SAPIENZA
UNIVERSITÀ DI ROMA

Ray Tracing: developing a 3D renderer from zero for "La Sapienza" University of Rome

Facoltà di Informatica, Matematica e Statistica

Corso di Laurea in Informatica

Candidate

Tommaso Maria Lopedote
ID number 1942920

Thesis Advisor

Prof. Maurizio Mancini

Academic Year 2023/2024

Ray Tracing: developing a 3D renderer from zero for "La Sapienza" University of Rome

Bachelor's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: tmlopedote@libero.it

Abstract

In this thesis I will discuss the end result of the work done during my internship at "La Sapienza" Universirty of Rome, where I developed a Ray Tracing/Path Tracing algorithm.

This thesis is a presentation and analysis of the code and algorithm developed in that time. It will features an analysis of the Ray Tracing algorithm, an analysis of the journey of ray from camera to world. The camera object will be scrutinized along side the shading processes and the BVH. Then we'll explain the process of generating some simple procedural textures, and we'll analyze the implementation of light sources and their results. Finally we'll look at how the renderer behaves in the Cornell Box and we'll compare it with Blender's results.

Contents

Chapter 1

Introduction - Objectives and learning progression

The field of Computer Graphics and in particular the 3D rendering of virtual worlds has been widely explored in the last 70 years, both in industry and in entertainment. 3D Computer graphics has reached a visual fidelity of such high quality in the last 20 years, that it can be indistinguishable from the real world. Such immaculate visual quality is the result of scientific studies, industrial endeavors and ingenuity. The Ray Tracing technique has been developed progressively by building upon initially simple components and by adding complexity through successive iterations. My goal in this thesis is to reproduce such path by learning and developing a working Ray Tracer/Path Tracer. The point of this journey is to understand what takes to reach such high standard by building a flexible and expandable working library of objects and methods that seamlessly connect with clear and simple interfaces in an attempt to kick start my career in Computer Graphics. In this paper I'm going to show the end result of my iterative process, I'll highlight the visual details and implementation techniques used to achieve such images as well as mathematically explain the correctness of such algorithms when necessary.

Chapter 2

A brief history of Computer Graphics

2.1 The Origins, the fields and the scientific research

Computer Graphics, often abbreviated in CG, was born as a subfield of Computer Science and it covers almost everything that is not text or sound on a computer, nowadays CG is synonymous of ways to represent tri-dimensional geometry and art trough computational algorithms. Today CG has expanded tremendously and has shaped new study and work fields:

- UI or User interface design
- Rendering
- Geometry processing
- Computer Animation
- Vector graphics
- 3D Modeling
- Scientific computing
- Image Processing
- Computational Geometry

Depending on the field of employment of the technology and the underlying knowledge base that services such technology changes. Computer Graphics can touch fields like psychology of perception in UI design to physics and optics of Computational Geometry.

2.2 A bit of history

The Origin of the idea of 3D rendering can be traced back to the Renaissance and the invention of the Camera Obscura and the prospective projection. Regarding Ray tracing as a technique we can pinpoint the inventor to Albercht Dürer who described multiple techniques for 3D projections onto a image plane, some scripts describe the Ray tracing technique properly, as in tracing a ray until the interception of an object along its way, other describe the method of tracing geometry onto a plane directly, method known as rasterization nowadays.

Regarding Computer Graphics, the first computer program to run a Ray Tracer was in 1968 by Arthur Appel, it was a non recursive and non stochastic Ray Tracer, today we call it Ray Caster. It sent a single ray from each pixel tracing its path until the first object intersection. Then knowing where the light was, it sent a shadow ray towards it to cast shadows.

We'll have to wait until the early 1980s to see result comparable with modern solution. In 1979, Whitted, Kay and Greenberg developed the very first recursive ray tracer, implementing accurate mirror reflections and density index dependent refraction and Shadows. They also found a solution for the stepping pixel effect known as Anti-Aliasing. But it's when in 1984 Cook at al. presented distribution Ray Tracing, later known as stochastic Ray Tracing, that a new standard where set achieving things like Depth of field effects, soft shadows and fuzzy reflections, standard features nowadays.

