것이다. 당신은 대신 논쟁을 사용할 수 있습니다; 그것은 취향의 문제입니다. 히트테이블과 재료는 서로를 알아야 하기 때문에 참조의 순환성이 있다. C++에서는 포인터가 아래 히트테이블 클래스의 "클래스 자료"가 하는 클래스에 대한 클래스라는 것을 컴파일러에 알리기만 하면 됩니다.

```
#include "rtweekend.h"

class material;

struct hit_record {
    point3 p;
    vec3 normal;
    shared_ptr<material> mat_ptr;
    double t;
    bool front_face;

    inline void set_face_normal(const ray& r, const vec3& outward_normal) {
        front_face = dot(r.direction(), outward_normal) < 0;
        normal = front_face ? outward_normal :-outward_normal;
    }
};
```

Listing 42: `[hittable.h]` *Hit record with added material pointer*

What we have set up here is that material will tell us how rays interact with the surface. `hit_record` is just a way to stuff a bunch of arguments into a struct so we can send them as a group. When a ray hits a surface (a particular sphere for example), the material pointer in the `hit_record` will be set to point at the material pointer the sphere was given when it was set up in `main()` when we start. When the `ray_color()` routine gets the `hit_record` it can call member functions of the material pointer to find out what ray, if any, is scattered.

이것을 달성하기 위해, 우리는 우리의 구체 클래스가 withinhit`hit_record` 반환하기 위한 자료에 대한 참조가 있어야 한다. 아래에서 강조 표시된 줄을 보세요:

```
class sphere : public hittable {
    public:
        sphere() {}
        sphere(point3 cen, double r, shared_ptr<material> m)
            : center(cen), radius(r), mat_ptr(m) {};

        virtual bool hit(
            const ray& r, double t_min, double t_max, hit_record& rec) const override;

    public:
        point3 center;
        double radius;
        shared_ptr<material> mat_ptr;
};

bool sphere::hit(const ray& r, double t_min, double t_max, hit_record& rec) const {
    ...

    rec.t = root;
    rec.p = r.at(rec.t);
    vec3 outward_normal = (rec.p - center) / radius;
    rec.set_face_normal(r, outward_normal);
    rec.mat_ptr = mat_ptr;

    return true;
}
```

Listing 43: `[sphere.h]` *Ray-sphere intersection with added material information*

9.3. 빛 산란과 반사율 모델링

우리가 이미 가지고 있는 람베르티안(확산)의 경우, 그것은 항상 흩어져 있고 반사율에 의해 약화될 수 있다.R, 또는 감쇠 없이 흩어질 수 있지만 분수를 흡수할 수 있다.$1 - R$ 광선의, 아니면 그 전략의 혼합일 수도 있다. 람베르티안 자료의 경우 우리는 이 간단한 수업을 받는다:

```
class lambertian : public material {
    public:
        lambertian(const color& a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            auto scatter_direction = rec.normal + random_unit_vector();
            scattered = ray(rec.p, scatter_direction);
            attenuation = albedo;
            return true;
        }

    public:
        color albedo;
};
```

Listing 44: `[material.h]` *The lambertian material class*

우리는 약간의 확률로만 흩어질 수 있다는 것을 알아두세요.ㅛㅁ리고 감쇠가 있어알베도/p. 네 선택이야.

위의 코드를 주의 깊게 읽으면, 장난의 작은 가능성을 알게 될 것이다. 우리가 생성하는 무작위 단위벡터가 정규 벡터와 정확히 반대라면, 둘은 0으로 합산되며, 이는 제로 산란 방향 벡터가 될 것이다. 이것은 나중에 나쁜 시나리오(무한대와 NaN)로 이어지기 때문에, 우리는 그것을 전달하기 전에 조건을 가로채야 한다.

이를 위해, 우리는 벡터가 모든 차원에서 0에 매우 가까운 경우 true를 반환하는 새로운 벡터 메서드인 `vec3::near_zero()`를 만들 것입니다.

```
class vec3 {
    ...
    bool near_zero() const {
        // Return true if the vector is close to zero in all dimensions.
        const auto s = 1e-8;
        return (fabs(e[0]) < s) && (fabs(e[1]) < s) && (fabs(e[2]) < s);
    }
    ...
};
```

Listing 45: `[vec3.h]` *The vec3::near_zero() method*

```
class lambertian : public material {
    public:
        lambertian(const color& a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            auto scatter_direction = rec.normal + random_unit_vector();

            // Catch degenerate scatter direction
            if (scatter_direction.near_zero())
                scatter_direction = rec.normal;

            scattered = ray(rec.p, scatter_direction);
            attenuation = albedo;
            return true;
        }

    public:
        color albedo;
};
```

Listing 46: `[material.h]` *Lambertian scatter, bullet-proof*

9.4. 미러링된 빛 반사

평활금속의 경우 광선은 무작위로 흩어지지 않을 것이다. 핵심 수학은: 광선은 금속 거울에서 어떻게 반사되는가? 벡터 수학은 여기서 우리의 친구야:

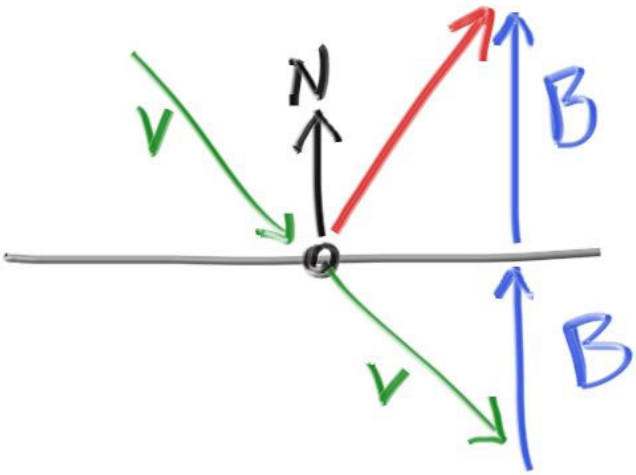


그림 11: *레이 반사*

빨간색으로 반사된 광선 방향은 그냥$\mathbf{v}+2\mathbf{b}$. 우리의 디자인에서,$\mathbf{n}$단위 벡터이지만,$\mathbf{n}$일 수도 있어. 의 길이$\mathbf{b}$그래야 해$\mathbf{v}\cdot\mathbf{n}$. 왜냐하면$\mathbf{v}$포인트, 우리는 마이너스 기호가 필요할 것이다.

```
vec3 reflect(const vec3& v, const vec3& n) {
    return v - 2*dot(v,n)*n;
}
```

Listing 47: `[vec3.h]` *vec3 reflection function*

금속 재료는 그 공식을 사용하여 광선을 반사한다:

```
class metal : public material {
    public:
        metal(const color& a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            vec3 reflected = reflect(unit_vector(r_in.direction()), rec.normal);
            scattered = ray(rec.p, reflected);
            attenuation = albedo;
            return (dot(scattered.direction(), rec.normal) > 0);
        }

    public:
        color albedo;
};
```

