Introduction to Real-Time Ray Tracing

Morgan McGuire, Peter Shirley, and Chris Wyman

SIGGRAPH 2019

COURSE NOTES

SIGGRAPH '19 Courses, July 28 - August 01, 2019, Los Angeles, CA, USA
ACM 978-1-4503-6307-5/19/07.
10.1145/3305366.3328047

1. Images.

Whenever you start a renderer, you need a way to see an image. The most straightforward way is to write it to a file. The catch is, there are so many formats and many of those are complex. I always start with a plain text ppm file. Here's a nice description from Wikipedia:

PPM example [edit]

This is an example of a color RGB image stored in PPM format. There is a newline character at the end of each line.

```
P3
# The P3 means colors are in ASCII, then 3 columns and 2 rows,
# then 255 for max color, then RGB triplets
3 2
255
255   0   0     0 255   0     0   0 255
255 255   0   255 255 255     0   0   0
```

Image
(magnified)

Let's make some C++ code to output such a thing:

```
#include <iostream>

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    for (int j = ny-1; j >= 0; j--) {
       for (int i = 0; i < nx; i++) {
           float r = float(i) / float(nx);
           float g = float(j) / float(ny);
           float b = 0.2;
           int ir = int(255.99*r);
           int ig = int(255.99*g);
           int ib = int(255.99*b);
           std::cout << ir << " " << ig << " " << ib << "\n";
       }
    }
}
```

There are some things to note in that code:

1. The pixels are written out in rows with pixels left to right.
2. The rows are written out from top to bottom.
3. By convention, each of the red/green/blue components range from 0.0 to 1.0. We will relax that later when we internally use high dynamic range, but before output we will tone map to the zero to one range, so this code won't change.
4. Red goes from black to fully on from left to right, and green goes from black at the bottom to fully on at the top. Red and green together make yellow so we should expect the upper right corner to be yellow.

Opening the output file (in ToyViewer on my mac, but try it in your favorite viewer and google "ppm viewer" if your viewer doesn't support it) shows:

Hooray! This is the graphics "hello world". If your image doesn't look like that, open the output file in a text editor and see what it looks like. It should start something like this:

```
P3
200 100
255
0 253 51
1 253 51
2 253 51
3 253 51
5 253 51
6 253 51
7 253 51
8 253 51
10 253 51
11 253 51
12 253 51
14 253 51
15 253 51
```

If it doesn't, then you probably just have some newlines or something similar that is confusing the image reader.

If you want to produce more image types than PPM, I am a fan of stb_image.h available on github.

2. The vec3 class

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.

Here's the top part of my vec3 class:

```
#include <math.h>
#include <stdlib.h>
#include <iostream>

class vec3  {
public:
    vec3() {}
    vec3(float e0, float e1, float e2) { e[0] = e0; e[1] = e1; e[2] = e2; }
    inline float x() const { return e[0]; }
    inline float y() const { return e[1]; }
    inline float z() const { return e[2]; }
    inline float r() const { return e[0]; }
    inline float g() const { return e[1]; }
    inline float b() const { return e[2]; }

    inline const vec3& operator+() const { return *this; }
    inline vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
    inline float operator[](int i) const { return e[i]; }
    inline float& operator[](int i) { return e[i]; };

    inline vec3& operator+=(const vec3 &v2);
    inline vec3& operator-=(const vec3 &v2);
    inline vec3& operator*=(const vec3 &v2);
    inline vec3& operator/=(const vec3 &v2);
    inline vec3& operator*=(const float t);
    inline vec3& operator/=(const float t);

    inline float length() const {
        return sqrt(e[0]*e[0] + e[1]*e[1] + e[2]*e[2]); }
    inline float squared_length() const {
        return e[0]*e[0] + e[1]*e[1] + e[2]*e[2]; }
    inline void make_unit_vector();

    float e[3];
};
```

I use floats here, but in some ray tracers I have used doubles. Neither is correct-- follow your own tastes. Everything is in the header file, and later on in the file are lots of vector operations:

```
inline std::istream& operator>>(std::istream &is, vec3 &t) {
    is >> t.e[0] >> t.e[1] >> t.e[2];
    return is;
}

inline std::ostream& operator<<(std::ostream &os, const vec3 &t) {
    os << t.e[0] << " " << t.e[1] << " " << t.e[2];
    return os;
}

inline void vec3::make_unit_vector() {
    float k = 1.0 / sqrt(e[0]*e[0] + e[1]*e[1] + e[2]*e[2]);
    e[0] *= k; e[1] *= k; e[2] *= k;
}

inline vec3 operator+(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] + v2.e[0], v1.e[1] + v2.e[1], v1.e[2] + v2.e[2]);
}

inline vec3 operator-(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] - v2.e[0], v1.e[1] - v2.e[1], v1.e[2] - v2.e[2]);
}

inline vec3 operator*(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] * v2.e[0], v1.e[1] * v2.e[1], v1.e[2] * v2.e[2]);
}

inline vec3 operator/(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] / v2.e[0], v1.e[1] / v2.e[1], v1.e[2] / v2.e[2]);
}

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

inline vec3 operator/(vec3 v, float t) {
    return vec3(v.e[0]/t, v.e[1]/t, v.e[2]/t);
}

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

inline float dot(const vec3 &v1, const vec3 &v2) {
    return v1.e[0] *v2.e[0] + v1.e[1] *v2.e[1]  + v1.e[2] *v2.e[2];
}

inline vec3 cross(const vec3 &v1, const vec3 &v2) {
    return vec3( (v1.e[1]*v2.e[2] - v1.e[2]*v2.e[1]),
                (-(v1.e[0]*v2.e[2] - v1.e[2]*v2.e[0])),
                (v1.e[0]*v2.e[1] - v1.e[1]*v2.e[0]));
}
```

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

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

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

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

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

inline vec3& vec3::operator/=(const float t) {
    float k = 1.0/t;

    e[0]  *= k;
    e[1]  *= k;
    e[2]  *= k;
    return *this;
}

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

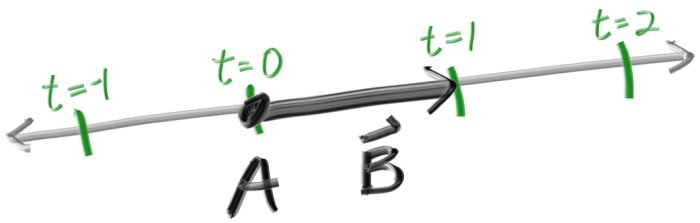
Now we can change our main to use this:

```
#include <iostream>
#include "vec3.h"

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    for (int j = ny-1; j >= 0; j--) {
       for (int i = 0; i < nx; i++) {
           vec3 col(float(i) / float(nx), float(j) / float(ny), 0.2);
           int ir = int(255.99*col[0]);
           int ig = int(255.99*col[1]);
           int ib = int(255.99*col[2]);

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

Rays, a simple camera, and background

The one thing that all ray tracers have is a ray class, and a computation of what color is seen along a ray. Let's think of a ray as a function $\boldsymbol{p}(t) = \boldsymbol{A} + t^*\boldsymbol{B}$. Here $\boldsymbol{p}$ is a 3D position along a line in 3D. $\boldsymbol{A}$ is the ray origin and $\boldsymbol{B}$ is the ray direction. The ray parameter t is a real number (float in the code). Plug in a different t and $\boldsymbol{p}(t)$ moves the point along the ray. Add in negative t and you can go anywhere on the 3D line. For positive t, you get only the parts in front of $\boldsymbol{A}$,

and this is what is often called a half-line or ray. The example $\boldsymbol{C} = \boldsymbol{p}(2)$ is shown here:



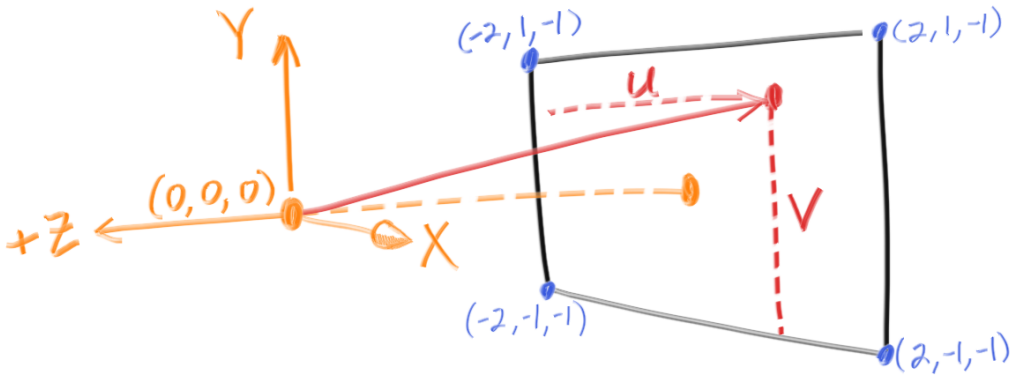
The function $\boldsymbol{p}(t)$ in more verbose code form I call "point_at_parameter(t)":

```
#ifndef RAYH
#define RAYH
#include "vec3.h"

class ray
{
    public:
        ray() {}
        ray(const vec3& a, const vec3& b) { A = a; B = b; }
        vec3 origin() const       { return A; }
        vec3 direction() const    { return B; }
        vec3 point_at_parameter(float t) const { return A + t*B; }

        vec3 A;
        vec3 B;
};

#endif
```

Now we are ready to turn the corner and make a ray tracer. At the core of a ray tracer is to send rays through pixels and compute what color is seen in the direction of those rays. This is of the form *calculate which ray goes from the eye to a pixel, compute what that ray intersects, and 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 *color(ray)* function that returns the color of the background (a simple gradient).

I've often gotten into trouble using square images for debugging because I transpose x and y too often, so I'll stick with a 200x100 image. I'll put the "eye" (or camera center if you think of a camera) at (0,0,0). I will have the y-axis go up, and the x-axis to the right. In order to respect the convention of a right handed coordinate system, into the screen is the negative z-axis. I will traverse the screen from the lower left hand corner and use two offset vectors along the screen sides to move the ray endpoint across the screen. Note that I do not make the ray direction a unit length vector because I think not doing that makes for simpler and slightly faster code.



Below in code, the ray **r** goes to approximately the pixel centers (I won't worry about exactness for now because we'll add antialiasing later):

```
#include <iostream>
#include "ray.h"

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

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    vec3 lower_left_corner(-2.0, -1.0, -1.0);
    vec3 horizontal(4.0, 0.0, 0.0);
    vec3 vertical(0.0, 2.0, 0.0);
    vec3 origin(0.0, 0.0, 0.0);
    for (int j = ny-1; j >= 0; j--) {
       for (int i = 0; i < nx; i++) {
           float u = float(i) / float(nx);
           float v = float(j) / float(ny);
           ray r(origin, lower_left_corner + u*horizontal + v*vertical);
           vec3 col = color(r);
           int ir = int(255.99*col[0]);
           int ig = int(255.99*col[1]);
           int ib = int(255.99*col[2]);

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

The *color(ray)* function linearly blends white and blue depending on the up/downess of the y coordinate. I first made it a unit vector so -1.0 < *y* < 1.0. I then did a standard graphics trick of scaling that to 0.0 < t < 1.0. When t=1.0 I want blue. When t = 0.0 I want white. In between, I want a blend. This forms a "linear blend", or "linear interpolation", or "lerp" for short, between two things. A lerp is always of the form: *blended_value = (1-t)*start_value + t*end_value*, with t going from zero to one. In our case this produces:

3. Adding a sphere

Let's add a single object to our ray tracer. People often use spheres in ray tracers because calculating whether a ray hits a sphere is pretty straightforward. Recall that the equation for a sphere centered at the origin of radius R is $x*x + y*y + z*z = R*R$. The way you can read that equation is "for any (x, y, z), if $x*x + y*y + z*z = R*R$ then (x,y,z) is on the sphere and otherwise it is not". It gets uglier if the sphere center is at (cx, cy, cz):

$$(x-cx)*(x-cx) + (y-cy)*(y-cy) + (z-cz)*(z-cz) = R*R$$

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 $\boldsymbol{C} = (cx,cy,cz)$ to point $\boldsymbol{p} = (x,y,z)$ is $(\boldsymbol{p} - \boldsymbol{C})$. And $dot((\boldsymbol{p} - \boldsymbol{C}),(\boldsymbol{p} - \boldsymbol{C})) = (x-cx)*(x-cx) + (y-cy)*(y-cy) + (z-cz)*(z-cz)$. So the equation of the sphere in vector form is:

$$dot((\boldsymbol{p} - \boldsymbol{c}),(\boldsymbol{p} - \boldsymbol{c})) = R*R$$

We can read this as "any point p that satisfies this equation is on the sphere". We want to know if our ray $\boldsymbol{p}(t) = \boldsymbol{A} + t*\boldsymbol{B}$ ever hits the sphere anywhere. If it does hit the sphere, there is *some* t for which $\boldsymbol{p}(t)$ satisfies the sphere equation. So we are looking for any t where this is true:

$$dot((\boldsymbol{p}(t) - \boldsymbol{c}),(\boldsymbol{p}(t) - \boldsymbol{c})) = R*R$$

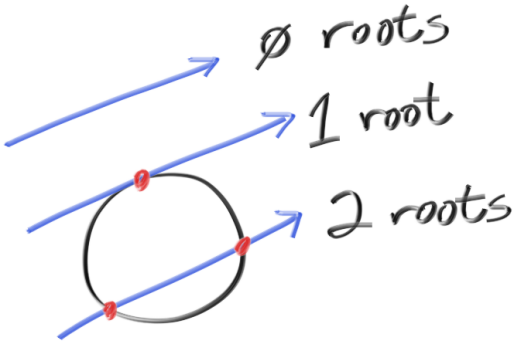
or expanding the full form of the ray $\boldsymbol{p}(t)$:

$$dot((\boldsymbol{A} + t*\boldsymbol{B} - \boldsymbol{C}),(\boldsymbol{A} + t*\boldsymbol{B} - \boldsymbol{C})) = R*R$$

The rules of vector algebra are all that we would want here, and if we expand that equation and move all the terms to the left hand side we get:

$$t*t*dot(\boldsymbol{B},\boldsymbol{B}) + 2*t*dot(\boldsymbol{B},\boldsymbol{A}-\boldsymbol{C}) + dot(\boldsymbol{A}-\boldsymbol{C},\boldsymbol{A}-\boldsymbol{C}) - R*R = 0$$

The vectors and R in that equation are all constant and known. The unknown is t, and the equation is a quadratic, like you probably saw in your high school math class. You can solve for t and there is a square root part that is either positive (meaning two real solutions), negative (meaning no real solutions), or zero (meaning one real solution). In graphics, the algebra almost always relates very directly to the geometry. What we have is:



If we take that math and hard-code it into our program, we can test it by coloring red any pixel that hits a small sphere we place at -1 on the z-axis:

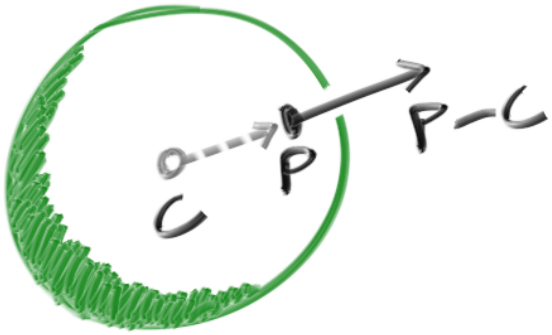
```
bool hit_sphere(const vec3& center, float radius, const ray& r) {
   vec3 oc = r.origin() - center;
   float a = dot(r.direction(), r.direction());
   float b = 2.0 * dot(oc, r.direction());
   float c = dot(oc, oc) - radius*radius;
   float discriminant = b*b - 4*a*c;
   return (discriminant > 0);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

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

What we get is this:

Now this lacks all sorts of things-- like shading and reflection rays and more than one object-- but we are closer to halfway done than we are to our start! One thing to be aware of is that we tested whether the ray hits the sphere at all, but $t < 0$ solutions work fine. If you change your sphere center to $z = +1$ you will get exactly the same picture because you see the things behind you. This is not a feature! We'll fix those issues next.

4. Surface normals and multiple objects.

First, let's get ourselves a surface normal so we can shade. This is a vector that is perpendicular to the surface, and by convention, points out. One design decision is whether these normals (again by convention) are unit length. That is convenient for shading so I will say yes, but I won't enforce that in the code. This could allow subtle bugs, so be aware this is personal preference as are most design decisions like that. For a sphere, the normal is in the direction of the hitpoint minus the center:



On the earth, this implies that the vector from the earth's center to you points straight up. Let's throw that into the code now, and shade it. We don't have any lights or anything yet, so let's just visualize the normals with a color map. A common trick used for visualizing normals (because it's easy and somewhat intuitive to assume ***N*** is a unit length vector– so each component is between -1 and 1) is to map each component to the interval from 0 to 1, and then map x/y/z to r/g/b. For the normal we need the hit point, not just whether we hit or not. Let's assume the closest hit point (smallest t). These changes in the code let us compute and visualize ***N***:

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

vec3 color(const ray& r) {
    float t = hit_sphere(vec3(0,0,-1), 0.5, r);
    if (t > 0.0) {
        vec3 N = unit_vector(r.point_at_parameter(t) - vec3(0,0,-1));
        return 0.5*vec3(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)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
}
```

And that yields this picture:

Now, how about several spheres? While it is tempting to have an array of spheres, a very clean solution is the make an “abstract class” for anything a ray might hit and make both a sphere and a list of spheres just something you can hit. What that class should be called is something of a quandary-- calling it an “object” would be good if not for “object oriented” programming. “Surface” is often used, with the weakness being maybe we will want volumes. “Hitable”

emphasizes the member function that unites them. I don't love any of these but I will go with "hitable".

This hitable 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 tmin to tmax, so the hit only "counts" if $tmin < t < tmax$. For the initial rays this is positive t, but as we will see, it can help some details in the code to have an interval *tmin to tmax*. One design question is whether to do things like compute the normal if we hit something; we might end up hitting something closer as we do our search, and we will only need the normal of the closest thing. I will go with the simple solution and compute a bundle of stuff I will store in some structure. I know we'll want motion blur at some point, so I'll add a time input variable. Here's the abstract class:

```
#ifndef HITABLEH
#define HITABLEH

#include "ray.h"

struct hit_record {
    float t;
    vec3 p;
    vec3 normal;
};

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

#endif
```

And here's the sphere (note that I eliminated a bunch of redundant 2's that cancel each other out):

```
#ifndef SPHEREH
#define SPHEREH

#include "hitable.h"

class sphere: public hitable  {
    public:
        sphere() {}
        sphere(vec3 cen, float r) : center(cen), radius(r)  {};
        virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
        vec3 center;
        float radius;
};

bool sphere::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = dot(oc, r.direction());
    float c = dot(oc, oc) - radius*radius;
    float discriminant = b*b - a*c;
    if (discriminant > 0) {
        float temp = (-b - sqrt(b*b-a*c))/a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
        temp = (-b + sqrt(b*b-a*c))/a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
    }
    return false;
}


#endif
```

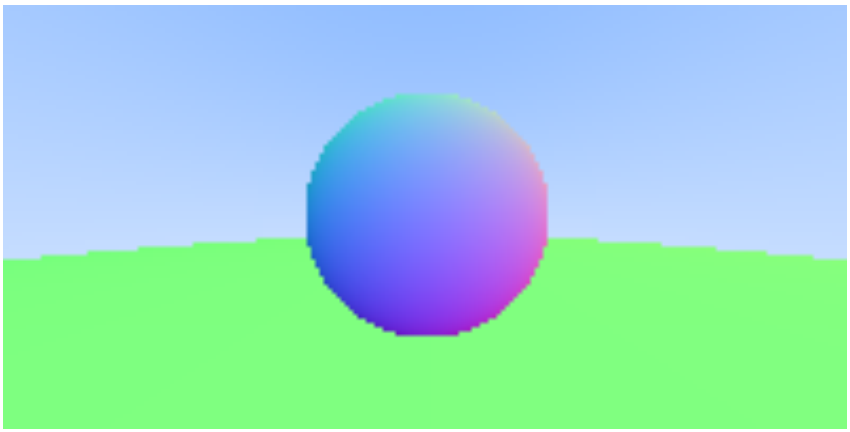
And a list of objects:

```
#ifndef HITABLELISTH
#define HITABLELISTH

#include "hitable.h"

class hitable_list: public hitable  {
    public:
        hitable_list() {}
        hitable_list(hitable **l, int n) {list = l; list_size = n; }
        virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
        hitable **list;
        int list_size;
};

bool hitable_list::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
        hit_record temp_rec;
        bool hit_anything = false;
        double closest_so_far = t_max;
        for (int i = 0; i < list_size; i++) {
            if (list[i]->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
```

And the new main:

```
#include <iostream>
#include "sphere.h"
#include "hitable_list.h"
#include "float.h"

vec3 color(const ray& r, hitable *world) {
    hit_record rec;
    if (world->hit(r, 0.0, MAXFLOAT, rec)) {
        return 0.5*vec3(rec.normal.x()+1, rec.normal.y()+1, rec.normal.z()+1);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5*(unit_direction.y() + 1.0);
        return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
    }
}

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    vec3 lower_left_corner(-2.0, -1.0, -1.0);
    vec3 horizontal(4.0, 0.0, 0.0);
    vec3 vertical(0.0, 2.0, 0.0);
    vec3 origin(0.0, 0.0, 0.0);
    hitable *list[2];
    list[0] = new sphere(vec3(0,0,-1), 0.5);
    list[1] = new sphere(vec3(0,-100.5,-1), 100);
    hitable *world = new hitable_list(list,2);
    for (int j = ny-1; j >= 0; j--) {
       for (int i = 0; i < nx; i++) {
           float u = float(i) / float(nx);
           float v = float(j) / float(ny);
           ray r(origin, lower_left_corner + u*horizontal + v*vertical);

           vec3 p = r.point_at_parameter(2.0);
           vec3 col = color(r, world);
           int ir = int(255.99*col[0]);
           int ig = int(255.99*col[1]);
           int ib = int(255.99*col[2]);

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

This yields a picture that is really just a visualization of where the spheres are along with their surface normal. This is often a great way to look at your model for flaws and characteristics.



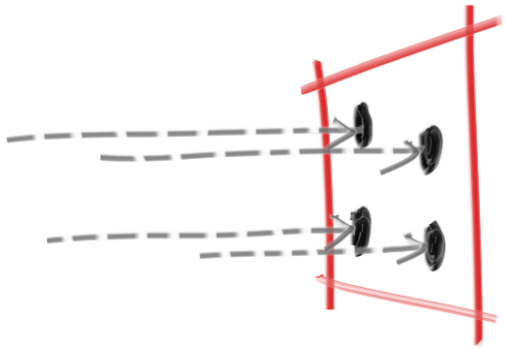
5. Antialiasing

When a real camera takes a picture, there are usually no jaggies along edges because the edge pixels are a blend of some foreground and some background. We can get the same effect by averaging a bunch of samples inside each pixel. We will not bother with stratification, which is controversial but is usual for my programs. For some ray tracers it is critical, but the kind of general one we are writing doesn't benefit very much from it and it makes the code uglier. We abstract the camera class a bit so we can make a cooler camera later.

One thing we need is a random number generator that returns real random numbers. C++ did not traditionally have a standard random number generator but most systems have drand48() tucked away someplace and that is what I use here. However, newer versions of C++ have addressed this issue with the <random> header (if imperfectly according to some experts). Whatever your

infrastructure, find a function that returns a canonical random number which by convention returns random real in the range 0 <= ran < 1. The “less than” before the 1 is important as we will sometimes take advantage of that.

For a given pixel we have several samples within that pixel and send rays through each of the samples. The colors of these rays are then averaged:



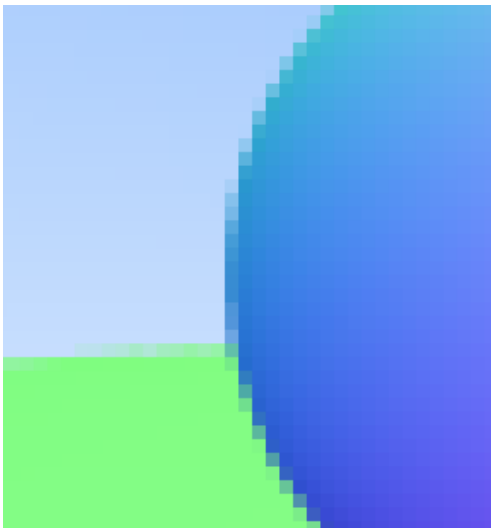
Putting that all together yields a camera class encapsulating our simple axis-aligned camera from before:

```
#ifndef CAMERAH
#define CAMERAH

#include "ray.h"

class camera {
    public:
        camera() {
            lower_left_corner = vec3(-2.0, -1.0, -1.0);
            horizontal = vec3(4.0, 0.0, 0.0);
            vertical = vec3(0.0, 2.0, 0.0);
            origin = vec3(0.0, 0.0, 0.0);
        }
        ray get_ray(float u, float v) { return ray(origin, lower_left_corner + u*horizontal + v*vertical - origin); }

        vec3 origin;
        vec3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};

#endif
```

Main is also changed:

```
int main() {
    int nx = 200;
    int ny = 100;
    int ns = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    hitable *list[2];
    list[0] = new sphere(vec3(0,0,-1), 0.5);
    list[1] = new sphere(vec3(0,-100.5,-1), 100);
    hitable *world = new hitable_list(list,2);
    camera cam;
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(0, 0, 0);
            for (int s=0; s < ns; s++) {
                float u = float(i + drand48()) / float(nx);
                float v = float(j + drand48()) / float(ny);
                ray r = cam.get_ray(u, v);
                vec3 p = r.point_at_parameter(2.0);
                col += color(r, world);
            }
            col /= float(ns);
            int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);
            std::cout << ir << " " << ig << " " << ib << "\n";
        }
    }
}
```

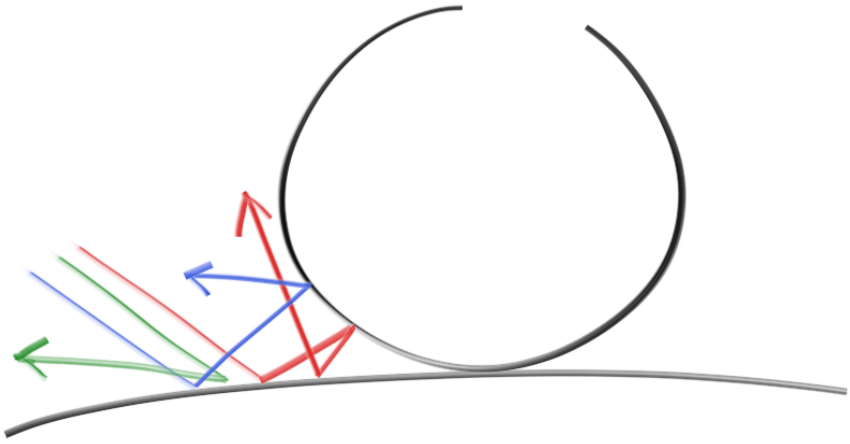
Zooming into the image that is produced, the big change is in edge pixels that are part background and part foreground:



6. Diffuse Materials

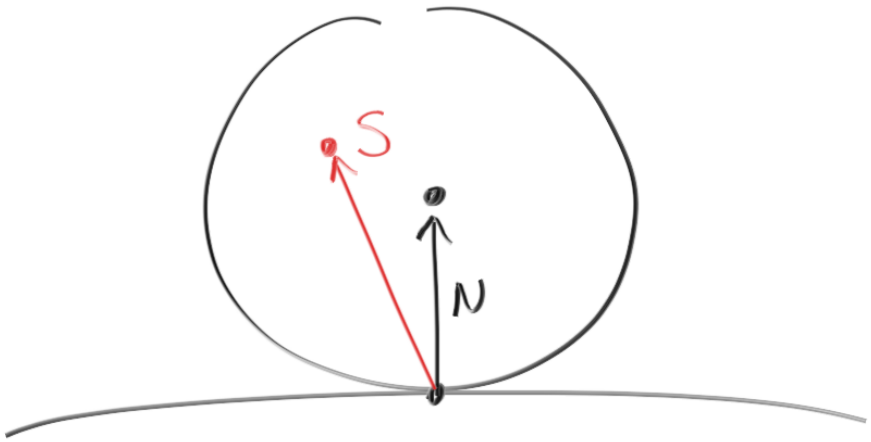
Now that we have objects and multiple rays per pixel, we can make some realistic looking materials. We'll start with diffuse (matte) materials. One question is whether we can mix and match shapes and materials (so we assign a sphere a material) or if it's put together so the geometry and material are tightly bound (that could be useful for procedural objects where the geometry and material are linked). We'll go with separate-- which is usual in most renderers-- but do be aware of the limitation.

Diffuse objects that don't emit light merely take on the color of their surroundings, but they modulate that with their own intrinsic color. Light that reflects off a diffuse surface has its direction randomized. So, if we send three rays into a crack between two diffuse surfaces they will each have different random behavior:



They also might be absorbed rather than reflected. The darker the surface, the more likely absorption is. (That's why it is dark!) Really any algorithm that randomizes direction will produce surfaces that look matte. One of the simplest ways to do this turns out to be exactly correct for ideal diffuse surfaces. (I used to do it as a lazy hack that approximates mathematically ideal Lambertian.)

Pick a random point **s** from the unit radius sphere that is tangent to the hitpoint, and send a ray from the hitpoint **p** to the random point **s**. That sphere has center (**p**+**N**):

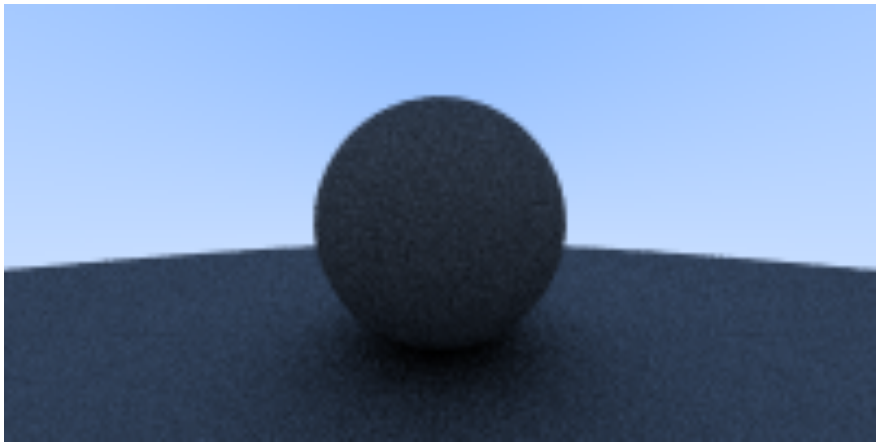


We also need a way to pick a random point in a unit radius sphere centered at the origin. We'll use what is usually the easiest algorithm: a rejection method. First, we pick a random point in the unit cube where x, y, and z all range from -1 to +1. We reject this point and try again if the point is outside the sphere. A do/while construct is perfect for that:

```
vec3 random_in_unit_sphere() {
    vec3 p;
    do {
        p = 2.0*vec3(drand48(),drand48(),drand48()) - vec3(1,1,1);
    } while (p.squared_length() >= 1.0);
    return p;
}
```

```
vec3 color(const ray& r, hitable *world) {
    hit_record rec;
    if (world->hit(r, 0.0, MAXFLOAT, rec)) {
        vec3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5*color( ray(rec.p, target-rec.p), world);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5*(unit_direction.y() + 1.0);
        return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
    }
}
```

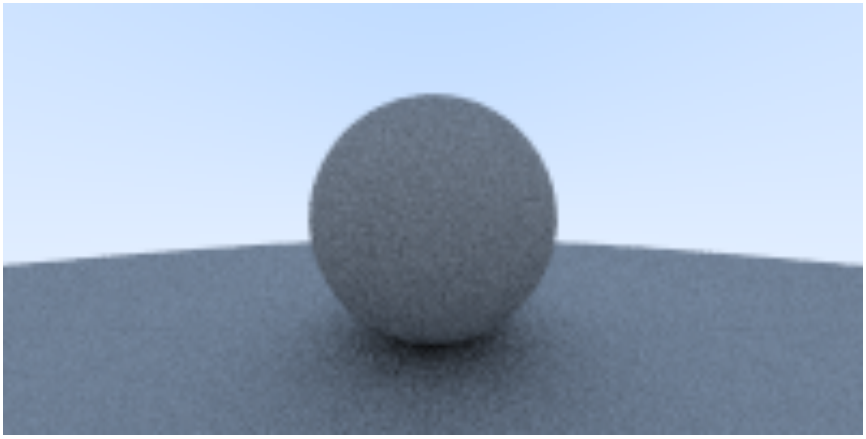
This gives us:



Note the shadowing under the sphere. This picture is very dark, but our spheres only absorb half the energy on each bounce, so they are 50% reflectors. If you can't see the shadow, don't worry, we will fix that now. These spheres should look pretty light (in real life, a light grey). The reason for this is that almost all image viewers assume that the image is "gamma corrected", meaning the 0 to 1 values have some transform before being stored as a byte. There are many good reasons for that, but for our purposes we just need to be aware of it. To a first approximation, we can use "gamma 2" which means raising the color to the power 1/gamma, or in our simple case ½, which is just square-root:

```
col /= float(ns);
col = vec3( sqrt(col[0]), sqrt(col[1]), sqrt(col[2]) );
int ir = int(255.99*col[0]);
int ig = int(255.99*col[1]);
int ib = int(255.99*col[2]);
std::cout << ir << " " << ig << " " << ib << "\n";
```

That yields light grey, as we desire:



There's also a subtle bug in there. Some of the reflected rays hit the object they are reflecting off of not at exactly t=0, but instead at t=-0.0000001 or t=0.00000001 or whatever floating point approximation the sphere intersector gives us. So we need to ignore hits very near zero:

```
if (world->hit(r, 0.001, MAXFLOAT, rec)) {
```

This gets rid of the shadow acne problem. Yes it is really called that.

7. Metal

If we want different objects to have different materials, we have a design decision. We could have a universal material with lots of parameters and different material types just zero out some of those parameters. This is not a bad approach. Or we could have an abstract material class that encapsulates behavior. I am a fan of the

latter approach. For our program the material needs to do two things:

1. produce a scattered ray (or say it absorbed the incident ray)
2. if scattered, say how much the ray should be attenuated

This suggests the abstract class:

```
class material  {
    public:
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const = 0;
};
```

The hit_record is to avoid a bunch of arguments so we can stuff whatever info we want in there. You can use arguments instead; it's a matter of taste. Hitables and materials need to know each other so there is some circularity of the references. In C++ you just need to alert the compiler that the pointer is to a class, which the "class material" in the hitable class below does:

```
#ifndef HITABLEH
#define HITABLEH
#include "ray.h"

class material;

struct hit_record
{
    float t;
    vec3 p;
    vec3 normal;
    material *mat_ptr;
};

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

#endif
```

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

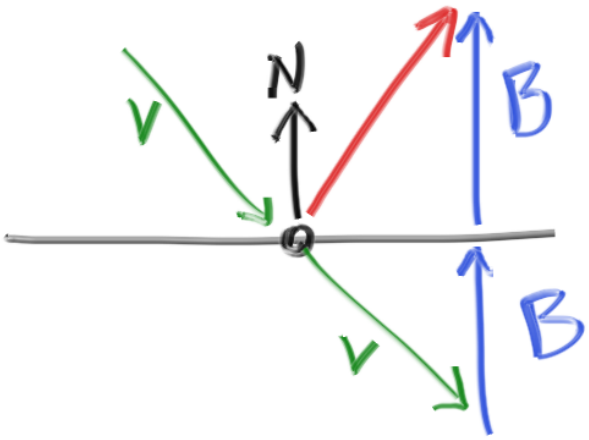
ray hits a surface (a particular sphere for example), the material pointer in the hit_record will be set to point at the material pointer the sphere was given when it was set up in *main()* when we start. When the color() routine gets the hit_record it can call member functions of the material pointer to find out what ray, if any, is scattered.

For the Lambertian (diffuse) case we already have, it can either scatter always and attenuate by its reflectance R, or it can scatter with no attenuation but absorb the fraction 1-R of the rays. Or it could be a mixture of those strategies. For Lambertian materials we get this simple class:

```
class lambertian : public material {
    public:
        lambertian(const vec3& a) : albedo(a) {}
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const  {
             vec3 target = rec.p + rec.normal + random_in_unit_sphere();
             scattered = ray(rec.p, target-rec.p);
             attenuation = albedo;
             return true;
        }

        vec3 albedo;

};
```

Note we could just as well only scatter with some probability p and have attenuation be albedo/p. Your choice.

For smooth metals the ray won't be randomly scattered. The key math is: how does a ray get reflected from a metal mirror? Vector math is our friend here:



The reflected ray direction in red is just (**v** + 2**B**). In our design, **N** is a unit vector, but **v** may not be. The length of B should be dot(**v**,**N**). Because v points in, we will need a minus sign yielding:

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

The metal material just reflects rays using that formula:

```
class metal : public material {
    public:
        metal(const vec3& a) : albedo(a) {}
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const  {
            vec3 reflected = reflect(unit_vector(r_in.direction()), rec.normal);
            scattered = ray(rec.p, reflected);
            attenuation = albedo;
            return (dot(scattered.direction(), rec.normal) > 0);
        }
        vec3 albedo;
};
```

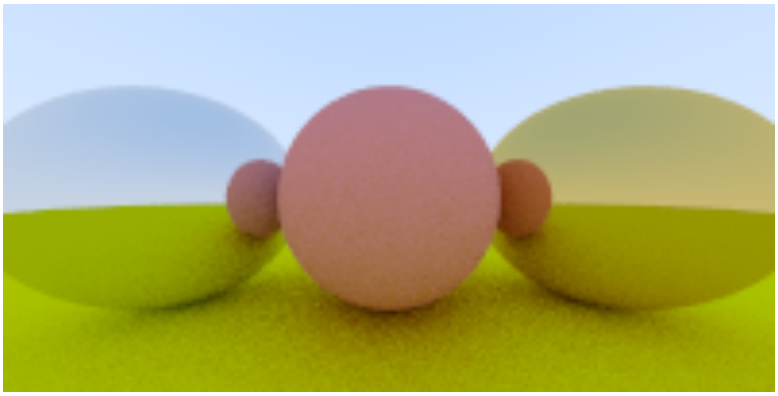
We need to modify the color function to use this:

```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, attenuation, scattered)) {
             return attenuation*color(scattered, world, depth+1);
        }
        else {
            return vec3(0,0,0);
        }
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5*(unit_direction.y() + 1.0);
        return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
    }
}
```

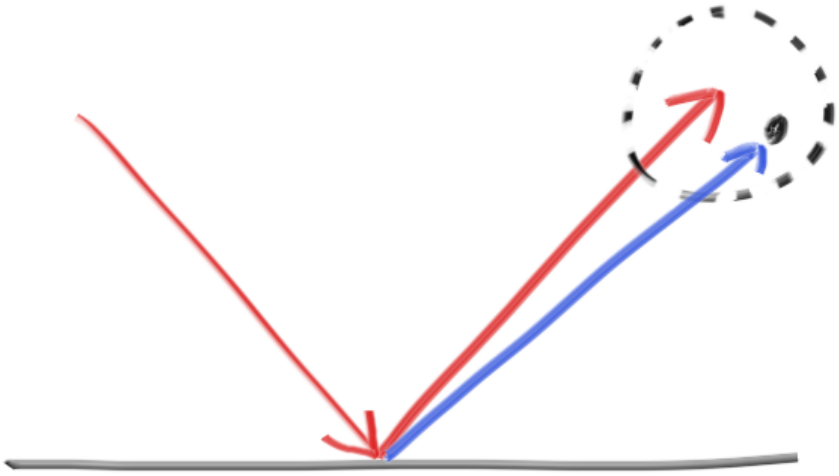
You will also need to modify the sphere class to have a material pointer to it. And add some metal spheres:

```
int main() {
    int nx = 200;
    int ny = 100;
    int ns = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    hitable *list[4];
    list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.8, 0.3, 0.3)));
    list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)));
    list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2)));
    list[3] = new sphere(vec3(-1,0,-1), 0.5, new metal(vec3(0.8, 0.8, 0.8)));
    hitable *world = new hitable_list(list,4);
    camera cam;
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(0, 0, 0);
            for (int s=0; s < ns; s++) {
                float u = float(i + drand48()) / float(nx);
                float v = float(j + drand48()) / float(ny);
                ray r = cam.get_ray(u, v);
                vec3 p = r.point_at_parameter(2.0);
                col += color(r, world,0);
            }
            col /= float(ns);
            col = vec3( sqrt(col[0]), sqrt(col[1]), sqrt(col[2]) );
            int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);
            std::cout << ir << " " << ig << " " << ib << "\n";
        }
    }
}
```

Which gives:



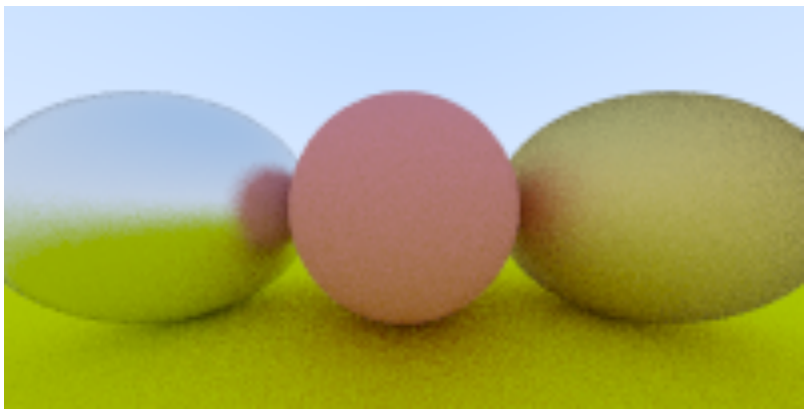
We can also randomize the reflected direction by using a small sphere and choosing a new endpoint for the ray:



The bigger the sphere, the fuzzier the reflections will be. This suggests adding a fuzziness parameter that is just the radius of the sphere (so zero is no perturbation). The catch is that for big spheres or grazing rays, we may scatter below the surface. We can just have the surface absorb those. We'll put a maximum of 1 on the radius of the sphere which yields:

```
class metal : public material {
    public:
        metal(const vec3& a, float f) : albedo(a) { if (f < 1) fuzz = f; else fuzz = 1; }
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const  {
            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);
        }
        vec3 albedo;
        float fuzz;
};
```

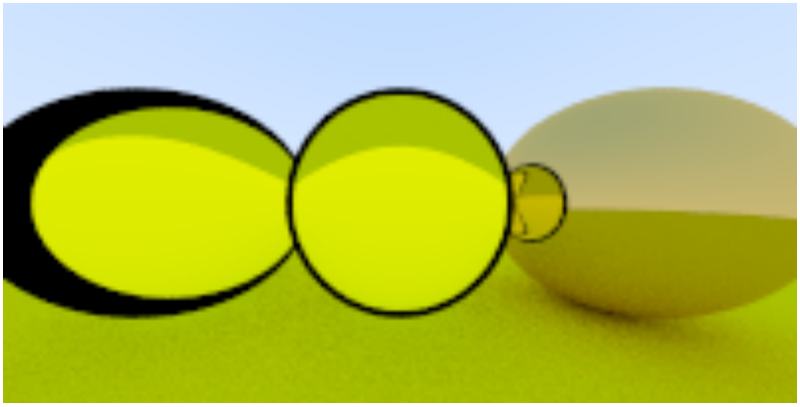
We can try that out by adding fuzziness 0.3 and 1.0 to the metals:



8. Dielectrics

Clear materials such as water, glass, and diamonds are dielectrics. When a light ray hits them, it splits into a reflected ray and a refracted (transmitted) ray. We'll handle that by randomly choosing between reflection or refraction and only generating one scattered ray per interaction.

The hardest part to debug is the refracted ray. I usually first just have all the light refract if there is a refraction ray at all. For this project, I tried to put two glass balls in our scene, and I got this (I have not told you how to do this right or wrong yet, but soon!):

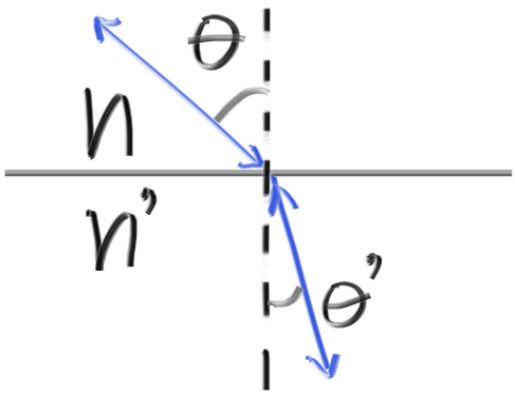


Is that right? Glass balls look odd in real life. But no, it isn't right. The world should be flipped upside down and no weird black stuff. I just printed out the ray straight through the middle of the image and it was clearly wrong. That often does the job.

The refraction is described by Snell's law:

$$n \sin(\theta) = n' \sin(\theta')$$

Where n and n′ are the refractive indices (typically air = 1, glass = 1.3-1.7, diamond = 2.4) and the geometry is:



One troublesome practical issue is that when the ray is in the material with the higher refractive index, there is no real solution to Snell's law and thus there is no refraction possible. Here all the light is reflected, and because in practice that is usually inside solid objects, it is called "total internal reflection". This is why sometimes the water-air boundary acts as a perfect mirror when you are submerged. The code for refraction is thus a bit more complicated than for reflection:

```
bool refract(const vec3& v, const vec3& n, float ni_over_nt, vec3& refracted) {
    vec3 uv = unit_vector(v);
    float dt = dot(uv, n);
    float discriminant = 1.0 - ni_over_nt*ni_over_nt*(1-dt*dt);
    if (discriminant > 0) {
        refracted = ni_over_nt*(uv - n*dt) - n*sqrt(discriminant);
        return true;
    }
    else
        return false;
}
```

And the dielectric material that always refracts when possible is:

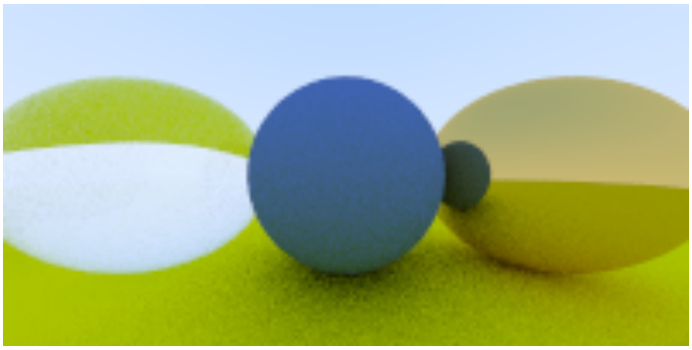
```
class dielectric : public material {
    public:
        dielectric(float ri) : ref_idx(ri) {}
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const  {
             vec3 outward_normal;
             vec3 reflected = reflect(r_in.direction(), rec.normal);
             float ni_over_nt;
             attenuation = vec3(1.0, 1.0, 0.0);
             vec3 refracted;
             if (dot(r_in.direction(), rec.normal) > 0) {
                  outward_normal = -rec.normal;
                  ni_over_nt = ref_idx;
             }
             else {
                  outward_normal = rec.normal;
                  ni_over_nt = 1.0 / ref_idx;
             }
             if (refract(r_in.direction(), outward_normal, ni_over_nt, refracted)) {
                scattered = ray(rec.p, refracted);
             }
             else {
                scattered = ray(rec.p, reflected);
                return false;
             }
             return true;
        }

        float ref_idx;
};
```

Attenuation is always 1-- the glass surface absorbs nothing. The *attenuation = vec3(1.0, 1.0, 0.0)* above will also kill the blue channel which is the type of color bug that can be hard to find-- it will give a color shift only. Try it the way it is and then change it to *attenuation = vec3(1.0, 1.0, 1.0)* to see the difference.

If we try that out with these parameters:

```
list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.1, 0.2, 0.5)));
list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)));
list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2)));
list[3] = new sphere(vec3(-1,0,-1), 0.5, new dielectric(1.5));
```

We get:



(The reader Becker has pointed out that when there is a reflection ray the function returns false so there are no reflections. He is right and that is why there are none in the image above. I am leaving this

in rather than correcting this because it is a very interesting example of a major bug that still leaves a reasonably plausible image. These sleeper bugs are the hardest bugs to find because we humans are not designed to find fault with what we see.)

Now real glass has reflectivity that varies with angle-- look at a window at a steep angle and it becomes a mirror. There is a big ugly equation for that, but almost everybody uses a simple and surprisingly simple polynomial approximation by Christophe Schlick:

```
float schlick(float cosine, float ref_idx) {
    float r0 = (1-ref_idx) / (1+ref_idx);
    r0 = r0*r0;
    return r0 + (1-r0)*pow((1 - cosine),5);
}
```

This yields our full glass material:

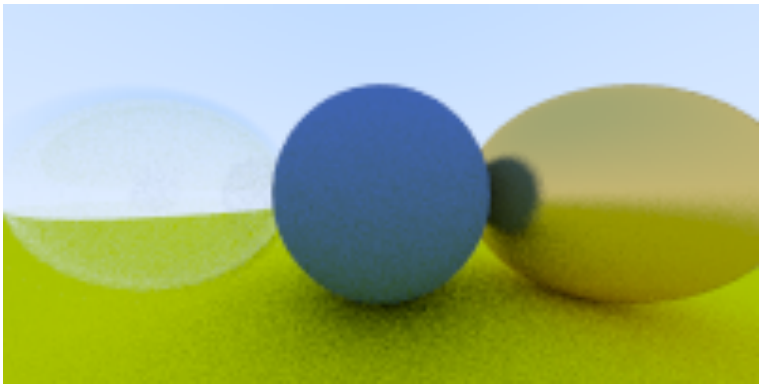
```
class dielectric : public material {
    public:
        dielectric(float ri) : ref_idx(ri) {}
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const  {
             vec3 outward_normal;
             vec3 reflected = reflect(r_in.direction(), rec.normal);
             float ni_over_nt;
             attenuation = vec3(1.0, 1.0, 1.0);
             vec3 refracted;
             float reflect_prob;
             float cosine;
             if (dot(r_in.direction(), rec.normal) > 0) {
                  outward_normal = -rec.normal;
                  ni_over_nt = ref_idx;
                  cosine = ref_idx * dot(r_in.direction(), rec.normal) / r_in.direction().length();
             }
             else {
                  outward_normal = rec.normal;
                  ni_over_nt = 1.0 / ref_idx;
                  cosine = -dot(r_in.direction(), rec.normal) / r_in.direction().length();
             }
             if (refract(r_in.direction(), outward_normal, ni_over_nt, refracted)) {
                reflect_prob = schlick(cosine, ref_idx);
             }
             else {
                scattered = ray(rec.p, reflected);
                reflect_prob = 1.0;
             }
             if (drand48() < reflect_prob) {
                scattered = ray(rec.p, reflected);
             }
             else {
                scattered = ray(rec.p, refracted);
             }
             return true;
        }

        float ref_idx;
};
```

(note: the first appearance of "scattered =" above is not needed and in fast is always over-written later. Doesn't affect result but is a performance bug).

An interesting and easy trick with dielectric spheres is to note that if you use a negative radius, the geometry is unaffected but the surface normal points inward, so it can be used as a bubble to make a hollow glass sphere:

```
list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.1, 0.2, 0.5)));
list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)));
list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2)));
list[3] = new sphere(vec3(-1,0,-1), 0.5, new dielectric(1.5));
list[4] = new sphere(vec3(-1,0,-1), -0.45, new dielectric(1.5));
```

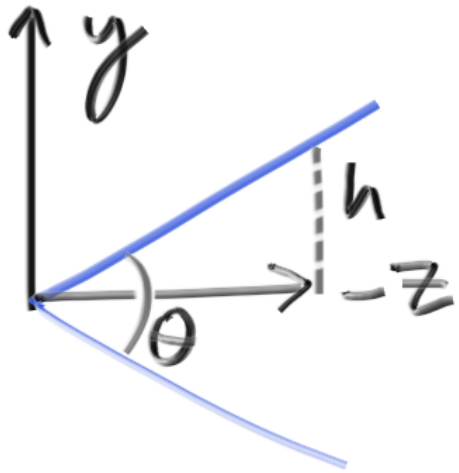
This gives:



9. Positionable camera

Cameras, like dielectrics, are a pain to debug. So I always develop mine incrementally. First, let's allow an adjustable field of view (fov). This is the angle you see through the portal. Since our image is not square, the fov is different horizontally and vertically. I always use vertical fov. I also usually specify it in degrees and change to radians inside a constructor-- a matter of personal taste.

I first keep the rays coming from the origin and heading to the z=-1 plane. We could make it the z=-2 plane, or whatever, as long as we made h a ratio to that distance. Here is our setup:



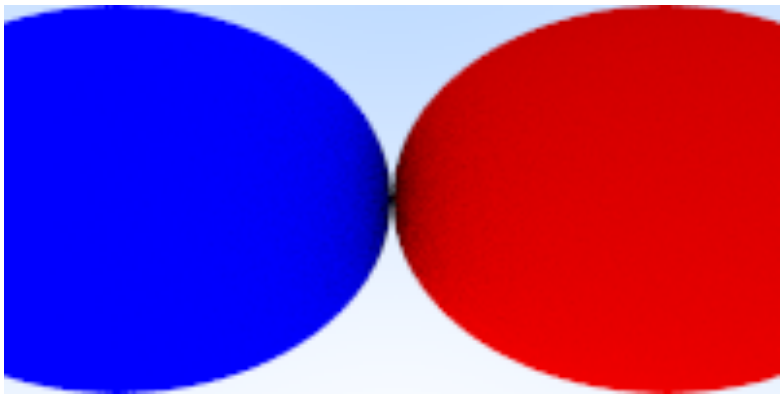
This implies h = tan(theta/2). Our camera now becomes:

```
#ifndef CAMERAH
#define CAMERAH

#include "ray.h"

class camera {
    public:
        camera(float vfov, float aspect) { // vfov is top to bottom in degrees
            float theta = vfov*M_PI/180;
            float half_height = tan(theta/2);
            float half_width = aspect * half_height;
            lower_left_corner = vec3(-half_width, -half_height, -1.0);
            horizontal = vec3(2*half_width, 0.0, 0.0);
            vertical = vec3(0.0, 2*half_height, 0.0);
            origin = vec3(0.0, 0.0, 0.0);
        }
        ray get_ray(float u, float v) { return ray(origin, lower_left_corner + u*horizontal + v*vertical - origin); }

        vec3 origin;
        vec3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};

#endif
```

With calling it camera cam(90, float(nx)/float(ny)) and these spheres:

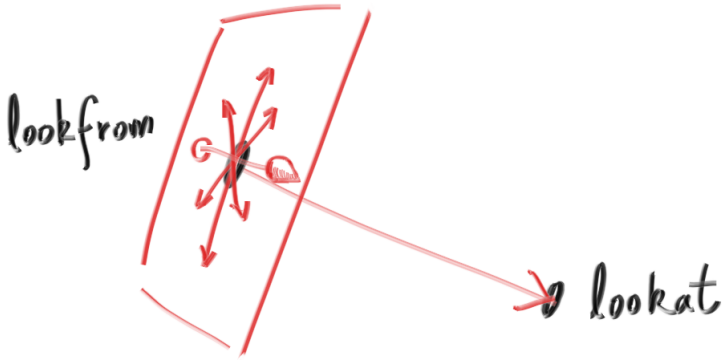
```
float R = cos(M_PI/4);
list[0] = new sphere(vec3(-R,0,-1), R, new lambertian(vec3(0, 0, 1)));
list[1] = new sphere(vec3( R,0,-1), R, new lambertian(vec3(1, 0, 0)));
hitable *world = new hitable_list(list,2);
```

gives:



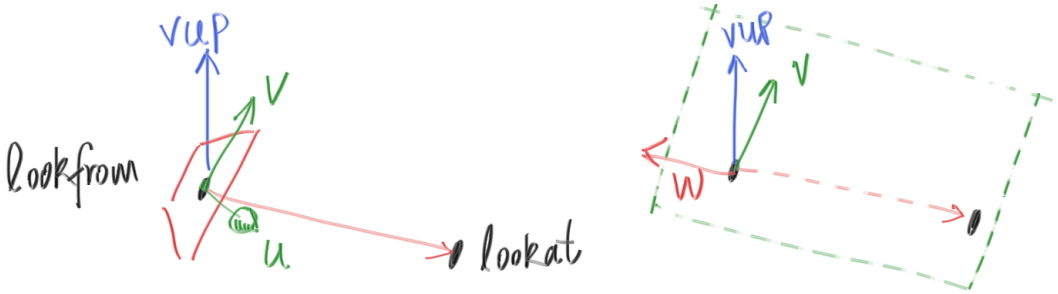
To get an arbitrary viewpoint, let's first name the points we care about. We'll call the position where we place the camera *lookfrom*, and the point we look at *lookat*. (Later, if you want, you could define a direction to look in instead of a point to look at.)

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 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. Notice we already we already have a plane that the up vector should be in, the plane orthogonal to the view direction.



We can actually use any up vector we want, and simply project it onto this plane to get an up vector for the camera. I use the common

convention of naming a "view up" (vup) vector. A couple of cross products, and we now have a complete orthonormal basis (u,v,w) to describe our camera's orientation.



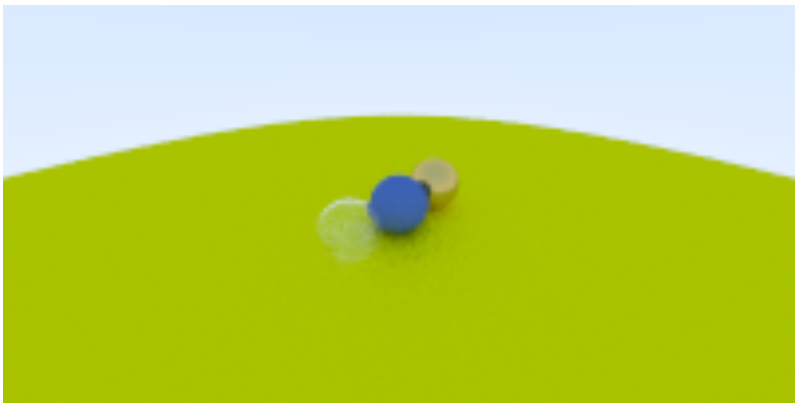
Remember that vup, v, and w are all in the same plane. Note that, like before when our fixed camera faced -Z, our arbitrary view camera faces -w. And keep in mind that we can-- but we don't have to-- use world up (0,1,0) to specify vup. This is convenient and will naturally keep your camera horizontally level until you decide to experiment with crazy camera angles.

```
#ifndef CAMERAH
#define CAMERAH

#include "ray.h"

class camera {
    public:
        camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect) { // vfov is top to bottom in degrees
            vec3 u, v, w;
            float theta = vfov*M_PI/180;
            float half_height = tan(theta/2);
            float half_width = aspect * half_height;
            origin = lookfrom;
            w = unit_vector(lookfrom - lookat);
            u = unit_vector(cross(vup, w));
            v = cross(w, u);
            lower_left_corner = vec3(-half_width, -half_height, -1.0);
            lower_left_corner = origin  - half_width*u -half_height*v - w;
            horizontal = 2*half_width*u;
            vertical = 2*half_height*v;
        }
        ray get_ray(float s, float t) { return ray(origin, lower_left_corner + s*horizontal + t*vertical - origin); }

        vec3 origin;
        vec3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};
#endif
```

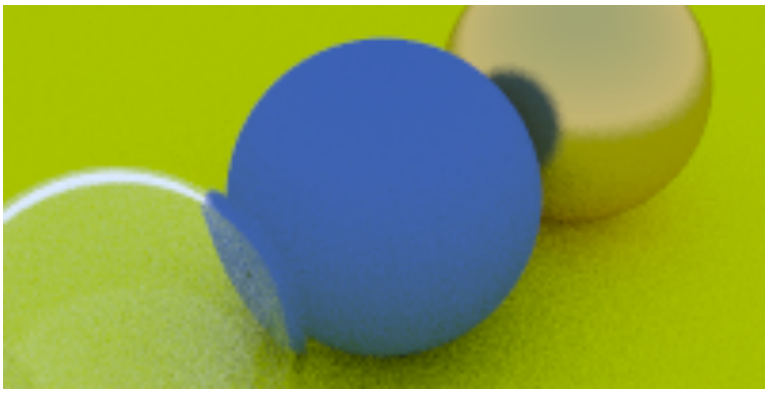
This allows us to change the viewpoint:

```
camera cam(vec3(-2,2,1), vec3(0,0,-1), vec3(0,1,0), 90, float(nx)/float(ny));
```

to get:



And we can change field of view to get:



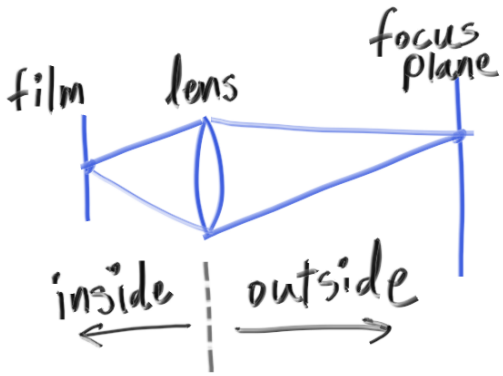
10. Defocus Blur

Now our final feature: *defocus blur*. Note, all photographers will call it "depth of field" so be aware of only using "defocus blur" among friends.

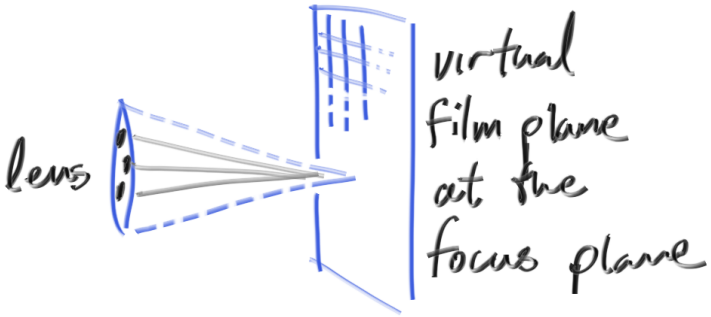
The reason we defocus blur in real cameras is because they need a big hole (rather than just a pinhole) to gather light. This would defocus everything, but if we stick a lens in the hole, there will be a certain distance where everything is in focus. The distance to that plane where things are in focus is controlled by the distance between the lens and the film/sensor. That is why you see the lens move

relative to the camera when you change what is in focus (that may happen in your phone camera too, but the sensor moves). The "aperture" is a hole to control how big the lens is effectively. For a real camera, if you need more light you make the aperture bigger, and will get more defocus blur. For our virtual camera, we can have a perfect sensor and never need more light, so we only have an aperture when we want defocus blur.

A real camera has a compound lens that is complicated. For our code we could simulate the order: sensor, then lens, then aperture, and figure out where to send the rays and flip the image once computed (the image is projected upside down on the film). Graphics people usually use a thin lens approximation.



We also don't need to simulate any of the inside of the camera. For the purposes of rendering an image outside the camera, that would be unnecessary complexity. Instead I usually start rays from the surface of the lens, and send them toward a virtual film plane, by finding the projection of the film on the plane that is in focus (at the distance focus_dist).



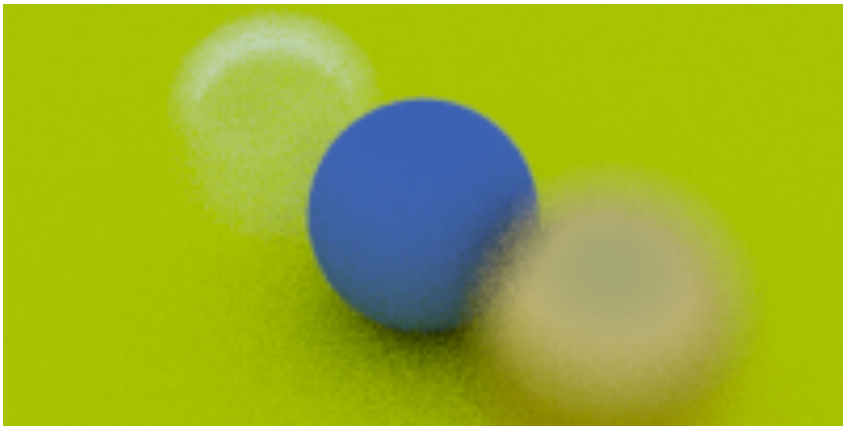
For that we just need to have the ray origins be on a disk around lookfrom rather than from a point:

```
#ifndef CAMERAH
#define CAMERAH
#include "ray.h"

vec3 random_in_unit_disk() {
    vec3 p;
    do {
        p = 2.0*vec3(drand48(),drand48(),0) - vec3(1,1,0);
    } while (dot(p,p) >= 1.0);
    return p;
}

class camera {
    public:
        camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect, float aperture, float focus_dist) {
            lens_radius = aperture / 2;
            float theta = vfov*M_PI/180;
            float half_height = tan(theta/2);
            float half_width = aspect * half_height;
            origin = lookfrom;
            w = unit_vector(lookfrom - lookat);
            u = unit_vector(cross(vup, w));
            v = cross(w, u);
            lower_left_corner = origin  - half_width*focus_dist*u -half_height*focus_dist*v - focus_dist*w;
            horizontal = 2*half_width*focus_dist*u;
            vertical = 2*half_height*focus_dist*v;
        }
        ray get_ray(float s, float t) {
            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);
        }

        vec3 origin;
        vec3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
        vec3 u, v, w;
        float lens_radius;
};
#endif
```

Using a big aperture:

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

camera cam(lookfrom, lookat, vec3(0,1,0), 20, float(nx)/float(ny), aperture, dist_to_focus);
```

We get:



Calculus is a core tool in computer graphics across not just physically based rendering but also other areas such as animation and geometric processing. You've already seen many applications of integrals, integral equations, and derivatives in physically-based rendering in this book. However, we've avoided evaluating any of the complicated integrals explicitly and never actually solved any integral equations at all.

Contents

We avoided solving the integral equations because they aren't just hard, they are *impossible* to solve analytically. This chapter makes a brief digression from rendering to introduce tools for **numerical** differentiation and integration. These methods have vast applications throughout simulation and high performance computing. We will of course apply them to rendering immediately in the Path Tracing chapter.

Numerical integration is slightly more challenging than numerical differentiation, mostly because we must consider some more adversarial cases and introduce randomized algorithms to avoid them. Consider that differentiation involves an infinitesimal interval and integration involves an infinite number of intervals. That approximating an infinite sum is more complicated than differentiation should not be a surprise. However, the simplicity of numerical integration

compared to manual integration should be a pleasant surprise.

1. Definition of a Derivative

The **derivative** of a function is another function. We compute it with respect to one of the parameters. For example, the derivative of $g(x)$ with respect to x is written $dg(x)/dx$. Evaluating the derivative function at a location gives the rate of change of the original function with respect to that variable, which is the slope in a graph. For example, $\frac{dg}{dx}(3)$ is how fast $g(x)$ is changing relative to x at $x = 3$.

The derivative operator applied to a function of one real variable is formally defined as: the limit of the change in the function's value over a small interval divided by the length of the interval, as that interval's length approaches zero:

$$\frac{d\,g(x)}{dx} = \lim_{\Delta x \to 0} \frac{g(x + \Delta x) - g(x - \Delta x)}{2\Delta x} \tag{1}$$

Here, I intentionally avoid naming the function $f(x)$, which is what you might expect in a math curriculum. That's because f is reserved to denote is the scattering function in rendering, and we'll soon apply these ideas to the rendering equation.

If the limit does not exist for a specific x, then then the derivative may not exist at that point. This happens with discontinuous functions, such as $\tan x$ at $x = \pi/2$, where its values slightly before and after that location diverge to $\pm\infty$. Continuous functions can also have locations at which the derivative does not exist because the derivative is discontinuous and the limit does not converge. Under the two-sided definition of the derivative used in this chapter, the derivative of the function:

$$g(x) = 0 \quad \text{for } x < 0 \quad \text{and } g(x) = x \quad \text{otherwise}$$

has no derivative at $x = 0$ even though g is continuous.

I make note of cases where the derivative does not exist as a caution. Often in graphics, we work with shading functions that have either discontinuities or discontinuous derivatives. These arise from the inclusion of step functions *or branches* (for example, `if` statements) in the implementation. Keep in the back of your mind that nonexistent derivative locations can cause problems. For example, some techniques that will be introduced later such as automatic MIP-mapping and geometric level of detail that attempt to combat aliasing effects such as jagged lines and flickering pixels by using derivatives, and when the derivatives do not exist they may fail.

In general mathematics terminology, a **differentiable** function is one where the derivative exists across the *entire* domain of the function. In the field of computer graphics, we often use a less constrained definition and treat functions such as $\tan(x)$ as "differentiable" except at their discontinuities. For $tan(x)$, those discontinuities are found at $x = (k + 1/2)\pi$ for integer k. There are also functions where the derivative does not exist at *any* point in the domain even if they are continuous, such as the Weierstrauss function and other fractals.

One way to a derivative is **analytically**, or "symbolically", by applying theorems that manipulate the algebraic symbols of a function's definition according to certain patterns. Performing differentiation in that way gives an exact result, and is likely the way that you were first exposed to differentiation in a calculus course.

For example, the **power rule** of differentiation states that $\frac{d\,ax^b}{dx} = a \cdot b \cdot x^{b-1}$ when a and b are constants. You might be familiar with other such rules, such as that the derivative of the sum of terms is the sum of their derivatives: $\frac{d\,f(x)+g(x)}{dx} = \frac{d\,f(x)}{dx} + \frac{d\,g(x)}{dx}$.

2. Numerical Differentiation

Moany applications perform analytic differentiation, including including Mathematica, Maple, and Matlab. These are indispensible tools for scientists when facing complex derivatives. They are also a pleasant discovery for students who didn't enjoy the process of manually computing derivatives. However, analytic differentiation is complex to implement and not always applicable or efficient.

Fortunately, it is very simple to implement **numerical differentiation** techniques. These approximate the value of a derivative at a location instead of computing the entire derivative function explicitly. Implementing numerical differentiation can be an entertaining programming exercise and suprisingly easy to do for all functions compared to computing a single analytic derivative by hand.

Numerical differentiation estimates the derivative by replacing the limit as $\Delta x \to 0$ with an approximate, small-but-nonzero Δx constant. The following listing introduces a function `derivative` that takes a function `g` as an argument and returns a new function that is the derivative of `g`. That returned derivative function applies numerical differentiation to `g` at its argument each time it is called.

Python C++

```
    # returns a new function, dg/dx
1   def derivative(g):
2       delta = 1e-10
3       return lambda x : (g(x + delta) - g(x -
    delta)) / (2 * delta)

    # x^3
4   def g(x): return x**3

5   # 3x^2
6   dg = derivative(g)

7   # 3*(2^2) = 12
8   print(dg(2))
```

When run on the example of $g(x) = x^3$ and evaluated at $x = 12$, the Python version of this program prints `12.000000992884452`. The correct answer is exactly 12. The slight error is due to the nonzero `delta` value. Making `delta` smaller can increase the accuracy of the approximation, but at some point the denominator `2 * delta` exceeds the precision available in floating point for the division operation, and error will increase again because the result is not representable with a limited number of bits.

I started with an example on a function that we can easily differentiate by hand so that we could verify the result. The utility of numerical differentiation becomes more obvious when applied to a more complicated function. Let's extending the previous example with:

Python C++

```
import math
def h(x):
    if x > 10:
        return 10 / x
    elif (math.tan(x) > 1):
        return math.cos(2 * math.pi * x) * x**5 / (2 * x**4 + abs(x))
    else:
        return math.sqrt(abs(x)) + x**2 + 10 * abs(x)**1.5 + math.floor(2.5 * x)

print(derivative(h)(-4.5))
```

This example is not one I'd like to differentiate by hand. Yet, the program instantly produces $\frac{dh}{dx}(-4.5) \approx -41.05551454358647$. You're welcome to verify the correctness of that manually.

The example with the branching `h` function is actually relatively simple compared to some of the shading and scattering functions that we need to differentiate in rendering. So, despite the precision limitations in certain cases, numerical differentiation is extremely useful for graphics.