Stochastic Ray Tracing involves using a Monte Carlo approach to approximate and resolve color and luminosity on a surface, and it's used specifically to find an accurate color value for a pixel of a raster image. In 1986 we see the first paper documenting the implementation of a Monte Carlo recursive approach applied to every object in a scene, capable of correctly approximate light radiosity, later we'll call that method path-tracing.

The greatest wall to surpass in the 80s to make Ray Tracing a viable option for 3D rendering of complex scenes was the lack of pure computational prowess. 20 years later we see the first films being produced with that very same tech. Finally 15 years later consumer level electronics and software makes Ray Tracing available to everyone, making it more accessible then ever.

Reading through the incredible evolution of this technology and being always passionate about Video Games and Computer graphics sprung my interest in this field even more. The idea of expressing the beauty of reality through physics and math is the perfect representation and synthesis of what the world is. Personally I believe I managed to answer the question "Why study so much Math?" and got a satisfactory answer through Computer Graphics.

Chapter 3

3D rendering through the Ray Tracing Algorithm

3.1 Ray Tracing: It's all about rays

Ray Tracing is an image-centric technique that solves the visibility problem, meaning that "given a set of obstacles in a Euclidian space, two points in that space are said to be visible to each other if the line segment that joins them does not intersect any obstacles."

It does so by defining a system of coordinates of 3D space called World space where objects and a Camera reside. Defining a Camera is as simple as defining a point in space, that we'll call the camera Eye, then in between the Camera eye and the object we slide a grid of pixels, known as a raster image, and we project multiple rays from the Camera eye through each of the pixel of the raster and from there to space. Once rays are traveling through space we have to test if there exists an intersection with an object in the world, and we'll do so by looping over a list of objects and solving an intersection function that gives us the coordinates in space of said intersection, by doing so we can decide the color of the pixel. But if the ray doesn't hit, it returns a background color defined at compile time or by a sky box.

Once we make sure that a ray has hit an object we have to decide what color the pixel has to take. That is on its own a very complex matter, because in the real world, objects do not look like a flat color. They may have imperfection, shiny, transparent, metallic or matte. They may consist of different colors depending on where the light hits and most importantly they interact with other objects that may have reflected light may cast a shadow on them. This part of the rendering process is called Shading and it's a key element of realistic rendering.

But the ray doesn't and shouldn't stop its trip there. After the intersection is calculated and the trip of the Primary Ray (that's the technical name of the ray spawned by the camera) has ended a new series of rays will be born, called Secondary rays, which are the result of the bounces from a surface. These rays should carry all the information that the ray has gathered up to that point. That's why this algorithm is called Ray tracing, because it keeps track of subsequent bounces of a series of rays, otherwise it would be called Ray casting algorithm.

In other words, this kind of Ray tracer is known as a recursive Ray tracer, as the Ray Casting function continues until the ray doesn't get killed by the program itself.

3.2 Drawbacks of Ray Tracing

The main concern with Ray Tracing in production-level renderers is the memory consumption and the testing for ray-to-geometry intersection, both of these problems are physiological to a Ray Tracer:

- Memory consumption: the renderer has to keep in memory all of the objects in a scene all the time, and this can be proven extremely costly in terms of memory for very large scenes with tons of high polygon counts and high definition shaders.
- Geometry intersection: testing the intersection of a single ray with an object in space means looping over all of the objects in memory, and for big production stages that means thousands of assets per ray.

For my renderer, these issues are not of concern as the goal of this internship was learning the INs and OUTs of a functioning photo-realistic Ray Tracing Algorithm

Chapter 4

Language choices and shortcuts

Before moving to the explanation of the code that I've produced it's necessary to explain the implementation choices made in this implementation:

- Language: I've been interested in gaining confidence C++ since I started using it in 2023 in Computer Graphics with Professor Pellacini during his course. The cutting-edge renderers and game engines are developed with the same language, so for my future career it's key to gain a good knowledge of this language. Furthermore, C++ is object-oriented language but at the same time has full access to all the low level control I may need, like memoey control thanks to manual and smart allocation.
- No true Montecarlo integration: but it does feature random ray sampling in a pixel and an average factor to approximate color over an area (Antialiasing) correctly.
- No photometry: this render doesn't feature any kind of photometry technique or algorithm as present in production-level Raytracers/Pathtracers as I expected it would take me too much time to understand and develop, but as the code stands it can be added with no refactoring necessary.
- No Shadow rays: Shadow rays are used to reduce the noise of the image at render time when lights are present (I will come back to it later) and also to better calculate luminosity on intersection point. It entices casting a ray towards the light when the Primary Ray trips over an object, and tests for intersections with other geometry in the path towards the light to test for shadows. Nonetheless, once light-emitting materials are added to the scene objects will cast shadows and will be realistically illuminated as I will explain.

Chapter 5

World, Camera, and Rays: a progressive approach

5.1 Fundamental Building blocks: vectors, rays, matrices, and quaternions

Before indulging in the analysis of the core function of the project, the Ray Tracing algorithm, I believe it's necessary to show the work done with the base math classes used throughout the entire renderer. The following have been developed and expanded upon in large and rapresent the fundamental building blocks of every Ray Tracer. All of the following classes and methods here are self explanatory.

```
class ray{
private: // attributes
    vec3f _origin;
    vec3f _dir;
public:
    // constructors
    ray() {} // default constructor defines a null ray
    ray(const vec3f origin, const vec3f direction) : _origin{origin}, _dir{direction} {}

    // getters
    const vec3f& origin() const { return _origin; }
    const vec3f& direction() const { return _dir; }

    // methods
    vec3f at(const float t) const {
        return _origin + (t * _dir);
    }

};

inline const ostream& operator<< (ostream& out, const ray& r) {
    return out << r.origin() << " " << r.direction();
}
```

Figure 5.1. Tri-Dimensional Ray: ray class

```cpp
class interval{
public:
    /**
     * An interval class to define a continuos interval of real values
     */
    // attributes
    float min, max;

    // constructors
    interval() : min(-infinity), max(+infinity) {}
    interval(float m, float M) : min{m}, max{M} {}
    interval(interval _x, interval _y){
        min = _x.min <= _y.min ? _x.min : _y.min;
        max = _x.max >= _y.max ? _x.max : _y.max;
    }

    // methods
    bool size() const{
        return max - min;
    }

    bool includes_or_equals(float x) const{
        return min <= x && x <= max;
    }

    bool includes(float x) const{ // maybe redundant
        return min < x && x < max;
    }

    bool excludes(float x){
        return x < min || x > max;
    }

    float clamp(float x) const{
        if (x < min) return min;
        else if (x > max) return max;
        return x;
    }

    interval expand(float delta){
        float half_delta = delta / 2.0f;
        return interval( m: min-half_delta,  M: max+half_delta);
    }

    // constants declaration
    static const interval empty;
    static const interval universe;
};

// constants definitions
const interval interval::empty    = interval( m: +infinity,  M: -infinity);
const interval interval::universe = interval( m: -infinity,  M: +infinity);
```

Figure 5.2. Floating point Intervals: intervals class

```
class vec3f{
public: // attributes
    float x, y, z;

    // constructors
    vec3f() : x{0}, y{0}, z{0} {} // default behaviour
    vec3f(float x, float y, float z) : x{x}, y{y}, z{z} {} // if args are defined

    // custom operators
    const float& operator[](const int i) const{
        return (&x)[i];
    }

    float& operator[](const int i){
        return  (&x)[i];
    }

    vec3f operator-() const{ // negative
        return {-x, -y, -z};
    };

    vec3f& operator+=(const vec3f v) {
        x += v.x;
        y += v.y;
        z += v.z;
        return *this;
    };

    vec3f& operator*=(const float t) {
        x *= t;
        y *= t;
        z *= t;
        return *this;
    }

    vec3f& operator/=(const float t) {
        return *this *= (1/t);
    }

    // methods
    float sqr_length() const {
        // each of the vector component is squared
        return x*x + y*y + z*z;
    }

    float length() const {
        // length of a vector which is the sqrt of the square length
        return sqrt( x: sqr_length());
    }

    bool near_zero() const{
        auto s : double  = 1e-8;
        return (fabs(this->x) < s) && (fabs( x: this->y) < s) && (fabs( x: this->z) < s);
    }

    vec3f random(){
        return { x: this->x = random_float(),
                 y: this->y = random_float(),
                 z: this->z = random_float()};
    }

    vec3f random(float min, float max){
        return { x: this->x = random_float(min, max),
                 y: this->y = random_float(min, max),
                 z: this->z = random_float(min, max)};
    }
}; // vec3f class
```