Listing 48: `[material.h]` *Metal material with reflectance function*

We need to modify the `ray_color()` function to use this:

```
color ray_color(const ray& r, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    if (world.hit(r, 0.001, infinity, rec)) {
        ray scattered;
        color attenuation;
        if (rec.mat_ptr->scatter(r, rec, attenuation, scattered))
            return attenuation * ray_color(scattered, world, depth-1);
        return color(0,0,0);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 49: `[main.cc]` *Ray color with scattered reflectance*

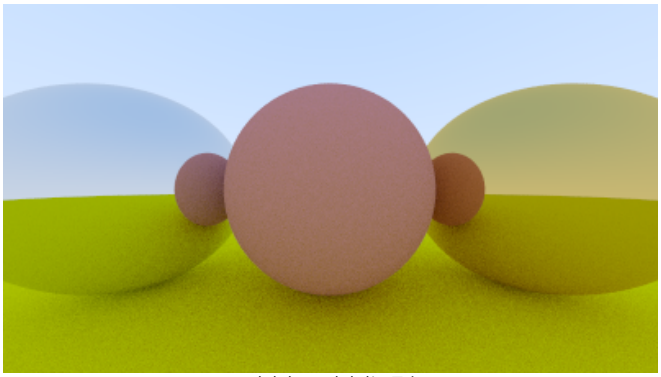
9.5. 금속 구체가 있는 장면

이제 우리 장면에 금속 구체를 추가해 봅시다:

```
...
#include "material.h"
...
int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 100;
    const int max_depth = 50;

    // World

    hittable_list world;

    auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
    auto material_center = make_shared<lambertian>(color(0.7, 0.3, 0.3));
    auto material_left   = make_shared<metal>(color(0.8, 0.8, 0.8));
    auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2));

    world.add(make_shared<sphere>(point3( 0.0, -100.5, -1.0), 100.0, material_ground));
    world.add(make_shared<sphere>(point3( 0.0,    0.0, -1.0),   0.5, material_center));
    world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),   0.5, material_left));
    world.add(make_shared<sphere>(point3( 1.0,    0.0, -1.0),   0.5, material_right));

    // Camera

    camera cam;

    // Render

    std::cout << "P3\n" << image_width << " " << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0, 0, 0);
            for (int s = 0; s < samples_per_pixel; ++s) {
                auto u = (i + random_double()) / (image_width-1);
                auto v = (j + random_double()) / (image_height-1);
                ray r = cam.get_ray(u, v);
                pixel_color += ray_color(r, world, max_depth);
            }
            write_color(std::cout, pixel_color, samples_per_pixel);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 50: `[main.cc]` *Scene with metal spheres*

주는 것:



이미지 11: 반짝이는 금속

9.6. 퍼지 반사

우리는 또한 작은 구를 사용하고 광선에 대한 새로운 끝점을 선택하여 반사된 방향을 무작위화할 수 있습니다:

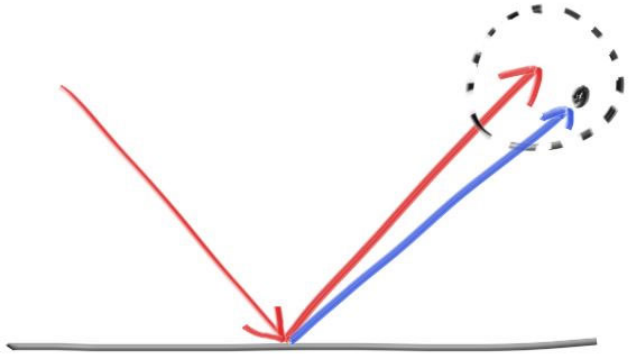


그림 12: *퍼지 반사 광선 생성*

구체가 클수록, 반사는 더 흐릿해질 것이다. 이것은 구의 반경인 흐릿함 매개 변수를 추가하는 것을 제안한다(따라서 0은 섭동이 아니다). 캐치는 큰 구체나 방목 광선이 표면 아래에 흩어질 수 있다는 것이다. 우리는 그것들을 표면 흡수할 수 있다.

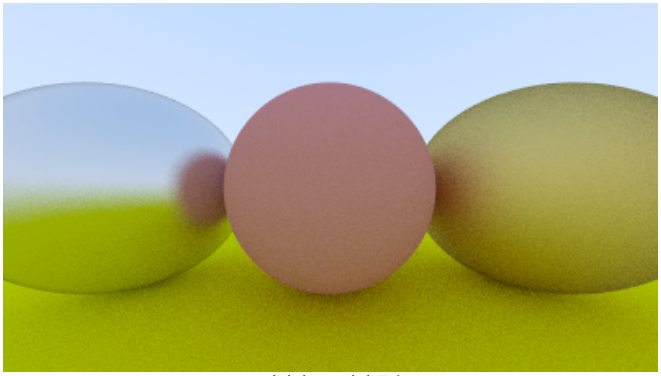
```
class metal : public material {
    public:
        metal(const color& a, double f) : albedo(a), fuzz(f < 1 ? f : 1) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            vec3 reflected = reflect(unit_vector(r_in.direction()), rec.normal);
            scattered = ray(rec.p, reflected + fuzz*random_in_unit_sphere());
            attenuation = albedo;
            return (dot(scattered.direction(), rec.normal) > 0);
        }

    public:
        color albedo;
        double fuzz;
};
```

Listing 51: `[material.h]` *Metal material fuzziness*

우리는 금속에 흐릿함 0.3과 1.0을 추가하여 그것을 시도할 수 있습니다:

```
int main() {
    ...
    // World

    auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
    auto material_center = make_shared<lambertian>(color(0.7, 0.3, 0.3));
    auto material_left   = make_shared<metal>(color(0.8, 0.8, 0.8), 0.3);
    auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 1.0);
    ...
}
```

Listing 52: `[main.cc]` *Metal spheres with fuzziness*



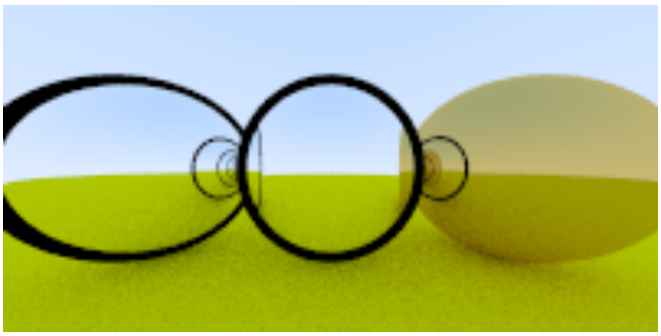
이미지 12: 퍼지 금속

10. 유전체

물, 유리, 다이아몬드와 같은 투명한 물질은 유전체이다. 광선이 그들에게 닿으면, 반사된 광선과 굴절된 (전송된) 광선으로 들어간다. 우리는 반사 또는 굴절 사이에서 무작위로 선택하고, 상호 작용당 하나의 흩어진 광선만 생성함으로써 그것을 처리할 것이다.