3. Definition of an Integral

There are multiple definitions of the integral. Let's use the Reimann definition from an introductory calculus course for now. It is familiar and leads to a corresponding numerical method. This should be relatively intuitive. When applying numerical integration in other chapters, we can then revisit whether that was the best choice.

The **indefinite integral** of a function of one real variable, $G(x) = \int g(x)\, dx$, is another function. When $G(x)$ is evaluated at a location, its value is the (signed) measure of the area bounded by $g(x)$ and x-axis, and between negative infinity up to that location. This is colloquially referred to as "the area under the curve".

In many applications including rendering, we are concerned with the related **definite integral** $\int_a^b g(x)\, dx = \int g(x)\, dx]_a^b$, which is the difference of the indefinite integral evaluated at each end of the interval $[a, b]$. This is the signed area under $g(x)$ on from $x = a$ to $x = b$. Simplifying some of the details, the Reimann definition of integration is the following limit:

$$\int_a^b g(x)\, dx = \lim_{N \to \infty} \sum_i^N g(x_i) \Delta x_i \qquad (2)$$

where a and b are constants, each $a \leq x_i \leq b$, and $\sum \Delta x_i = b - a$.

We've already seen many cases where the domain of integration is a 2D region, using notation such as $\int_A g(X)\, dX$ for the 2D integration case. This is shorthand for multiple 1D real integrals. For example, if A is a rectangular region of the 2D plane,

$$\int_A g(X)\, dX = \int_{x_0}^{x_1} \int_{y_0}^{y_1} g(x, y)\, dy\, dx. \qquad (3)$$

In spherical coordinates, the integral over a solid angle spherical rectangular patch domain Γ can be expressed in expanded form:

$$\int_{\Gamma} g(\hat{\omega})\, d\hat{\omega} = \int_{\psi_0}^{\psi_1} \int_{\theta_0}^{\theta_1} g(\theta, \psi)\ \sin\theta\, d\theta\, d\psi. \qquad (4)$$

We rarely require such explicit expression of these expansions. I mention them here so that you'll know they are just notation and not new concepts on their own. Single-variable integration sufficies for rendering applications, and multidimensional expressions are a convenient shorthand when we don't need to see the details.

The integral equal to the **antiderivative**. The first fundamental theorem of calculus states that the derivative and indefinite integral are inverse operations. This is either profound or obvious, depending on your expertise and context. In many cases, including rendering glass surfaces with the path tracing algorithm, we solve problems by taking the integral of the derivative of the function in which we are interested but cannot directly access. The first fundamental theorem tells is that this is equivalent to evaluating that implicit function.

4. Analytic Integration

Consider the operation of evaluating a specific definite integral. This operation maps a function and a region in its domain to a value in the product of the domain and range of the function. For example, for the function $g(x) = x^2$ and the region $A = [1, 2]$ in the domain of real numbers, the value of the definite integral $\int_A g(x)\, dx$ is:

$$\int_1^2 x^2 dx = \frac{x^3}{3}\Big]_1^2 = \frac{1}{3}[2^3 - 1^3] = \frac{7}{3}. \qquad (5)$$

I evaluated that definite integral by **symbolic** (a.k.a. **analytic**) integration. First, I mapped the integrand $g(x)$ to

its integral function using the power rule. Second, I evaluated that integral function at both ends of the domain and took their difference. This is the method of integration presented in introductory calculus courses. It requires knowing the integrals for specific patterns, such as polynomials (the power rule), trigonometric functions, function composition (the chain rule), and products (integration by parts), and then applying different memorized rules for each pattern.

Unfortunately, many functions that have definite integrals are not analytically integrable. The patterns aren't just unknown, they provably don't exist for many functions. For these, there is no analytic way to construct an exact integral function that can be evaluated at the ends of the integration domain, *even though the integral itself does exist*.

This may be surprising, but is probably not unprecedented in your mathematics experience. For example, the trivial algebraic process for solving linear equations of one variable and the analytic solution to quadratic equations are tools you have used frequently in rendering already to compute ray intersections with planes and spheres. As you may be aware, there are also (albeit less friendly) analytic solutions for the roots of third- and fourth-order polynomials. However, there is provably no analytic solution for the roots of a fifth-order polynomial. These roots obviously exist in many cases, and you can find them by ad-hoc methods, for example the roots of x^5 are all $x = 0$, however there can be no general process for finding the roots of an arbitrary quintic equations.

Up to this point, the discussion of integration sounded analogous to the previous differentiation discussion. We've now hit a key distinction due to integrals that are not just time-consuming to evaluate analytically, but ones that are *impossible* to evaluate analytically. The solution of course will be numerical integration. However, in this case it is not an issue of efficiency or convenience. It is an issue of necessity.

Because it makes analytically-unsolvable problems solvable in practice, numerical integration is one of the great contributions of mechanical computing to science and engineering. *Efficient* numerical integration algorithms are also a great contribution of computer science to general mathematics. Take pride and be inspired that this has been explored extensively, and in some cases pioneered by, computer graphics... and also remains an important and open area of research to which you may contribute.

5. Numerical Integration

A black-box definite integral algorithm should work for both cases that can be solved analytically and those that cannot, provided that the definite integral exists. **Numerical (definite) integration** algorithms can solve both cases, albeit usually only as an approximation.

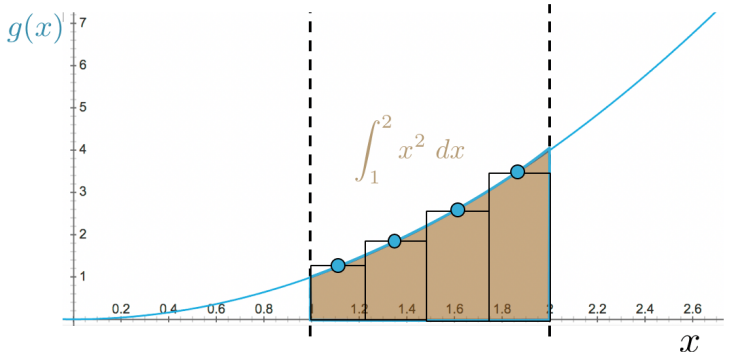
The simplest numerical integration algorithm for functions from reals to reals divides the integration interval into N uniform pieces and evaluates the function at those. This is equivalent to estimating the average value of the function from a series of **samples** and then multiplying that average by the measure of the region.

I break with our naming conventions for variables and use a capital N for the integer number of samples to help distinguish it from the surface normal vector, $\hat{n}$, which will naturally appear soon, when we apply numerical integration to the rendering equation.

The following Python code implements that method using a fixed 100 steps and applies it to our running example of integrating $\int_A g(x)\,dx = \int_1^2 x^2\,dx$:

```
1  def deterministicEstimateIntegral(g, low,
   high):
       # number of sampled subintervals
2      N = 100
3      sum = 0
```

```
        # the measure of each regular subinterval
4       dx = (high - low) / N

        # sample the center (+0.5) of each subinterval
5       sampleLocations = [low + (i + 0.5) * dx
    for i in range(0, N)]

6       for x in sampleLocations:
7           sum += g(x) * dx
8       return sum

    # our example function
9   def g(x): return x * x

10  print(deterministicEstimateIntegral(g, 1, 2))
```



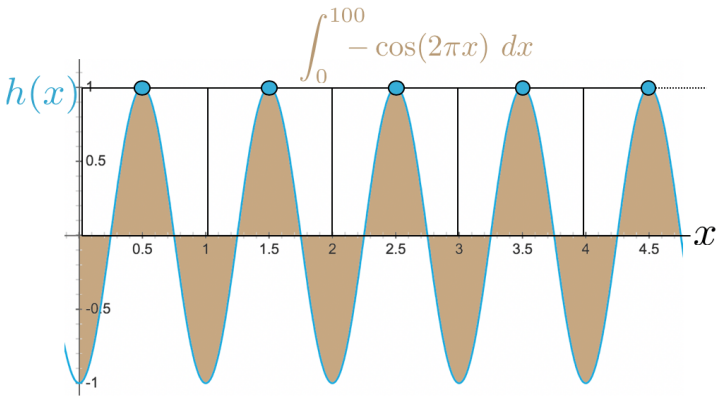
The curve $g(x) = x^2$ and its definite integral from $x = 1$ to $x = 2$, which is the measure of the shaded area under the curve. The black outline boxes are the areas summed by the deterministic estimator to approximate the shaded area at $N = 4$.

When run, this program prints `2.333325`, which is only slightly lower than the exact result, $7/3 = 2.3\overline{3}$. Given a larger number of samples, the program can produce an even more accurate estimate.

This simple, deterministic numerical integrator has a severe limitation. If the integrand takes on values on each subinterval that are very different from the value at the centers of the regular intervals, then the estimate of the definite integral can be arbitrarily wrong because only the centers are sampled.

For example, consider the definite integral $\int_0^{100} -\cos 2\pi x\,dx$. The correct result by symbolic evaluation

is $\frac{-1}{2\pi} \sin 2\pi x]_0^{100} = \frac{1}{2\pi}[0-0] = 0$. By inspection, the cosine wave is symmetric about the x axis and has net integral of zero over an even number of periods.



The definite integral of $-\cos 2\pi x$ is zero over any full period; from $x = 0$ to $x = 100$ there are exactly 100 periods, so its definite integral on that interval is zero. Deterministic, regular sampling at $x = i + 0.5$ always captures the peaks and gives the misleading estimate of 100 for the integral.

The deterministic program produces tremendous error in this case. To see this, append the following lines to the previous program:

```
import math
def h(x): return -math.cos(x * 2 * math.pi)
print(deterministicEstimateIntegral(h, 0,
100))
```

The deterministic estimator's result for the integral of the cosine function is `100.0` instead of zero. That is because the cosine function violates the assumption that the center of each subinterval is representative of the value of the function on that subinterval. In fact, the cosine function is adversarial: it hits a local maximum value exactly at the points that the integrator sampled, distorting the estimate as much as possible.

6. A Tale of Two Cities

Most problems in applied computer science are today solved

by **randomized algorithms**, also known as **nondeterministic** and **stochastic** methods. Most of the world's computing resources are devoted to running randomized algorithms. They are pervasive in fields as diverse as finance, climate modeling, biological simulation, and of course, computer graphics.

Traditional, **deterministic algorithms** guarantee a run time bound to produce the exactly correct answer. Randomized algorithms leverage random values to guarantee either good run time bounds or error bounds, but rarely both at once. Randomized algorithms ideally seek to avoid distinctive and biasing patterns in the results, although not all achieve this.

Las Vegas

The **Las Vegas** class of randomized algorithms are named after the United States city with permissive gambling laws and a large, festive gambling industry. These algorithms guarantee that the result that they estimate will be within a bound of the true answer, but their running times are only low (i.e., fast) with high probability. That is, these guarantee *accuracy*. The median-of-median pivot selection algorithm for quicksort is an example of a well-known Las Vegas algorithm. "Randomized algorithms" computer science theory textbooks and courses often focus on Las Vegas algorithms.

Monte Carlo

The **Monte Carlo** class of randomized algorithms are named for a region of Monaco that contains a famous European casino. These algorithms guarantee that the running time of the algorithm is bounded, but their results are within an error bound only with high probability. That is, these guarantee *performance*. The path tracing algorithm, and most simulation and forecasting algorithms, are in the Monte Carlo class of randomized algorithms. Applied math and computer science "numerical methods" textbooks and

courses often focus on Monte Carlo algorithms.

7. Monte Carlo Integration

There is a standard Monte Carlo randomized algorithm for definite integration. It is strictly superior to the deterministic, regular subinterval method because it too works well in cases where the deterministic method succeeds, but the randomized method can better handle unknown, worst-case, and adversarial scenarios where the deterministic method fails.

The general Monte Carlo (definite) Integral estimator for a function $g(m) : \mathbf{M} \to \mathbf{G}$ from domain $\mathbf{M}$ to range $\mathbf{G}$ integrated over a region $A \subset \mathbf{M}$ is:

$$\int_{A \subset \mathbf{M}} \underbrace{g(m \in \mathbf{M})}_{\mathbf{G}} \underbrace{dm}_{\subset \mathbf{M}} \approx \sum_{j=0}^{N-1} \frac{g(m_j) \in \mathbf{G}}{p(m_j) \in \mathbf{M}^{-1}} \frac{1}{N} \tag{6}$$

where
$p(m) : \mathbf{M} \to \mathbf{G}^{-1}$; $p(m) \geq 0 \; \forall \; m \in \mathbf{A}$; $\int_A p(m)dm = 1 \in \mathbf{G}^{-1}$ is a **probability density function**. This density function must integrates to unity over area A and be nonzero anywhere that $g(m)$ is nonzero. It is otherwise unconstrained and may even take on infinite (but not negative) values. One implication of this definition is that the measure of A is incorporated into p. For example, if p is a uniform distribution an A is a 1D real interval, then $p(m) = 1/\|A\|$ and the estimator becomes the straight average that you would expect: $\frac{\|A\|}{N} \sum g(m)$.

The definite integral's value is in the set $\mathbf{M} \cdot \mathbf{G}$. By inspection, the running time of the right hand side is linear in N (the number of evaluations of g), but the quality of the estimate highly depends on the properties of both p and g.

How does this estimator perform on our previous

adversarial estimate of the integral of the cosine function? Here is an implementation using a uniform probability density function:

Python C++

```python
import random, math

def monteCarloEstimateIntegral(g, sample, A, N):
    I = 0
    for j in range(0, N):
        # m is the sample taken from the distribution, pm = p(m) is the value of the pdf at m
        (m, pm) = sample(A)
        I += g(m) / (pm * N)

    return I

def uniformSample(A): return (random.uniform(A[0], A[1]), 1 / (A[1] - A[0]))

def h(x): return -math.cos(x * 2 * math.pi)
print(monteCarloEstimateIntegral(h, uniformSample, (0, 100), 100))
```

When run with $N = 100$ samples, it produces estimates that mostly range from about -8 to +8. The correct answer is zero, so that's not perfect, but it is a much better estimate than the value of 100 produced by the deterministic estimator (depending on how you measure, you could say this is 80% better). Run with $N = 10^6$, the Monte Carlo estimator gives values close to zero. So, given more computation, it can be driven towards a better result without concern about hitting worst cases for functions that are reasonably smooth, even if they exhibit a lot of variation.

8. An Energy Example

To develop intution for Monte Carlo integration, let's apply the estimator to an energy consumption problem. This problem has units and an integrand that is nonnegative, so it

will be somewhat closer to our goal of a rendering application than our earlier abstract problem. I'm still using sine/cosine waves because they are a conveniently simple function that has varying derivatives, but there's nothing particularly important about the use of trigonometric functions.

Let $g(m \in \mathbb{R}_+ \text{ s}) \in \mathbb{R}_+ \text{ W} = \sin(m \cdot \pi/(86 \times 10^3 \text{ s})) \cdot 50 \text{ W} + 2 \text{ W}$ be the power consumed by my laptop computer as a function of the time of day m in seconds since midnight. The domain $\mathbf{M}$ is positive real-number time, in seconds, and the integration region is one day, $A = [0 \text{ s}, 86 \times 10^3 \text{ s}]$. The range $\mathbf{G}$ is positive real-number power, measured in watts. (The function and set names are odd because I'm using the notation from the previous section.)

The following Python program implements the Monte Carlo integration algorithm as literally as possible, using a uniform distribution for $p(m)$:

Python C++

```
    # Samples m from region A and returns tuple (m,p(m))
1   def g(m): return math.sin(m * math.pi / 86e3)
    * 50 + 2

2   A = (0, 86e3)

3   print(monteCarloEstimateIntegral(g,
    uniformSample, A, 1000))
```

When run with $N = 1000$ samples, this program prints the estimate 2940659.0100626466 for the value of the definite integral (it will be slightly different each time that it is run, due to the random numbers). In SI notation with our units, the numerical result was $2.94 \times 10^6 \text{ J}$. To evaluate the accuracy of the algorithm, in this particular case, we can analytically compute the true value of the definite integral as:

$$\int_{0\,\mathrm{s}}^{86\times10^3\,\mathrm{s}} \sin\left(m \cdot \frac{\pi}{86\times10^3\,\mathrm{s}}\right) \cdot 50\,\mathrm{W} + 2\,\mathrm{W}\; dm$$

$$= \left[-\cos\left(m \frac{\pi}{86\times10^3\,\mathrm{s}}\right) \frac{4.3\times10^6}{\pi}\,\mathrm{W}\cdot\mathrm{s} + m\cdot 2\,\mathrm{W}\right]_{m=0\,\mathrm{s}}^{m=86\times10^3\,\mathrm{s}}$$

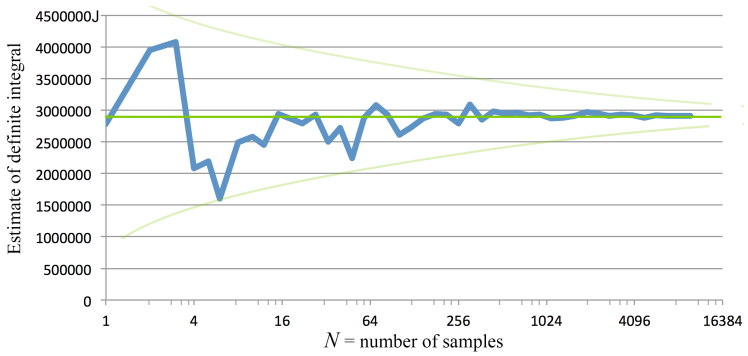
$$= 2.9\times10^6\,\mathrm{W}\cdot\mathrm{s} = 2.9\times10^6\,\mathrm{J} \qquad (7)$$

(Is this a lot of energy? In the units that electric utilities use, 2.9 megajoules is $0.81\,\mathrm{kWh}$. In the US, that would cost about ten cents on an electric bill today and is less energy than a space heater consumes in an hour.)

So, using only $N = 1000$ samples, the algorithm has come within about 1% of the true value. That is pretty good. With only $N = 1$ sample, the error is of course much higher. I observed a result of $0.5\times10^6\,\mathrm{J}$, which is 82% error.

As N increases and the result becomes accurate, we say that the process has nearly **converged** to producing the true value. Individual estimates for any given N may be too high or too low, but they tend towards increasingly expected accuracy under the law of large numbers as N grows.

The following figure visualizes this convergence and allows us to evaluate the behavior of the integrator with respect to the N parameter. The plot shows independent estimates of the integral for N samples, with a logarithmic horizontal axis. I generated this by running the above program for exponentially-spaced values of N and then showing the results produced. If the random number generator is also initialized independently for each run, then the actual estimate produced for a given run is independent. That is, the values in the following figure are artifacts of the Python random number generator, but the general trend of the difference between those values and the mean decreasing as N increases is a valid observation.



Estimates by the simple uniform-sampling integrator for the power example as integral value vs. log N .

The horizontal green line is the true result, to which the blue estimates appear to converge as the number of samples increases. We can tell that the mean of the estimate is corect because it is centered on that line.

The curved green lines are trendlines for the error, expressed as two standard deviations. If we ran the experiment many times, the blue curve would jump around but mostly lie between those curved lines. This Monte Carlo estimator's accuracy is known to improve with the square root of the number of the samples, which can be seen from those curves (keep in mind that the horizontal axis is logarithmic). That is, its improvement is sublinear in cost, so most of the gain is in increasing N at small values. We see only constant improvement for doubling the number of samples.

This problem was a convenient one to use to test correctness and convergence, since it has an easy analytic solution. The Monte Carlo integral estimator is of course most valuable for integrals that admit no analytic solution. For example, the integral in the rendering equation. In this case, the error in the estimate at each pixel will manifest as intensity noise across the image. This appears similar to an extreme example of the sensor noise that you can observe from any digital camera in real life. In a sense it arises from the same cause: real images are made by the discrete “samples” of light that are photons, and in a dark room, there aren't enough of them to average out the stochastic noise.

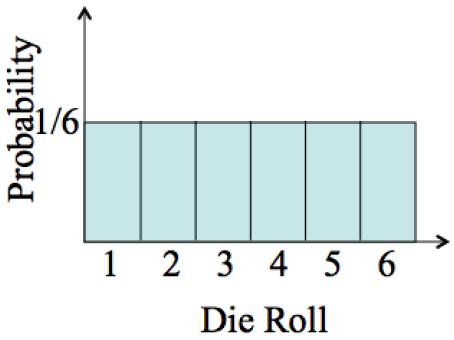
Light scatters when it strikes a surface, typically in many directions in different amounts. The **bidirectional scattering distribution function** (BSDF) models the distribution of exitant radiance [$\mathrm{W}(\mathrm{m}^2\mathrm{sr})$] from a beam of light. This is the function $f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o)$, which I also refer to as simply "the scattering function".

Contents

The Surface Geometry chapter defined a surface as the interface between two homogeneous media and divided a surface into geometry and material. It then presented a triangle mesh model for the geometry of the surface. This chapter presents models for the **material** properties of a surface. These are the **scattering function** $f_{X,\hat{n}}$, and two techniques for representing detail with images instead of meshes: **bump**/normal/displacement maps and **partial coverage** ("alpha").

1. Probability Density

The scattering function describes how a surface distributes incident radiance. That is, it models a *distribution* of exiting radiance for each incident direction $\hat{\omega}_i$. This distribution is like a probability distribution. Unlike a probability distribution, however, it does not integrate to one because some light will be absorbed at the surface. I say

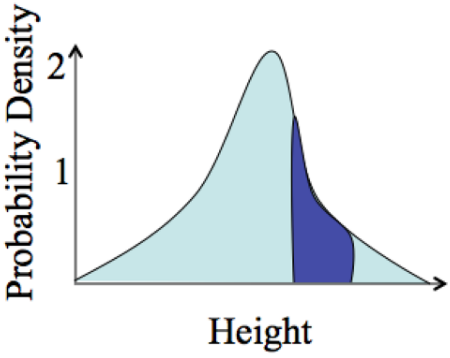


A discrete probability mass function.

"integrate" instead of "sum" because this is a distribution over a continuous space of directions.

Actions like flipping a coin or rolling a die have a discrete number of possible outcomes, so we model the probability of their outcomes using **probability mass** (often also just called probability). There you can say, "the probability of rolling a 4 is $P(\text{“}x = 4\text{ ”}) = 1/6$ ".

When considering a continuous space like distance, you can't say "the probability of my child growing to a height of exactly $1.8\ \text{m}$ tall is..." because there are infinite number of possible heights for a person to be. The probability of being "exactly" one of them is necessarily zero.



Continuous probability density.

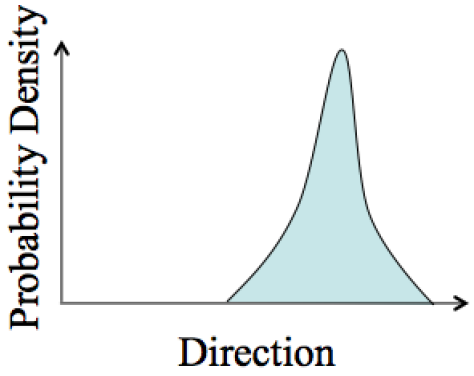
Continuous distributions are expressed in terms of **probability density** (also known as differential probability). You can integrate over a range to obtain a probability mass, e.g., "the probability of my child growing to a height between 1.7 and $1.9\ \text{m}$ tall is..." has a nonzero value. Like distance, time and directions have continuous domains, so we must use probability density with them.

Probability *mass* functions must *sum* to 1.0 to account for all collectively exhaustive, mutually exclusive events. The probablity of any one event is necessarily between 0 and 1.

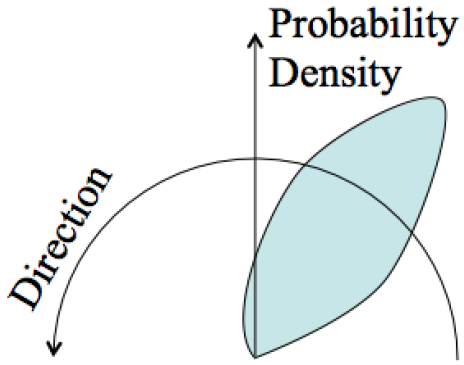
Probability *density* functions must *integrate* to 1.0 and be non-negative everywhere, but the density at any particular value may be arbitrarily large. That is because the probability of a value falling within some range is the area under the density function, e.g., the blue region in the figure on the right. Even a function containing *infinite* density at some location may integrate to 1.0 so long as it does so over a vanishingly small range.

Set aside the die-rolling and height examples and consider a simple situation more closely related to the scattering function: a ball rolling across a road and bouncing off a rough cobblestone curb. Since the cube is rough, the ball might bounce off it in a variety of directions. You might it expect to bounce more frequently with the outgoing nearly angle equal to and opposite the incoming angle, since that is the behavior observed with a smooth curb. Yet you also know that

there is a chance that it will bounce in a seemingly arbitrary direction. This is because sometimes the ball will strike a rock facet at an angle and bounce in an unusual direction, possibly even right back in the way that it arrived. You could write down a function that gives the probability density for any outgoing direction in terms of the incoming direction. If we graph the outgoing direction for a fixed 45-degree incoming direction, it might peak at 45-degrees on the other side, but be non-zero over a wide range:



Since we're considering radial directions, it is perhaps more intuitive to visualize the probability density function in polar form, where the outgoing angle is in fact an angle and the density is the radius of the plot:



Photons reflect in ways very similar to macroscopic balls in many cases. Of course, just as the curb in this example defines a smooth boundary to the road from a distance but is rough when observed up close, many seemingly smooth surfaces are microscopically rough at optical scales and can cause photons to bounce in arbitrary directions. To stretch the analogy a bit, you might also expect that different kinds of road edges and balls might produce different bounce distributions: a bowling ball at a steel curb would react differently than a beach ball at a muddy ditch. Photons can vary in frequency and surfaces can vary in their chemical structure, so the geometry of the sitation does not always dominate the scattering distribution.

This curb defines a smooth line from a distance but is rough when viewed up close.

2. The BSDF

The scattering function is formally known in computer graphics as the **bidirectional scattering distribution function** (BSDF). It is a function that models a scattering distribution in terms of two (i.e., "bi") directions.

It was formally defined by Nicodemus et al. [77] (as a national standard intended for the design of lighting fixtures!) as the ratio of scattered radiance to incident *irradiance*:

$$f_{X\in\mathbb{R}^3,\hat{n}\in S^2}(\hat{\omega}_i\in S^2,\hat{\omega}_o\in S^2)\in \mathrm{sr}^{-1} = \frac{\overbrace{dL_s(X,\hat{\omega}_o)}^{\mathrm{W/(m^2\,sr)}}}{\underbrace{L_i(X,\hat{\omega}_i)}_{\mathrm{W/(m^2\,sr)}}\,|\hat{\omega}_i\cdot\hat{n}|\,\overbrace{d\hat{\omega}_i}^{\mathrm{sr}}} = \frac{\Delta L_o(X,\hat{\omega}_o)}{\Delta E_i(X)} \tag{1}$$

In optics, a surface is called **reflective** if it scatters light so that the incoming and outgoing directions are in the same hemisphere with respect to the surface normal. A "reflective" surface does *not* have to be mirror-like under this definition. Nicodemus et al. were mostly concerned with painted walls and other opaque materials in buildings, so they only considered reflective surfaces in their original definition. Technically, they defined the **bidirectional reflectance distribution function** (BRDF). However, today, most rendering simply models its generalization to the BSDF in order to accomodate

a model of **transmission** of light through materials such as glass.

All BSDFs in the real world are constrained by thermodynamics to obey three properties [Veach1998Thesis] [Heitz2014Masking]:

1. **non-negative**,
$$f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o) \geq 0 \quad \forall \quad \hat{\omega}_i, \hat{\omega}_o, X$$

2. **energy conserving**
    1. Do not create energy:
$$\int_{S^2} f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o) \, |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i \leq 1 \quad \forall \quad \hat{\omega}_o, X$$
    2. No direction may reflect more light than was incident in some other direction.
    3. For every outgoing direction, all incident light must be reflected when the surface's net reflectivity is 1.0 (the "white furnace" test).

3. **reciprocal**,
$$\frac{f(X, \hat{\omega}_i, \hat{\omega}_o)}{\eta_o^2} = \frac{f(X, \hat{\omega}_o, \hat{\omega}_i)}{\eta_i^2}$$

The symmetry of reciprocity is particularly useful for two reasons. The first is that we only have to implement "half" of each BSDF in some sense. The second is that when done with care, we can trace light transport backwards from the camera instead of forward from the light, which we've implicitly assumed this far.

Note that the BSDF does not model the distribution of scattered photons, so it is not exactly analogous to the bouncing ball example in the previous section. Instead, it models the *radiance* transported by the photons. Because of this, it takes the projected area of the incident beam into account with a dot product. That is, the BSDF is defined in the reference frame of the outgoing light ray, so the radiance of the incoming light ray must be adjusted for the (differential) area over which it is distributed before it scatters. The fact that this scales the effective incoming radiance by the cosine of the angle of incidence is known as Lambert's Law.

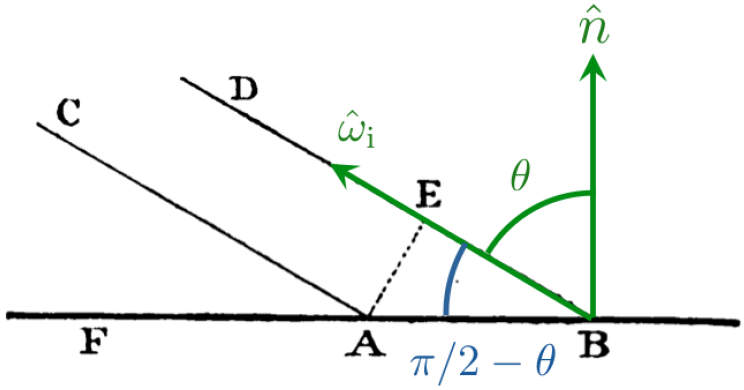
Beware that most popular computer graphics BSDFs do not satisfy all

of these physical properties. For efficiency, BSDFs are typically modeled from combinations of dot products with certain symmetries. Energy conservation and reciprocity are hard to enforce simultaneously in an efficient BSDF.

Edwards et al. [2006] and Heitz [2014] showed that placing a perfectly reflective white sphere in a "white furnace" that radiates uniformly *should* result in an all-white image, but that the most popular BSDFs all produce a clearly-visible sphere with darkened edges. Heitz explains that this is because the BSDFs were derived assuming that light scatters only once off the surfaces. For rough surfaces observed at glancing angles, that assumption does not hold and leads to a loss of light.

Lambert's Law

Lambert's Law, also known as Lambert's reflection law, Lambert's cosine law, and Lambert's projected area law, states that the *observed* area of a locally-planar surface is proportional to the cosine of the angle of incidence (of the observation direction), measured from the surface normal. An example of this is that, in a rain on a windless day, a flat roof gathers more water than a sloped roof of the same area. The rain is falling straight down and the sloped roof presents less area. You've probably felt a similar pheonmenon with air when holding a hand out of an automobile window: when your hand is horizontal, less wind hits it than when it is vertical.



Lambert's original diagram depicting the projected area law that now bears his name [Lambert1760Photometrie], with modern annotations in color.

Lambert showed why this is the case using the geometry of a right triangle. His diagram shows a beam of light (or wind, or rain) approaching a horizontal surface. Let the vector pointing back along the beam be $\hat{\omega}_\mathrm{i}$, which is at an angle with a measure of θ angle to the vertical, $\hat{n}$. The beam's width is defined by line segment AE in his figure. That is, if we measured the radiance transported along a ray

in the beam, that measurement would be power per area per solid angle, and the relevant area would be that of AE. But in the reference frame of the surface, the beam is spread over the area of AB when it strikes. Thus the measurement of the incident radiance is too high if directly applied in the surface's frame--it was measured over a small area, and must now be distributed over a larger one. Note that triangle ABE is a right triangle because the original measurement segment AE is necessarily orthogonal to $\hat{\omega}_i$ (we always measure radiance through a region orthogonal to the direction of the ray). This means that AB is the hypotenuse of a triangle and AE is one side. They are related by the sine of the acute angle ABE, which has measure $\pi/2 - \theta$. Because $|\sin(\pi/2 - \theta)| = |\cos(\theta)|$,

$$\|AE\| = |\cos\theta| \|AB\| \tag{2}$$

$$= |\hat{\omega}_i \cdot \hat{n}| \|AB\| \tag{3}$$

thus, any measurement relative to AE must be scaled by the cosine of the angle of incidence, or the dot product of the normal with the direction of incidence, before applying it in the plane of AB.

As a historical note, Lambert originally expressed his law in a different reference frame than we do today. He measured the angle of incidence from the horizontal, not from the surface normal, and considered the area of the incident beam instead of the area on the surface. That is, his diagram gives the angle of incidence as ABE, and computes the effective area of AE relative to AB: "Wenn man nun AB als Radius und als Einheit annimmt, so wird AE der Sinus des Incidenzwinkels sein"; "If we assume that AB has unit length, then AE is the sine of the angle of incidence" [Lambert1760Photometrie p22]. This leads to the interesting conclusion that Lambert would have called "Lambert's Cosine Law" the "Sine Law".

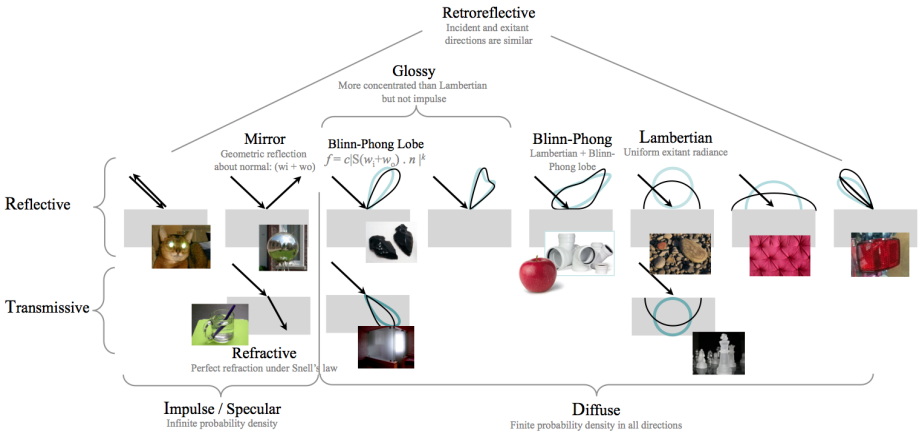
3. Scattering Functions

Common Terms

The **scattering distribution** is often modeled as a sum of terms that are themselves distributions. To communicate about scattering, we name terms that correspond to observable illumination phenomena, computationally convenient expressions, and that are amenable to specific sampling strategies. It is also common to describe a

scattering event by the term that modeled it, e.g., “a diffuse reflection event”.

The diagram below plots several scattering terms (density functions, in black) in polar form. Each is shown for $\hat{\omega}_i$ at 45-degrees from the normal. The grey box represents the medium on the opposite side of the normal. The exiting **photon distribution** (depicted with a fat blue curves) is the radiance distribution diminished by the cosine of the angle of exitance. This occurs because a higher density of scattering points is observed at glancing angles, and thus proportionally fewer photons yield the same radiance. That is, this is an application of Lambert's law.



Common scattering function terms. BSDF f in black, photon scattering distribution in blue.

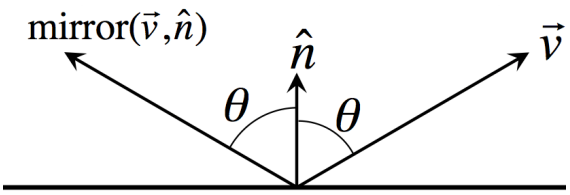
The names in the diagram are drawn from modern physically based rendering and physics, in which precise scattering terminology is important. Beware that it is common to casually apply imprecise phenomenological graphics terminology from the 70's. This primarily occurs in the real-time rendering/games community. For example, in that community, “specular” means glossy, “diffuse” means Lambertian, and “reflection” means mirror.

The examples shown are evocative, not exhaustive, and only cover common rendering cases. Also, the overall surface appearance is often due to non-local effects that cannot be accurately modeled by a BSDF. For example, the soft appearance of skin is due to a *sequence* of diffuse transmission events that are collectively called subsurface scattering, so no BSDF can correctly model this effect within a single event.