Figure 5.3. Floating point Tri-Dimensional Vector : vec3f class

```
class vec4f {
public:
    float x, y, z, w;

    // constructors
    vec4f() : x{0}, y{0}, z{0}, w{0} {}; // default behaviour
    vec4f(float x, float y, float z, float w) : x{x}, y{y}, z{z}, w{w} {} // if args are defined
    vec4f(vec3f v, float s) : x{v.x}, y{v.y}, z{v.z}, w{s} {}

    // operators
    inline float& operator[](int i) {
        return (&x)[i];
    }

    inline const float& operator[](int i) const {
        return (&x)[i];
    }

    inline vec4f operator-() const { // negative
        return {-x, -y, -z, w};
    };

    inline vec4f& operator+=(const vec4f& v) {
        x += v.x;
        y += v.y;
        z += v.z;
        w += v.w;
        return *this;
    };

    inline vec4f& operator-=(const vec4f& v){
        x -= v.x;
        y -= v.y;
        z -= v.z;
        w -= v.w;
        return *this;
    }

    inline vec4f& operator*=(const float& t) {
        x *= t;
        y *= t;
        z *= t;
        w *= t;
        return *this;
    }

    vec4f& operator/=(const float& t) {
        return *this *= (1/t);
    }

    // methods
    float sqr_length() const {
        // each of the vector component is squared
        return x*x + y*y + z*z + w*w;
    }

    float length() const {
        // length of a vector or norm or ||q||
        return sqrt( x: sqr_length());
    }

    bool near_zero() const{
        auto s : double  = 1e-8;
        return (fabs(x) < s) && (fabs( x: y) < s) && (fabs( x: z) < s) && (fabs( x: w) < s);
    }

    vec3f vector() const {
        return {x, y, z};
    }

    float scalar() const {
        return w;
    }

}; // vec4f
```

Figure 5.4. Floating point Four-Dimensional Vector - Quaternions : vec4f class

```
class matrix44f {
public:
    vec4f x = { x: 0,  y: 0,  z: 0,  w: 0}; // xx xy xz            | xx
    vec4f y = { x: 0,  y: 0,  z: 0,  w: 0}; // yx yy yz            |    yy
    vec4f z = { x: 0,  y: 0,  z: 0,  w: 0}; // zx zy zz  rotations |       zz
    vec4f t = { x: 0,  y: 0,  z: 0,  w: 0}; // t translations      |          tt scaling

    // constructor
    matrix44f() {};
    matrix44f(float id) { x.x = id; y.y = id; z.z = id; t.w = id; };
    matrix44f(vec4f x, vec4f y, vec4f z, vec4f t) : x{x}, y{y}, z{z}, t{t} {};

    // methods
    inline vec4f& operator[] (int i) {
        return (&x)[i];
    }

    inline const vec4f& operator[](int i) const {
        return (&x)[i];
    }

    inline void operator+=(matrix44f &f) {
        x += f.x;
        y += f.y;
        z += f.z;
        t += f.t;
    }

    inline void operator-=(matrix44f &f) {
        x -= f.x;
        y -= f.y;
        z -= f.z;
        t -= f.t;
    }

    inline void operator*=(matrix44f &f) {
        x = x * f.x;
        y = y * f.y;
        z = z * f.z;
        t = t * f.t;
    }

    inline void operator*=(float f) {
        x *= f;
        y *= f;
        z *= f;
        t *= f;
    }

    inline const float determinant() const {
        // from https://semath.info/src/determinant-four-by-four.html
        return  x.x*y.y*z.z*t.w + x.x*y.z*z.w*t.y + x.x*y.w*z.y*t.z
        - x.x*y.w*z.z*t.y - x.x*y.z*z.y*t.w - x.x*y.y*z.w*t.z
        - x.y*y.x*z.z*t.w - x.z*y.x*z.w*t.y - x.w*y.x*z.y*t.z
        + x.w*y.x*z.z*t.y + x.z*y.x*z.y*t.w + x.y*y.x*z.w*t.z
        + x.y*y.z*z.x*t.z + x.z*y.w*z.x*t.y + x.w*y.y*z.x*t.z
        - x.w*y.z*z.x*t.y - x.z*y.y*z.x*t.w - x.y*y.w*z.x*t.z
        - x.y*y.z*z.w*t.x - x.z*y.w*z.y*t.x - x.w*y.y*z.z*t.x
        + x.w*y.z*z.y*t.x + x.z*y.y*z.w*t.x + x.y*y.w*z.z*t.x;
    }

    matrix44f diagonal() const {
        return { x: vec4f{x.x,  y: 0,  z: 0,  w: 0},
                 y: vec4f{ x: 0, y.y,  z: 0,  w: 0},
                 z: vec4f{ x: 0,  y: 0, z.z,  w: 0},
                 t: vec4f{ x: 0,  y: 0,  z: 0, t.w}};
    }

    matrix44f transpose() const {
        return {
                 x: {x.x,  y: y.x,  z: z.x,  w: t.x},
                 y: { x: x.y, y.y,  z: z.y,  w: t.y},
                 z: { x: x.z,  y: y.z, z.z,  w: t.z},
                 t: { x: x.w,  y: y.w,  z: z.w,  t.w}
                };
    }
}; // class matrix44f
```