10.1. 굴절

디버깅하기 가장 어려운 부분은 굴절된 광선이다. 나는 보통 굴절 광선이 있다면 먼저 모든 빛의 굴절을 가지고 있다. 이 프로젝트를 위해, 나는 우리 장면에 두 개의 유리 공을 넣으려고 노력했고, 나는 이것을 얻었다 (나는 아직 이것을 옳고 그름을 하는 방법을 말하지 않았지만, 곧!):



이미지 13: 유리 먼저

그게 맞나요? 유리 공은 실생활에서 이상하게 보인다. 하지만 아니, 그건 옳지 않아. 세상은 거꾸로 뒤집혀야 하고 이상한 검은 물건은 없어야 한다. 방금 이미지 가운데를 통해 광선을 인쇄했는데 분명히 틀렸어. 그건 종종 그 일을 해.

10.2. 스넬의 법칙

굴절은 스넬의 법칙에 의해 설명된다:

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

어디에θ그리고θ'정상에서의 각도인가요, 그리고η그리고η'("에타"와 "에타 프라임"로 발음)은 굴절률이다 (일반적으로 공기 = 1.0, 유리 = 1.3–1.7, 다이아몬드 = 2.4). 기하학은:

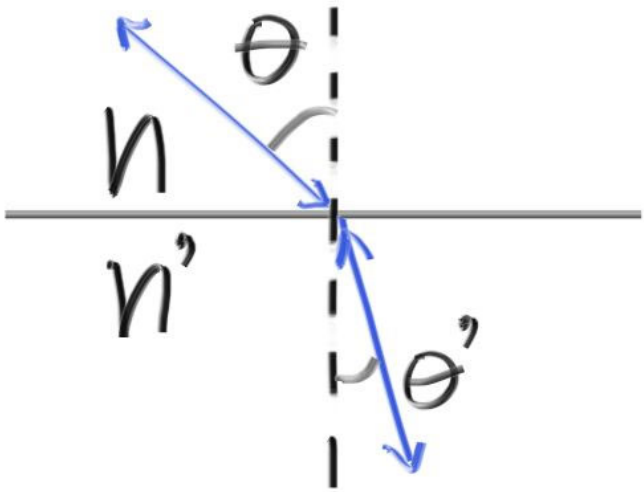


그림 13: *광선 굴절*

굴절된 광선의 방향을 결정하기 위해, 우리는 해결해야 한다.죄 θ' :

$$\text{죄}\,\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

표면의 굴절된 면에는 굴절된 광선이 있다.$\mathbf{R}'$그리고 평범한$\mathbf{n}'$, 그리고 각도가 존재한다,θ', 그들 사이에. 우리는 나눌 수 있어요.$\mathbf{R}'$수직인 광선의 부분으로$\mathbf{n}'$그리고 평행하게$\mathbf{n}'$:

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

만약 우리가 해결한다면$\mathbf{R}'_{\perp}$그리고$\mathbf{R}'_{\parallel}$우리는 얻는다:

$$\mathbf{R}'_{\perp} = \frac{\eta}{\eta'}(\mathbf{R} + \cos\theta\mathbf{n})$$

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

당신이 원한다면 이것을 스스로 증명할 수 있지만, 우리는 그것을 사실로 취급하고 계속 나아갈 것입니다. 그 책의 나머지 부분은 당신이 증거를 이해할 것을 요구하지 않을 것이다.

우리는 여전히 해결해야 해$\cos\theta$. 두 벡터의 내적은 그들 사이의 각도의 코사인 측면에서 설명될 수 있다는 것은 잘 알려져 있다:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}|\cos\theta$$

우리가 제한한다면$\mathbf{a}$그리고$\mathbf{b}$단위 벡터가 되기 위해:

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

우리는 이제 다시 쓸 수 있어$\mathbf{R}'_{\perp}$알려진 양의 관점에서:

$$\mathbf{R}'_{\perp} = \frac{\eta}{\eta'}(\mathbf{R} + (-\mathbf{R} \cdot \mathbf{n})\mathbf{n})$$

우리가 그것들을 다시 결합할 때, 우리는 계산할 함수를 쓸 수 있다.$\mathbf{R}'$:

```
vec3 refract(const vec3& uv, const vec3& n, double etai_over_etat) {
    auto cos_theta = fmin(dot(-uv, n), 1.0);
    vec3 r_out_perp =  etai_over_etat * (uv + cos_theta*n);
    vec3 r_out_parallel = -sqrt(fabs(1.0 - r_out_perp.length_squared())) * n;
    return r_out_perp + r_out_parallel;
}
```

Listing 53: `[vec3.h]` *Refraction function*

그리고 항상 굴절되는 유전체 물질은 다음과 같다:

```
class dielectric : public material {
    public:
        dielectric(double index_of_refraction) : ir(index_of_refraction) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            attenuation = color(1.0, 1.0, 1.0);
            double refraction_ratio = rec.front_face ? (1.0/ir) : ir;

            vec3 unit_direction = unit_vector(r_in.direction());
            vec3 refracted = refract(unit_direction, rec.normal, refraction_ratio);

            scattered = ray(rec.p, refracted);
            return true;
        }

    public:
        double ir; // Index of Refraction
};
```

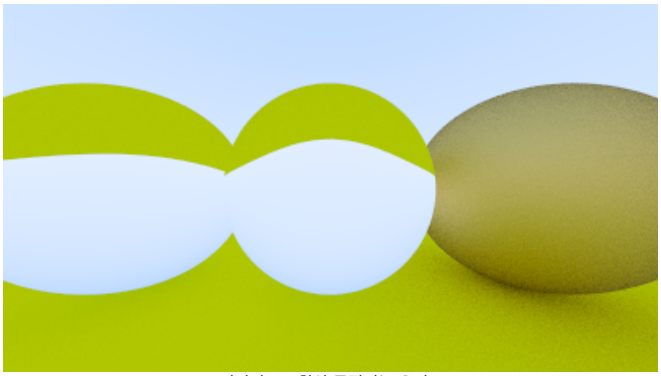
Listing 54: `[material.h]` *Dielectric material class that always refracts*

이제 우리는 왼쪽과 중앙 구체를 유리로 바꾸기 위해 장면을 업데이트할 것입니다:

```
auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
auto material_center = make_shared<dielectric>(1.5);
auto material_left   = make_shared<dielectric>(1.5);
auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 1.0);
```

Listing 55: `[main.cc]` *Changing left and center spheres to glass*

이것은 우리에게 다음과 같은 결과를 준다:



이미지 14: 항상 굴절되는 유리 구

10.3. 완전한 내부 반영

그건 확실히 옳지 않아 보여. 한 가지 성가신 실용적인 문제는 광선이 더 높은 굴절률을 가진 물질에 있을 때, 스넬의 법칙에 대한 진정한 해결책이 없기 때문에 굴절이 불가능하다는 것이다. 만약 우리가 스넬의 법칙과 파생을 다시 언급한다면죄 θ' :

$$\text{죄}\,\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

광선이 유리 안에 있고 외부가 공기라면 ($\eta = 1.5$ 그리고$\eta' = 1.0$):

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

의 가치죄 θ' 1보다 클 수 없어. 그래서, 만약,

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

,

방정식의 양쪽 사이의 평등은 깨졌고, 해결책은 존재할 수 없다. 용액이 존재하지 않는다면, 유리는 굴절될 수 없으므로 광선을 반사해야 한다:

```
if (refraction_ratio * sin_theta > 1.0) {
    // Must Reflect
    ...
} else {
    // Can Refract
    ...
}
```

Listing 56: `[material.h]` *Determining if the ray can refract*

여기서 모든 빛은 반사되며, 실제로는 보통 고체 물체 안에 있기 때문에, 그것은 "총 내부 반사"라고 불린다. 이것이 때때로 당신이 물에 잠길 때 물-공기 경계가 완벽한 거울 역할을 하는 이유입니다.