The following sections describe several of these phenomena in more detail and give concrete, implementable scattering functions.

Mirror Reflection

An optically smooth surface, such as a glass mirror or polished metal, reflects a ray of light with the angle of reflection equal to the angle of incidence:



$$\text{mirror}(v \vec{\in} \mathbb{R}^3 u, \hat{n} \in S^2) \in \mathbb{R}^3 u$$

$$=$$

$$2\hat{n}(v \vec{\cdot} \hat{n}) - v \vec{}$$

(4)

In this notation, $\hat{\omega}_o = \text{mirror}(\hat{\omega}_i, \hat{n})$. This is analogous to a hard ball bouncing off a hard, smooth surface.

The scattering term that describes mirror reflection has an integral equal to the fraction $\varrho(\hat{\omega}_i)$ of radiance that is reflected for a given direction $\hat{\omega}_i$:

$$\varrho(\hat{\omega}_i) = \int_{S^2} f(\hat{\omega}_i, \hat{\omega}_o) d\hat{\omega}_o \tag{5}$$

For a perfect mirror, $\varrho(\hat{\omega}_i) = 1$ since no energy is lost.

There is zero probability density at any direction except for the mirror direction, i.e., $f(\hat{\omega}_i, \hat{\omega}_o) = 0$ for $\hat{\omega}_o \neq \text{mirror}(\hat{\omega}_i, \hat{n})$. The probability density at the mirror direction is problematic, however. It must integrate to $\varrho(\hat{\omega}_i)$ over a solid angle whose measure is *zero*, since there is a single direction in which all outgoing radiance is reflected. This means that

$$f(\hat{\omega}_i, \text{mirror}(\hat{\omega}_i, \hat{n})) = \frac{\varrho(\hat{\omega}_i)}{0 \cdot |\hat{\omega}_i \cdot \hat{n}|} \tag{6}$$

That means the value of the scattering function f must be *infinite* at the mirror direction. Physicists model this with a delta function, which is the derivative of a Heaviside step function:

$$f(\hat{\omega}_i, \hat{\omega}_o) = \frac{\delta(\hat{\omega}_o, \text{mirror}(\hat{\omega}_i, \hat{n}))\, \varrho(\hat{\omega}_i)}{|\hat{\omega}_i \cdot \hat{n}|} \tag{7}$$

The delta function is meaningless outside of an integral, but within an integrand (say, the integrand found in the Rendering Equation), the value of whole integral is well-defined.

Unfortunately, delta functions are just a notation game. In practice, we can't implement the scattering function using a delta function. While floating point representation can express "infinity", it doesn't have a way to encode the relative scales of infinity needed for a density function. A mirror that reflects 50% of the incident light and one that reflects 100% of the incident light both have infinite density functions, but those functions integrate to 0.5 and 1.0, respectively. We need a representation that can distinguish these cases when implemented in software.

A solution is to model the infinities in the scattering function in a similar way to how we resolved them for direct illumination from a point source. In the direct illumination case, we had to consider integrals of the form $\int L f$ where L was infinite for one direction and zero elsewhere. Instead of trying to express L directly, we represented the value of the integral. For a mirror, we have the opposite case for a similar integral, where f is infinite for one direction. The solution is to express the result of the entire integral rather than evaluate f.

To simplify the notation, assume that ϱ is invariant with direction, so that the mirror is equally reflective no matter what angle it is viewed from. Also note that $\text{mirror}(\hat{\omega}_\text{o}, \hat{n}) = \hat{\omega}_\text{i}$ when $\text{mirror}(\hat{\omega}_\text{i}, \hat{n}) = \hat{\omega}_\text{o}$. Applying these and omitting the integral in favor of its value, we can model a mirror as a different form of the Rendering Equation:

$$L_\text{o}(X, \hat{\omega}_\text{o}) = L_\text{e}(X, \hat{\omega}_\text{o}) + \varrho_X \, L_\text{i}(X, \text{mirror}(\hat{\omega}_\text{o}, \hat{n})) \tag{8}$$

when both directions are in the positive hemisphere, i.e., $\hat{\omega}_\text{i} \cdot \hat{n} > 0$ and $\hat{\omega}_\text{o} \cdot \hat{n} > 0$. Of course, rather than adding a special case to a program to evaluate the Rendering Equation differently at mirrors, it is good practice to provide a general way to handle infinite density. We say that the scattering function has an **impulse** at the mirror direction, which is what the delta function is attempting to model symbolically.

Implementing Impulses

An impulse has a direction and a magnitude — it is a vector. However, since we'll later want to represent magnitude that varies with the frequency of light, it is useful to represent these with separate state, e.g.,

```
class Impulse {
public:
    // ϱ
    Color3  magnitude;

    // ω̂i
    Vector3 direction;
};
```

We can now think of a BSDF implemented on a surfel as a method that evaluates the finite part of the scattering density (which we have yet to encounter) and an array of impulses:

```
class Surfel {
public:
    ...
    virtual void getImpulses(const Vector3& wo,
Array<Impulse>& impulses) const;

    // Finite part of f_{X,n̂}(ω̂i, ω̂o)
    virtual void evaluateFiniteDensity(const Vector3&
wi, const Vector3& wo) const;
};
```

Note that Lambert's law does not apply in the case of mirror reflection. That's because Lambert's argument is for the area over which a *beam* of light distributes its radiance. In the case of a mirror, we're evaluating perfect specular reflection of an individual *ray* of light, not a beam. No matter what angle the ray strikes the mirror at, it always distributes its radiance over a single point.

> ***Exercise* [mirror]:** A mirror creates a virtual image that is identical to the real image that you would perceive if the entire world was inverted left-to-right and viewed through a window instead of the mirror. Why does a mirror invert left-to-right, but not top-to-bottom? *Hint: think about your reference frame.*

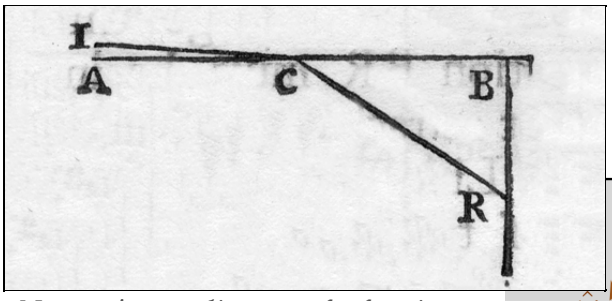
Transmission and Refraction

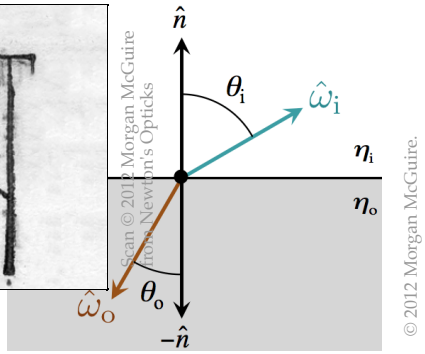
A material that **transmits** light, such as glass, is described by a scattering function that is non-zero for $\hat{\omega}_i$ and $\hat{\omega}_o$ in opposite hemispheres with respect to the normal, $\hat{n}$. Transmission may diffuse, such as through frosted glass, in which case incident rays are

arbitrarily perturbed and the scattering function is finite everywhere in the transmissive hemisphere. Or scattering may be "specular" (to abuse etymology somewhat), in which case there is an impulse in the scattering function in the transmissive hemisphere.

Let us consider only the case of perfect, or specular, transmission. Recall that I originally defined a surface to be the interface between two homogeneous media, and defined homogeneity based primarily on the the speed of propagation of light within a medium. The real index of refraction, η, is inversely proportal to the speed of propagation of light within a medium. When a plane wave (of anything, not just light) passes between regions in which it propagates at different rates, it changes direction. This is called **refraction**. The incident and outgoing angles are related by Snell's law,

$$\begin{aligned} \frac{\sin\theta_o}{\sin\theta_i} &= \frac{\vec{v_o}}{\vec{v_i}} = \frac{\lambda_o}{\lambda_i} = q = \frac{\eta_i}{\eta_o} \end{aligned} \tag{9}$$





Newton's [1704] diagram of refraction of ray IC to CR at interface AB.

Snell's law in vector form gives the direction of the refractive impulse in the BSDF as

$$\text{refract}(\hat{\omega}_i \in \mathbb{S}^2, \hat{n} \in \mathbb{S}^2, q \in \mathbb{R}) \in \mathbb{S}^2 = \hat{\omega}_o$$

$$= -q(\hat{\omega}_i - (\hat{\omega}_i \cdot \hat{n})\hat{n}) - (\sqrt{1 - q^2(1 - (\hat{\omega}_i \cdot \hat{n})^2)}\,)\hat{n}$$

(10)

where $q = \eta_i/\eta_o$. An imaginary result for $\hat{\omega}_o$ indicates total internal reflection [Heckbert1989Refraction] [RISUPM p24].

The other property that may change at an interface is the absorption rate of the material, which is related to the imaginary part of the index of refraction, $\varkappa$. It is common in graphics to model transmissive surfaces as if they were of fixed thickness (and to neglect the fact that the thickness observed varies with the angle of incidence). In this case, one can model the absorption that occurs for that thickness by decreasing the magnitude of the impulse. A more correct solution is to compute the attenuation of transmitted light based on the distance actually travelled through the medium from X to Y, which is given by the Beer-Lambert Law:

For points $X \in \mathbb{R}^3\,\mathrm{m}$ and $Y \in \mathbb{R}^3\,\mathrm{m}$ in a homogeneous medium with absorption coefficient $\alpha \in \mathbb{R}_+\,\mathrm{m}^{-1}$,

$$L_i(X, \hat{\omega}) = L_o(Y, -\hat{\omega})e^{-\alpha\|Y-X\|} \quad , \qquad (11)$$

where $\hat{\omega} = S(Y - X)$.

In the real world, any surface that transmits light must also specularly reflect it, with the ratio given by the Fresnel equations (which are described at the end of the chapter).

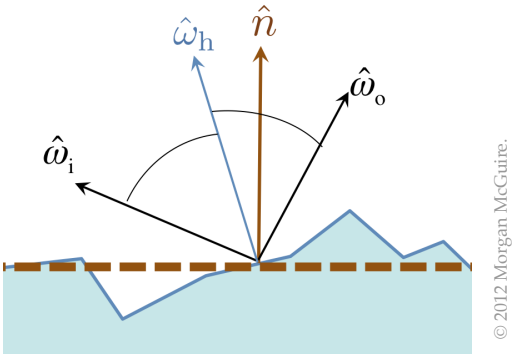
Microfacets

Specular surfaces are visually interesting, but are also less common in everyday scenes than glossy and matte surfaces. Specular reflection arose from the idealized condition of an optically flat surface. Since most surfaces are not flat at a microscopic level, light does not reflect from them specularly. Instead, at a microscopic scale light hits tiny **microfacets** of the surface. These may be individually specular, but collectively allow the light to reflect in arbitrary directions from the macrosurface. We say that the reflection is **diffuse** in this case. For example, brushed aluminum contains tiny scratches that make it glossy but not mirror reflective. Likewise, sandblasted

(a.k.a. frosted) glass diffuses the light that transmits through it. These cases are analogous to the rough curbstone that reflects a rolled ball at arbitrary angles.

The amount of diffusion of light varies with the smoothness of the surface. A specular surface is infinitely smooth. A glossy surface, like an apple, is less smooth. A nearly matte surface such as dry earth is extremely rough. The Blinn-Phong model of glossy reflection [Blinn1977Reflection] characterizes the smoothness of a surface with a scalar s_X, where $s_X = \infty$ is a mirror, $s_X \approx 0$ is nearly matte, and values in between give varying levels of gloss.

Were a ray of light to reflect off a specular microfacet from $\hat{\omega}_i$ to $\hat{\omega}_o$ under the Blinn-Phong model, it *must* have reflected off some facet whose normal was $\hat{\omega}_h$ satisfying $\text{mirror}(\hat{\omega}_i, \hat{\omega}_h) = \hat{\omega}_o$ since f is zero for all other angles. This microfacet normal $\hat{\omega}_h$ is called the **half vector** since it is halfway between $\hat{\omega}_i$ and $\hat{\omega}_o$:



Geometry of the half vector.

$$\hat{\omega}_h \in S^2 = S\left(\frac{\hat{\omega}_i + \hat{\omega}_o}{2}\right) \tag{12}$$

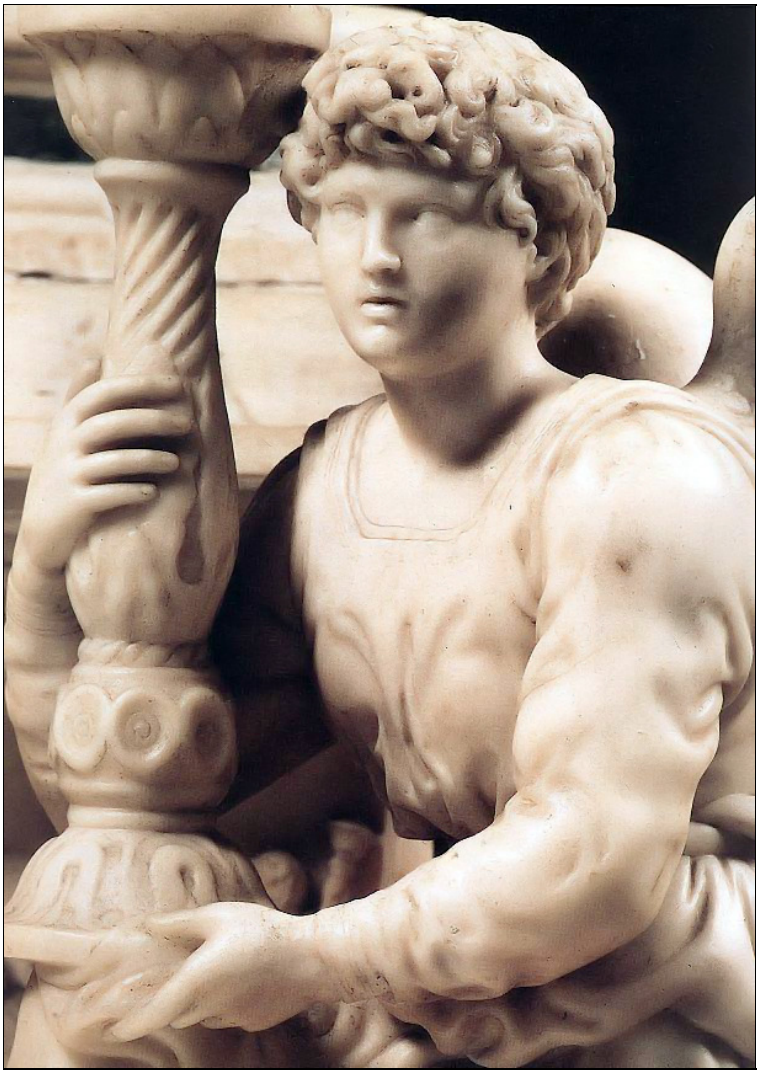
The cosine of the angle between $\hat{\omega}_o$ and $\hat{\omega}_h$ is the same as the cosine of the angle between $\hat{\omega}_i$ and $\hat{\omega}_h$, and it is given by $\hat{\omega}_o \cdot \hat{\omega}_h = \hat{\omega}_i \cdot \hat{\omega}_h$. In polar form, cosine forms a fat lobe. Raising cosine to an exponent narrows it, since the cosine function is always less than 1. The Blinn-Phong model exploits this to efficiently distribute scattering density about $\hat{\omega}_h$ in a way that yields a narrow density lobe for high smoothness (s_X) and a broad lobe for low smoothness:

$$f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o) \propto \max(\hat{\omega}_o \cdot \hat{\omega}_h, 0)^{s_X} \tag{13}$$

While this function is physically plausible, there is nothing physically-based about this choice of function — it uses the dot product and exponent solely because they are a computationally efficient and convenient way of expressing a lobe of concentrated density.

Subsurface Scattering

When light strikes an electrically insulating material, such as skin, plastic, or rubber, it tends to penetrate some distance into the material. When the light scatters, it then scatters *within* the medium, rather than strictly at the surface. If the medium is highly absorptive, like black rubber, the light may not travel very far. For materials like skin and marble, light can travel a few millimeters before being completely absorbed or emerging back out of the medium. When it emerges at a slightly different location from where it was incident, the appearance of the surface is softened because the light is effectively blurred over the surface. That is why skin and marble appear "soft" — and why marble statues look lifelike. The direction of propagation of the outgoing light is also essentially random after this subsurface scattering.



Michelangelo 1494

Michelangelo's Angel with Candlestick appears soft, like human skin, because light scatters beneath the surface of the marble.

When a medium is fairly transmissive, such as water or smoke, light can travel a relatively long distance while still retaining most of its energy. The direction of propagation is also preserved somewhat in

these cases, so you can see moderatly far through mist or smoke and fairly far through water. In fact, air scatters and absorbs some light passing through it, but the amount is negligible over meters and the effect is only visible on the scale of kilometers in most cases.

Surfaces that exhibit subsurface scattering *also* exhibit specular or glossy scattering from their surface. This is why analytic BSDFs are often modeled with a sum of terms. Only light that did not scatter at the surface can be transmitted, so the total energy from specular reflective impulses and the glossy portion of the finite BSDF should be removed before computing specularly transmitted and subsurface terms.

There are general models of subsurface scattering and special case ones for important natural materials such as clouds, human skin, and foliage. These have varying cost depending on the accuracy, but are generally more expensive than the glossy microfacet model. However, one extremely simple model is employed by almost all analytic BSDFs--the Lambertian diffuse term.

Lambertian Reflection

Lambert observed that many surfaces have the property that they appear equally bright from all viewing orientations [Lambert1760Photometrie]. For example, the ideal matte wall paint produces a wall that looks equally bright when you are looking at it from across the room as when your head is next to the wall and you are looking along it. Such a surface is called **Lambertian**, or occasionally "perfectly diffuse". We take this property for granted and often think of it as the default appearance for a material because most natural surfaces exhibit approximately this behavior.

Most Lambertian materials appear that way because subsurface scattering causes the outgoing *radiance* to be nearly equally distributed in all directions. Thus at a point X on a Lambertian surface, $L_o(X, \hat{\omega}_o)$ is nearly constant with respect to direction $\hat{\omega}_o$ (in the hemisphere above the surface). It also follows that a Lambertian BSDF is trivially constant:

$$f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o) \in \mathrm{sr}^{-1} = \frac{\rho \in \mathbb{R}}{\pi \,\mathrm{sr}} \tag{14}$$

when $\hat{\omega}_i \cdot \hat{n} > 0$ and $\hat{\omega}_o \cdot \hat{n} > 0$, and $f = 0$ otherwise. The (unitless)

numerator $0 \leq \varrho \leq 1$ is the **reflectivity** of the surface. The denominator is $\int_{S^2} \max(\hat{n} \cdot \hat{\omega}_i, 0) d\hat{\omega}_i = \pi$, which normalizes the BSDF.

Note that a Lambertian surface distributes *radiance* equally. It does not distribute *photons* equally. It must distribute the photons according to a cosine distribution — i.e., according to Lambert's law. To see this, follow a simple thought experiment. Look through a cardboard tube at a matte wall, orthogonally. You perceive a disk at some brightness. Now walk in a circle about the point that you see, always keeping it in view. You're changing $\hat{\omega}_o$ from along the normal towards perpendicular to the normal. As you do so, the disk that you perceive becomes an elipse in the reference frame of the wall (although you can't tell). You also perceive that the brightness of that "disk" is unchanged. That is, you are exposed to the same number of photons regardless of the angle from which you view the wall. For the number of photons to remain constant while the area over which you are collecting them increases, the density of photons must be inversely proportional to the area that you observe. The area that you observe through the tube is proportional to $1/|\hat{\omega}_o \cdot \hat{n}|$, so the number of photons must be proportional to $|\hat{\omega}_o \cdot \hat{n}|$.

Also note that while Lambertian surfaces are mathematically convenient and perfect, there is nothing physically perfect about them. That is, they happen to distribute radiance uniformly, but there are surfaces with subsurface scattering that distribute radiance unevenly as well.

One very interesting surface material is Earth's moon. Because we are far away from the moon, its BSDF combines subsurface effects, glossy/specular effects (including retroreflection), and "microfacets" that are meters wide but are imperceptibly small when viewed from Earth. The surface of the moon appears about equally bright over all illuminated regions, and then suddenly plunges into darkness at the terminator line (this is what causes the cycles of the moon — illumination from different directions by the sun, and self-shadowing). For the moon's appearance ($L \cdot f \cdot |\hat{\omega}_i \cdot \hat{n}$) to be invariant of $\hat{\omega}_i$, it must have a BSDF that is based on $\hat{\omega}_i$. In fact the BSDF must be proportional reciprocal of the cosine of the angle of incidence to cancel the $|\hat{\omega}_i \cdot \hat{n}|$ term in the Rendering Equation. Consider how that differs from the appearance of a rubber Lambertian ball, which shows a gradient of intensity that falls continuously to zero at the terminator line.

The Fresnel Effect

All materials become more specular/glossy at glancing angles, and are more transmissive/Lambertian towards normal incidence. For example, the table in the image below reflects the white box significantly more when viewed on the right at a glancing angle as compared to the image on the left taken at a higher elevation angle.

This wood table shows strong glossy reflections only at glancing angles.

This effect is predicted and described by the Fresnel equations, which are different for conductors and insulators. The curves described by those equations are simpler than they appear in the math. Schlick noted that they can be well-approximated for many materials using a very efficient approximation of the amount of specularity:

Math | C++ | GLSL

$$F_S(\theta \in \text{Rrad}) \in \text{R}$$

$$= F_0 + (1 - F_0)(1 - \max(0, \cos\theta))^5 \tag{15}$$

[Schlick1993Reflectance]

$$F_S(\hat{\omega}_i, \hat{\omega}_o) = F_S(\hat{\omega}_o, \hat{\omega}_i) \tag{16}$$

$$= F_0 + (1 - F_0)(1 - \max(0, \hat{\omega}_h \cdot \hat{\omega}_i))^5 \tag{17}$$

$$= F_0 + (1 - F_0)(1 - \max(0, \hat{\omega}_h \cdot \hat{\omega}_o))^5 \tag{18}$$

where $\hat{\omega}_h = S(\hat{\omega}_i + \hat{\omega}_o)$.

[Shirley1997Reflection]

In these equations, F_0 is the measured or estimated Fresnel reflectance at **normal incidence** ($\hat{\omega}_i = \hat{n}$). The amount of transmission is simply $F_T = 1 - F_S$.

4. Partial Coverage

Sometimes it is convenient to approximate a volume that contains a complex mixture of a singule medium with air (or other surrounding transmissive medium) as if it had a single medium. For example, to model smoke particles in air as a "volume of smoke", or a window screen or thin cloth as a single object instead of a collection of threads. In this case, the surfaces and media are a very coarse approximation when viewed up close, but may be sufficiently accurate when a viewed at a reasonable distance. Distance is of course relative--we might model the columns of a building and air between them as a single mixed medium from several kilometers away.

The surface bounding such a mixed medium is said to have **partial coverage**, meaning that the material described is intended to only model a fraction of the area covered by the geometric surface in the model. Partial coverage is historically represented by the variable α, which ranges from no coverage (all air) $\alpha = 0$ to full coverage (all material) $\alpha = 1$.

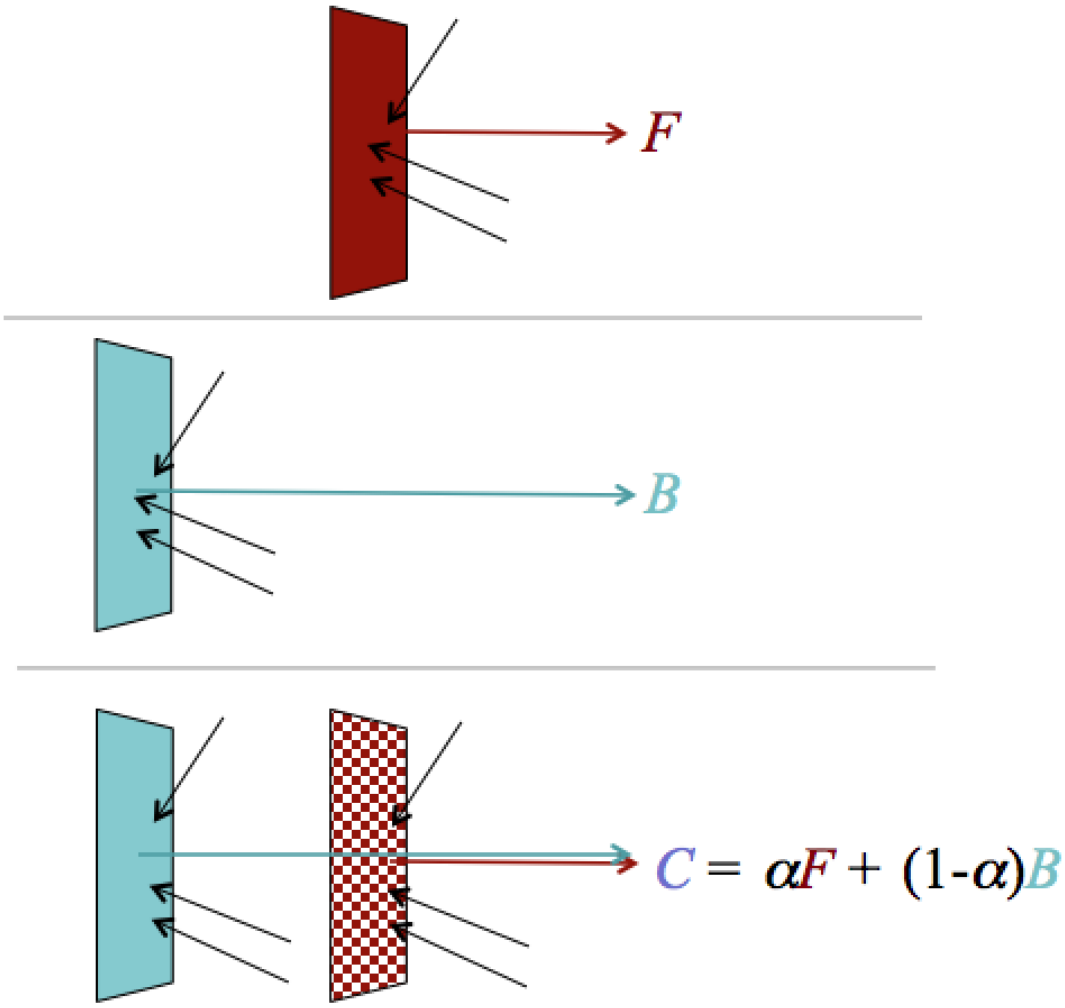
Compositing

Porter and Duff [1984] published the first comprehensive description of modeling partial coverage in the context of a single pixel covered by multiple surfaces. Perhaps the most frequently employed result from their work is the **over** operator. It describes a way to compute the net radiance along a ray that passes through an opaque surface with partial coverage. Let B be the radiance along the ray before it strikes the surface, i.e., the background radiance. Let F be the radiance reflected from the surface, i.e., the foreground radiance. Let α be the partial coverage of the surface. In this case, the net net radiance of "F over B" is the **composite** C given by:

$$C = \alpha F + (1 - \alpha)B = B + (F - B)\alpha \qquad (19)$$

Note that this is equivalent to linear interpolation between F and B

by α.



*F composited **over** B with partial coverage α*

This leads to a straightforward algorithm for shading a surface with partial coverage. Consider the case where a primary ray has been traced to a surfel F. If there is non-zero partial coverage, then we can simply bump the ray and continue tracing:

```
1  Radiance3 Lo(const Surfel& F, const Vector3& wo) {
2      Radiance3 L = ...;

3      if (F.partialCoverage < 1) {
           // Continue tracing the ray
4          L += ...
5      }

6      return L;
7  }
```

Alpha Cutout

Just as it is common to represent a surface with varying BSDF parameters using a texture map, you can vary the partial coverage

value over a surface. This so-called "alpha channel" in a texture map is often packed in memory next to the Lambertian components. That conveniently aligns 8-bit red, green, and blue values to 32-bit word boundaries by inserting an 8-bit alpha value.

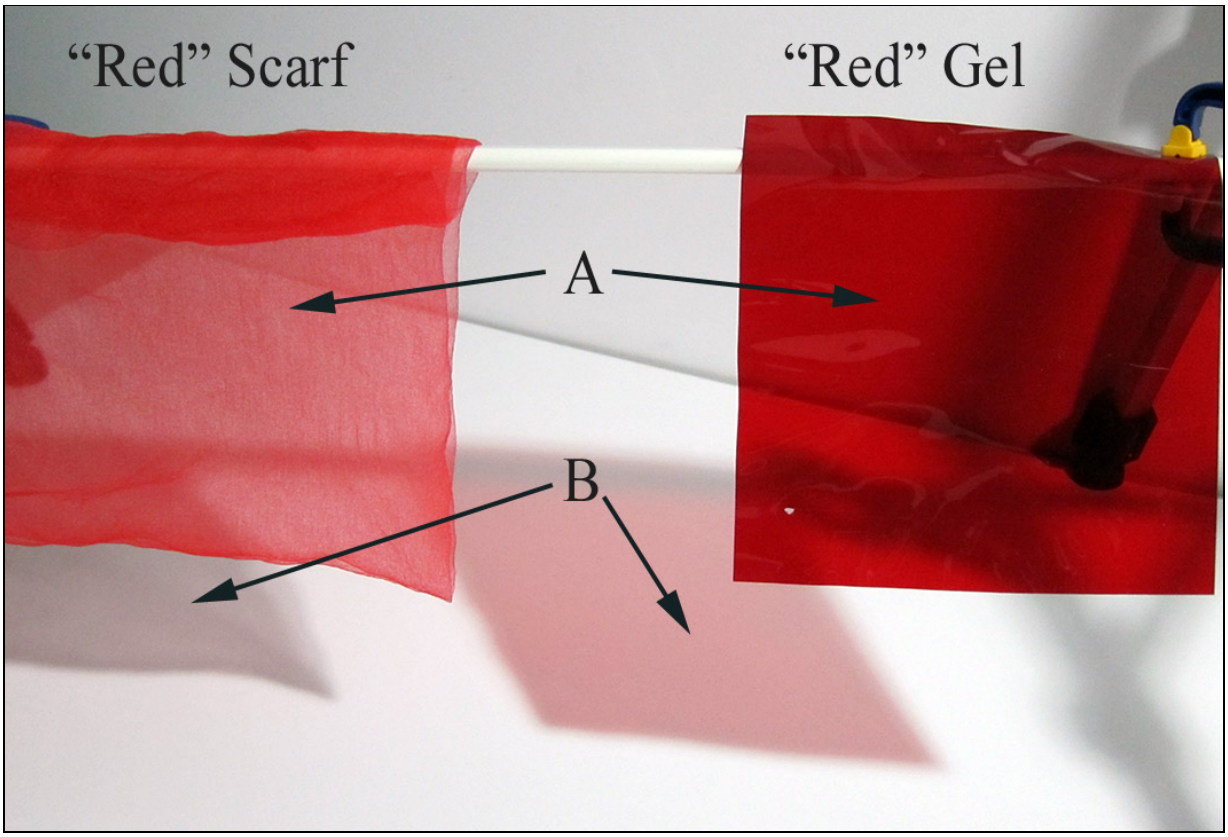
Since any hole can be represented as an arbitrary material with $\alpha = 0$, partial coverage is also convenient for modelling thin, irregular surfaces such as tree leaves. In this case, the leaf could be modelled as a quadrilateral with a texture map of varying α values. The areas outside of the leaf would have $\alpha = 0$, the areas inside the leave would have $\alpha = 1$, and pixels immediately adjacent to the border of the leaf would have fractional values.

Perceived Color

We often casually refer to the "color" of a material. This is necessary for effective communication, yet it is important to recognize that materials do not really have color and be aware of the ways in which color is misleading. Color is a complex phenomenon that is the human visual system's *perception* of spectral distribution. It is relative to the content of an image, the observer's recent history, and the individual's biology. Even in a controlled laboratory setting a single individual will perceive different spectral distributions as identical colors due to the sensory limitations of the human retina.

Some of these limitations are unavoidable given today's display technology, the frequent choice of only simulating a small number of frequencies of light, and the inescapable biology of the human visual system. However, at the most practical level for modeling scenes, it is important to distinguish between phenomena which yield similar appearance in some situations but different appearance in others. In particular, the distinction between color resulting from transmission and from reflection can easily be modeled fairly accurately if one considers a few representative cases.

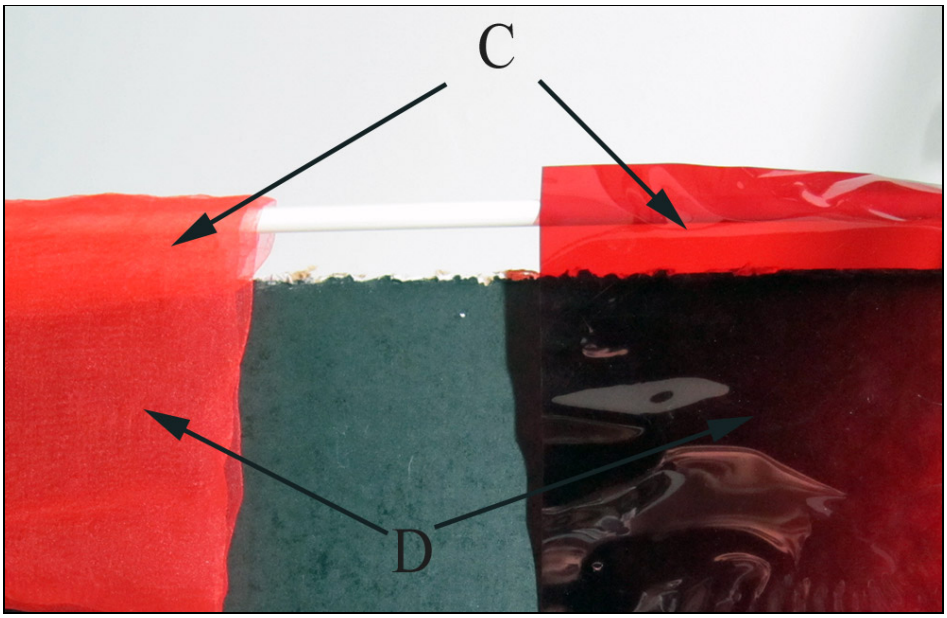
Consider the two objects in the following photograph, each of which appears red. On the left is a scarf and on the right is a theatre gel.



Perceived color of A) surface and B) shadow for “red” reflective and transmissive materials.

The scarf appears red because it reflects almost all light at low frequencies and absorbs almost all light at medium and high frequencies. Yet we can see through it. That is because the perceived geometric surface of the scarf has only approximately $\alpha = 0.5$ coverage; it is composed of threads and holes between them. The gel appears red and we can see through it because it transmits about 50% of the low frequency light that is incident from behind, and it absorbs other frequencies. However, it has $\alpha = 1.0$ coverage because there are no holes. These are very different models: red reflective with partial coverage and red transmissive with full coverage. They happen to produce similar appearance at A, on the body of each surface over a white background. But note that the shadows at B are different colors. The scarf casts a gray shadow because it does not selectivly transmit light with respect to frequency. The gel casts a red shadow because it transmits red light in both directions.

Likewise, if we examine the color of these objects against a black background, we can see that the reflective scarf still reflects low frequency light, but the transmissive gel appears nearly black because now there are few photons with low frequencies passing through from behind.



"Red" reflective and transmissive materials against C) white and D) black backgrounds.

When modeling a translucent surface, always consider these issues. Is the surface actually an approximation of a mixture of holes and opaque material, or is it a truly continuous surface that transmits light? What color should its shadow be? What color would it appear against black and against white?