Figure 5.5. 4x4 Matrices : matrix44f class

5.2 The Camera

The camera represents the eye of the observer in the 3D space, it mimics the behavior of a Camera Obscura or more similarly a digital camera. In fact exactly like a digital camera, our virtual camera has a matrix of photosensitive pixels oriented towards a pinhole, but sitting in front of our pinhole. The way we follow the light is reversed too, as the direction the light follows, going from the eye towards one of the raster pixels, and from there to the outside world to intersect the scene's geometry. Then the trip continues bouncing off the object (a maximum number of times before dying) until it doesn't hit a light emitting material. In short, rays from the camera go through the path of eye-object-light instead of light-object-eye. It has been done this way to optimize the number of rays hitting the eye. If we cast rays directly from the light source the chances of one of those rays entering our camera would be extremely low and Ray Tracing would result computationally extremely expensive.

5.2.1 Initialization of the camera - Aspect ratio

The aspect ratio decides the format of the resulting image. It's the ratio of the image width to the image height. By default our camera will default to 16/9 aspect ratio, and in tandem with a given image width, the camera will automatically define the image height.

```
// define image resolution
auto img_h = int(float (image_width) / aspect_ratio); // h = w * (1/aspect_ratio)
image_height = img_h < 1 ? 1 : img_h; // image_height must be at least 1
```

Figure 5.6. Enter Caption

5.2.2 Initialization of the camera - Camera Frame

By convention, a static camera is always born at the origin of the axis and uses the eye position to define the position of the camera in the world, while the orientation is defined by the eye position plus the distance between the eye and the projection canvas, which by default is pointing towards the z-axis. But that's not very useful. A simple and intuitive way to move the camera is by giving the camera a frame and the ability to point towards any point in World space. Additionally, we need a way to hang the camera to an axis to define an "upwards" direction that is by convention the y axis.

```
// looking frame - camera orientation
vec3f look_from;     // camera reference
vec3f look_at;       // objective reference
vec3f view_up;       // vertical reference
```

(a) Camera frame variables: (u,v,w)

```
// define camera frame
w = unit( v: look_from-look_at);          // z axis
u = unit( v: cross( u: view_up,  v: w));  // y axis
v = cross( u: w, v: u);                   // x axis
```

(b) Camera frame definition formula

Figure 5.7. Camera frame definition

5.2.3 Initialization of the camera - Viewport definition and functions

The Viewport is the raster image or the projection canvas positioned orthogonal to the eye and into the scene and acts like the sensor of a digital camera, but instead of sitting behind the eye, it sits in front, and it's used to cast rays towards the outside world.