우리는 삼각 특성을 사용하여 `sin_theta` 해결할 수 있다:

$$\text{죄} = \sqrt{1 - \text{쾌냐}^2\theta}$$

그리고

$$\cos\theta = R \cdot n$$

```
double cos_theta = fmin(dot(-unit_direction, rec.normal), 1.0);
double sin_theta = sqrt(1.0 - cos_theta*cos_theta);

if (refraction_ratio * sin_theta > 1.0) {
    // Must Reflect
    ...
} else {
    // Can Refract
    ...
}
```

Listing 57: `[material.h]` *Determining if the ray can refract*

그리고 (가능한 경우) 항상 굴절되는 유전체 물질은 다음과 같다:

```
class dielectric : public material {
    public:
        dielectric(double index_of_refraction) : ir(index_of_refraction) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            attenuation = color(1.0, 1.0, 1.0);
            double refraction_ratio = rec.front_face ? (1.0/ir) : ir;

            vec3 unit_direction = unit_vector(r_in.direction());
            double cos_theta = fmin(dot(-unit_direction, rec.normal), 1.0);
            double sin_theta = sqrt(1.0 - cos_theta*cos_theta);

            bool cannot_refract = refraction_ratio * sin_theta > 1.0;
            vec3 direction;

            if (cannot_refract)
                direction = reflect(unit_direction, rec.normal);
            else
                direction = refract(unit_direction, rec.normal, refraction_ratio);

            scattered = ray(rec.p, direction);
            return true;
        }

    public:
        double ir; // Index of Refraction
};
```

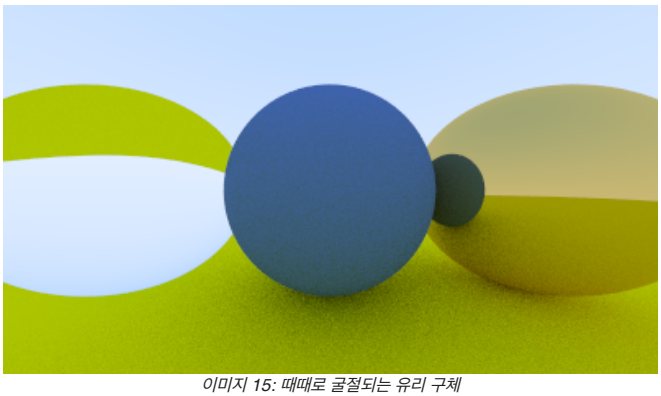
Listing 58: `[material.h]` *Dielectric material class with reflection*

감쇠는 항상 1이다 - 유리 표면은 아무것도 흡수하지 않는다. 만약 우리가 이 매개 변수들로 그것을 시도한다면:

```
auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
auto material_center = make_shared<lambertian>(color(0.1, 0.2, 0.5));
auto material_left   = make_shared<dielectric>(1.5);
auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 0.0);
```

Listing 59: `[main.cc]` *Scene with dielectric and shiny sphere*

우리는 얻는다:



이미지 15: 때때로 굴절되는 유리 구체

10.4. 슐릭 근사치

이제 진짜 유리는 각도에 따라 달라지는 반사율을 가지고 있다 - 가파른 각도로 창문을 보면 거울이 된다. 그것에 대한 큰 추악한 방정식이 있지만, 거의 모든 사람들이 크리스토프 슐릭의 저렴하고 놀라울 정도로 정확한 다항식 근사치를 사용한다. 이것은 우리의 완전한 유리 재료를 산출한다:

```
class dielectric : public material {
    public:
        dielectric(double index_of_refraction) : ir(index_of_refraction) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered
        ) const override {
            attenuation = color(1.0, 1.0, 1.0);
            double refraction_ratio = rec.front_face ? (1.0/ir) : ir;

            vec3 unit_direction = unit_vector(r_in.direction());
            double cos_theta = fmin(dot(-unit_direction, rec.normal), 1.0);
            double sin_theta = sqrt(1.0 - cos_theta*cos_theta);

            bool cannot_refract = refraction_ratio * sin_theta > 1.0;
            vec3 direction;
            if (cannot_refract || reflectance(cos_theta, refraction_ratio) > random_double())
                direction = reflect(unit_direction, rec.normal);
            else
                direction = refract(unit_direction, rec.normal, refraction_ratio);

            scattered = ray(rec.p, direction);
            return true;
        }

    public:
        double ir; // Index of Refraction

    private:
        static double reflectance(double cosine, double ref_idx) {
            // Use Schlick's approximation for reflectance.
            auto r0 = (1-ref_idx) / (1+ref_idx);
            r0 = r0*r0;
            return r0 + (1-r0)*pow((1 - cosine),5);
        }
};
```

Listing 60: `[material.h]` *Full glass material*

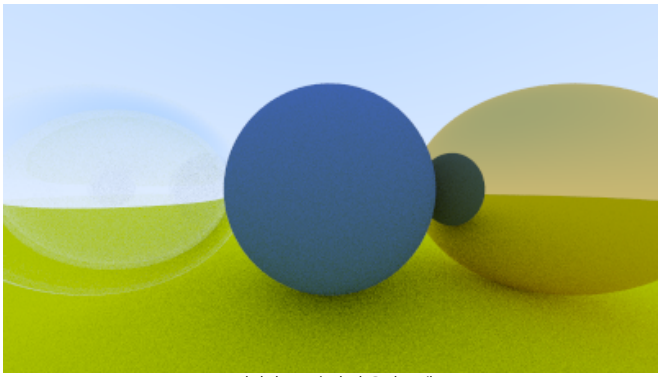
10.5. 빈 유리 구체 모델링하기

유전체 구체의 흥미롭고 쉬운 트릭은 음의 반경을 사용하면 기하학은 영향을 받지 않지만 표면의 법선점은 안쪽으로 향한다는 점에 유의하는 것입니다. 이것은 속이 빈 유리 구체를 만들기 위해 거품으로 사용될 수 있다:

```
world.add(make_shared<sphere>(point3( 0.0, -100.5, -1.0), 100.0, material_ground));
world.add(make_shared<sphere>(point3( 0.0,    0.0, -1.0),   0.5, material_center));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),   0.5, material_left));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),  -0.4, material_left));
world.add(make_shared<sphere>(point3( 1.0,    0.0, -1.0),   0.5, material_right));
```

Listing 61: `[main.cc]` *Scene with hollow glass sphere*

이것은 다음을 준다:



이미지 16: 속이 빈 유리 구체

11. 위치 가능한 카메라

유전체와 같은 카메라는 디버깅하기가 힘들다. 그래서 나는 항상 내 것을 점진적으로 발전시킨다. 먼저, 조정 가능한 시야(*fov*)를 허용합시다. 이것은 당신이 포털을 통해 보는 각도입니다. 우리의 이미지는 정사각형이 아니기 때문에, fov는 수평과 수직으로 다르다. 나는 항상 수직 fov를 사용한다. 나는 또한 보통 그것을 각도로 지정하고 생성자 내부의 라디안으로 바꾼다 - 개인적인 취향의 문제이다.