[Blinn1978Bump]

References

[Blinn1977Reflection]

Models of Light Reflection for Computer Synthesized Pictures
James F. Blinn
in *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, p. 192-198, ACM, New York, NY, USA, 1977.
Official URL: http://dl.acm.org/citation.cfm?id=563893
Free URL: https://www.microsoft.com/en-us/research/wp-content/uploads/1977/01/p192-blinn.pdf
http://doi.acm.org/10.1145/563858.563893

[Blinn1978Bump]

Simulation of Wrinkled Surfaces
James F. Blinn
SIGGRAPH Comput. Graph. 12:3, p. 286-292, ACM, New York, NY, USA, August1978.
Official URL: http://doi.acm.org/10.1145/965139.507101

[Edwards2006BRDF]

The Halfway Vector Disk for BRDF Modeling
Dave Edwards, Solomon Boulos, Jared Johnson, Peter Shirley, Michael Ashikhmin, Michael Stark, and Chris Wyman
ACM Transactions on Graphics 25:1, p. 1-18, ACM, New York, NY, USA, 2006.
http://doi.acm.org/10.1145/1122501.1122502

[Edwards2006BRDF]

The Halfway Vector Disk for BRDF Modeling
Dave Edwards, Solomon Boulos, Jared Johnson, Peter Shirley, Michael Ashikhmin, Michael Stark, and Chris Wyman
ACM Transactions on Graphics 25:1, p. 1-18, 2006.
Official URL: http://dl.acm.org/citation.cfm?id=1122502
Free URL: http://www.cs.utah.edu/~boulos/papers/brdftog.pdf

[Heitz2014Masking]

Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs
Eric Heitz
JCGT 3:2, p. 48-107, 2014.
Free URL: http://jcgt.org/published/0003/02/03/paper.pdf

[Lambert1760Photometrie]

Photometrie
Johann Heinrich Lambert
p. 433, W. Engelmann, 1760.
Free URL: http://archive.org/details/lambertsphotome00lambgoog

[Newton1704Opticks]

Opticks
Isaac Newton
Sam Smith and Benj. Walford, 1704.
Free URL: https://archive.org/details/Optics_285

[Nicodemus1977Reflectance]

Geometrical Considerations and Nomenclature for Reflectance
F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis
in *Final Report National Bureau of Standards, Washington, DC. Inst. for Basic Standards.*, Ott (ed.), October1977.
Official URL: http://www.amazon.com/Geometrical-considerati

ons-nomenclature-reflectance-Standards/dp/B003TSD18S

[Porter1984Compositing]

Compositing Digital Images
Thomas Porter and Tom Duff
SIGGRAPH Comput. Graph. 18:3, p. 253-259, ACM, New York, NY, USA, January1984.
Official URL: http://doi.acm.org/10.1145/964965.808606

[Veach1998Thesis]

Robust Monte Carlo Methods for Light Transport Simulation
Eric Veach
Adviser-Leonidas J. Guibas, Stanford University, Stanford, CA, USA, 1998.

Illumination emitted from the light sources directly into the camera, and that **direct illumination** scattered into the camera together account for the highest-magnitude terms in the Rendering Equation. Because these two kinds of light transport paths are short, they can be evaluated without tackling the full integral equation. Proper direct illumination computation also requires computing the line-of-sight from the light source to each point visible to the camera in order to produce shadows.

Contents

Although these two terms contribute the most to the rendering equation, considering only emitted and scattered direct illumination necessarily underestimates the total illumination and the image will be too dark. This is especially true in shadows, which will be completely black if there is a single light source. An **ambient term** is a common coarse estimate of the missing illumination intended to improve the accuracy and appearance of the final image without incuring significant computational cost.

1. Radiance is Conserved Along a Ray

Recall that the simple black-and-white ray caster computed the shade at each pixel in an `App::L_i(X, wi)` method. This will be our implementation of $L_{\mathrm{i}}(X, \hat{\omega}_{\mathrm{i}})$, and it is now time to replace the placeholder implementation.

Also recall that radiance is conserved along a ray through vacuum between points X and Y if there is no object between them, i.e.,

$$L_{\mathrm{i}}(X, \hat{\omega}_{\mathrm{i}}) = L_{\mathrm{o}}(Y, -\hat{\omega}_{\mathrm{i}}) \qquad (1)$$

where $\hat{\omega}_{\mathrm{i}} = (Y - X)/\|Y - X\|$.

Thus we can trivially implement the expression for the incoming light at X in two steps. First, identify point Y (as before, in the simple ray caster). Second, delegating the actual radiance estimation incident at X to a new helper method that solves for the radiance exitant at Y:

```
Radiance3 App::L_i(const Point3& X, const Vector3& wi)
{
    // Find the first intersection
    const shared_ptr<Surfel>& surfelY =
findFirstIntersection(X, wi);

    if (notNull(surfelY)) {
        // Compute the light leaving Y, which is the same as
        // the light entering X when the medium is non-absorptive
        return L_o(surfelY, -wi);
    } else {
        return Radiance3(0, 0, 0);
    }
}
```

2. Direct Illumination

The Rendering Equation tells us that the outgoing radiance at Y is the sum of emitted (L_e) and scattered radiance. The scattered radiance may have come in from any direction. In brief:

$$\underbrace{L_o}_{\text{Outgoing}} = \underbrace{L_e}_{\text{Emitted}} + \underbrace{\int L_i \cdot f \cdot |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i}_{\text{Scattered}} \qquad (2)$$

The integral presents two challenges. First, it is over a continuous domain, implying that we need to iterate over an infinite number of directions to evaluate it exactly. Second, it would create an infinite recursion were we to directly implement this relation in code, since we just implemented L_i in terms of L_o.

For this chapter, I will address both of these challenges by assuming that the only light incident at a surface arrives directly from some light source. That is, the renderer will only consider **direct illumination**. Furthermore, I will assume that all light sources are spheres with very small radii compared to their distance from the shaded surfaces. This is called a **point light** representation for the luminaires.

Considering only direct illumination means that the renderer will always underestimate the radiance at Y, and then at X. It also means that shadows will be perfectly black. For now we'll accept these limitations on the quality of the images in exchange for a straightforward implementation of shading. In the next few chapters

we'll then remove those limitations by dramatically increasing the complexity of the shading algorithm.

Under the assumptions of this chapter, the outgoing radiance is the sum of emitted radiance (which the surfel will directly provide) and the direct illumination that is scattered at Y in direction $\hat{\omega}_o$:

```
1  Radiance3 App::L_o(const shared_ptr<Surfel>& surfelX,
   const Vector3& wo) {
2      return surfelX->emittedRadiance(wo) +
   L_scatteredDirect(surfelX, wo);
3  }
```

Note that actual parameter `surfelY` is now formal parameter `surfelX` in method `L_o`. I changed the frame of reference by advancing one step closer to the light along a transport path. As in all equations so far, X is the point at which radiance is being scattered and Y is the next node closer to the light on the light transport path.

3. Scattered Radiance

The scattered direct illumination term is an estimate of

$$\int_{S^2} L_i(X, \hat{\omega}_o) f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i . \qquad (3)$$

Because direct illumination only considers light that arrived directly from a luminaire, L_i is zero for all directions except those directly towards one of the luminaires. Furthermore, the luminaires are points, so there are only a finite number of directions in which we need to evaluate the integrand; i.e., the integral is a summation. This is very good because it means that we can use a loop with a finite number of iterations to exactly compute the scattering term.

However, point lights produce *infinite* radiance L_i, since they emit finite power from a zero-area surface (over the entire 4π sr sphere), and $L = \Phi/(\|A\|\|\Gamma\|)$. By modeling an idealized yet *impossible* point emitter, I introduced an *impossible* term into our equation. If the incident radiance is infinite, will the scattered radiance also be infinite? No...because by definition of the integral $\|d\hat{\omega}_i\| = 0$ sr in the limit. So we're integrating an infinite quantity over zero steradians.

At this point, calculus cannot help us because we presented it with a ridiculous model. If we wish to proceed in taking the integral, we must somehow guarantee that that all of the limits implicit in the integral and derivative expressions are being approached at exactly the right rates to cancel one another. If they are, then we can obtain a correct

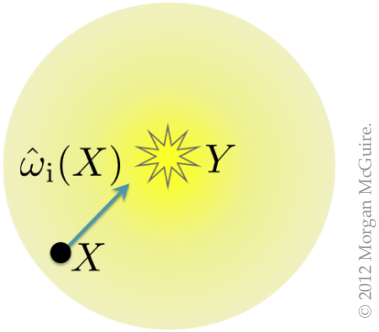
result by carefully applying the definitions of integral and derivative operations, which include limits themselves. However, this is a confusing and error-prone approach. Instead, I'll step back one level and reason about power instead of spatial and angular derivatives of power.

Biradiance

Let Φ be the total power emitted by a small sphere emitter centered at Y. This power is emitted equally in all directions. The distance from Y to X is $|Y - X|$. At that distance, the sphere surrounding the emitter has surface area $4\pi|Y - X|^2$, measured in square meters. All of the power emitted at Y passes through that sphere, so at X (and every other point at the sphere), the power-per-area due to the emitter at Y is:

$$\beta(Y, X) \in \mathrm{W/m^2} = \frac{\overset{\mathrm{W}}{\Phi}}{\underset{\mathrm{m^2}}{4\pi|Y - X|^2}} \tag{4}$$

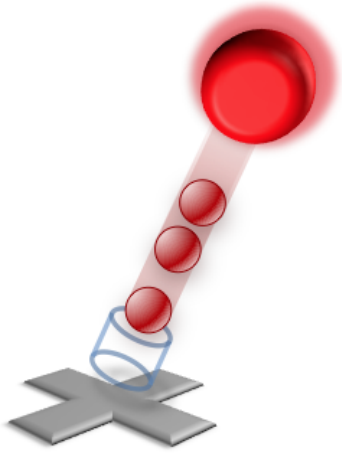
This quantity is called **biradiance**; it is the solid-angle weighted radiance at X due to the entire emitter around Y. The relationship only holds under the assumption that $|Y - X|$ is large relative to the radius of the emitter — note that β goes to infinity when the distance goes to zero.



This biradiance is exactly the quantity that we need to estimate scattered radiance. Adjusting it for projected area and applying the scattering function gives an expression for the scattered radiance due to all lights:

$$\int_{S^2} L_i(X, \hat{\omega}_i) f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i = \sum_{Y \in \text{lights}} \beta(Y, X) f_{X,\hat{n}}(S(Y - X), \hat{\omega}_o) |S(Y - X) \cdot \hat{n}| \tag{5}$$

It may help your intuition to observe that we could also construct this expression by abusing the notation and explicitly writing solid-angle weighted radiance as:



$$\beta(Y \in \mathrm{R}^3\mathrm{m}, X \in \mathrm{R}^3\mathrm{m}) \in \mathrm{W/m}^2 = \frac{\Phi}{4\pi |Y - X|^2} = \int_{\Gamma} \underbrace{L_i(X, S(Y - X))}_{\mathrm{W/(m^2\,sr)}} \underbrace{d\hat{\omega}_i}_{\mathrm{sr}} \qquad (6)$$

where Γ is the (small) solid angle subtended by the source at Y about X, and that source has bounding radius $r \ll |Y - X|$ [PP3].

Under the assumptions that the lights are relatively far from the surface and Γ is small, the quantities L_i, $|\hat{\omega}_i \cdot \hat{n}|$, and f are all constant over the domain. So, we can compute their integrals separately and then take the product.

Implementing Point Lights

At this point, we have a viable expression for scattered radiance. Iterating over each of the lights and applying it gives:

```
1  Radiance3 App::L_o(const shared_ptr<Surfel>& surfelX,
   const Vector3& wo) {
       // Begin with the emitted radiance
2      Radiance3 L = surfelX->emittedRadiance(wo);
3      const Point3& X = surfelX->position;
4      const Vector3& n = surfelX->shadingNormal;

       // Add direct illumination from each light
5      for (const shared_ptr<Light>& light : lightArray)
   {

6          const Point3& Y = light->position().xyz();

7          if (visible(X, Y)) {
8              const Vector3& wi = (Y - X).direction();
```

```
                Biradiance3& Bi = light->biradiance(X);
                const Color3& f = surfelX->finiteScatteringDensity(wi, wo);
                L += Bi * f * abs(wi.dot(n));
            }
        }

        return L;
    }
```

There are only three unimplemented functions present, `visible`, `biradiance` and `finiteScatteringDensity`. For the moment, assume that `visible` always returns true. I just derived the key expression for the biradiance function in the previous section. We do not yet have a good model of how to implement the scattering function f. You have two choices for how to implement it. One choice is to invoke `G3D::Surfel::finiteScatteringDensity` as a black box. That method implements $f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o)$ under certain limiting assumptions. Your second choice is to implement f yourself using some approximation of scattering based on observation. Let's say that you implement it yourself. One way to do this is to assume that all surfaces scatter radiance equally in all directions over the positive hemisphere. This translates to:

$$f_{X,\hat{n}}(\hat{\omega}_i, \hat{\omega}_o) = \varrho/\pi \tag{7}$$

if $\hat{\omega}_i \cdot \hat{n} > 0$ and $\hat{\omega}_o \cdot \hat{n} > 0$, and $f = 0$ otherwise. This is the scattering function of an ideally "Lambertian" matte surface with spectral reflectivity ϱ. In G3D, you can access the reflectivity of a surfel as `G3D::Surfel::reflectivity`. That method takes a random number generator because it computes the value by numerical integration of the underlying scattering function f.

You should now be able to extend the ray caster with direct illumination and emission. Of course, the result fails to consider transport paths that have more than one scattering event and treats all surfaces as matte. But for some scenes, the resulting error is acceptable. What is likely unacceptable is that there are no shadows. This is because we assumed that every surface receives direct illumination from every light (because `visible` always returned true), and this assumption clearly does not hold in most scenes.

4. Shadows

The Visibility Function

A point X is in **shadow** with respect to a small emitter at Y if any

surface lies on the line between them between them. This is formally represented in computer graphics with the binary **visibility function** $V(Y, X)$ that is zero if some surface intersects the *open* line segment XY and $V(Y, X) = 1$ otherwise.

A line segment is a subset of a ray, so the visibility function reduces to an already-solved problem: ray casting. Cast a ray from near the light at $Y + \epsilon S(X - Y)$ in direction $S(X - Y)$, (or from $X + \epsilon S(Y - X)$ in direction $S(Y - X)$). Let $V = 1$ if the first intersection found is at X (or Y if casting towards the light), otherwise $V = 0$. The ϵ value is a small distance to "bump" the ray to approximate the open line segment instead of the closed one. In your program, implement `visible` as casting the bumped ray and returning 1 or 0. Note that the ray trace can terminate after discovering *any* intersection on the line segment rather than iterating until the first intersection is discovered, since if any intersection exists the entire function is zero.

Inserting the visibility function into the complete expression yields:

$$L_o(X, \hat{\omega}_o) = L_e(X, \hat{\omega}_o) \tag{8}$$

$$+ \sum_{Y \in \text{lights}} \left[\beta(Y, X) f_{X,\hat{n}}(S(Y - X), \hat{\omega}_o) \, |S(Y - X) \cdot \hat{n}| \, V(Y, X) \right] \tag{9}$$

> ***Exercise* [light + shadow]:** Implement `visible(Y, X)` and then `L_o` for lambertian surfaces and point lights, without calling `biradiance` or `finiteScatteringDensity`.

5. Ambient Illumination

Eliminating the integral from the Rendering Equation makes it computationally trivial to evaluate but is a poor approximation. For example, the shadows created by $V(Y, X) = 0$ are perfectly black. It is common to attempt to minimize the bias of an direct illumination-only approximation by adding an estimate of the missing indirect illumination. An **ambient** term accounts for this. The ambient term may be constant or a function of the normal to account for directional variation in indirect illumination. The ambient term is obviously another approximation; it cannot capture the richness of the full incident light field.

There are two further drawbacks of the ambient term. Because the indirect illumination that it approximates depends on the content of the scene, the ambient term must be adjusted based on heuristics (or

typically, manually) based on an understanding of the scene. It is therefore an explicit approximation in the lighting environment for a scene specification instead of an approximation in the algorithm. Thus it creates a user-assisted rendering scenario instead of a robust, automatic solution.

The second drawback of the ambient term is that it should be modulated by the scattering function when computing the outgoing light at X , but doing so properly would eliminate much of the computational benefit of the approximation in the first place. One common technique is to require an explicit Lambertian reflectivity term in the scattering function and use that to modulate ambient illumination [Phong1973Thesis]. This does not work very well for glossy objects, such as a car with black paint, because it misses the glossy reflection of the environment. Another approach is to precompute the convolution of the ambient lighting environment with several canonical scattering functions and then blend these together at runtime to approximate a particular material's appearance. This is called **environment mapping** [Blinn1976Texture] for the specular reflection term in a scattering function and **irradiance mapping** for diffuse (i.e., all non-specular) terms.

References

[Blinn1976Texture]

Texture and Reflection in Computer Generated Images
James F. Blinn and Martin Newell
Communications of the ACM 19:10, p. 542-547, ACM, New York, NY, USA, October1976.
Official URL: http://doi.acm.org/10.1145/360349.360353
Free URL: http://ddm.ace.ed.ac.uk/lectures/DDM/Intro_Digital_Media/Lecture3/slides/p542-blinn.pdf
10.1145/360349.360353

[Phong1973Thesis]

Illumination for Computer-Generated Images
Bui Tuong Phong
The University of Utah, 1973.
Free URL: http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA008786

A Gentle Introduction To DirectX Raytracing

What is DirectX Raytracing?

At the 2018 Game Developer's Conference, Microsoft introduced an addition to DirectX 12 called *DirectX Raytracing* (DXR), an API allowing easy use of GPU-accelerated ray tracing in DirectX and allowing simple interoperability with traditional DirectX rasterization. Unlike most other GPU-accelerated ray tracing APIs, there are not specialized interoperability considerations to allow a rasterizer and ray tracer to efficiently share GPU resources. Instead, the same resources can easily be reused between rasterization and ray tracing.

Additionally, as a Microsoft standard, this should enable writing cross-vendor ray tracers. If your GPU drivers do not support DXR today, Microsoft has a software fallback layer that should work on Windows 10 RS4 using any GPU with Shader Model 6.0 support.

We also presented a SIGGRAPH 2018 course covering the basics in an "Introduction to DirectX Raytracing". You can find detailed slides and other resources on the course webpage.

Resources for Learning Ray Tracing

My goal is *not* to teach you ray tracing. Instead, I largely assume you understand the basics of ray tracing, e.g., from an introductory graphics text. If you are looking for resources to get you to that stage, some good options include:

- Ray Tracing In One Weekend, which has a devoted following on my Twitter feed.
- Fundamentals of Computer Graphics, a widely-used introductory graphics text.
- Realistic Ray Tracing, a text focusing on implementing a realistic ray tracer.
- Ray Tracing From The Ground Up, an introductory graphics text focusing on ray tracing.
- Graphics Codex, an online resource with bits of information about all things graphics.

Tutorial Goals

Frequently, when new APIs are released, documentation is accessible and understandable only by experts in the area. This is especially true with modern graphics APIs, like DirectX 12, which operate close to the hardware. These tutorials are ***explicitly designed*** to help more naive GPU programmers get a simple ray tracer running quickly and allow for easy experimentation.

That means we have highly abstracted the CPU-side, C++ DirectX Raytracing API. If you want to learn the specifics of the host-side API, other tutorials can help you (e.g., the Falcor DXR API tutorials or Microsoft's DXR SDK). To abstract the API, we use NVIDIA's Falcor framework, an open-source prototyping infrastructure we built at NVIDIA Research to help improve our research productivity. I added an additional syntactic sugar layer to further simplify the interface.

The tutorials build on each other, so each adds only an incremental, easy-to-grasp chunk of code. Our abstraction relies on a simple, modular render pass architecture that localizes changes and simplifies code reuse between tutorials. A resource manager abstracts creation and management of texture resources shared between these modular render passes.

After finishing the tutorials, you will be able to quickly and easily:

- Open a window using a DirectX context,
- Run raster, ray tracing, and compute passes that interact with each other,
- Load Falcor scene files (ASCII text-based and can include .obj and .fbx files),
- Generate an easily-customizable G-Buffer using either rasterization or ray tracing,
- Spawn deferred shading rays from this G-Buffer for hybrid ray-raster rendering,
- Use traditional ray tracing effects including: ambient occlusion, mirror reflections, shadow rays, and path tracing,
- Use a simple GGX material model for direct and indirect lighting,
- Easily add complex camera models introducing depth-of-field,
- Accumulate samples temporally for generating ground truth ray traced images.

Tutorial Requirements

Requirements depend on if you are using a release version of DirectX Raytracing (in Windows 10 RS5 or later) or are using the experimental version of DirectX Raytracing (in Windows 10 RS4). We encourage everyone to develop on RS5 or later, as it removes multiple sources of crypic errors.

For all DirectX Raytracing, you need:

- A 64-bit version of Windows 10. Check the version by running *winver.exe*.
    - Make sure you have version 1809 is Windows 10 RS5 or later.
    - Earlier versions cause cryptic errors, black screens, or compilation errorss.
    - *Please* double check your version of Windows. This is *the* most common issue.
- *Visual Studion 2017*. Note: the free Visual Studio Community Edition works just fine.
- A graphics card and driver that support DirectX Raytracing
    - This includes NVIDIA Pascal, Volta, and Turing cards.
    - For consumer cards NVIDIA, specifically including: GeForce GTX 1070, 1080, 1080 Ti, 1660, 2060, 2070, 2080, 2080 Ti.
    - Drivers should be 416.xx or later (use 430.86 or later for non-20xx series cards).
- *Windows 10 SDK 10.0.17763.0*. Get it here.
    - Later SDK versions may work, but you need to know how to update the SDK used in Visual Studio.

We have not had a chance (or the hardware) to test the software fallback layer. In theory, these tuorials run on GPUs without hardware-accelerated ray tracing that support DirectX 12.

Getting the Code

You can grab the tutorials:

- The code, readme, and precompiled binaries for Windows 10 RS5 and later.

Prebuilt binaries come with all dependencies included (except the GPU, driver, and OS). Please read the README in the binaries package for more details.

Code comes with most dependencies. However, we rely on the open source Falcor framework, which has a number of external dependencies. These are downloaded via a simple script which runs as a build step in Visual Studio (see the README), so your first build may take longer than usual.

Contents

1. Tutorial 1: Open a window, and clear it to a user-controllable color.

As discussed above, our goal is to provide a simple infrastructure for getting a DirectX Raytracing application up and running without digging around in low-level API specification documents. So before digging into the meat of these tutorials, we focus a little on just getting started.

These tutorials build up, step-by-step, to demonstrate a quite complex path tracer with global illumination, depth-of-field, and physically-based materials. However, the first few tutorials are fairly simple and focus on understanding the infrastructure we will take for granted in the more advanced tutorials. If you wish to jump ahead to a tutorial with actual DirectX Raytracing programming, jump ahead to Tutorial 4.

1.1. Our Infrastructure

These tutorials use Falcor, a prototypying framework I use every day in my job at NVIDIA Research. This is an open source, multi-platform wrapper around DirectX and Vulkan that is tested daily on NVIDIA, AMD, and Intel hardware, and it provides common tools needed for baseline graphics experimentation. This includes a GUI, window management, scene and asset loading, and common abstractions over graphics APIs suitable for developing graphics algorithms.

I added some additional wrappers around Falcor functionality to simplify creating a modular set of tutorials and reduce boilerplate code common in many complex graphics APIs. These wrappers are in the directory `DXRTutors\SharedUtils`, but you should not need to look there unless you need to modify or extend these tutorials in fairly complex ways.

1.2. Building a Simple Tutorial Application

If you open the project `DXRTutors\01-OpenWindow`, the main file is `Tutor01-OpenWindow.cpp`, and is just about 20 lines long.

```
#include "Falcor.h"
#include "../SharedUtils/RenderingPipeline.h"
#include "Passes/ConstantColorPass.h"

int WINAPI WinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance,
   _In_ LPSTR lpCmdLine, _In_ int nShowCmd)
{
  // Create our rendering pipeline
  RenderingPipeline *pipeline = new RenderingPipeline();

  // Add passes into our pipeline
  pipeline->setPass(0, ConstantColorPass::create());

  // Define a set of configuration for our program
  SampleConfig config;
  config.windowDesc.title =
    "Tutorial 1: Open a window and set up a simple rendering pipeline";
  config.windowDesc.resizableWindow = true;

  // Start our program!
  RenderingPipeline::run(pipeline, config);
}
```

This starts with some #includes for basic Falcor functionality, the required Falcor extensions for the DirectX Raytracing API, and the *`RenderingPipeline`* abstraction. Each of the tutorials is structured as a *`RenderingPipeline`*, which contains an arbitrarily-long sequence of *`RenderingPasses`*. A frame is rendered by executing each of the render passes in sequence.

In this simple example, there is a single pass called *`ConstantColorPass`*, which displays a constant color across the entire screen.

The *`SampleConfig`* structure contains various parameters that control the window created by Falcor. In this case, we specify a title and note that the created window should be resizable.

Finally, we start execution by calling the static method *`RenderingPipeline::run`* with our tutorial's rendering pipeline and the window configuration.

1.3. Defining a New Render Pass

Before running the program, let's look into what it takes to build a simple *`RenderPass`*. Our *`ConstantColorPass`* is located in the *`Passes\`* directory. First let's examine the boilerplate:

```
#include "../SharedUtils/RenderPass.h"

class ConstantColorPass : public RenderPass,
      inherit_shared_from_this< RenderPass, ConstantColorPass >
{
public:
  using SharedPtr = std::shared_ptr< ConstantColorPass >;

  static SharedPtr create() { return SharedPtr(new ConstantColorPass()); }
  virtual ~ConstantColorPass() = default;

protected:
  ConstantColorPass()
      : SimpleRenderPass("Constant Color Pass", "Constant Color Options") {}

  // Actual functional methods declared below here
  ...
};
```

Here, *RenderPass* is the abstract class defining the interface a render pass needs to implement, and this abstract class resides in my directory of wrapper utilities.

Falcor makes heavy use of smart pointers (e.g., based on *std::shared_ptr*), and the Falcor utility class *inherit_shared_from_this* ensures derived classes don't end up inheriting multiple copies of the internal pointers. Most Falcor classes define a *SharedPtr* type that simplifies the template notation when using smart pointers, and our wrappers keep up this tradition.

We want all *RenderPasses* to be dynamically created. We enforce this with a *create* constructor. The inputs to the *SimpleRenderPass* constructor are strings that will be used by the GUI display. The first is the name to appear in the list of passes. The second is the name of this pass' GUI window that contains any user-controllable options it exposes.

The important rendering methods and data in *ConstantColorPass* are as follows:

```
bool initialize(RenderContext::SharedPtr pRenderContext,
                ResourceManager::SharedPtr pResManager) override;
```

This method is called when the program and render pass are initialized. As part of initialization, a render pass receives a Falcor *RenderContext*, which allows you to create and access DirectX state and resources. Additionally, the *ResourceManager* is a utility class that enables sharing of resources between *RenderPasses*.

```
void renderGui(Gui* pGui) override;
```

The *renderGui* method is called when drawing the application's GUI and allows you to explicitly control placement of widgets in the GUI window for this pass. Falcor currently uses Dear Imgui for GUI rendering, and the *Falcor::Gui* class is a light wrapper around Imgui's functionality.

```
void execute(RenderContext::SharedPtr pRenderContext) override;
```

The *execute* method is invoked when rendering a frame. If your pipeline contains multiple passes, their *execute* methods are called in the order of inclusion in the pipeline.

1.4. Implementing a New Render Pass

Now that we've looked at the header for our *ConstantColorPass*, let's look at how we implement each of the three important methods:

```
bool ConstantColorPass::initialize(RenderContext::SharedPtr pRenderContext,
                                   ResourceManager::SharedPtr pResManager)
{
  // Remember our resource manager, so we can ask for resources later
  mpResManager = pResManager;

  // Tell our resource manager that we need access to the output channel
  mpResManager->requestTextureResource(ResourceManager::kOutputChannel);

  // If we return false, this pass will be removed from our pipeline.
  return true;
}
```

For our simple *ConstantColorPass*, we do two important initialization tasks:

1. Keep a copy of our resource manager, so we can access shared textures and buffers later. *mpResManager* is a member variable declared in the base class *RenderPass*.
2. Tell the resource manager that this pass requires access to the *kOutputChannel* texture. This texture is a required output from any pipeline, as it gets displayed onscreen after all *RenderPasses* execute.

```
void ConstantColorPass::renderGui(Gui* pGui)
{
  // Add a GUI widget allowing us to dynamically change the displayed color
  pGui->addFloat3Var("Color", mConstColor, 0.0f, 1.0f);
}
```

Our *renderGui* defines what widgets are available to interact with when we open the GUI window for the *ConstantColorPass*. In this case, we'll have an RGB color control (i.e., a *float3*) where all components can freely be changed between the values of *0.0f* and *1.0f*.

```
void ConstantColorPass::execute(RenderContext::SharedPtr pRenderContext)
{
  // Get a pointer to a Falcor texture resource of our output channel
  Texture::SharedPtr outTex =
      mpResManager->getTexture(ResourceManager::kOutputChannel);

  // Clear the texture to the appropriate color
  mpResManager->clearTexture(outTex, vec4(mConstColor, 1.0f));
}
```

The *execute* method is invoked when the pipeline wants us to perform our rendering tasks. For a simple pass like *ConstantColorPass*, we have exactly two goals:

1. Get a handle to the buffer that will be displayed on screen, and
2. Clear this buffer to the user-specified constant color.

To do the first task, we simply ask our resource manager for the requested texture. This only works if we called *requestTextureResource* with the corresponding texture during pass initialization. The resource manager handles creation, destruction, and sharing of all textures that any pass has requested. It also handles resizing of screen-sized buffers (all textures default to screen size unless a different size is requested).

We then call the `clearTexture()` utility method that is part of the `ResourceManager` class. This hides some DirectX details (i.e., how to clear a resource varies depending on what *resource views* it be bound with).

1.5. What Does it Look Like?

That covers the important points of this tutorial. Now if you run it, you get a result similar to this:

Result of running Tutorial 1

To open the GUI window for the `ConstantColorPass`, click the little box to the left of the text "Constant Color Pass" in the main window GUI.

Hopefully, this tutorial clarified how to set up a simple application with our framework and define a basic render pass to control what appears on screen. We will skip many of these basics and ignore the boilerpate in future tutorials.

When you are ready, continue on to Tutorial 2, where we build a *slightly* more complex raster pass with a basic HLSL shader.

2. Tutorial 2: Adding a basic full-screen rasterization pass.

As discussed in our tutorial introduction, our goal is to provide a simple infrastructure for getting a DirectX Raytracing application up and running without digging around in low-level API specification documents. Tutorial 2 continues with our sequence covering some infrastructure basics before we get to the meat of implementing a path tracer. If you wish to move on to a tutorial with actual DirectX Raytracing programming, jump ahead to Tutorial 4.

2.1. Adding a More Complex RenderPass

Tutorial 1 showed you how to get a basic window open and inset a simplistic `RenderPass` to clear your screen to a user-controllable color. This tutorial shows how to setup and use a more complex rasterization pass with a programmable HLSL shader.

The specific shader is not particularly important, rather the point of this tutorial is to demonstrate the wrappers we have for encapsulating programmable shaders of various kinds and how you interact and pass parameters to these shaders from your C++ based *RenderPasses*.

If you open up *Tutor02-SimpleRasterShader.cpp*, you will find it looks remarkably similar to the main program from Tutorial 1. The key difference is in what *RenderPass* we add to our pipeline:

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, SinusoidRasterPass::create());
```

In this tutorial, we simply swap out the *ConstantColorPass* for our more complex *SinusoidRasterPass*. The key files to look at are then:

- *Passes\SinusoidRasterPass.h*
- *Passes\SinusoidRasterPass.cpp*
- *Data\Tutorial02\sinusoid.ps.hlsl*

Looking at the *SinusoidRasterPass.h* header, you will largely see the same class declaration boilerplate from the *ConstantColorPass*, and in fact the only differences are in the member variables:

```
FullscreenLaunch::SharedPtr   mpSinusoidPass;
GraphicsState::SharedPtr      mpGfxState;
uint32_t                      mFrameCount = 0;
float                         mScaleValue = 0.1f;
```

For now ignore *mFrameCount* and *mScaleValue*, which are parameters to control the image displayed by our shader.

The *GraphicsState* is a Falcor class that wraps up the pipeline state of a DirectX rasterization pipeline. This includes things like culling, depth testing, blending, rasterization parameters, etc. For our ray tracing tutorials, we only need a simple *default* graphics state, so the key to remember is we need *some* graphics state to setup a DirectX pipeline. Don't worry too much about what settings it controls.

The *FullscreenLaunch* class is an abstraction in the *SharedUtils* directory that easily allows you to launch very simple full-screen rasterization draw calls. Similar abstractions, *RasterLaunch* and *RayLaunch*, exist for drawing more complex raster and ray calls using loaded scene geometry. We'll use those in future tutorials, and they behave quite similar to our *FullscreenLaunch*.

2.2. Defining the SinusoidRasterPass

As in the case of our simpler *ConstantColorPass*, we need to define the *initialize*, *renderGui*, and *execute* methods. Looking in *SinusoidRasterPass.cpp*, the *renderGui* should be self explanatory:

```
pGui->addFloatVar()"Sin multiplier", mScaleValue, 0.0f, 1.0f, 0.00001f);
```

It simply adds a GUI widget that allows control over a single floating point value, allowing variations in the range [0...1] in increments of 0.00001.

Our *initialize* pass adds two new lines that were not part of *ConstantColorPass*:

```
mpGfxState = GraphicsState::create();
mpSinusoidPass = FullscreenLaunch::create( "Tutorial02\\sinusoid.ps.hlsl" );
```

The first creates a default DirectX raster pipeline state we'll use when rendering. The second line says we'll be doing a full-screen draw call using the specified HLSL shader. Falcor's full-screen pass uses a default vertex shader, so you only need to specify a pixel shader.

The *execute* method is more complex, so let's step through the entire function:

```
void SinusoidRasterPass::execute(RenderContext::SharedPtr pRenderContext)
{
  // Create a framebuffer object to render into.
  Fbo::SharedPtr outputFbo =
      mpResManager->createManagedFbo({ ResourceManager::kOutputChannel });
  mpGfxState->setFbo(outputFbo);

  // Set shader parameters for our shader
  auto shaderVars = mpSinusoidPass->getVars();
  shaderVars["PerFrameCB"]["gFrameCount"] = mFrameCount++;
  shaderVars["PerFrameCB"]["gMultValue"]  = mScaleValue;

  // Execute our shader
  mpSinusoidPass->execute(pRenderContext, mpGfxState);
}
```

In this case, we're executing a raster shader. This means we need a framebuffer to draw into, and we can't simply clear our texture (as we did in *ConstantColorPass*).

Our first line asks our *ResourceManager* to create a framebuffer for us, and to bind the *kOutputChannel* as the framebuffer object's color channel. Ideally, you would not create a new framebuffer each frame, but this is left as an exercise for more advanced readers. We then update our DirectX pipeline state to send results to this new FBO when we run our shader.

The next three lines set shader parameters in the *sinusoid.ps.hlsl* shader. The first line gets an object that can access and transfer data to the variables in your shader. The next two lines have very simple syntax: on the left of the assignment operator you specify the variable in your shader, on the right is the value you want to transfer to HLSL. The assignment operator is overloaded and (in a Debug build) will do some basic type checking (at runtime) to ensure the value is the right type to assign to the specified HLSL variable.

The syntax *shaderVars["PerFrameCB"]["gMultValue"] = mScaleValue* reads as: in the specified shader, find the variable *gMultValue* in the constant buffer *PerFrameCB* and set the value to *mScaleValue*. This is a very simple and clear assignment between CPU and GPU variables. It is not the most efficient way to update shader values, but for the purposes of these tutorials (and most of my research prototypes), this overhead is not significant enough to favor more efficient, but cryptic, data transfers.

Finally, we execute our full-screen shader. This requires two parameters: the current DirectX render context and the graphics state to use for rendering.

2.3. Interacting with the Sinusoidal HLSL Shader

The final step to understanding this tutorial is to look at our GPU shader in *sinusoid.ps.hlsl*:

```
// A constant buffer of shader parameters populated from our C++ code
cbuffer PerFrameCB
{
  uint gFrameCount;
  float gMultValue;
}

float4 main(float2 texC : TEXCOORD, float4 pos : SV_Position) : SV_Target0
{
  // Get integer screen coordinates (e.g., in [0..1920] x [0..1080])
  uint2 pixelPos = (uint2)pos.xy;

  // Compute a per-pixel sinusoidal value
  float sinusoid = 0.5 * (1.0f + sin( 0.001f * gMultValue *
                                      dot(pixelPos, pixelPos) +
                                      gFrameCount ));

  // Save our color out to our framebuffer
  return float4(sinusoid, sinusoid, sinusoid, 1.0f);
}
```

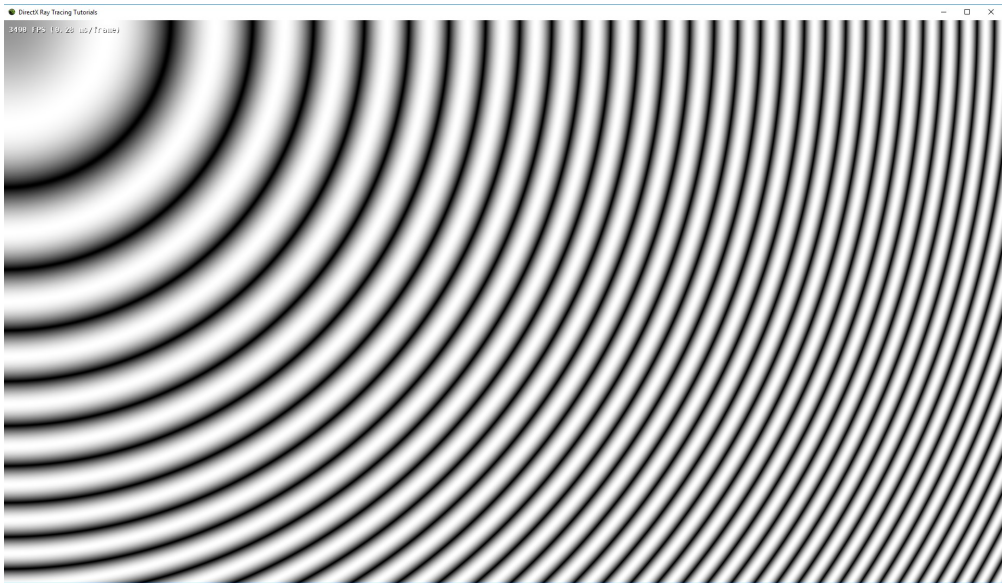
The first couple lines define our constant buffer with the parameters we set from the *`execute`* method of our *`SinusoidRenderPass`*.

We then define our shader *`main`* that takes two variables (*`texC`* and *`pos`*) from Falcor's default vertex shader and writes out to our framebuffer's color buffer 0 (i.e., *`SV_Target0`*).

The math is fairly unimportant, but generates a grayscale sinuosoid that slowly changes over time (since *`gFrameCount`* gets incremented each frame) and whose scale is controllable by the *`gMultValue`* variable.

2.4. What Does it Look Like?

That covers the important points of this tutorial. Now if you run it, you get a result similar to this:



Result of running Tutorial 2

Hopefully, this tutorial demonstrated how to use a very basic raster shader and send it parameters from

our simple RenderPass architecture.

When you are ready, continue on to Tutorial 3, where we use a more traditional rasterization pass to generate a G-Buffer and allow the user to selectively display different G-Buffer parameters.

3. Tutorial 3: Creating a basic, rasterized G-buffer

As discussed in our tutorial introduction, our goal is to provide a simple infrastructure for getting a DirectX Raytracing application up and running without digging around in low-level API specification documents. Tutorial 3 continues with our sequence covering some infrastructure basics before we get to the meat of implementing a path tracer. If you wish to move on to a tutorial with actual DirectX Raytracing programming, jump ahead to Tutorial 4.

3.1. Why Create a G-Buffer?

Tutorial 2 showed you how to use a more complex `RenderPass` to launch a simple HLSL pixel shader. Before moving on to actually using ray tracing in Tutorial 4, we'll walk through how to interact with Falcor-loaded scene files and create a set of traditional vertex and pixel shaders that run over this geometry during rasterization.

The shaders we use to demonstrate this will create a G-Buffer that we can use to accelerate ray tracing in later tutorials. In fact, Tutorial 5 uses a *hybrid renderer* that rasterizes primary visibility and only uses DirectX Raytracing to shoot shadow rays.

As an additional benefit, in order to extract the data to populate our G-buffer, we walk through various Falcor shader utilities that allow you to access scene properties like textures and materials.

3.2. A More Complex Rendering Pipeline

If you open up `Tutor03-RasterGBuffer.cpp`, you will find a slightly more complex main program that defines the following `RenderingPipeline`:

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, SimpleGBufferPass::create());
pipeline->setPass(1, CopyToOutputPass::create());
```

Now, there are *two* render passes: `SimpleGBufferPass` and `CopyToOutputPass`. For every frame, these are executed in sequence. First `SimpleGBufferPass` is executed, and it stores its output in textures managed by our `ResourceManager`. This allows subsequent passes, like `CopyToOutputPass` to access and reuse these intermediate results. This structure allows us to build modular and reusable code. In fact, we'll reuse the `SimpleGBufferPass` we write here in Tutorial 5, without modification.