We already defined the size and format of our raster image, by defining an aspect ratio and width with which we can calculate the height. The next step is to define a Viewport height and width and to do so we need to know the vertical field of view of the camera and the focus distance, or the distance between the eye and the focus plane which for now we will assume equals to the focal length (the focal length in photography is the distance between the camera center and the image plane). In a later chapter, we'll talk about focal length and focus distance in more detail while emulating the presence of a thin lens and the Defocus Blur effect. Once we have our Viewport height and width we need to define two vectors aligned with our camera frame's x and y axis to help us navigate the raster image pixel by pixel, together with step vectors that will give us the dimension of each pixel according to the width and height of the image. Finally, we need the origin point of our raster image, the top-left corner, and the coordinates on the plane of the center of the pixel (0,0) from which the ray can be cast.

```
// define viewport dimensions
float vfov_rad = degrees_to_radians( degrees: vfov);
float h = tan( x: vfov_rad/2);
viewport_height = 2.0f * h * focus_dist;
viewport_width = viewport_height * (float(image_width) / float (image_height));

// viewport vectors
viewport_u = viewport_width * u;
viewport_v = viewport_height * -v;

// pixel deltas, small vector that define a step to the next pixel of the viewport
pixel_delta_u = viewport_u / float (image_width);
pixel_delta_v = viewport_v / float (image_height);

// location of upper left corner pixel
viewport_upper_left = eye_point - (focus_dist * w) - viewport_u/2 - viewport_v/2;
pixel00_location = viewport_upper_left + 0.5 * (pixel_delta_u + pixel_delta_v);
```

Figure 5.8. Viewport definitions formulas

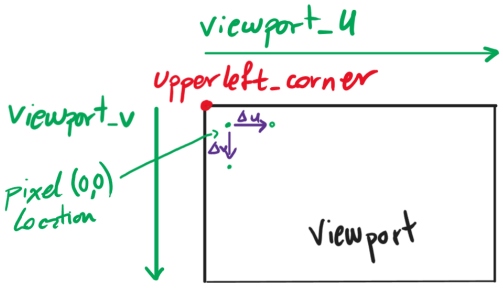
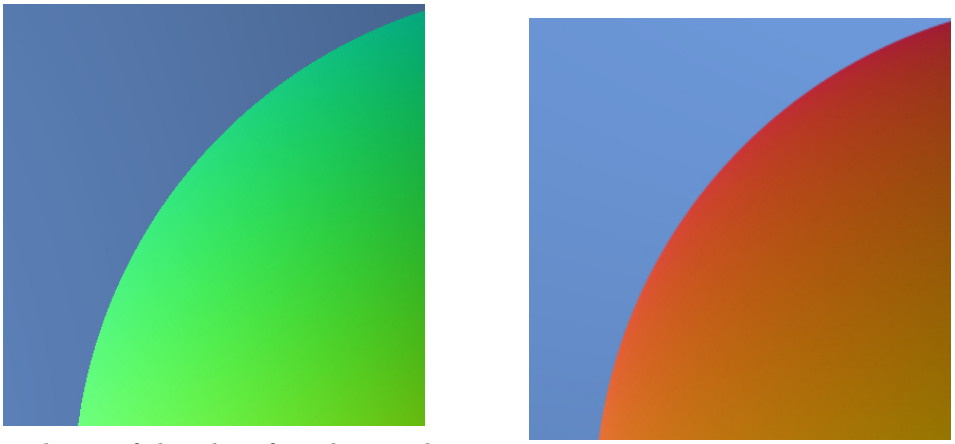


Figure 5.9. Viewport schema

5.2.4 The Rendering Routine and Anti-Aliasing

The rendering routine is at the core of the camera class, its primary function, after initialization of the camera, is parsing the Viewport Raster line by line, pixel by pixel, and sample per sample. For each pixel, the renderer throws several rays from the eye of the camera (or from around a defocus disk, as shown in the next chapter) to the outside world.