11.1. 카메라 보기 기하학

나는 먼저 원점에서 오는 광선을 유지하고$z = -1$ 비행기. 우리는 해낼 수 있어.$z = -2$ 비행기든 뭐든, 우리가 만든 한h그 거리에 대한 비율. 여기 우리의 설정이 있습니다:

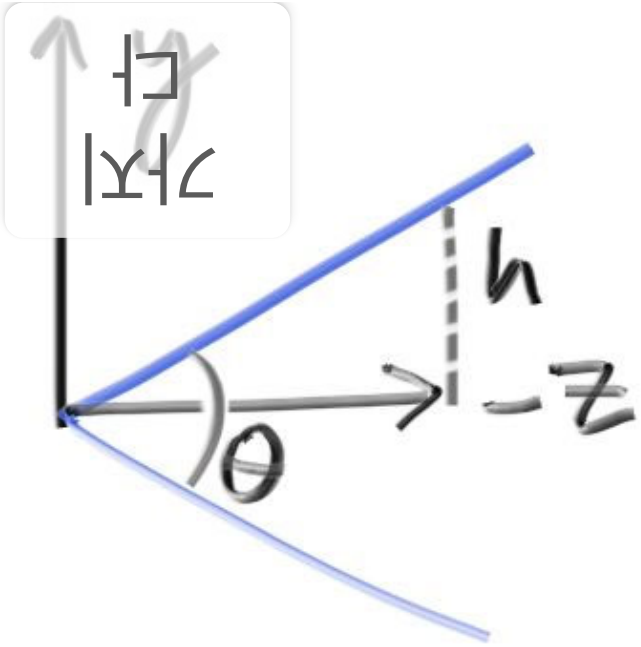


그림 14: *카메라 보기 기하학*

이것은 암시한다 $h = \tan(\frac{\theta}{2})$. 우리의 카메라는 이제:

```
class camera {
    public:
        camera(
            double vfov, // vertical field-of-view in degrees
            double aspect_ratio
        ) {
            auto theta = degrees_to_radians(vfov);
            auto h = tan(theta/2);
            auto viewport_height = 2.0 * h;
            auto viewport_width = aspect_ratio * viewport_height;

            auto focal_length = 1.0;

            origin = point3(0, 0, 0);
            horizontal = vec3(viewport_width, 0.0, 0.0);
            vertical = vec3(0.0, viewport_height, 0.0);
            lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0, focal_length);
        }

        ray get_ray(double u, double v) const {
            return ray(origin, lower_left_corner + u*horizontal + v*vertical - origin);
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};
```

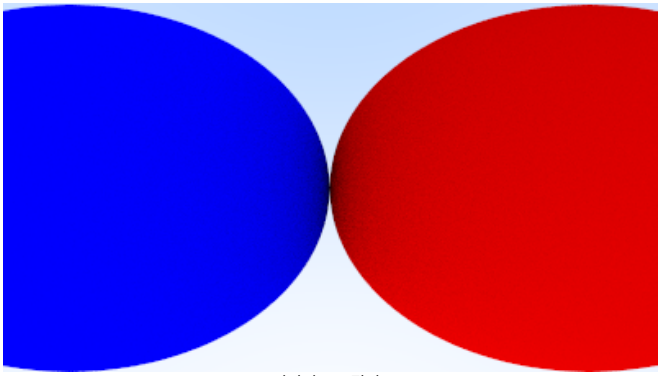
Listing 62: `[camera.h]` *Camera with adjustable field-of-view (fov)*

카메라 `cam(90, aspect_ratio)`과 이 구체로 호출할 때:

```
int main() {
    ...
    // World

    auto R = cos(pi/4);
    hittable_list world;

    auto material_left  = make_shared<lambertian>(color(0,0,1));
    auto material_right = make_shared<lambertian>(color(1,0,0));

    world.add(make_shared<sphere>(point3(-R, 0, -1), R, material_left));
    world.add(make_shared<sphere>(point3( R, 0, -1), R, material_right));

    // Camera

    camera cam(90.0, aspect_ratio);

    // Render

    std::cout << "P3\n" << image_width << " " << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
    ...
```

Listing 63: `[main.cc]` *Scene with wide-angle camera*

주는:



이미지 17: 광각도

11.2. 카메라의 포지셔닝과 방향 지정

임의적인 관점을 얻기 위해, 우리가 신경 쓰는 점의 이름을 짓기 위해. 우리는 카메라를 배치하는 위치와 우리가 보는 지점을 부를 것이다. (나중에, 당신이 원한다면, 볼 지점 대신 볼 방향을 정의할 수 있습니다.)

We also need a way to specify the roll, or sideways tilt, of the camera: the rotation around the lookat-lookfrom axis. Another way to think about it is that even if you keep `lookfrom` and `lookat` constant, you can still rotate your head around your nose. What we need is a way to specify an “up” vector for the camera. This up vector should lie in the plane orthogonal to the view direction.

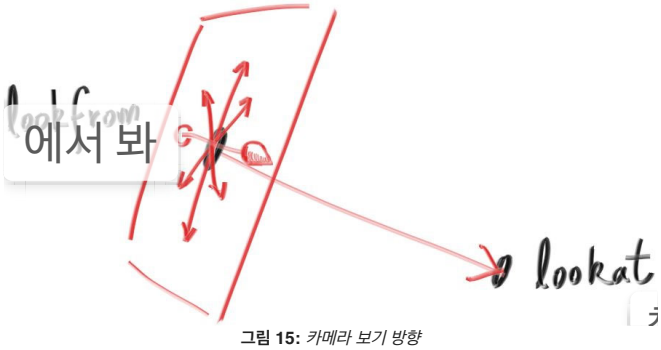


그림 15: *카메라 보기 방향*

우리는 실제로 우리가 원하는 모든 업 벡터를 사용할 수 있으며, 단순히 이 비행기에 투사하여 카메라의 업 벡터를 얻을 수 있습니다. 나는 "뷰업"(*vup*) 벡터를 명명하는 일반적인 규칙을 사용한다. 몇 개의 교차 제품, 그리고 우리는 이제 완전한 직교 정규 기반을 가지고 있다. (u, v, w) 우리 카메라의 방향을 설명하기 위해.