In this particular tutorial, `SimpleGBufferPass` creates a G-Buffer containing each pixel's position, surface normal, diffuse color, specular color, and z-buffer. `CopyToOutputPass` simply allows the user to select, via the GUI, which of those outputs to show and then copies the appropriate buffer to the `kOutputChannel` to display.

3.3. Handling the Falcor Scene and Launching Rasterization

Start by looking in *SimpleGBufferPass.h*. This should look familiar, as the boilerplate is nearly identical to that from the *RenderPasses* we wrote in Tutorials 1 and 2. The major difference is in our pass' member variables:

```
GraphicsState::SharedPtr    mpGfxState;  // DirectX state
Scene::SharedPtr            mpScene;     // Falcor scene abstraction
RasterLaunch::SharedPtr     mpRaster;    // Encapsulates rasterization pass
```

As in Tutorial 2, the *GraphicsState* class encapsulates various DirectX rendering state like the depth, rasterization, blending, and culling settings. The *Scene* class encapsultes Falcor's scene representation. It has a variety of accessor methods to provide access the cameras, lights, geometry, and other details. For these tutorials, we will mostly pass *mpScene* into our rendering wrappers and let Falcor automatically send the data to the GPU.

The *RasterLaunch* is similar to the *FullscreenLaunch* class from Tutorial 2, except it encapsulates state for rasterizing complex scene geometry (rather than a screen-aligned quad).

3.4. Initializing our G-Buffer Pass

Our *SimpleGBufferPass::initialize()* method is slightly more complex that in our prior passes:

```
bool SimpleGBufferPass::initialize(RenderContext::SharedPtr pRenderContext,
                                   ResourceManager::SharedPtr pResManager)
{
  // Stash a copy of our resource manager for later
  mpResManager = pResManager;

  // Tell our resource manager that we expect to write these channels
  mpResManager->requestTextureResource("WorldPosition");
  mpResManager->requestTextureResource("WorldNormal");
  mpResManager->requestTextureResource("MaterialDiffuse");
  mpResManager->requestTextureResource("MaterialSpecRough");
  mpResManager->requestTextureResource("MaterialExtraParams");
  mpResManager->requestTextureResource("Z-Buffer",
                                       ResourceFormat::D24UnormS8,
                                       ResourceManager::kDepthBufferFlags);

  // Define our raster pipeline state (though we use the defaults)
  mpGfxState = GraphicsState::create();

  // Create our wrapper for a scene rasterization pass.
  mpRaster = RasterLaunch::createFromFiles("gBuffer.vs.hlsl",
                                           "gBuffer.ps.hlsl");
  mpRaster->setScene(mpScene);
  return true;
}
```

There's a couple of important things to note here:

- We no longer write to *kOutputChannel*. Thus, *SimpleGBufferPass* does not form a complete rendering pipeline. If no subsequent pass uses our intermediate results to write to *kOutputChannel*, nothing will appear on screen!
- When requesting buffers, the names are unimportant. If a subsequent pass requests access to a buffer with the same name, it will be shared.
- For the *Z-Buffer*, we use a more complex *requestTextureResource()* call. The second

parameter specifies the resource format (using a 24 bit depth and 8 bit stencil). We also need to specify how it can be bound, since DirectX depth buffers have different limitations. The constant *kDepthBufferFlags* stores good defaults for a depth buffer.

- When not using the more complex request, buffers default to RGBA textures using 32-bit floats for each channel. The bind flags default to *kDefaultFlags*, which provide good defaults for all textures that are not used for depth or stencil.

We then create our raster wrapper *mpRaster* by pointing it to our vertex and pixel shaders. For our *RasterLaunch*, it needs to know what scene to use. In case our scene has already been loaded prior to initialization, we pass the scene into our wrapper.

3.5. Handling Scene Loading

Our tutorial application automatically adds a GUI button to allow users to open a scene file. When Falcor loads a scene, all passes have the option to process it by overriding the *RenderPass::initScene()* method:

```
void SimpleGBufferPass::initScene(RenderContext::SharedPtr pRenderContext,
                                  Scene::SharedPtr pScene)
{
  // Stash a copy of the scene
  mpScene = pScene;

  // Update our raster pass wrapper with this scene
  if (mpRaster)
      mpRaster->setScene(mpScene);
}
```

For our G-buffer class, this is very simple:

- Store a copy of the scene pointer so we can access it later.
- Tell our raster pass that we're using a new scene.

3.6. Launching our G-Buffer Rasterization pass

Now that we initialized our rendering resources and loaded our scene file, we can launch our G-buffer rasterization.

```
void SimpleGBufferPass::execute(RenderContext::SharedPtr pRenderContext)
{
  // Create a framebuffer for rendering.  (Should avoid doing each frame)
  Fbo::SharedPtr outputFbo = mpResManager->createManagedFbo(
      { "WorldPosition", "WorldNormal", "MaterialDiffuse",
        "MaterialSpecRough", "MaterialExtraParams" },
      "Z-Buffer" );

  // Clear all color buffers to (0,0,0,0), depth to 1, stencil to 0
  pRenderContext->clearFbo(outputFbo.get(), vec4(0, 0, 0, 0), 1.0f, 0);

  // Rasterize!  Note: Falcor will populate many built-in shader variables
  mpRaster->execute(pRenderContext, mpGfxState, outputFbo);
}
```

First, we need a framebuffer to write the results of our rendering pass. As in Tutorial 2, we call

createManagedFbo(), albeit with a more complex set of parameters. Again, this creation should not occur once per frame for performance reasons, though here we do for simplicity and clarity.

When calling *createManagedFbo*, the first parameter is a list of names of resources managed by our *ResourceManager*. (Note that these buffers were all requested during initialization.) These will be the color buffers in our framebuffer, and are bound in the order specified (so *"WorldPosition"* is *SV_Target0* in our DirectX shader and *"MaterialExtraParams"* is *SV_Target4*). The second parameter is the name of the resource to bind as a depth texture.

We then clear this newly created framebuffer using a Falcor built-in. This method clears all 5 color buffers to black, clears the depth buffer to 1.0f and the stencil buffer to 0.

Finally, we launch our rasterization pass. *execute()* requres the DirectX context, the DirectX graphics state to use, and the framebuffer to store results.

3.7. The DirectX HLSL for Our G-Buffer Rasterization

Our vertex shader appears somewhat cryptic, since we use Falcor utility functions to access the scene data and pass it to our pixel shader appropriately:

```
// ---- gBuffer.vs.hlsl ----
#include "VertexAttrib.h"
__import ShaderCommon;
__import DefaultVS;

VertexOut main(VertexIn vIn)
{
  return defaultVS(vIn);
}
```

Falcor has a default vertex shader called *defaultVS* that we can use after the *__import DefaultVS;* (The code is in the file *DefaultVS.slang*.) This default shader accesses standard scene attributes (see *VertexAttrib.h*), applies appropriate viewing, animation, and camera matrices, and stores the results into a *VertexOut* structure (which is also defined in *DefaultVS.slang*). Note that the *__import* lines are not standard HLSL, but rather invoke our framework's shader preprocessor / special-purpose compiler, Slang.

Fundamentally, this is a very simple vertex shader that applies a few matrices to the vertex positions and normals, but the default shader gracefully handles different scenes geometry that may or may not have any combination of: normals, bitangents, texture coordinates, lightmaps, geometric skinning, plus a few other advanced features.

The more interesting shader, our pixel shader, follows:

```
// ---- gBuffer.ps.hlsl ----
__import Shading;     // To get ShadingData structure & shading helper funcs
__import DefaultVS;   // To get the VertexOut declaration

struct GBuffer
{
    float4 wsPos    : SV_Target0;  // Specific bindings here determined by the
    float4 wsNorm   : SV_Target1;  //     order of buffers in the call to
    float4 matDif   : SV_Target2;  //     createManagedFbo() in our method
    float4 matSpec  : SV_Target3;  //     SimpleGBufferPass::execute()
    float4 matExtra : SV_Target4;
};

GBuffer main(VertexOut vsOut, float4 pos: SV_Position)
{
    // A Falcor built-in to extract geometry and material data suitable for
    //    shading.  (See ShaderCommon.slang for the structure and routines)
    ShadingData hitPt = prepareShadingData(vsOut, gMaterial, gCamera.posW);

    // Dump out our G buffer channels
    GBuffer gBufOut;
    gBufOut.wsPos    = float4(hitPt.posW, 1.f);
    gBufOut.wsNorm   = float4(hitPt.N, length(hitPt.posW - gCamera.posW) );
    gBufOut.matDif   = float4(hitPt.diffuse, hitPt.opacity);
    gBufOut.matSpec  = float4(hitPt.specular, hitPt.linearRoughness);
    gBufOut.matExtra = float4(hitPt.IoR,
                              hitPt.doubleSidedMaterial ? 1.f : 0.f,
                              0.f, 0.f);
    return gBufOut;
}
```

The first couple lines include Falcor built-ins by asking our Slang shader preprocessor to import various common definitions and functions.

We then declare the structure of our output framebuffer's render targets. This needs to match what we specified in the C++ code (when we created our framebuffer via *`createManagedFbo`*).

Finally, we define our *`main`* routine for our pixel shader to take in the *`VertexOut`* structure from our vertex shader and outputs to the framebuffer of the approriate format. We start by calling a Falcor built-in that uses our interploated geometry attributes, the scene's materials (in the Falcor-defined shader variable *`gMaterial`*) and the current camera position (in Falcor variable *`gCamera`*) to extract commonly used data needed for shading. We then store some of this data out into our G-buffer:

1. Our pixel's world-space position *`hitPt.posW`*
2. Our pixel's world-space normal *`hitPt.N`* and distance from fragment to the camera.
3. Our diffuse material (including texture) color and alpha value.
4. Our specular reflectance and surface roughness.
5. A miscellanous buffer containing index-of-refraction and a flag determining if the surface should be considered double-sided when shading.

This is an extremely verbose G-buffer, using five 128-bit buffers, which is significantly more than most people would consider reasonable. However, this is done for simplicity and clarity. It should be straightforward to compress this data into a more compact format.

3.8. Implementing our CopyToOutputPass

As noted above, our *SimpleGBufferPass* does not write to the shared resource *kOutputChannel*, so another pass is required to generate an image. Our *CopyToOutputPass* executes the following code when it renders:

```
void CopyToOutputPass::execute(RenderContext::SharedPtr pRenderContext)
{
  // Get the Falcor texture we're copying and our output buffer
  Texture::SharedPtr inTex  = mpResManager->getTexture(mSelectedBuffer);
  Texture::SharedPtr outTex = mpResManager->getClearedTexture(
                                        ResourceManager::kOutputChannel,
                                        vec4(0.0, 0.0, 0.0, 0.0));

  // If we selected an invalid texture, return.
  if (!inTex) return;

  // Copy the selected input buffer to our output buffer.
  pRenderContext->blit( inTex->getSRV(), outTex->getRTV() );
}
```

From the *ResourceManager* we get the texture the user requested and our output buffer. Here we use the method *getClearedTexture()* to first clear the output to black before returning it.

If our input texture is valid, we then copy the input to the output using the Falcor built-in *blit()*. (Blit is an older graphics term that often means *copy*, or more specifically *block image transfer*.)

We allow the user to select *mSelectedBuffer* via a GUI dropdown widget from the list of options in *mDisplayableBuffers*:

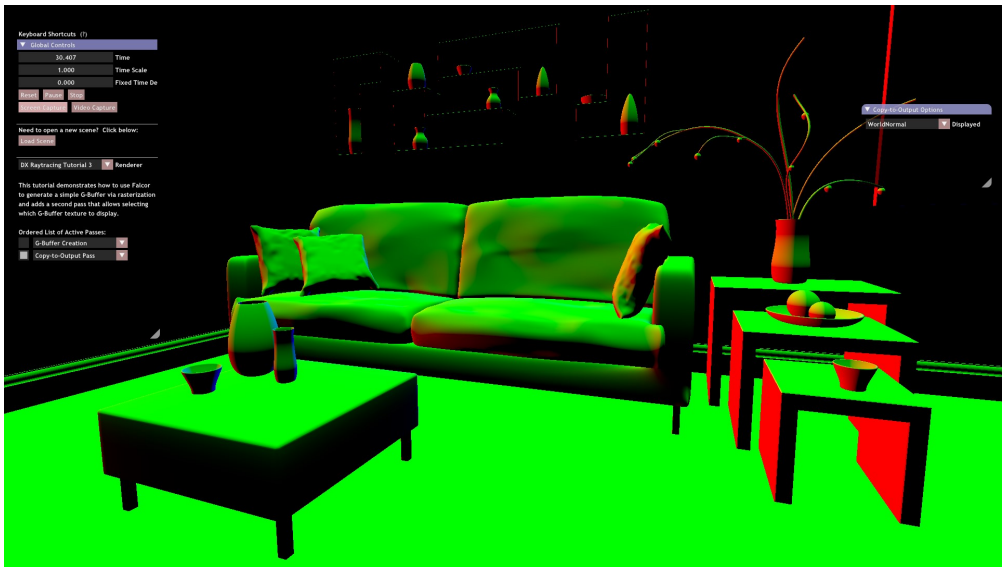
```
void CopyToOutputPass::renderGui(Gui* pGui)
{
  pGui->addDropdown("Displayed", mDisplayableBuffers, mSelectedBuffer);
}
```

A new method in *CopyToOutputPass* is *pipelineUpdated()*, which gets called whenever passes get added or removed from our *RenderingPipeline*. The idea here is to create a GUI dropdown list containing all possible textures in the resource manager that we can display:

```
void CopyToOutputPass::pipelineUpdated(ResourceManager::SharedPtr pResManager)
{
  // If our resource manager changed, stash the new pointer
  mpResManager = pResManager;

  // Clear our GUI list
  mDisplayableBuffers.clear();

  // Look through all available texture resources
  for (uint32_t i = 0; i < mpResManager->getTextureCount(); i++)
  {
     // If this one isn't the output buffer, add it to the displayables list
     if (i == mpResManager->getTextureIndex(ResourceManager::kOutputChannel))
        continue;
     mDisplayableBuffers.push_back({int(i), mpResManager->getTextureName(i)});
  }
}
```

3.9. What Does it Look Like?

That covers the important points of this tutorial. Now if you run it, you get a result similar to this:



Result of running Tutorial 3, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated:

- How to build pipelines of multiple `RenderPasses` that share resources.
- How to access Falcor scenes inside your render passes.
- How to rasterize these scenes using fairly basic HLSL vertex and fragment shaders.

When you are ready, continue on to Tutorial 4, where we finally learn how to spawn rays using DirectX Raytracing.

4. Tutorial 4: Create the same G-buffer, this time using DirectX ray tracing

Our first three tutorials focused on getting up to speed with our simple tutorial infrastructure and making simple DirectX rasterization calls (to create a basic G-Buffer). This tutorial finally moves on to actually spawning some ray tracing with DirectX. To focus on how ray tracing and rasterization are different, this tutorial builds a G-Buffer identical to the one created in Tutorial 3.

4.1. Our Basic Ray Tracing Render Pipeline

If you open up `Tutor04-RayTracedGBuffer.cpp`, you will find the only change is we swaped out `SimpleGBufferPass` for the new `RayTracedGBufferPass` we'll build below. We reuse the same `CopyToOutputPass` we created in Tutorial 3.

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, RayTracedGBufferPass::create());
pipeline->setPass(1, CopyToOutputPass::create());
```

For those who skipped the first three tutorials, this code can be read as "create a rendering pipeline with two components: a pass that generates a ray-traced G-buffer followed by a pass that copies a selectable buffer onscreen."

4.2. Creating and Launching DirectX Raytracing Work

Start by looking in *RayTracedGBufferPass.h*. This should look familiar, as the boilerplate is nearly identical to that from the *RenderPasses* from prior tutorials. The major difference is in our pass' member variables:

```
RtScene::SharedPtr      mpScene;     // Falcor scene abstraction
RayLaunch::SharedPtr    mpRays;      // Encapsulates DirectX Raytracing pass
```

The *RtScene* class encapsultes Falcor's scene representation, with additions for ray tracing. In the rest of our tutorials, we will use the *RtScene* class (which derives from the *Scene* class from Tutorial 3 but adds functionality required for ray tracing).

The *RayLaunch* is similar to the *RasterLaunch* class from Tutorial 3, except it exposes methods for setting up all the new HLSL ray tracing shader types and setting their variables.

4.3. Initializing our Ray Traced G-Buffer Pass

Our *RayTracedGBufferPass::initialize()* looks quite similar to the one for *SimpleGBufferPass*:

```
bool RayTracedGBufferPass::initialize(RenderContext::SharedPtr pRenderContext,
                                      ResourceManager::SharedPtr pResManager)
{
  // Stash a copy of our resource manager; tell it what shared texture
  //     resources we need for this pass.
  mpResManager = pResManager;
  mpResManager->requestTextureResources({"WorldPosition", "WorldNormal",
                                         "MaterialDiffuse", "MaterialSpecRough",
                                         "MaterialExtraParams"});

  // Create our wrapper for our DirectX Raytracing launch.
  mpRays = RayLaunch::create("rtGBuffer.hlsl", "GBufferRayGen");
  mpRays->addMissShader("rtGBuffer.hlsl", "PrimaryMiss");
  mpRays->addHitShader("rtGBuffer.hlsl", "PrimaryClosestHit", "PrimaryAnyHit");

  // Compile our shaders and pass in our scene
  mpRays->compileRayProgram();
  mpRays->setScene(mpScene);
  return true;
}
```

A couple important notes:

- The method *requestTextureResources* is identical to calling *requiresTextureResource* multiple times, with each of the specified names, simplifying the code somewhat.
- We don't request a Z-buffer, because ray tracing does not generate one by default. (Of course, if desired you can output a Z-buffer manually.)

We then create our ray tracing wrapper *mpRays* by pointing it to our shader file *rtGBuffer.hlsl* and specifying the function where our *ray generation shader* starts, in this case the function *GBufferRayGen()* in our HLSL file. Because we plan to launch rays from this shader, we also have to specify the *miss shader* to use, named *PrimaryMiss()*, and our *closest-hit* and *any hit shaders*, named *PrimaryClosestHit()* and *PrimaryAnyHit()*.

When ray tracing, we can have multiple ray types. This means we can call *addMissShader* multiple times. The first time it is called, we specify miss shader #0. The second time, we specify miss shader #1. The next, miss shader #2. Et cetera. Similarly, when called multiple times, *addHitShader* specifies hit group #0, 1, 2, etc. This integer identifier is important, as calling *TraceRay()* in HLSL requires you specify the correct IDs.

4.4. Launching Ray Tracing and Sending Data to HLSL

Now that we initialized our rendering resources, we can create our ray traced G-buffer. The *RayTracedGBufferPass::execute* pass is somewhat complex, so let's split it into multiple pieces:

```
void RayTracedGBufferPass::execute(RenderContext::SharedPtr pRenderContext)
{
  // Color used to clear our G-buffer
  vec4 black    = vec4(0, 0, 0, 0);

  // Load our textures; ask the resource manager to clear them first
  //    Note:  'auto' type used for brevity; actually Texture::SharedPtr
  auto wsPos    = mpResManager->getClearedTexture("WorldPosition",       black);
  auto wsNorm   = mpResManager->getClearedTexture("WorldNormal",         black);
  auto matDif   = mpResManager->getClearedTexture("MaterialDiffuse",     black);
  auto matSpec  = mpResManager->getClearedTexture("MaterialSpecRough",   black);
  auto matExtra = mpResManager->getClearedTexture("MaterialExtraParams", black);
  auto matEmit  = mpResManager->getClearedTexture("Emissive",            black);
```

First, we need access the textures we'll use to write out our G-buffer. Unlike in Tutorials 2 and 3, we do not need to create a framebuffer object. Instead for ray tracing, we'll write directly into them (using UAVs). We also ask our resource manager to clear them to black prior to returning the textures.

```
  // Pass our background color down to miss shader #0
  auto missVars = mpRays->getMissVars(0);
  missVars["MissShaderCB"]["gBgColor"] = mBgColor;  // Color for background
  missVars["gMatDif"] = matDif;                     // Where to store bg color
```

Next, we set the HLSL variables for our *miss shader #0*. These variables are scoped so they are *only* usable while executing this specific miss shader. This uses the same syntax we used for raster shaders, except we need to specifically ask (*getMissVars(0)*) for variables for miss shader #0. In this case, if our rays miss we'll store the specified background color in the *"MaterialDiffuse"* texture.

```
  // Pass down variables for our hit group #0
  for (auto pVars : mpRays->getHitVars(0))
  {
    pVars["gWsPos"] = wsPos;
    pVars["gWsNorm"] = wsNorm;
    pVars["gMatDif"] = matDif;
    pVars["gMatSpec"] = matSpec;
    pVars["gMatExtra"] = matExtra;
  }
```

In this snippet, we are setting the HLSL variables for hit group #0 (i.e, the *closest-hit* and *any-hit* shaders). This code is more complex than for miss shaders, since your variables may change for each geometry instance in your scene. This is the cause of the *for* loop, as we loop over all the instances. In this case, the variables specifed are scoped so they are only visibile by the closest-hit and any-hit shaders executed while intersecting the specified geometry instance.

Above, we are specifying the output textures *gWsPos*, *gWsNorm*, *gMatDif*, *gMatSpec*, and *gMatExtra*. We will store our G=Buffer output in our *closest-hit* shader, which can thought of as similar to a pixel shader.

```
    // Launch our ray tracing
    mpRays->execute( pRenderContext, mpResManager->getScreenSize() );
}
```

Finally, we launch our ray tracing pass. *execute()* requres the DirectX context and the number of rays to launch (often dependant on screen resolution).

4.5. The DirectX Raytracing HLSL for Our G-Buffer

Just like when starting to read a C++ program it often makes sense to start reading at *main()*, you should probably start reading HLSL ray tracing code at the ray generation shader. As specified above during our pass initialization, our shader's ray generation shader is named *GBufferRayGen()*:

```
[shader("raygeneration")]
void GBufferRayGen()
{
    // Convert our ray index into a ray direction in world space.
    float2 currenPixelLocation = DispatchRaysIndex() + float2(0.5f, 0.5f);
    float2 pixelCenter = currenPixelLocation / DispatchRaysDimensions();
    float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 rayDir = normalize( ndc.x * gCamera.cameraU +
                               ndc.y * gCamera.cameraV +
                                       gCamera.cameraW );

    // Initialize a ray structure for our ray tracer
    RayDesc ray = { gCamera.posW, 0.0f, rayDir, 1e+38f };

    // Initialize our ray payload (a per-ray, user-definable structure).
    SimplePayload payload = { false };

    // Trace our ray
    TraceRay(gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );
}
```

The first few lines lookup the current pixel location. The number of rays launched (i.e., the parmaeter passed to *mpRays->execute*) is accessible via HLSL's *DispatchRayDimensions()* intrinsic, and the current pixel being processed is given by *DispatchRaysIndex()*. After converting the pixel location into a value in [–1...1], we use the current camera parameters to generate a world-space vector that describes the ray from the camera through this pixel. The camera variable *gCamera* is automatically passed down by our framework.

We then create a ray using the DirectX data type *RayDesc*. The first entry in the structure (named *Origin*) specifies where the ray starts. The second entry (named *TMin*) specifies the minimum distance we need to travel along our ray before reporting a valid hit. The third entry (named *Direction*) specifies the ray directions. The fourth entry (named *TMax*) specifies the maximum distance along our ray to return a valid hit.

When tracing rays in DirectX, there is a user-definable *payload* structure that accompanies each ray and allows you to stash temporary data during ray tracing. In this example shader this payload is unused, so is not particularly important.

Finally, we call *TraceRay()* to launch our ray with the following parameters:

- The first parameter, *gRtScene*, is the ray acceleration structure. Our framework populates this HLSL variable for you automatically.
- Second, we specify flags to control ray behavior. Here, we use no special flags.
- Third, we specify an instance mask. This allows you to skip some instances during traversal. A value *0xFF* tests all geometry in the scene (skipping none).
- Fourth, we specify which hit group to use. In the DXR spec, this is a somewhat complex computation. For most simple usage, this corresponds directly to the ID of your hit group, as determined in *RayTracedGBufferPass::initialize*. (In our case, there is only one hit group so this takes a value of 0).
- Fifth, we specify how many hit groups there are. In our case this is 1. (Our framework computes this and stores it in the varaible *hitProgramCount*, if you wish to avoid hard coding this value.)
- Sixth, we specify which miss shader to use. This directly corresponds to the miss shader ID determined in *RayTracedGBufferPass::initialize*.
- Seventh, we specify the ray to trace.
- Finally, we specify the payload data structure to use while tracing this ray.

After calling *TraceRay()*, some processing happens internally. The next shader executed depends on the geometry, ray direction, and internal data structures. *Any-hit shaders* get launched for some potential hits, *closest-hit shaders* get launched when you have identified the single closest hit along the ray, and *miss shaders* get launched if no valid hit occurs.

```
// A dummy payload for this simple ray; never used
struct SimplePayload {
    bool dummyValue;
};

// Our miss shader's variables
cbuffer MissShaderCB {
    float3  gBgColor;
};

// The output textures.  See bindings in C++ code.
//     -> gMatDif is visible in the miss shader
//     -> All textures are visible in the hit shaders
RWTexture2D<float4> gWsPos, gWsNorm, gMatDif, gMatSpec, gMatExtra;

[shader("miss")]
void PrimaryMiss(inout SimplePayload)
{
   gMatDif[DispatchRaysIndex()] = float4( gBgColor, 1.0f );
}
```

This miss shader is fairly straightforward. When a ray misses all geometry, write out the background color into the specified texture. Where in the texture do we write? That depends on the current pixel location we are working on (using the same *DispatchRaysIndex()* intrinsic we used in our ray generation shader).

```
[shader("anyhit")]
void PrimaryAnyHit(inout SimpleRayPayload, BuiltinIntersectionAttribs attribs)
{
   if (alphaTestFails(attribs)) IgnoreHit();
}
```

Our any hit shader is also straightforward. In most cases, when ray traversal detects a hit point, it is a valid intersection. However, if the geometry uses alpha-testing and has a textured alpha mask, we need to query it before confirming the intersection. Here we call a utility function *alphaTestFails()* (see tutorial code for specifics) that loads the alpha texture from Falcor's internal scene structure and determines if we hit a transparent texel. If so, this hit is ignored via the *IgnoreHit()* intrinsic, otherwise the hit is accepted.

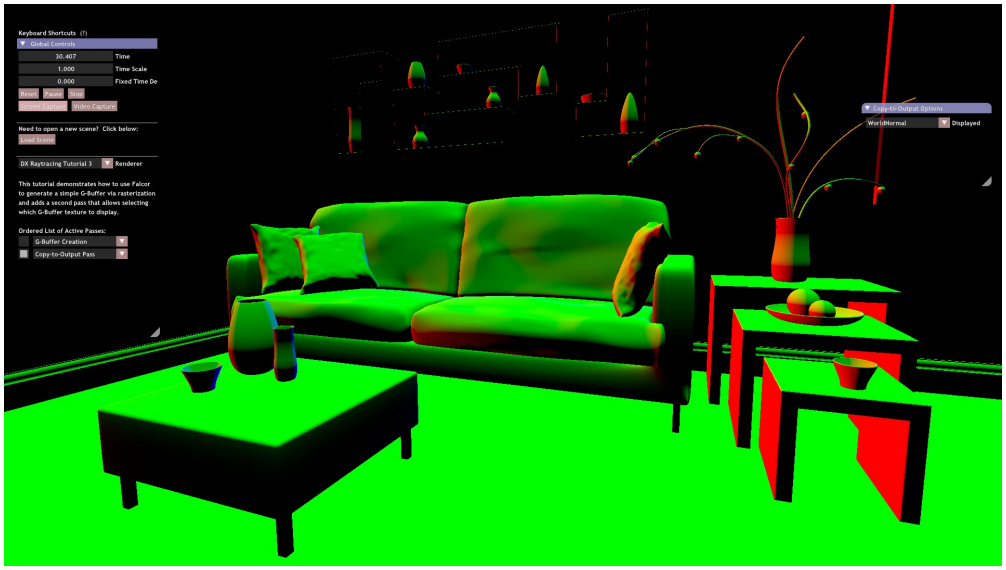
```
[shader("closesthit")]
void PrimaryClosestHit(inout SimpleRayPayload,
                       BuiltinIntersectionAttribs attribs)
{
   // Which pixel spawned our ray?
   uint2 idx = DispatchRaysIndex();

   // Run helper function to compute important data at the current hit point
   ShadingData shadeData = getShadingData( PrimitiveIndex(), attribs );

   // Save out our G-buffer values to the specified output textures
   gWsPos[idx] = float4(shadeData.posW, 1.f);
   gWsNorm[idx] = float4(shadeData.N, length(shadeData.posW - gCamera.posW));
   gMatDif[idx] = float4(shadeData.diffuse, shadeData.opacity);
   gMatSpec[idx] = float4(shadeData.specular, shadeData.linearRoughness);
   gMatExtra[idx] = float4(shadeData.IoR,
                           shadeData.doubleSidedMaterial ? 1.f : 0.f, 0.f, 0.f);
}
```

Our closest hit shader looks remarkably like ou pixel shader in Tutorial 3. We first find which pixel onscreen this ray belongs to (using the *DispatchRaysIndex()* intrinsic), then we get our *ShadingData* using Falcor utility functions, finally we output our G-buffer data to the corresponding output buffers.

As in Tutorial 3, this G-buffer is extremely verbose, and you could likely compress the same data into many fewer bits. This format is used for clarity.

4.6. What Does it Look Like?

That covers the important points of this tutorial. When running, you get basically the same result as in Tutorial 3:



Result of running Tutorial 4, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated:

- How to launch DirectX Raytracing work.
- Write basic ray tracing HLSL shaders.
- Use intrinsic structures and functions like `RayDesc`, `DispatchRaysIndex` and `TraceRay`.

When you are ready, continue on to Tutorial 5, where we build on our existing G-buffers to render ray traced ambient occlusion.

5. Tutorial 5: Ambient occlusion. Start from G-buffer, trace one AO ray per pixel.

In Tutorial 4, we walked through our first example of spawning rays in DirectX and the new HLSL shader types needed to control them. This tutorial combines the rasterized G-Buffer we created in Tutorial 3 and then launches ambient occlusion rays from the locations of pixels in that G-buffer.

5.1. Our Basic Ambient Occlusion Pipeline

If you open up `Tutor05-AmbientOcclusion.cpp`, you will find our new pipeline combines the `SimpleGBufferPass` from Tutorial 3 with the new `AmbientOcclusionPass` we'll build below.

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, SimpleGBufferPass::create());
pipeline->setPass(1, AmbientOcclusionPass::create());
```

This pipeline is a hybrid rasterization and ray tracing pipeline, as it uses a standard rasterization pass to defer rendering, saving our primary hit points in a G-buffer and shading them in a later pass. In this example, our later pass will shade these rasterized points using ray traced ambient occlusion.

5.2. Launching DirectX Ambient Occlusion Rays From a G-Buffer

Continue by looking in `AmbientOcclusionPass.h`. This should look familiar, as the boilerplate is nearly identical to that from `RayTracedGBufferPass` in Tutorial 4. The major difference is our

member variables to control the ambient occlusion:

```
float     mAORadius = 0.0f;     // Our ambient occlusion radius
uint32_t  mFrameCount = 0;      // Used for unique random seeds each frame
```

Ambient occlusion gives an approximation of ambient lighting and shadowing due to nearby geometry at each pixel. Usually, ambient occlusion is limited to nearby geometry by having a radius that defines "nearby geometry." In our rendering, this is controlled by *`mAORadius`*.

Ambient occlusion is a stochastic effect, so we'll need a random number generator to select our rays. Our shader uses a pseudo random number generator to generate random samples, and to avoid generating the same rays every frame we need to update our random seeds each frame. We do this by hashing the pixel location and a frame count to initialize our random generator.

5.3. Initializing our Ambient Occlusion Pass

Our *`AmbientOcclusionPass::initialize()`* is fairly straightforward:

```
bool AmbientOcclusionPass::initialize(RenderContext::SharedPtr pRenderContext,
                                      ResourceManager::SharedPtr pResManager)
{
   // Stash a copy of our resource manager; request needed buffer resources
   mpResManager = pResManager;
   mpResManager->requestTextureResources( {"WorldPosition", "WorldNormal",
                                           ResourceManager::kOutputChannel });

   // Create our wrapper for our DirectX Raytracing launch.
   mpRays = RayLaunch::create("aoTracing.hlsl", "AoRayGen");
   mpRays->addMissShader("aoTracing.hlsl", "AoMiss");
   mpRays->addHitShader("aoTracing.hlsl", "AoClosestHit", "AoAnyHit");

   // Compile our shaders and pass in our scene
   mpRays->compileRayProgram();
   mpRays->setScene(mpScene);
   return true;
}
```

We first request our rendering resources. We need our G-buffer fields *`"WorldPosition"`* and *`"WorldNormal"`* as inputs and we're outputting a displayable color to *`kOutputChannel`*.

Then we create a ray tracing wrapper *`mpRays`* that starts tracing rays at in the ray generation shader *`AoRayGen()`* in the file *`aoTracing.hlsl`* and has one ray type with shaders *`AoMiss`*, *`AoAnyHit`*, and *`AoClosestHit`*.

5.4. Launching Ambient Occlusion Rays

Now that we initialized our rendering resources, we can shoot our ambient occlusion rays. Let's walk through the *`AmbientOcclusionPass::execute`* pass below:

```
void AmbientOcclusionPass::execute(RenderContext::SharedPtr pRenderContext)
{
   // Get our output buffer; clear it to black.
   auto dstTex = mpResManager->getClearedTexture(
                         ResourceManager::kOutputChannel,
                         vec4(0.0f, 0.0f, 0.0f, 0.0f));

   // Set our shader variables for our ambient occlusion shader
   auto rayGenVars = mpRays->getRayGenVars();
   rayGenVars["RayGenCB"]["gFrameCount"]  = mFrameCount++;
   rayGenVars["RayGenCB"]["gAORadius"]    = mAORadius;
   rayGenVars["RayGenCB"]["gMinT"]        = mpResManager->getMinTDist();
   rayGenVars["gPos"]    = mpResManager->getTexture("WorldPosition");
   rayGenVars["gNorm"]   = mpResManager->getTexture("WorldNormal");
   rayGenVars["gOutput"] = dstTex;

   // Shoot our AO rays
   mpRays->execute( pRenderContext, mpResManager->getScreenSize() );
}
```

First, we grab our output buffer and clear it to black.

Then we pass our G-buffer inputs, output texture, and user-controllable variables down to the ray generation shader. Unlike Tutorial 4, we do not send variables to either the miss shader or closest hit shader. Instead, our variables will only be visibile in the ray generation shader.

We pass down our *`mFrameCount`* to seed our per-pixel random number generator, our user-controllable *`mAORadius`* to change the amount of occlusion seen, and a minimum *t* value to use when shooting our rays. This *`gMinT`* variable will control how much bias to add at the beginning of our rays to avoid self-intersection on surfaces.

We also pass down our input textures *`gPos`* and *`gNorm`* from our prior G-buffer pass and our default pipeline output to *`gOutput`*.

5.5. DirectX Raytracing for Ambient Occlusion

Let's start by reading our ambient occlusion shader's ray generation shader *`AoRayGen()`*:

```
cbuffer RayGenCB {
   float gAORadius, gMinT;
   uint  gFrameCount;
}

Texture2D<float4> gPos, gNorm;
RWTexture2D<float4> gOutput;

[shader("raygeneration")]
void AoRayGen()
{
   // Where is this thread's ray on screen?
   uint2 pixIdx = DispatchRaysIndex();
   uint2 numPix = DispatchRaysDimensions();

   // Initialize a random seed, per-pixel and per-frame
   uint randSeed = initRand( pixIdx.x + pixIdx.y * numPix.x, gFrameCount );

   // Load the position and normal from our g-buffer
   float4 worldPos  = gPos[pixIdx];
   float3 worldNorm = gNorm[pixIdx].xyz;

   // Default ambient occlusion value if we hit the background
   float aoVal = 1.0f;

   // worldPos.w == 0 for background pixels; only shoot AO rays elsewhere
   if (worldPos.w != 0.0f)
   {
      // Random ray, sampled on cosine-weighted hemisphere around normal
      float3 worldDir = getCosHemisphereSample(randSeed, worldNorm);

      // Shoot our ambient occlusion ray and update the final AO value
      aoVal = shootAmbientOcclusionRay(worldPos.xyz, worldDir,
                                       gMinT, gAORadius);
   }

   gOutput[pixIdx] = float4(aoVal, aoVal, aoVal, 1.0f);
}
```

First, we discover what pixel we're at and how big our ray launch is. We use this data to initialize our psuedo random number generator (see tutorial code for details on *`initRand()`*) and to load the surfaces stored in our G-buffer (i.e., in *`gPos`* and *`gNorm`*).

Next we need to test if our G-buffer actually stores geometry or it the pixel represents background. (Our G-buffer stores this information in the alpha channel of the position: *`0.0f`* means a background pixel, other values represent valid hit points.) For background pixels, our default AO value will be *`1.0f`* representing fully lit.

For pixels containing valid G-buffer hits, *`getCosHemisphereSample()`* computes a random ray (see code for details or various places online for the math of cosine-weighted sampling).

We trace our ambient occlusion ray by calling our helper function *`shootAmbientOcclusionRay()`*:

```
struct AORayPayload {
   float aoVal;  // Stores 0 on a ray hit, 1 on ray miss
};

float shootAmbientOcclusionRay( float3 orig, float3 dir,
                                float minT,  float maxT )
{
   // Setup AO payload.  By default, assume our AO ray will *hit* (value = 0)
   AORayPayload  rayPayload = { 0.0f };

   // Describe the ray we're shooting
   RayDesc       rayAO = { orig, minT, dir, maxT };

   // We're going to tell our ray to never run the closest-hit shader and to
   //     stop as soon as we find *any* intersection
   uint rayFlags = RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH |
                   RAY_FLAG_SKIP_CLOSEST_HIT_SHADER;

   // Trace our ray.
   TraceRay(gRtScene, rayFlags, 0xFF, 0, 1, 0, rayAO, rayPayload );

   // Return our AO value out of the ray payload.
   return rayPayload.aoVal;
}
```

This helper function sets up our *`RayDesc`* with the minimum *t* to avoid self intersection and the specified maximum *t* (of *`gAoRadius`*). It creates a ray payload that will toggle between 1.0f (it no intersection occurs) and 0.0f (if we hit a surface), then we trace a ray that never executes any closet hit shaders and stops as soon as it gets any confirmed intersections.

We only have one hit group and one miss shader specified, so the 4th and 6th parameters to *`TraceRay()`* must be 0 and the 5th parameter must be 1.

Our miss and hit shaders for AO rays are extremely simple:

```
[shader("miss")]
void AoMiss(inout AORayPayload rayData)
{
   rayData.aoVal = 1.0f;
}
```

When our ambient occlusion ray misses, we need to toggle our payload value to store *`1.0f`*, which is the value representing illuminated.

```
[shader("anyhit")]
void AoAnyHit(inout AORayPayload rayData, BuiltinIntersectionAttribs attribs)
{
   if (alphaTestFails(attribs)) IgnoreHit();
}
```

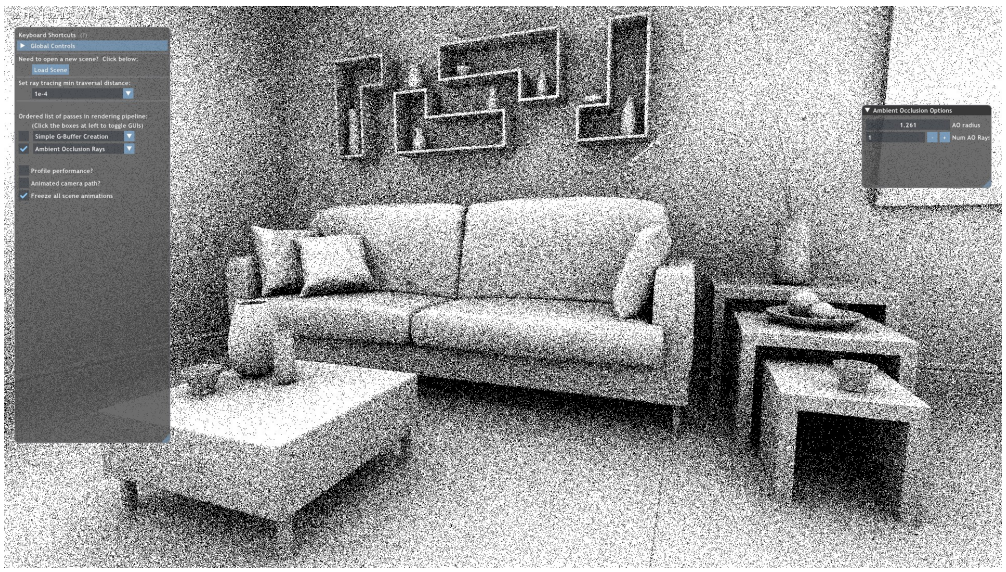
When we have a potential hit, make sure it is valid by checking if there is an alpha mask and if the hit fails the alpha test.

```
[shader("closesthit")]
void AoClosestHit(inout AORayPayload, BuiltinIntersectionAttribs)
{
}
```

Out closest hit shader is never executed, so it can be a trivial no-op. Alternatively, this function could be removed from the shader and a null closest-hit shader specified in `AmbientOcclusionPass::initialize()`.

5.6. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:



Result of running Tutorial 5, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated:

- How to launch DirectX rays from a previously rasterized G-buffer
- Write basic ray tracing HLSL shaders to output ambient occlusion

When you are ready, continue on to Tutorial 6, which accumulates ambient occlusion samples over time so you can get ground-truth ambient occlusion rather than noisy one sample per pixel results.

6. Tutorial 6: Adding a simple temporal accumulation pass

In Tutorial 5, we built a hybrid ray tracer that generates a rasterized G-Buffer and then in a second pass shoots rays into the environment to discover nearby occluders. However, for most stochastic algorithms our random rays introduce a lot of noise. The easiest way to reduce the noise is to improve our approximation by firing more rays — either all at once, or by accumulating them over multiple frames.

In this demo, we are going to create a new `RenderPass` that keeps internal state to average multiple frames to get a high quality ambient occlusion rendering. We will reuse this `SimpleAccumulationPass` in most of our subsequent tutorials to allow generation of very high quality images.

6.1. Our Improved Temporal Accumulation Rendering Pipeline

If you open up `Tutor06-TemporalAccumulation.cpp`, you will find our new pipeline combines the `SimpleGBufferPass` from Tutorial 3, the `AmbientOcclusionPass` from Tutorial 5 and the new `SimpleAccumulationPass` we'll build below.

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, SimpleGBufferPass::create());
pipeline->setPass(1, AmbientOcclusionPass::create());
pipeline->setPass(2,
SimpleAccumulationPass::create(ResourceManager::kOutputChannel));
```

This *SimpleAccumulationPass* takes as input the shared texture to accumulate. Logically, this pipeline (a) renders a G-buffer, (b) computes ambient occludion, storing the output in *kOutputChannel*, then (c) accumulates the image in *kOutputChannel* with the renderings from prior frames and outputs the accumulate result back to *kOutputChannel*.

Of course, our *SimpleAccumulationPass* is a bit more sophisticated, as it allows clearing the accumulation when you move the camera or make other changes to program settings.

6.2. Accumulating Images Temporally

Continue by looking in *SimpleAccumulationPass.h*. Unlike the past few demos, this pass does not require any ray tracing. Instead, we use rasterization over the full screen (i.e., the *FullscreenLaunch* wrapper) to do our temporal accumulation. Thus, this pass is closer in appearance to our sinusoid rendering in Tutorial 2.

Hopefully, the *RenderPass* declaration boilerplate is starting to look familiar. There are a couple key changes. For instance, this pass overrides a few new *RenderPass* methods:

```
void resize(uint32_t width, uint32_t height) override;
void stateRefreshed() override;
```

The *resize()* callback gets executed when the screen resolution changes. Since our accumulation pass averages results over multiple frames, when the resolution changes we need to update our internal storage to match the new frame size.

The *stateRefreshed()* callback gets executed whenever any other pass notes that its settings have changed. In this case, our image will likely change significantly and we should reset our accumulated result.

We also store various new internal data:

```
Texture::SharedPtr  mpLastFrame;         // Our accumulated result
uint32_t            mAccumCount = 0;     // Total frames have we accumulated
Scene::SharedPtr    mpScene;             // What scene are we using?
mat4                mpLastCameraMatrix;  // The last camera matrx
Fbo::SharedPtr      mpInternalFbo;       // A temp framebuffer for rendering
```

In particular, we need a place to store our accumulated color from prior frames (we put that in *mpLastFrame*) and a count of how many frames we have accumulated (in *mAccumCount*).

To avoid accumulating during motion (which gives an ugly smeared look), we need to detect when the camera moves, so we can stop accumulating. We do this by remember the scene and comparing the current camera state with last frame (*mpScene* and *mpLastCameraMatrix*).

We also need a framebuffer object for our full-screen raster pass, and we create one on initialization (in *mpInternalFbo*) and reuse it every frame.

6.3. Initializing our Accumulation Pass

Our *SimpleAccumulationPass::initialize()* is straightforward:

```
bool SimpleAccumulationPass::initialize(RenderContext::SharedPtr pRenderContext,
                                        ResourceManager::SharedPtr pResManager)
{
   // Stash a copy of our resource manager; request needed buffer resources
   mpResManager = pResManager;
   mpResManager->requestTextureResource( mAccumChannel );  // Pass input

   // Create our graphics state and an accumulation shader
   mpGfxState = GraphicsState::create();
   mpAccumShader = FullscreenLaunch::create("accumulate.ps.hlsl");
   return true;
}
```

First we ask to access the texture we're going to accumulate into. This is the channel name passed as input to our pass constructor in *Tutor06-TemporalAccumulation.cpp*.

Then we create a rasterization pipeline graphics state and our wrapper for our fullscreeen accumulation shader.

6.4. Accumulating Current Frame With Prior Frames

Now that we initialized our rendering resources, we can shoot do temporal accumulation. Let's walk through the *SimpleAccumulationPass::execute* pass below:

```
void SimpleAccumulationPass::execute(RenderContext::SharedPtr pRenderContext)
{
   // Get our output buffer; clear it to black.
   auto accumTex = mpResManager->getTexture(mAccumChannel);

   // If camera moved, reset accumulation
   if (hasCameraMoved()) {
      mAccumCount = 0;
      mLastCameraMatrix = mpScene->getActiveCamera()->getViewMatrix();
   }

   // Set our shader variables for our accumulation shader
   auto shaderVars = mpAccumShader->getVars();
   shaderVars["PerFrameCB"]["gAccumCount"] = mAccumCount++;
   shaderVars["gLastFrame"] = mpLastFrame;   // Running accumulation total
   shaderVars["gCurFrame"]  = accumTex;      // Current frame to accumulate

   // Execute the accumulation shader
   mpGfxState->setFbo( mpInternalFbo );
   mpAccumShader->execute( pRenderContext, mpGfxState );

   // Copy accumulated result (to output and our temporary buffer)
   auto outputTex = mpInternalFbo->getColorTexture(0);
   pRenderContext->blit( outputTex->getSRV(), accumTex->getRTV() );
   pRenderContext->blit( outputTex->getSRV(), mpLastFrame->getRTV() );
}
```

First, we grab our input texture that we're accumulating.

Next, we check if we need to reset our accumulation due to any camera movement in the last frame. See the *`hasCameraMoved()`* utility below.

Then we run a simple accumulation shader that takes our prior accumulated result, our current frame, and combines them (with appropriate weights) to get the new average considering the additional frame of input.

Finally, we copy our accumulated result back to our output buffer and our temporary buffer (that keeps our running total for next frame). Note, we use a Falcor utility *`blit()`* to do this copy.

SRV means *shader resource view*, which is a DirectX term meaning roughly "a texture accessor used for reading in our shader". *RTV* means *render target view*, which is a DirectX term meaning roughly "a texture accessor used to write to this texture as output".

6.5. Resetting Accumulation

When do we need to stop accumulating with prior frames and instead resetart our accumulation from scratch? We identify three cases we want to handle:

- Whenever the camera moves
- Whenever the screen is resized
- Whenever our *`RenderPasses`* in our pipline tell us they have changed state.

The camera motion is handled in *`SimpleAccumulationPass::execute`*, with a little help from the following utility function:

```
bool SimpleAccumulationPass::hasCameraMoved() {
   // No scene?  No camera?  Then the camera hasn't moved.
   if (!mpScene || !mpScene->getActiveCamera())
      return false;

   // Check:  Has our camera moved?
   return (mpLastCameraMatrix != mpScene->getActiveCamera()->getViewMatrix());
}
```

We can reset accumulation when the window is resized by creating a *`resize()`* callback, which is explicitly called when the window changes size. This *`resize()`* callback is also called on initialization, since the screen dimensions change from 0 pixels to the actual size.

```
bool SimpleAccumulationPass::resize(uint32_t width, uint32_t height) {
   // Resize our internal temporary buffer
   mpLastFrame = Texture::create2D(width, height, ResourceFormat::RGBA32Float,
                                   1, 1, nullptr,
ResourceManager::kDefaultFlags);

   // Recreate our framebuffer with the new size
   mpInternalFbo = ResourceManager::createFbo(width, height,
   ResourceFormat::RGBA32Float);

   // Force accumulation to restart by resetting the counter
   mAccumCount = 0;
}
```

This resizes our two internal resources (the texture *mpLastFrame* and the framebuffer *mpInternalFbo*). We use a Falcor utility to create our texture (the last 4 parameters specify how many array slices and mip levels the texture has, what data it is initialized with, and how it can be bound for rendering). We use our resource manager to create our framebuffer object.

Finally we reset the *mAccumCount* which is actually what ensures we don't reuse any samples from last frame.

Our last important C++ class method is *stateRefreshed()*, which is a callback that gets executed when any other *RenderPasses* call their *setRefreshFlag()* method. This pair of functions is a simple way for passes to communicate: *setRefreshFlag()* says "I just changed state significantly" and *stateRefreshed()* allows other passes to process this.

```
void SimpleAccumulationPass::stateRefreshed() {
   mAccumCount = 0;
}
```

In this case, our *stateRefreshed()* callback is very simple... it just resets accumulation by setting *mAccumCount* back to zero.

6.6. DirectX Accumulation Shader

Our accumulation shader is extremely simple:

```
cbuffer PerFrameCB {
   uint gAccumCount;  // How many frames have we already accumulated?
}

Texture2D<float4> gLastFrame, gCurFrame;  // Last and current frame inputs

float4 main(float2 texC : TEXCOORD, float4 pos : SV_Position) : SV_Target0
{
   uint2 pixelPos = (uint2)pos.xy;          // Where is this pixel on screen?
   float4 curColor = gCurFrame[pixelPos];   // Pixel color this frame
   float4 prevColor = gLastFrame[pixelPos]; // Pixel color last frame

   // Do a weighted sum, weighing last frame's color based on total count
   return (gAccumCount * prevColor + curColor) / (gAccumCount + 1);
}
```

This shader is simply a weighted sum. Instead of averaging this frame with the last frame, it weights last frame based on the total number of samples it contains. (The current frame always gets a weight of 1.)

6.7. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 6, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated:

- How to use a full-screen pass to run a simple post processing pass
- Accumulate multiple frames together temporally
- Use some of the more advanced callbacks in the `RenderPass` class.

When you are ready, continue on to Tutorial 7, which introduces a camera jitter. When combined with the temporal accumulation from this tutorial, this allows you to get high quality antialiasing to avoid the jaggies seen in the image above.

7. Tutorial 7: Antialising using jittered camera samples

In Tutorial 6, we built a `SimpleAccumulationPass` that allows averaging multiple frames in sequence to get high-quality, noise-free results. However, this leaves numerous problems, including obvious aliasing along geometric edges.

In this tutorial, we add a simple antialiasing scheme by *jittering* the camera position and using the `SimpleAccumulationPass` to average the jittered camera locations. While this antialiasing scheme is quite simple, camera jitter is used in many more advanced schemes including today's state-of-the-art temporal antialiasing (TAA) techniques.

7.1. Our Antialised Rendering Pipeline

If you open up `Tutor07-SimpleAntialiasing.cpp`, you will find our new pipeline combines a new `JitteredGBufferPass`, the `AmbientOcclusionPass` from Tutorial 5, and the new `SimpleAccumulationPass` from Tutorial 6.

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, JitteredGBufferPass::create());
pipeline->setPass(1, AmbientOcclusionPass::create());
pipeline->setPass(2,
    SimpleAccumulationPass::create(ResourceManager::kOutputChannel));
```

This *JitteredGBufferPass* builds on the *SimpleGBufferPass* from Tutorial 3 but computes a per-frame camera jitter. You can think of this jitter as moving the center of each pixel slightly every frame, by +/- 0.5 pixel in any direction.

The compilable tutorial code provides two different ways of selecting this jitter: using a set of discrete samples (the positions used for 8x MSAA) or using a randomly selected offset each frame. Both techniques work identically, the only change is how the offset in [−0.5..+0.5] is computed. The discussion below only walks through the random jitter.

7.2. Setting up Our Jittered Camera Pass

Continue by looking in *JitteredGBufferPass.h*, which should look similar to *SimpleGBufferPass.h* from Tutorial 3.

Key changes are a number of new variables related to random number selection:

```
bool                                mUseJitter = true;
std::uniform_real_distribution<float> mRngDist;
std::mt19937                        mRng;
```

mUseJitter is a user-controllable variable in the GUI that allows toggling camera jitter. *mRng* and *mRngDist* are random number generators from the C++ standard library. It turns out the default initialization for *mRngDist* is exactly what we want. We initialize *mRng* in *JitteredGBufferPass::initialize* by seeding it with the current time (which we also get from the C++ standard library).

```
bool JitteredGBufferPass::initialize(RenderContext::SharedPtr pRenderContext,
                                     ResourceManager::SharedPtr pResManager)
{
  ...
  auto now    = std::chrono::high_resolution_clock::now();
  auto msTime = std::chrono::time_point_cast<std::chrono::milliseconds>(now);
  mRng        = std::mt19937( uint32_t(msTime.time_since_epoch().count()) );
  ...
}
```

7.3. Rendering With a Jittered Camera Pass

Now that we have set up our random number generator, we can add camera jitter in a very straightforward way during our *JitteredGBufferPass::execute()*, which is a direct copy of the *SimpleGBufferPass* except the following additional lines:

```
void JitteredGBufferPass::execute(RenderContext::SharedPtr pRenderContext)
{
  ...
  if (mUseJitter && mpScene && mpScene->getActiveCamera()) {
     uint2 screenSz = mpResManger->getScreenSize();
     float xJitter  = (mRngDist(mRng) - 0.5f) / float(screenSz.x);
     float yJitter  = (mRngDist(mRng) - 0.5f) / float(screenSz.y);
     mpScene->getActiveCamera()->setJitter( xJitter, yJitter );
  }
  ...
}
```

The `if` is protection against undefined results, since we can't jitter the camera if we don't have a valid scene or camera.

We then get a random number. `mRngDist(mRng)` returns a random value between 0 and 1. We offset that to get a random value between −0.5 and +0.5.

Falcor's camera system already has a built-in method for camera jittering, but it requires the jitter to be relative to the entire screen, rather than an individual pixel, so we need to divide by the screen resolution prior to calling `setJitter()`.

Interestingly, adding camera jitter requires no changes to our shader code, so at this point we are done.

7.4. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 7, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated how to add randomized jitter to the camera to achieve simple antialiasing when combined with temporal accumulation.

When you are ready, continue on to Tutorial 8, which uses a thin lens camera model rather than the pinhole camera model usually used in interactive computer graphics. Using a thin lens camera allows us to generate dynamic depth of field.

8. Tutorial 8: Depth-of-field using a simple thin-lens camera

In Tutorial 7, we added random jitter to the camera position to allow antialiasing by accumulating samples temporally over multiple frames. This tutorial modifies our `RayTracedGBufferPass` from Tutorial 4 to use a simple thin lens camera model. Each pixel randomly selects the camera origin somewhere on the lens of this camera. Temporally accumulating these random camera origins over multiple frames allows us to model dynamic depth of field.

8.1. Our Antialised Rendering Pipeline

If you open up *Tutor08-ThinLensCamera.cpp*, you will find our new pipeline combines a new *ThinLensGBufferPass*, the *AmbientOcclusionPass* from Tutorial 5, and the *SimpleAccumulationPass* from Tutorial 6.

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, ThinLensGBufferPass::create());
pipeline->setPass(1, AmbientOcclusionPass::create());
pipeline->setPass(2,
SimpleAccumulationPass::create(ResourceManager::kOutputChannel));
```

This *ThinLensGBufferPass* builds on the *RayTracedGBufferPass* from Tutorial 4 but adds our new per-pixel random camera origin. Essentially, camera jitter from Tutorial 7 perturbs the camera ray *direction*; a thin lens model also perturbs the camera *origin*.

In order to combine antialiasing, our thin lens camera model, and to allow the user to toggle them both on and off, the *ThinLensGBufferPass* also reuses the random jittering code from Tutorial 7.

8.2. Setting up Our Thin Lens

Continue by looking in *ThinLensGBufferPass.h*. The key changes are the introduction of a number of variables related to lens parameters:

```
bool      mUseThinLens = false;  // Currently using thin lens?  (Or pinhole?)
float     mFNumber = 32.0f;      // The f-number of our thin lens
float     mFocalLength = 1.0f;   // The distance to our focal plane
float     mLensRadius;           // The camera aperature.  (Computed)
```

mUseThinLens is a user-controllable variable in the GUI that allows toggling camera jitter. *mFNumber* and *mFocalLength* are the user-controllable parameters for the thin lens. *mFNumber* controls the virtual f-number. *mFocalLength* controls our camera's focal length, which is the distance from the camera where all rays contributing to a pixel converge.

Since we define default values for all our thin lens parameters in the header file, there are no camera-specific additions to our *ThinLensGBufferPass::initialize()* method, so we move on to the changes required in *ThinLensGBufferPass::execute()*

```
void ThinLensGBufferPass::execute(RenderContext::SharedPtr pRenderContext)
{
...
// Compute lens radius based on our user-exposed controls
mLensRadius = mFocalLength / (2.0f * mFNumber);

// Specify our HLSL variables for our thin lens
auto rayGenVars = mpRays->getRayGenVars();
rayGenVars["RayGenCB"]["gLensRadius"]  = mUseThinLens ? mLensRadius : 0.0f;
rayGenVars["RayGenCB"]["gFocalLen"]    = mFocalLength;

// Compute our camera jitter
float xJitter = mUseJitter ? mRngDist(mRng) : 0.0f;
float yJitter = mUseJitter ? mRngDist(mRng) : 0.0f;
rayGenVars["RayGenCB"]["gPixelJitter"] = vec2(xOff, yOff);
...
}
```

The first addition computes *mLensRadius* based on our user-specified focal length and f-number. We then pass down our thin lens parameters to our DirectX ray generation shader, where we'll use it to determine what rays to shoot from our camera. Note: A thin lens camera model degenerates to a pinhole camera if the lens radius is set to zero, so if the user chooses a pinhole camera the logic need not change.

We also pass down a random camera jitter to antialias geometry that is focus. This is slightly different that when rasterizing in Tutorial 7, where Falcor utilities handled everything to jitter the camera. Here we pass the pixel jitter down to our ray generation shader, where we'll take also this jitter into account when tracing our rays.

8.3. DirectX Ray Generation for Jittered, Thin-Lens Camera Rays

The final step in this tutorial is updating our G-buffer's ray generation to perturb our camera origin and ray directions:

```
[shader("raygeneration")]
void GBufferRayGen()
{
    // Get our pixel's position on the screen
    uint2 rayIdx       = DispatchRaysIndex();
    uint2 rayDim       = DispatchRaysDimensions();

    // Convert our ray index into a jittered ray direction.
    float2 pixelCenter = (rayIdx + gPixelJitter) / rayDim;
    float2 ndc         = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 rayDir      = ndc.x * gCamera.cameraU +
                         ndc.y * gCamera.cameraV +
                         gCamera.cameraW;
    ...
```

To start off, it looks very similar to our previous ray traced G-buffer. However, instead of using a fixed *0.5f* offset to shoot our ray through the center of each pixel, we'll use our computed camera jitter *gPixelJitter* as our sub-pixel offset. This gives us antialiasing as in Tutorial 7.

```
    ...
    // Find the focal point for this pixel.
    rayDir /= length(gCamera.cameraW);
    float3 focalPoint = gCamera.posW + gFocalLen * rayDir;
    ...
```

Next, we want to find the *focal point* for this pixel. All rays in the pixel pass through this focal point, so a ray from the camera center goes through it. Compute the right point by moving along this ray an appropriate distance. The division ensures this focal plane is planar (i.e., always the same distance from the camera along the viewing vector *cameraW*).

```
    ...
    // Initialize a random number generator
    uint randSeed = initRand(rayIdx.x + rayIdx.y * rayDim.x, gFrameCount);

    // Get point on lens (in polar coords then convert to Cartesian)
    float2 rnd    = float2( nextRand(randSeed) * M_2PI,
                            nextRand(randSeed) * gLensRadius );
    float2 uv     = float2( cos(rnd.x) * rnd.y, sin(rnd.x) * rnd.y );
    ...
```

Now we need to compute a random ray origin on our lens. To do this, we first initialize a random number generator and pick a random point on a canonical lens of the selected raidus (in polar coordinates), then convert this to a Cartesian location on the lens.

```
    ...
    // Use uv coordinate to compute a random origin on the camera lens
    float3 randomOrig = gCamera.posW + uv.x * normalize(gCamera.cameraU) +
                        uv.y * normalize(gCamera.cameraV);

    // Initialize a random thin lens camera ray
    RayDesc ray;
    ray.Origin    = randomOrig;
    ray.Direction = normalize(focalPoint - randomOrig);
    ray.TMin      = 0.0f;
    ray.TMax      = 1e+38f;
   ...
}
```

Finally, we compute the random thin lens camera ray to use. We convert our random sample into an actual world space position on the camera (using the origin and the camera's *u* and *v* vectors).

Once we have the random camera origin, we can compute our random ray direction by shooting from the random origin *through* this pixel's focal point (computed earlier).

The rest of our G-buffer pass is identical to `RayTracedGBufferPass` from Tutorial 4.

8.4. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 8, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated how to add add a simple thin lens model by randomly selecting ray origin and direction based on standard thin lens camera parameters.

When you are ready, continue on to Tutorial 9, which swaps out our simplistic ambient occlusion shading for a slightly more complex Lambertian material model using ray traced shadows.

9. Tutorial 9: A simple Lambertian material model with ray traced shadows

In Tutorial 8, we added random camera jitter over a camera lens to introduce dynamic depth of field, which is typically hard to achieve using rasterization.

This tutorial finally gets rid of the ambient occlusion shading introduced in Tutorial 5 and replaces it with a simple diffuse shading model (also known as a Lambertian material model). In addition to using Lambertian shading, we also trace shadow rays to each light in the scene. This gives us pixel-accurate hard shadows, which is still something of an open problem in rasterization (which has been plagued by shadow map artifacts for decades).

9.1. Our Lambertian Rendering Pipeline

If you open up *Tutor09-LambertianPlusShadows.cpp*, you will find our new pipeline combines a the *ThinLensGBufferPass* from Tutorial 8 and our new *LambertianPlusShadowPass*.

```
// Create our rendering pipeline
RenderingPipeline *pipeline = new RenderingPipeline();
pipeline->setPass(0, ThinLensGBufferPass::create());
pipeline->setPass(1, LambertianPlusShadowPass::create());
```

Just like the ambient occlusion pass our new *LambertianPlusShadowPass* replaces, we will read from a G-buffer, compute shading and ray traced shadows, and output the result to the displayed image *kOutputChannel*.

9.2. Setting up Our Lambertian Rendering Pass

Continue by looking in *LambertianPlusShadowPass.h*. This header consists entirely of standard boilerplate and *RayLaunch* wrappers you have seen in prior tutorials.

Similarly, looking at *LambertianPlusShadowPass::initialize()* shows a set of familar functionality: requesting appropriate inputs from our G-buffer pass and setting up our diffuse *RayLaunch* wrapper's ray generation, miss, closest-hit, and any-hit shaders appropriately.

Looking at *LambertianPlusShadowPass::execute()* is also straightforward, though it might be worth pointing out that in:

```
rayGenVars["RayGenCB"]["gMinT"] = mpResManager->getMinTDist();
```

getMinTDist() returns a selectable parameter from the GUI that specifies how far shadow rays should be offset to avoid self-intersections due to numerical precision.

9.3. Lambertian Shading and Shooting Shadow Rays

In our HLSL shader file *LambertianPlusShadows.rt.hlsl*, let's start by looking at out ray generation shader *LambertShadowsRayGen()*:

```
[shader("raygeneration")]
void LambertShadowsRayGen()
{
   ...
   float4 difMatlColor = gDiffuseMatl[launchIndex];

   // If we don't hit any geometry, our difuse material is our background color.
   float3 shadeColor = (worldPos.w != 0.0f) ? float3(0,0,0) : difMatlColor.rgb;
   ...
```

Most of the the beginning of this shader is identical to our ambient occlusion shader. However, we do load our diffuse material color from the G-buffer. The G-buffer's diffuse color serves two purposes in our implementation: for surfaces, it stores their Lambertian color; for background pixels, it stores the background color.

So, we initialize our final output color *`shadeColor`* to the diffuse color if our pixel falls on the background, otherwise we initialize our shaded color to black.

```
...
// Loop over all the lights in the scene
for (int lightIdx = 0; lightIdx < gLightsCount; lightIdx++)
{
   // Query our scene to get this info about the current light
   float  distToLight;   // How far away is it?
   float3 lightColor;    // What color is it?
   float3 toLight;       // What direction is it from our current hitpoint?