그림 16: *카메라 방향 보기*

vup, v, 그리고 w 모두 같은 비행기에 있다는 것을 기억하세요. 우리의 고정 카메라가 -Z를 마주했을 때와 마찬가지로, 우리의 임의 뷰 카메라는 -w를 향합니다. 그리고 우리가 세상을 사용할 수 있다는 것을 명심하세요. 하지만 그럴 필요는 없습니다.$(0,1,0)$ vup을 지정하기 위해. 이것은 편리하며 미친 카메라 각도로 실험하기로 결정할 때까지 자연스럽게 카메라를 수평으로 유지할 것입니다.

```
class camera {
    public:
        camera(
            point3 lookfrom,
            point3 lookat,
            vec3   vup,
            double vfov, // vertical field-of-view in degrees
            double aspect_ratio
        ) {
            auto theta = degrees_to_radians(vfov);
            auto h = tan(theta/2);
            auto viewport_height = 2.0 * h;
            auto viewport_width = aspect_ratio * viewport_height;

            auto w = unit_vector(lookfrom - lookat);
            auto u = unit_vector(cross(vup, w));
            auto v = cross(w, u);

            origin = lookfrom;
            horizontal = viewport_width * u;
            vertical = viewport_height * v;
            lower_left_corner = origin - horizontal/2 - vertical/2 - w;
        }

        ray get_ray(double s, double t) const {
            return ray(origin, lower_left_corner + s*horizontal + t*vertical - origin);
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};
```

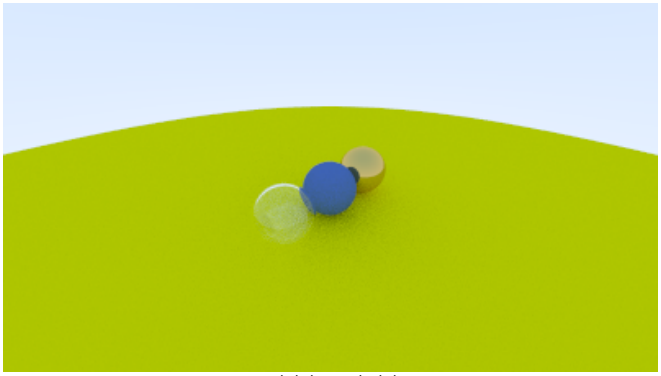
Listing 64: `[camera.h]` *Positionable and orientable camera*

우리는 이전 장면으로 돌아가서, 새로운 관점을 사용할 것이다:

```
hittable_list world;

auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
auto material_center = make_shared<lambertian>(color(0.1, 0.2, 0.5));
auto material_left   = make_shared<dielectric>(1.5);
auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 0.0);

world.add(make_shared<sphere>(point3( 0.0, -100.5, -1.0), 100.0, material_ground));
world.add(make_shared<sphere>(point3( 0.0,    0.0, -1.0),   0.5, material_center));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),   0.5, material_left));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0), -0.45, material_left));
world.add(make_shared<sphere>(point3( 1.0,    0.0, -1.0),   0.5, material_right));

camera cam(point3(-2,2,1), point3(0,0,-1), vec3(0,1,0), 90, aspect_ratio);
```

Listing 65: `[main.cc]` *Scene with alternate viewpoint*

얻기 위해:



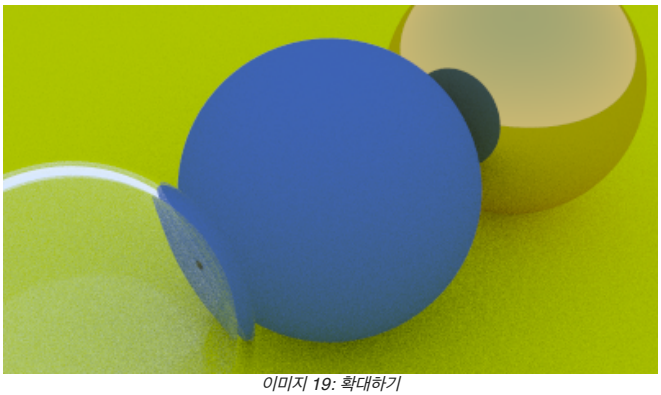
이미지 18: 먼 전망

그리고 우리는 시야를 바꿀 수 있다:

```
camera cam(point3(-2,2,1), point3(0,0,-1), vec3(0,1,0), 20, aspect_ratio);
```

Listing 66: `[main.cc]` *Change field of view*

얻기 위해:



이미지 19: 확대하기

12. 디포커스 블러

이제 우리의 마지막 특징: 디포커스 블러. 참고로, 모든 사진작가들은 그것을 "필드의 깊이"라고 부를 것이므로 친구들 사이에서 "디포커스 블러"만 사용하는 것을 주의하십시오.

우리가 실제 카메라에서 흐릿함을 끄는 이유는 빛을 모으기 위해 큰 구멍(단단 구멍이 아닌)이 필요하기 때문이다. 이것은 모든 것의 초점을 떨어뜨릴 것이지만, 우리가 구멍에 렌즈를 꽂으면, 모든 것이 초점을 맞춘 특정 거리가 있을 것이다. 렌즈를 이런 식으로 생각할 수 있습니다: 초점 거리의 특정 지점에서 오는 모든 광선은 이미지 센서의 단일 지점으로 구부러질 것입니다.

우리는 투영점과 모든 것이 완벽하게 초점을 맞춘 평면 사이의 거리를 *초점 거리라고* 부른다. 초점 거리는 초점 거리와 같지 않다는 점에 유의하십시오. *초점 거리는* 투영점과 이미지 평면 사이의 거리입니다.

물리적 카메라에서 초점 거리는 렌즈와 필름/센서 사이의 거리에 의해 제어된다. 그것이 초점이 바뀔 때 렌즈가 카메라에 상대적으로 움직이는 것을 보는 이유입니다(휴대폰 카메라에서도 발생할 수 있지만 센서는 움직입니다). "조공"은 렌즈가 얼마나 큰지 효과적으로 제어하는 구멍이다. 실제 카메라의 경우, 더 많은 빛이 필요하다면 조리개를 더 크게 만들고, 더 많은 디포커스 블러를 얻을 것이다. 가상 카메라의 경우, 우리는 완벽한 센서를 가질 수 있고 더 많은 빛이 필요하지 않으므로, 디포커스 블러를 원할 때만 조리개가 있습니다.

12.1. 얇은 렌즈 근사치

실제 카메라는 복잡한 복합 렌즈를 가지고 있다. 우리의 코드를 위해 우리는 순서를 시뮬레이션할 수 있다: 센서, 렌즈, 조리개. 그런 다음 우리는 광선을 어디로 보낼지 알아낼 수 있고, 계산된 후 이미지를 뒤집을 수 있다(이미지는 필름에 거꾸로 투사된다). 그러나 그래픽 사람들은 보통 얇은 렌즈 근사치를 사용한다:

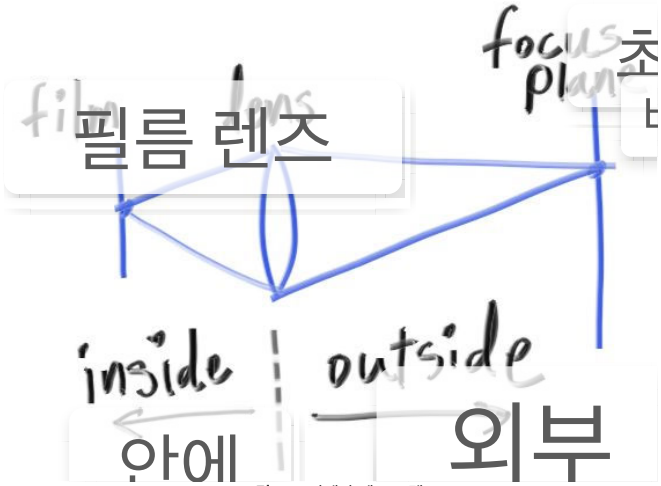


그림 17: *카메라 렌즈 모델*

우리는 카메라 내부를 시뮬레이션할 필요가 없습니다. 카메라 외부에서 이미지를 렌더링하기 위해, 그것은 불필요한 복잡성이 될 것이다. 대신, 나는 보통 렌즈에서 광선을 시작하고, 그 비행기의 모든 것이 완벽한 초점에 있는 초점 평면(렌즈에서 멀리 `focus_dist`)으로 보낸다.