   // A helper to query the Falcor scene to get light data
   getLightData(lightIdx, worldPos.xyz, toLight, lightColor, distToLight);

   // Compute our Lambertion term (NL dot L)
   float NdotL = saturate(dot(worldNorm.xyz, toLight));

   // Shoot our ray.  Return 1.0 for lit, 0.0 for shadowed
   float vis   = shootShadowRay(worldPos.xyz, toLight, gMinT, distToLight);

   // Accumulate our Lambertian shading color
   shadeColor += vis * NdotL * lightColor;
}

// Modulate based on the physically based Lambertian term (albedo/pi)
shadeColor *= difMatlColor.rgb / M_PI;
...
}
```

The next important part of our Lambertian shader is our loop over the scene's lights. Here *`gLightsCount`* is an automatically set Falcor global variable that specifies how many lights your currently loaded scene has. We need to loop over all these lights and accumulate the light's (potentially shadowed) contribution to the current pixel.

To do this we need a little more data about the light's properties. To avoid digging into specifcs of Falcor's scene representation, I wrote a simple helper *`getLightData()`* to extract this light data. (For reference, this function is available in header with HLSL helper functions in the tutorial's Visual Studio project.)

Once we have the direction to the light, we can compute the Lambertian color as light color times diffuse

BRDF times Lambertian term:

- Our diffuse BRDF is `difMatlColor.rgb / M_PI`
- Our Lambertian term is the dot product `NdotL`
- Our light color is `lightColor * vis`

Where `vis` is a visibility factor of either `1.0f` or `0.0f` depending on if our shadow ray determined the light illuminates the pixel.

We compute our visibility factor with our utility routine `shootShadowRay()`. It turns out that both shadow and ambient occlusion rays are asking the same question (i.e., "is anything between point A and point B?"). That means `shootShadowRay()` is identical to the `shootAoRay()` used in prior tutorials. (And our miss, closest-hit, and any-hit shaders are also the same.)

9.4. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 9, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated how to use a slightly more interesting and realistic shading model and how to do simplistic queries to Falcor scene data to shoot shadow rays towards scene-specified lights.

When you are ready, continue on to Tutorial 10, shows how to load an high dynamic range environment map and use a DirectX miss shader to return a background color that varies across the screen.

10. Tutorial 10: Using a miss shader to index into a high dynamic range light probe

In Tutorial 9, we added a Lambertian shading model with ray traced shadows. Before moving on to global illumination, we want to load and render with environment map light probes, which easily allow much more complex lighting.

For this tutorial, we assume our environment maps are parameterized using a lat-long projection, for instance the free environment maps available here. I encourage you to use high dynamic range

environments (e.g., with a .hdr filename), since they provide more realistic lighting with larger variations in light intensity.

10.1. Environment Maps in our Tutorials

Unlike most of the functionality we have added in our tutorials, using an environment map does not necessarily add a new *RenderPass*. Instead, it simply modifies existing ones, changing the behavior of our miss shaders as rays hit a more complex environment. In more complex pipelines, *every RenderPass* may need to use the environment map.

To simplify this tutorial to focus on exactly how environment maps are used, we add an environment to our *ThinLensGBufferPass* from Tutorial 8. Instead of storing a constant background color to our G-buffer when we hit the background, we will now store a color from our environment map.

To do this, in *Tutor10-LightProbeEnvironmentMap.cpp*, we swap out the *ThinLensGBufferPass* for our new *LightProbeGBufferPass*, which handles jittered antialiasing, a thin lens camera model, and rendering an environment mapped background all in a single, more complex *RenderPass*.

10.2. Setting up a Render Pass For Environment Mapping

Look in *LightProbeGBufferPass.h*. This header looks almost identical to the *ThinLensGBufferPass.h* except for the following addition:

```
bool usesEnvironmentMap() override { return true; }
```

The method *usesEnvironmentMap()* overrides an inherited method from *RenderPass* and tells the pipeline that our passes plan to use an environment light probe. Our tutorial framework does a couple things with this information. First, it adds a control to the GUI allowing you to dynamically load new environment maps. Second, it tells our *ResourceManager* to allocate a default environment map texture.

10.3. Loading an Environment Map

Falcor and our tutorial framework have various built-in texture loading utilites. After adding the *usesEnvironmentMap()* override above, you can dynamically load new light probes using the GUI. However, to specify the environment map loaded on initialization, we add the following line in *LightProbeGBufferPass::initialize()*:

```
mpResManager->updateEnvironmentMap( "MonValley_G_DirtRoad_3k.hdr" );
```

This tells the resource manager to creates an environment map and initialize it with the texture in *"MonValley_G_DirtRoad_3k.hdr"*. Note that if *updateEnvironmentMap()* is called multiple times (e.g., in different *RenderPass::initialize()* methods), the last one called wins.

Falcor uses FreeImage, so you can load any image formats it supports, including .hdr, .exr, .dds, .png, and .jpg (among many others).

10.4. Sending an Evironment Map to a Miss Shader

Look down in *LightProbeGBufferPass::execute()* and you will see the following new lines related to environment mapping:

```
// Pass our environment map down to our miss shader
auto missVars = mpRays->getMissVars(0);
missVars["gEnvMap"] =
mpResManager->getTexture(ResourceManager::kEnvironmentMap);
missVars["gMatDif"] = mpResManager->getTexture("MaterialDiffuse");
```

This sends down two variables (*gEnvMap* and *gMatDif*) for access in miss shader #0. Note: these variables will *only* be accessible in miss shader #0. (However, since the code *also* sets *"gMatDif"* as a hit shader variable later on, if is also accessible to hit group #0).

10.5. Using an Evironment Map in our Miss Shader

Now that we loaded an environment map and sent it to our miss shader, we can index into it as follows (see *LightProbeGBuffer.rt.hlsl*):

```
// The texture containing our environment map
Texture2D<float4> gEnvMap;

[shader("miss")]
void PrimaryMiss()  {
   float2 texDims;
   gEnvMap.GetDimensions( texDims.x, texDims.y );

   float2 uv = wsVectorToLatLong( WorldRayDirection() );
   gMatDif[ DispatchRaysIndex() ] = gEnvMap[ uint2(uv * texDims) ];
}
```

To start, we need to declare out HLSL texture *gEnvMap* to load our texture. The first two lines of *PrimaryMiss()* query the DirectX texture object for its size. The third line converts our ray direction into a (u,v) coordinate to lookup a color in light probe. Finally, we grab the color from our environment map and store it into our output buffer in the appropriate location for the current pixel.

Converting from a vector to a lat-long projection is fairly straightforward, using the following code

```
float2 wsVectorToLatLong(float3 dir)
{
   float3 p = normalize(dir);
   float u = (1.f + atan2(p.x, -p.z) * M_1_PI) * 0.5f;
   float v = acos(p.y) * M_1_PI;
   return float2(u, v);
}
```

Note: The texture lookup code in *PrimaryMiss()* uses nearest-neighbor indexing into the environment map. Instead of calling *gEnvMap[...]*, other DirectX texture lookups, like *gEnvMap.Sample(gTexSampler, uv)*, can be used if you prefer to use linear texture interpolation. Of course, this requires defining a *TextureSampler gTexSampler;* and passing it in from your C++ code.

It is fairly straightforward to use a more complex texture sampler, but this is not in this tutorial:

```
// Need to do stuff to configure mySampler as desired.
Sampler::SharedPtr mySampler = Sampler::create( ... );

missVars["gTexSampler"] = mySampler;
```

10.6. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result, though you need to rotate the camera a bit to see the background:

Result of running Tutorial 10, after loading the scene "pink_room.fscene"

Hopefully, this tutorial demonstrated how to use load an environment map, send it to your HLSL shader, and index into it in your miss shader.

When you are ready, continue on to Tutorial 11, which takes the Lambertian material shading pass from Tutorial 9 and shoots only a simple, random shadow ray for each pixel. This reduces rendering cost in scenes with many lights, but adds some noise in exchange.

11. Tutorial 11: Lambertian surface with one random shadow ray per pixel

In Tutorial 9, we added a Lambertian shading model that traced a shadow ray to every light. While accurate, there are many reasons tracing shadows rays for each light maybe a poor choice. A key consideration is costs increase linearly with additional lights in the scene.

In this tutorial, we will randomly select *just one* light to test for each pixel. This is a common strategy when path tracing with explicit direct lighting, but more importantly introduces the idea of Monte Carlo integration on a very simple quantity (the Lambertian term with direct illumination).

11.1. Changes to our C++ Render Pass

There is a single difference between the C++ code for this tutorial and Tutorial 9. Since our shader requires a random number generator, we pass down a changing "frame count" to allow us to reseed our pseudorandom number generator each frame.

Our header *DiffuseOneShadowRayPass.h* declares:

```
uint32_t  mFrameCount = 0x1337u;
```

This is a counter that increments each frame and only seeds our per-pixel random numbers. Because we don't want the same random number generator as *ThinLensGBufferPass* (or any of our other *RenderPasses*), we initialize *mFrameCount* to a different value in each pass. I picked somewhat arbitrary hexadecimal numbers, but there's nothing special about these values *except* each pass is initialized uniquely.

We also pass this *mFrameCount* down to our shader so we can use it:

```
auto rayGenVars = mpRays->getRayGenVars();
rayGenVars["RayGenCB"]["gFrameCount"] = mFrameCount++;
```

11.2. HLSL Changes for Random Light Selection

Open up *diffusePlus1Shadow.rt.hlsl*. Our first change is we initialize a pseudorandom number generator using our input frame count:

```
[shader("raygeneration")]
void LambertianShadowsRayGen()
{
   ...
   uint randSeed = initRand( launchIndex.x + launchIndex.y * launchDim.x,
                             gFrameCount );
   ...
}
```

Then when we are shading, we no longer loop over our lights:

```
// Old: loop over all lights
// for (int lightIndex = 0; lightIndex < gLightsCount; lightIndex++)

// New: pick a single random light in [0...gLightsCount-1]
int lightIndex = min( int(gLightsCount * nextRand(randSeed)), gLightsCount-1 );
```

The last consideration: because we use just a single light for shading, in a scene with N lights our random sampling will be too dark (by a factor of N). In Monte Carlo integration, the contribution of each random sample should be divided by the probability of selecting that sample.

Our uniform random light selection (above) samples each light with probability *1.0f / N*, so when shading with a random light, we should multiply that light's contribution by *N*. If using a more intelligent light sampling scheme, a more sophisticated probability must be computed. This means our shading computation changes:

```
// Old, per-light contribution when looping through all lights
// shadeColor += NdotL * lightColor * isLit * matlDiffuse / M_PI;

// New contribution for a single, randomly selected light
shadeColor = gLightsCount * NdotL * lightColor * isLit * matlDiffuse / M_PI;
```

11.3. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 11, after loading the scene "pink_room.fscene"

The tutorial defaults to running with temporal accumulation enabled, so if you turn away for a moment, you might not see the difference from Tutorial 9. However, if you disable temporal accumulation, you will notice the shadows are significantly noisier.

Something we observed in our research: it is easier to denoise direct lighting than indirect, global lighting. This means if you decide to use a filter (like our spatiotemporal variance-guded filter) to reconstruct noisy results for your indirect lighting, you can probably save the cost of extra shadow rays for direct lighting. By the time you have enough information to reconstruct indirect light, you probably have more than enough to reconstruct your direct light, too.

When you are ready, continue on to Tutorial 12, which adds one bounce diffuse global illumination.

12. Tutorial 12: Adding one bounce global illumination to our Lambertian shading

In Tutorial 11, we changed our Lambertian shading pass to stochastically shoot exactly one shadow ray per pixel, rather than shooting one ray per scene light. This demonstrates a very simplistic Monte Carlo integration process, which we will extend in this tutorial to compute one-bounce diffuse global illumination.

12.1. Changes to our C++ Render Pass

As in most of the later tutorials, differences in the C++ *`RenderPass`* are fairly minor. In our new *`SimpleDiffuseGIPass.h`*, we now add the following methods and variables:

```
// So we can use an environment map to illuminate the scene
bool usesEnvironmentMap() override { return true; }

// Some user controls allowing the UI to switch on/off certain rays
bool mDoIndirectGI = true;
bool mDoDirectShadows = true;
```

The major change in *`SimpleDiffuseGIPass::initialize()`* occurs because we now plan to shoot *two* types of rays: *shadow rays* and *indirect bounce rays*. This means, we need to define both types

when we initialize our *RayLaunce* wrapper class:

```
// Create our ray tracing wrapper and define the ray generation shader
std::string shaderFile = "simpleDiffuseGI.rt.hlsl";
mpRays = RayLaunch::create(shaderFile, "DiffuseRayGen");

// Define our miss shaders
mpRays->addMissShader(shaderFile, "ShadowMiss");   // Miss shader #0
mpRays->addMissShader(shaderFile, "IndirectMiss"); // Miss shader #1

// Define our hit groups (first is #0, second is #1)
mpRays->addHitShader(shaderFile, "ShadowClosest", "ShadowAny");
mpRays->addHitShader(shaderFile, "IndirectClosest", "IndirectAny");

// Finalize pass; compile and attach our scene
mpRays->compileRayProgram();
if (mpScene) mpRays->setScene(mpScene);
```

The only other changes our C++ code includes sending additional variables down to our DirectX shaders, see *SimpleDiffuseGIPass::execute()*:

```
// Send down our optional UI parameters
auto rayGenVars = mpRays->getRayGenVars();
rayGenVars["RayGenCB"]["gDoIndirectGI"] = mDoIndirectGI;
rayGenVars["RayGenCB"]["gDirectShadow"] = mDoDirectShadows;

// Set an environment map for indirect rays that miss geometry
auto missVars = mpRays->getMissVars(1);   // Indirect rays use miss shader #1
missVars["gEnvMap"] =
      mpResManager->getTexture(ResourceManager::kEnvironmentMap);
```

12.2. HLSL Changes for Diffuse Global Illumination

Open up *simpleDiffuseGI.rt.hlsl*. Our first change is we define a new set of shaders for our indirect ray. The miss shader is essentially copied from our environment map lookup in Tutorial 10 and the any-hit shader is our standard any-hit shader that performs alpha testing:

```
// Identical to our environment map code in Tutoral 10
[shader("miss")]
void IndirectMiss(inout IndirectPayload rayData)
{
   float2 dims;
   gEnvMap.GetDimensions(dims.x, dims.y);
   float2 uv = wsVectorToLatLong(WorldRayDirection());
   rayData.color = gEnvMap[uint2(uv * dims)].rgb;
}

// Identical to any hit shaders for most other rays we've defined
[shader("anyhit")]
void IndirectAny(inout IndirectPayload rayData,
                 BuiltinIntersectionAttribs attribs)
{
   if (alphaTestFails(attribs)) IgnoreHit();
}
```

Our new indirect ray's closest hit shader is somewhat more complex, but essentially reproduces the direct illumination code from Tutorial 11. Basically: when we our bounce rays hit another surface, we do

simple Lambertian shading at the hit points, using the same math we previously defined for primary hits. To save cost, we only shoot one shadow ray from each hit (rather than looping through all scene lights):

```
[shader("closesthit")]
void IndirectClosest(inout IndirectPayload rayData,
                     BuiltinIntersectionAttribs attribs)
{
   // Extract data from our scene description to allow shading this point
   ShadingData shadeData = getHitShadingData( attribs );

   // Pick a random light from our scene to shoot a shadow ray towards
   int lightToSample = min( int(gLightsCount * nextRand(rayData.rndSeed)),
   gLightsCount - 1 );

   // Query our scene to get this info about the current light
   float  distToLight;   // How far away is it?
   float3 lightColor;    // What color is it?
   float3 toLight;       // What direction is it from our current hitpoint?

   // A helper to query the Falcor scene to get light data
   getLightData(lightIdx, shadeData.posW, toLight, lightColor, distToLight);

   // Compute our Lambertion term (NL dot L)
   float NdotL = saturate(dot(shadeData.N, toLight));

   // Shoot our ray.  Return 1.0 for lit, 0.0 for shadowed
   float vis   = shootShadowRay(shadeData.posW, toLight,
   RayTMin(), distToLight);

   // Return the shaded Lambertian shading at this indirect hit point
   rayData.color = vis * NdotL * lightColor * shadeData.diffuse / M_PI;
}
```

We also add a utility function, similar to *shootShadowRay* that encapsulates the process of shooting an indirect ray and returning the incident color in the selected direction:

```
struct IndirectPayload
{
   float3 color;    // The color in the ray's direction
   uint   rndSeed;  // Our current random seed
};

float3 shootIndirectRay(float3 orig, float3 dir, float minT, uint seed)
{
   // Setup shadow ray and the default ray payload
   RayDesc         rayColor = { orig, minT, dir, 1.0e+38f };
   IndirectPayload payload  = { float3(0,0,0), seed };

   // Trace our indirect ray.  Use hit group #1 and miss shader #1 (of 2)
   TraceRay(gRtScene, 0, 0xFF, 1, 2, 1, rayColor, payload);
   return payload.color;
}
```

12.3. Changes to our Ray Generation Shader to Add Indirect Lighting

In our ray generation shader from Tutorial 11, we essentially have the following code:

```
[shader("raygeneration")]
void SimpleDiffuseGIRayGen()
{
    ...
    // Shoot a shadow ray for our direct lighting
    float vis = shootShadowRay(worldPos.xyz, toLight, gMinT, distToLight);
    shadeColor += vis * NdotL * lightColor * difMatlColor.rgb / M_PI;
    ...
}
```

After adding our color for *direct illumination* we need to add our color computations for *indirect illumination*:

```
...
// Pick a random direction for our indirect ray (in a cosine distribution)
float3 bounceDir = getCosHemisphereSample(randSeed, worldNorm.xyz);

// Get NdotL for our selected ray direction
float NdotL = saturate(dot(worldNorm.xyz, bounceDir));

// Shoot our indirect global illumination ray
float3 bounceColor = shootIndirectRay(worldPos.xyz, bounceDir,
gMinT, randSeed);

// Probability of sampling a cosine lobe
float sampleProb = NdotL / M_PI;

// Do Monte Carlo integration of indirect light (i.e., rendering equation)
//    -> This is:  (NdotL * incoming light * BRDF) / probability-of-sample
shadeColor += (NdotL * bounceColor * difMatlColor.rgb / M_PI) / sampleProb;
...
```

This is doing Monte Carlo integration of the rendering equation for just the indirect component of the lighting. In this tutorial, I explicitly define the sampling probability for our random ray.

In more efficient code, you would obviously cancel the duplicate terms. But for novices (and even experts), cancelling terms is an easy way to forget the underlying math. Once you forget the math, you might decide to select your ray differently and forget to update the accumulation term with a new sampling probability. Almost every rendering researcher and engineer has stories of "magic coefficients" used to get a good results in some code base—forgetting the underlying mathematics is usually the cause of these spurious magic numbers.

12.4. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 12, after loading the scene "pink_room.fscene"

With this tutorial, you now have one-bounce diffuse global illumination (and the UI allows you to turn on and off direct and indirect light for comparison). We're getting close to an interesting renderer. However, before continuing on to add more complex materials and multi-bounce global illumination, Tutorial 13 adds a detour to handle high dynamic range outputs.

When rendering simple materials and lighting, displaying colors in a monitor's arbitrary [0...1] range is typically sufficient. However, adding multiple bounces and HDR environment maps starts giving output colors that no longer fit in that range. This can give very dark or blown out images. In order to address this, Tutorial 13 allows you to turn on tone mapping to map a wider range of colors onto your monitor.

13. Tutorial 13: Adding tone mapping to handle high dyanmic range output

In Tutorials 10 and 12, we added high dynamic range environment maps and indirect illumination. With these two additions, we sometimes get renderngs that are perfectly correct but might be too dark or bright to clearly display on our monitors. As we add more complex materials (and, eventually, area lighting using actual measured light power) these problems quickly become worse.

This tutorial takes a quick detour to add a pass to tone map our rendered output. Tone mapping and high dynamic range imaging are widely studied areas. Rather than dive into details, I suggest you find a textbook on the topic (perhaps this one) for a guide on picking the right algorithm for your needs.

Falcor has a built-in tone mapping utility, and we are simply going to expose this utility in a new `SimpleToneMappingPass`.

13.1. A Render Pass Leveraging Falcor's Tone Mapper

Unlike the last few tutorials, let's start again from the beginning: the `main()` function in `Tutor13-SimpleToneMapping.cpp`. This this tutorial, we are adding a fourth render pass to out pipeline:

```
pipeline->setPass(0, LightProbeGBufferPass::create());
pipeline->setPass(1, SimpleDiffuseGIPass::create("HDRColorOutput"));
pipeline->setPass(2, SimpleAccumulationPass::create("HDRColorOutput"));
pipeline->setPass(3, SimpleToneMappingPass::create("HDRColorOutput",
ResourceManager::kOutputChannel));
```

Here what we are doing is:

- Creating our G-buffer
- Rendering our one-bounce diffuse global illumination into the *HDRColorOutput* texture
- Temporally accumulating from and storing the accumualted result into *HDRColorOutput*
- Tone mapping the result from *HDRColorOutput* into our final output *kOutputChannel*

13.2. Defining the Tone Mapping Render Pass

Our render pass definition in *SimpleToneMappingPass.h* is mostly boilerplate, with the key data being:

```
std::string                mInChannel;    // Input texture for tonemapping
std::string                mOutChannel;   // Output texure from tonemapping
GraphicsState::SharedPtr   mpGfxState;    // DirectX raster state
ToneMapping::UniquePtr     mpToneMapper;  // Falcor's tonemapping utility
```

Here *mInputChannel* and *mOutputChannel* are input and output buffers, copied from the constructor (so based on the initialization in *main()*, these are *"HDRColorOutput"* and *kOutputChannel*).

mpGfxState is a default DirectX pipeline state, as we used in Tutorial 2.

mpToneMapper is an instantiation of Falcor's tone mapping utility class. There are a number of methods to configure the tonemapping, including *setOperator()*, *setExposureKey()*, etc. The tone mapper class also provides a GUI, so the user can configure the settings dynamically at run time.

13.3. Applying our Tone Mapping

The key operations in *SimpleToneMappingPass::initialize()* include:

```
mpToneMapper = ToneMapping::create( ToneMapping::Operator::Clamp );
mpGfxState   = GraphicsState::create();
```

The first line instantiates the class and initializes it to use a clamping operator. The clamp operator does *no* tone mapping, essentially copying the input texture to the output. The second line initializes a default DirectX raster state (needed while tone mapping).

To perform tone mapping, our *SimpleToneMappingPass::execute()* looks as follows:

```
void SimpleToneMappingPass::execute(RenderContext::SharedPtr pRenderContext)
{
   Fbo::SharedPtr srcFbo = mpResManager->createManagedFbo({ mInChannel });
   Fbo::SharedPtr dstFbo = mpResManager->createManagedFbo({ mOutChannel });

   pRenderContext->pushGraphicsState(mpGfxState);
   mpToneMapper->execute(pRenderContext.get(), srcFbo, dstFbo);
   pRenderContext->popGraphicsState();
}
```

The first two lines get our input and output buffers as frame buffer objects (as required by `mpToneMapper`). `mpToneMapper->execute()` performs tone mapping, and the `pushGraphicsState()` and `popGraphicsState()` ensure changes in the tone mapper don't affect the rest of our program.

One last observation: in order to expose the class parameters in the UI, we need to add `mpToneMapper->renderUI()` to our `renderGui()` method:

```
void SimpleToneMappingPass::renderGui(Gui* pGui)
{
    mpToneMapper->renderUI(pGui, nullptr);
}
```

13.4. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 13, after loading the scene "pink_room.fscene"

With this tutorial, you can apply Falcor's built-in tone mapping to allow rendering of very dark or very bright environments.

Tutorial 14 changes our Lambertian material model to use Falcor's standard GGX materials and extends our one-bounce global illumination to an arbitrary, user-controllable number of bounces.

14. Tutorial 14: Swapping out a Lambertian BRDF for a GGX BRDF model

In Tutorial 12, we showed how to do recursive bounces for path tracing using a diffuse Lambertain material model. This tutorial explores how to swap out a Lambertian material model for a more complex model, the GGX model commonly used today in film and games. Additionally, we extend the one-bounce global illumination with an arbitrary number of bounces.

14.1. Changes to the C++ Code

The C++ code barely changes in this tutorial, other than changing the pass names. Inside

GGXGlobalIllumination.cpp, we ask for a few additional fields from our G-buffer in the `GGXGlobalIlluminationPass::initialize()` method, including the `MaterialSpecRough` and `Emissive` fields, which include properties needed to render specular materials and those that emit light directly. This means we can render scenes with no light sources (but materials marked "emissive" will act as lights).

Additionally, the method `GGXGlobalIlluminationPass::execute()` passes a few additional parameters to our shader, including a maximum ray depth (`gMaxDepth`) and our additional G-buffer textures.

14.2. New Microfacet Functions in the HLSL Code

A new file in the shader data direction is `microfacentBRDFUtils.hlsli`, which include a number of utility functions for rendering a GGX material. The form of the GGX BRDF is: `D * G * F / (4 * NdotL * NdotV)`. This form was introduced by Cook and Torrance (also available here) and is widely used across many microfacet BRDF models used today.

A microfacet BRDF model assumes the surface is made up of a large number of very tiny planar facets that are all perfectly reflective (i.e., they reflect along the mirror direction). In rough, diffuse surfaces these facets are oriented almost uniformly randomly so light is reflected evenly around the hemisphere. On glossy surfaces, these facets are much more likely to lie flat along the geometry. The `D` term is the microfacet distribution, which controls the probability an incoming ray sees a facet of a particular orientation.

We use a standard form of the GGX normal distribution for D (e.g., math taken from here):

```
float ggxNormalDistribution( float NdotH, float roughness )
{
   float a2 = roughness * roughness;
   float d = ((NdotH * a2 - NdotH) * NdotH + 1);
   return a2 / (d * d * M_PI);
}
```

Note: When building path tracers, it is important to maintain numerical robustness to avoid NaNs and Infs. In some circumstances, the last line in the `ggxNormalDistribution()` function may cause a divide by zero, so you may wish to clamp.

The `G` term in the Cook-Torrance BRDF model represents geometric masking of the microfacets. I.e., facets of various orientations will not always be visible; they may get occluded by other tiny facets. The model for geometric masking we use is from Schlick's BRDF model (or direct PDF). Usually other masking terms are used with GGX (see Naty Hoffman's SIGGRAPH Notes), but this model plugs in robustly without a lot of code massaging, which makes the tutorial code simpler to understand. This formulation for the Schlick approximation comes from Karas' SIGGRAPH 2013 notes from the Physically Based Shading course:

```
float schlickMaskingTerm(float NdotL, float NdotV, float roughness)
{
   // Karis notes they use alpha / 2 (or roughness^2 / 2)
   float k = roughness*roughness / 2;

   // Compute G(v) and G(l).  These equations directly from Schlick 1994
   //     (Though note, Schlick's notation is cryptic and confusing.)
   float g_v = NdotV / (NdotV*(1 - k) + k);
   float g_l = NdotL / (NdotL*(1 - k) + k);
   return g_v * g_l;
}
```

Finally, the `F` term in the Cook-Torrance model is the Fresnel term, which describes how materials become more reflective when seen from a grazing angle. Rarely do renderers implement the full Fresnel equations, which account for the wave nature of light. Since most real-time renderers assume geometric optics, we can ignore wave effects, and most renderers use Schlick's approximation, which comes from the same paper references above:

```
float3 schlickFresnel(float3 f0, float lDotH)
{
   return f0 + (float3(1.0f, 1.0f, 1.0f) - f0) * pow(1.0f - lDotH, 5.0f);
}
```

Finally, in addition to the three functions representing `D`, `G`, and `F`, the `microfacetBRDFUtils.hlsli` also includes the function `getGGXMicrofacet()` which gets a random microfacet orientation (i.e., a facet normal) that follows the distribution described by the function `ggxNormalDistribution()`. This allows us to randomly choose what direction a ray bounces when it leaves a specular surface:

```
// When using this function to sample, the probability density is:
//      pdf = D * NdotH / (4 * HdotV)
float3 getGGXMicrofacet(inout uint randSeed, float roughness, float3 hitNorm)
{
   // Get our uniform random numbers
   float2 randVal = float2(nextRand(randSeed), nextRand(randSeed));

   // Get an orthonormal basis from the normal
   float3 B = getPerpendicularVector(hitNorm);
   float3 T = cross(B, hitNorm);

   // GGX NDF sampling
   float a2 = roughness * roughness;
   float cosThetaH = sqrt(max(0.0f, (1.0-randVal.x)/((a2-1.0)*randVal.x+1) ));
   float sinThetaH = sqrt(max(0.0f, 1.0f - cosThetaH * cosThetaH));
   float phiH = randVal.y * M_PI * 2.0f;

   // Get our GGX NDF sample (i.e., the half vector)
   return T * (sinThetaH * cos(phiH)) +
          B * (sinThetaH * sin(phiH)) +
          hitNorm * cosThetaH;
}
```

14.3. Shading a Surface Point

When shading a point on a surface, we need to invoke these microfacet BRDF functions. To reduce the chance of error, we combine these into a function and call this function from multiple locations. In particular, inside the ray generation shader `ggxGlobalIllumination.rt.hlsl`, shading looks as

follows:

```
// Add any emissive color from primary rays
shadeColor = gEmitMult * pixelEmissive.rgb;

// Do explicit direct lighting to a random light in the scene
if (gDoDirectGI)
   shadeColor += ggxDirect(randSeed, worldPos.xyz, worldNorm.xyz, V,
                           difMatlColor.rgb, specMatlColor.rgb, roughness);

// Do indirect lighting for global illumination
if (gDoIndirectGI && (gMaxDepth > 0))
   shadeColor += ggxIndirect(randSeed, worldPos.xyz, worldNorm.xyz, V,
                           difMatlColor.rgb, specMatlColor.rgb, roughness, 0);
```

Basically, the color at any hitpoint is: the color a surface emits, plus any light directly visible from light sources, plus light that bounces in via additional bounces along the path.

When we fire indirect rays (see `indirectRay.hlsli`), we shade the closest hit using a similar process:

```
[shader("closesthit")]
void IndirectClosestHit(inout IndirectRayPayload rayData,
                        BuiltInTriangleIntersectionAttributes attribs)
{
   // Run a helper functions to extract Falcor scene data for shading
   ShadingData shadeData = getHitShadingData( attribs, WorldRayOrigin() );

   // Add emissive color
   rayData.color = gEmitMult * shadeData.emissive.rgb;

   // Do direct illumination at this hit location
   if (gDoDirectGI)
   {
      rayData.color += ggxDirect(rayData.rndSeed, shadeData.posW,
            shadeData.N, shadeData.V, shadeData.diffuse, shadeData.specular,
            shadeData.roughness);
   }

   // Do indirect illumination (if we haven't traversed too far)
   if (rayData.rayDepth < gMaxDepth)
   {
      rayData.color += ggxIndirect(rayData.rndSeed, shadeData.posW,
            shadeData.N, shadeData.V, shadeData.diffuse, shadeData.specular,
            shadeData.roughness, rayData.rayDepth);
   }
}
```

14.4. Direct Lighting Using a GGX Model

Direct lighting using a GGX model looks very similar to the direct lighting using Lambertian from Tutorial 12. In particutlar, we start by picing a random light, extracting it's information from the Falcor scene representation, and tracing a shadow ray to determine if it is visible.

Note that with many BRDFs, our GGX model consists of a specular lobe and a diffuse lobe. The math for the diffuse lobe is identical to that in Tutorial 12, we're just adding a new specular lobe to the diffuse term.

If visible, we perform shading using the GGX model (i.e., `D * G * F / (4* NdotL * NdotV)`). In this case, numerical robustness is improved significantly by cancelling `NdotL` terms in the GGX lobe to avoid potential divide-by-zero when light hits geometry at a grazing angle. I left in these cancelled `NdotL` terms in comments to make the math clear.

```
float3 ggxDirect(inout uint rndSeed, float3 hit, float3 N, float3 V,
                 float3 dif, float3 spec, float rough)
{
   // Pick a random light from our scene to shoot a shadow ray towards
   int lightToSample = min( int(nextRand(rndSeed) * gLightsCount),
                            gLightsCount - 1 );

   // Query the scene to find info about the randomly selected light
   float distToLight;
   float3 lightIntensity;
   float3 L;
   getLightData(lightToSample, hit, L, lightIntensity, distToLight);

   // Compute our lambertion term (N dot L)
   float NdotL = saturate(dot(N, L));

   // Shoot our shadow ray to our randomly selected light
   float shadowMult = float(gLightsCount) *
                      shadowRayVisibility(hit, L, gMinT, distToLight);

   // Compute half vectors and additional dot products for GGX
   float3 H = normalize(V + L);
   float NdotH = saturate(dot(N, H));
   float LdotH = saturate(dot(L, H));
   float NdotV = saturate(dot(N, V));

   // Evaluate terms for our GGX BRDF model
   float  D = ggxNormalDistribution(NdotH, rough);
   float  G = ggxSchlickMaskingTerm(NdotL, NdotV, rough);
   float3 F = schlickFresnel(spec, LdotH);

   // Evaluate the Cook-Torrance Microfacet BRDF model
   //     Cancel NdotL here to avoid catastrophic numerical precision issues.
   float3 ggxTerm = D*G*F / (4 * NdotV /* * NdotL */);

   // Compute our final color (combining diffuse lobe plus specular GGX lobe)
   return shadowMult * lightIntensity * ( /* NdotL * */ ggxTerm +
                                          NdotL * dif / M_PI);
}
```

14.5. Indirect Lighting Using a GGX Model

Bouncing an indirect ray is somewhat more complex. Since we have both a diffuse lobe and a specular lobe, we need to sample them somewhat differently; the cosine sampling used for lambertian shading doesn't have particularly good characteristics for GGX. One way would shoot two rays: one in the diffuse lobe and one in the specular lobe. But this gets costly, and they converge at different rates.

Instead, we randomly pick whether to shoot an indirect diffuse or indirect glossy ray (see `ggxIndirect()`):

```
// We have to decide whether we sample our diffuse or specular/ggx lobe.
float probDiffuse = probabilityToSampleDiffuse(dif, spec);
float chooseDiffuse = (nextRand(rndSeed) < probDiffuse);
```

In this case, we choose specular or diffuse based on their diffuse and specular albedos, though this isn't a particularly well thought out or principaled approach:

```
float probabilityToSampleDiffuse(float3 difColor, float3 specColor)
{
   float lumDiffuse = max(0.01f, luminance(difColor.rgb));
   float lumSpecular = max(0.01f, luminance(specColor.rgb));
   return lumDiffuse / (lumDiffuse + lumSpecular);
}
```

Going back to `ggxIndirect()`, if we sample our diffuse lobe, the indirect ray looks almost identical to that from Tutorial 12. We shoot a cosine-distributed ray, return the color, and divide by the probability of selecting this ray.

```
if (chooseDiffuse)
{
   // Shoot a randomly selected cosine-sampled diffuse ray.
   float3 L = getCosHemisphereSample(rndSeed, N);
   float3 bounceColor = shootIndirectRay(hit, L, gMinT, 0, rndSeed, rayDepth);

   // Accumulate the color: (NdotL * incomingLight * dif / pi)
   // Probability of sampling this ray:  (NdotL / pi) * probDiffuse
   return bounceColor * dif / probDiffuse;
}
```

If we choose to sample the GGX lobe, the behavior is fundamentally identical even though the code is more complex: select a random ray, shoot it and return a color, and divide by the probability of selecting this ray. The key is that when we sample according to `getGGXMicrofacet()` our probability density for our rays is different (and described by `D * NdotH / (4 * LdotH)`).

```
    // Otherwise we randomly selected to sample our GGX lobe
    else
    {
        // Randomly sample the NDF to get a microfacet in our BRDF
        float3 H = getGGXMicrofacet(rndSeed, rough, N);

        // Compute outgoing direction based on this (perfectly reflective) facet
        float3 L = normalize(2.f * dot(V, H) * H - V);

        // Compute our color by tracing a ray in this direction
        float3 bounceColor = shootIndirectRay(hit, L, gMinT, 0, rndSeed, rayDepth);

        // Compute some dot products needed for shading
        float  NdotL = saturate(dot(N, L));
        float  NdotH = saturate(dot(N, H));
        float  LdotH = saturate(dot(L, H));

        // Evaluate our BRDF using a microfacet BRDF model
        float  D = ggxNormalDistribution(NdotH, rough);
        float  G = ggxSchlickMaskingTerm(NdotL, NdotV, rough);
        float3 F = schlickFresnel(spec, LdotH);
        float3 ggxTerm = D * G * F / (4 * NdotL * NdotV);

        // What's the probability of sampling vector H from getGGXMicrofacet()?
        float  ggxProb = D * NdotH / (4 * LdotH);

        // Accumulate color:  ggx-BRDF * lightIn * NdotL / probability-of-sampling
        //    -> Note: Should really cancel and simplify the math above
        return NdotL * bounceColor * ggxTerm / (ggxProb * (1.0f - probDiffuse));
    }
```

14.6. What Does it Look Like?

That covers the important points of this tutorial. When running, you get the following result:

Result of running Tutorial 14, after loading the scene "pink_room.fscene"

With this tutorial, you can run Falcor's using a fairly feature-rich path tracer, even if the sampling is extremely naive. Moving forward, which is left as an exercise to the reader, you can add better importance sampling, multiple importance sampling, and using next-event estimation for better explicit

direct lighting. Additionally, we haven't handled refractive materials in this set of tutorials, though as described in Pete Shirley's Ray Tracing in One Weekend, this is fairly straightforward to add.