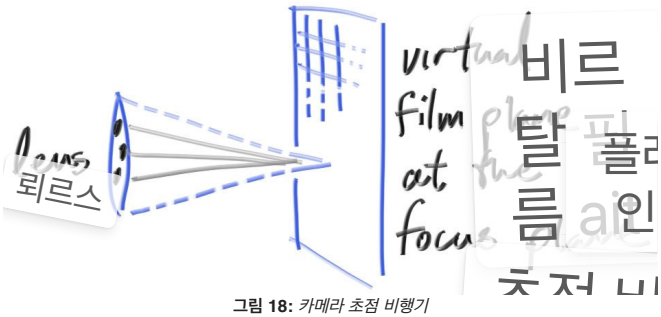


그림 18: *카메라 초점 비행기*

12.2. 샘플 광선 생성

Normally, all scene rays originate from the `lookfrom` point. In order to accomplish defocus blur, generate random scene rays originating from inside a disk centered at the `lookfrom` point. The larger the radius, the greater the defocus blur. You can think of our original camera as having a defocus disk of radius zero (no blur at all), so all rays originated at the disk center (`lookfrom`).

```
vec3 random_in_unit_disk() {
    while (true) {
        auto p = vec3(random_double(-1,1), random_double(-1,1), 0);
        if (p.length_squared() >= 1) continue;
        return p;
    }
}
```

Listing 67: `[vec3.h]` *Generate random point inside unit disk*

```
class camera {
    public:
        camera(
            point3 lookfrom,
            point3 lookat,
            vec3   vup,
            double vfov, // vertical field-of-view in degrees
            double aspect_ratio,
            double aperture,
            double focus_dist
        ) {
            auto theta = degrees_to_radians(vfov);
            auto h = tan(theta/2);
            auto viewport_height = 2.0 * h;
            auto viewport_width = aspect_ratio * viewport_height;

            w = unit_vector(lookfrom - lookat);
            u = unit_vector(cross(vup, w));
            v = cross(w, u);

            origin = lookfrom;
            horizontal = focus_dist * viewport_width * u;
            vertical = focus_dist * viewport_height * v;
            lower_left_corner = origin - horizontal/2 - vertical/2 - focus_dist*w;

            lens_radius = aperture / 2;
        }


        ray get_ray(double s, double t) const {
            vec3 rd = lens_radius * random_in_unit_disk();
            vec3 offset = u * rd.x() + v * rd.y();

            return ray(
                origin + offset,
                lower_left_corner + s*horizontal + t*vertical - origin - offset
            );
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
        vec3 u, v, w;
        double lens_radius;
};
```

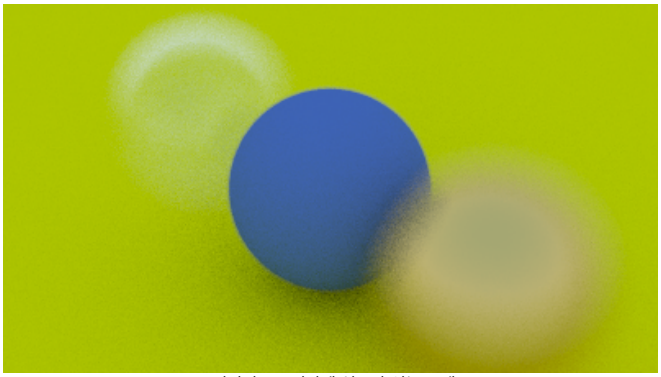
Listing 68: `[camera.h]` *Camera with adjustable depth-of-field (dof)*

큰 조리개 사용하기:

```
point3 lookfrom(3,3,2);
point3 lookat(0,0,-1);
vec3 vup(0,1,0);
auto dist_to_focus = (lookfrom-lookat).length();
auto aperture = 2.0;

camera cam(lookfrom, lookat, vup, 20, aspect_ratio, aperture, dist_to_focus);
```

Listing 69: `[main.cc]` *Scene camera with depth-of-field*

우리는 얻는다:



이미지 20: 피사계 심도가 있는 구체

13. 다음은 어디야?

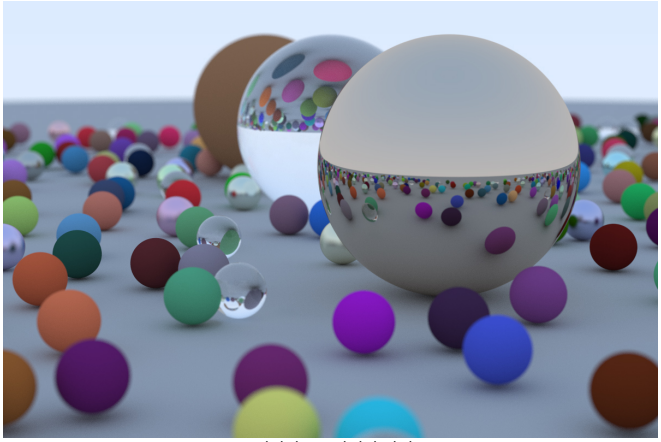
13.1. 최종 렌더링

먼저 이 책의 표지에 이미지를 만들어 봅시다 - 많은 무작위 구체:

```
hittable_list random_scene() {
    hittable_list world;

    auto ground_material = make_shared<lambertian>(color(0.5, 0.5, 0.5));
    world.add(make_shared<sphere>(point3(0,-1000,0), 1000, ground_material));

    for (int a = -11; a < 11; a++) {
        for (int b = -11; b < 11; b++) {
            auto choose_mat = random_double();
            point3 center(a + 0.9*random_double(), 0.2, b + 0.9*random_double());

            if ((center - point3(4, 0.2, 0)).length() > 0.9) {
                shared_ptr<material> sphere_material;

                if (choose_mat < 0.8) {
                    // diffuse
                    auto albedo = color::random() * color::random();
                    sphere_material = make_shared<lambertian>(albedo);
                    world.add(make_shared<sphere>(center, 0.2, sphere_material));
                } else if (choose_mat < 0.95) {
                    // metal
                    auto albedo = color::random(0.5, 1);
                    auto fuzz = random_double(0, 0.5);
                    sphere_material = make_shared<metal>(albedo, fuzz);
                    world.add(make_shared<sphere>(center, 0.2, sphere_material));
                } else {
                    // glass
                    sphere_material = make_shared<dielectric>(1.5);
                    world.add(make_shared<sphere>(center, 0.2, sphere_material));
                }
            }
        }
    }

    auto material1 = make_shared<dielectric>(1.5);
    world.add(make_shared<sphere>(point3(0, 1, 0), 1.0, material1));

    auto material2 = make_shared<lambertian>(color(0.4, 0.2, 0.1));
    world.add(make_shared<sphere>(point3(-4, 1, 0), 1.0, material2));

    auto material3 = make_shared<metal>(color(0.7, 0.6, 0.5), 0.0);
    world.add(make_shared<sphere>(point3(4, 1, 0), 1.0, material3));

    return world;
}

int main() {

    // Image

    const auto aspect_ratio = 3.0 / 2.0;
    const int image_width = 1200;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 500;
    const int max_depth = 50;

    // World

    auto world = random_scene();

    // Camera

    point3 lookfrom(13,2,3);
    point3 lookat(0,0,0);
    vec3 vup(0,1,0);
    auto dist_to_focus = 10.0;
    auto aperture = 0.1;

    camera cam(lookfrom, lookat, vup, 20, aspect_ratio, aperture, dist_to_focus);

    // Render

    std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        ...
}
```

Listing 70: `[main.cc]` *Final scene*

이것은 다음을 준다:



이미지 21: 마지막 장면

당신이 주목할 수 있는 흥미로운 점은 유리 공이 실제로 떠 있는 것처럼 보이게 하는 그림자가 없다는 것입니다. 이것은 벌레가 아닙니다 - 당신은 실생활에서 유리 공을 많이 볼 수 없으며, 유리 공도 약간 이상해 보이고, 실제로 흐린 날에 떠 있는 것처럼 보입니다. 유리 공 아래의 큰 구체의 한 지점은 하늘이 막히지 않고 재정렬되었기 때문에 여전히 많은 빛을 치고 있다.

13.2. 다음 단계

넌 이제 멋진 레이 트레이서를 가지고 있어! 다음은 뭐야?

1. 조명 — 그림자 광선을 빛으로 보내 명시적으로 할 수 있거나, 일부 물체가 빛을 방출하게 하고, 산란된 광선을 편향시킨 다음, 편향을 상쇄하기 위해 그 광선을 하향시킴으로써 암묵적으로 할 수 있습니다. 둘 다 일해. 나는 후자의 접근 방식을 선호하는 소수이다.
2. 삼각형 — 가장 멋진 모델은 삼각형 형태이다. 모델 I/O는 최악이며 거의 모든 사람들이 이것을 하기 위해 다른 사람의 코드를 얻으려고 한다.
3. 표면 텍스처 — 벽지처럼 이미지를 붙여넣을 수 있습니다. 꽤 쉽고 좋은 일이야.
4. 단단한 질감 — 켄 펄린은 그의 코드를 온라인에 가지고 있다. 앤드류 켄슬러는 그의 블로그에 아주 멋진 정보를 가지고 있다.
5. 볼륨과 미디어 — 멋진 것들과 당신의 소프트웨어 아키텍처에 도전할 것입니다. 나는 볼륨이 히트테이블 인터페이스를 가지고 있고 확률적으로 밀도에 기반한 교차점을 갖는 것을 선호한다. 당신의 렌더링 코드는 그 방법으로 볼륨이 있다는 것을 알 필요조차 없습니다.
6. 평행주의 — 실행N 당신의 코드 사본N 다른 무작위 씨앗이 있는 코어. 평균N 뛰어. 이 평균은 또한 계층적으로 수행될 수 있다. $N/2$ 한 쌍을 평균화할 수 있어$N/4$ 이미지와 그 쌍은 평균화될 수 있다. 그 병렬 처리 방법은 코딩이 거의 없는 수천 개의 코어로 잘 확장되어야 한다.

재밌게 놀아, 그리고 네 멋진 이미지 좀 보내줘!

14. 감사

원본 원고 도움말

데이브 하트
진 버클리

웹 출시

베르나 카바다예
로렌조 만치니
로리 휘플러 홀라쉬
로널드 워츨로

수정 및 개선

아리아만 바시슈타
앤드루 켄슬러
안토니오 가미즈
아푸르바 조시
아라스 프란케비치우스
베커
벤 켈
벤자민 서머턴
베넷 하드윅
댄 드러먼드
데이비드 챔버스
데이비드 하트
에릭 헤인즈
파비오 산시네티
필리페 스커
프랭크 그
게릿 베센도르프
그루 데브리
잉고 월드
제이슨 스톤
진 버클리
조이 조
존 킬패트릭
칸 에라슬란
로렌조 만치니
마나스 케일
마커스 오토슨
마크 크레이그
매튜 하임리히
나카타 다이스케
폴 멜리스
필 크리스텐슨
로널드 워츨로
숀 P. 리
가와지리 쇼타
오가와 타츠야
티아고 이제
바한 소소얀
제하오 첸

특별한 감사

수치에 도움을 주신 Limnu의 팀에게 감사드립니다.

이 책들은 전적으로 모건 맥과이어의 환상적이고 무료인 마크딥 도서관에서 쓰여졌다. 이것이 어떻게 생겼는지 보려면, 브라우저에서 페이지 소스를 보세요.

그녀의 https://github.com/RayTracing/ GitHub 조직을 이 프로젝트에 기부해 주신 Helen Hu에게 감사드립니다.

15. 이 책을 인용하기

일관된 인용은 이 작품의 출처, 위치 및 버전을 더 쉽게 식별할 수 있게 해준다. 이 책을 인용한다면, 가능하다면 다음 양식 중 하나를 사용해 보라고 요청합니다.

15.1. 기본 데이터

- **제목 (시리즈):** "한 주말 시리즈의 레이 트레이싱"
- **제목 (책):** "한 주말에 레이 트레이싱"
- **저자**: 피터 셜리
- **편집자**: 스티브 홀라쉬, 트레버 데이비드 블랙
- **버전/버전**: v3.2.3
- **날짜**: 2020-12-07
- **URL (시리즈):** https://raytracing.github.io/
- **URL (책):** https://raytracing.github.io/books/RayTracingInOneWeekend.html

15.2. 스니펫

15.2.1 마크다운

```
[_Ray Tracing in One Weekend_](https://raytracing.github.io/books/RayTracingInOneWeekend.html)
```

15.2.2 HTML

```
<a href="https://raytracing.github.io/books/RayTracingInOneWeekend.html">
    <cite>Ray Tracing in One Weekend</cite>
</a>
```

15.2.3 LaTeX와 BibTex

```
~\cite{Shirley2020RTW1}

@misc{Shirley2020RTW1,
   title = {Ray Tracing in One Weekend},
   author = {Peter Shirley},
   year = {2020},
   month = {December},
   note = {\small \texttt{https://raytracing.github.io/books/RayTracingInOneWeekend.html}},
   url = {https://raytracing.github.io/books/RayTracingInOneWeekend.html}
}
```

15.2.4 빕라텍스

```
\usepackage{biblatex}

~\cite{Shirley2020RTW1}

@online{Shirley2020RTW1,
   title = {Ray Tracing in One Weekend},
   author = {Peter Shirley},
   year = {2020},
   month = {December}
   url = {https://raytracing.github.io/books/RayTracingInOneWeekend.html}
}
```

15.2.5 IEEE

```
“Ray Tracing in One Weekend.” raytracing.github.io/books/RayTracingInOneWeekend.html
(accessed MMM. DD, YYYY)
```

15.2.6 MLA:

```
Ray Tracing in One Weekend. raytracing.github.io/books/RayTracingInOneWeekend.html
Accessed DD MMM. YYYY.
```