The reason for multiple sampling for each pixel is because of an adjustment called Anti-Aliasing, which fixes the "stair-like" look of the edges of geometry. The reason for such an artifact is because that the human eye doesn't perceive the world through discrete values, like a pixel sampling a single color from the scene. The human eye perceives light as a continuous function, while sampling from the center of the pixel makes said pixel the color of the first thing it hits without considering the immediate surroundings. To mitigate this effect we sample color around an area, by skewing the pixel randomly around the pixel surface. We then set the pixel's color to the mean value derived from the sum of all the ray saplings. Finally, we correct the color gamma and write to a png file thanks to the write_png() function.



(a) rendering of the edge of a sphere without Anti Aliasing and a false color gradient applied

(b) rendering of the edge of a sphere with Anti Aliasing and a false color gradient applied

Figure 5.10. Comparison of rendering results with and without Anti-aliasing

```
void render_png(const string& fname, hittable_list& world) {
    initialize();

    auto const start_timer : const time_point<...> {steady_clock::now()};
    vector<vector<pixel_i>> tuple_image = {};
    for (int i = 0; i < image_height; i++){ // for each row
        clog << "\rScanlines remaining: " << i << "/" << image_height << flush;
        vector<pixel_i> scanline = {};
        tuple_image.push_back(scanline);
        for (int j = 0; j < image_width; j++){ // for each column
            pixel_f color_sum( r: 0.0f, g: 0.0f, b: 0.0f);
            for (int s = 0; s < samples_number; s++){   // for random squares in pixel
                ray r = get_ray(i,j);
                color_sum += ray_color(r, &: world, recursive_depth: max_recursion_depth);
            }
            auto mean_color : pixel_f  = color_sum * mean_factor; // (sum(colors rays))/number_of_rays)
            auto correct_color : pixel_i  = correct_color_gamma( color_data: mean_color);
            tuple_image[i].push_back(correct_color);
        }
    }
    write_png( filename: fname, &: image_width, &: image_height, tuple_image);
    // clock
    auto const end_timer : const time_point<...>  {std::chrono::steady_clock::now()};
    const duration<float> elapsed_time { d: end_timer - start_timer};
    clog << "\rRENDERING DONE IN: " << int(elapsed_time.count()) << " seconds" << flush;
}
```

Figure 5.11. render_png() function of camera class

Chapter 6

Simulating a thin lens and Depth of field effects

Digital cameras are subjected to an effect called Depth of field or more precisely Defocus Blur. This happens because cameras are equipped with a diaphragm that opens and closes according to the light conditions of the scene. Ideally, you would make the pinhole so small that only a single ray of light could fit through, but that has the side effect of making the image blurry for objects not close enough to the objective, because the focus plane gets closer to the lens, together with having long exposure time. In computer graphics, the exposure time is not an issue, but the blurriness of the image can be simulated to make images as photorealistic as possible. We can mimic this by simulating a thin lens that sits on the eye of the camera and shoots rays toward a pixel of the View plane. The rays cast from the lens are slightly skewed before reaching the pixel, making the resulting image slightly blurry in a circle around the center.

```
    float defocus_radius = focus_dist * tan( x: degrees_to_radians( degrees: defocus_angle/2));
    defocus_disk_u = u * defocus_radius;
    defocus_disk_v = v * defocus_radius;
}

ray get_ray(const int& i, const int& j) {
    auto offset : vec3f  = sample_square(); // random offset in a square shaped area
    // pixel_center with added random offsets
    auto sampled_pixel : vec3f  = pixel00_location + ((i+offset.y) * pixel_delta_v) + ((j+offset.x) * pixel_delta_u);
    // origin sampling for fringe lens effect
    auto ray_origin : vec3f  = defocus_angle <= 0 ? eye_point : defocus_disk_sample();
    auto ray_direction : vec3f  = sampled_pixel - ray_origin;
    return { origin: ray_origin,  direction: ray_direction};
}

vec3f defocus_disk_sample(){
    auto rdm : vec3f  = random_in_unit_disk();
    return eye_point + (rdm.x * defocus_disk_u) + (rdm.y * defocus_disk_v);
}
```

Figure 6.1. get_ray() function from the camera class

The following render has been realized by setting the camera defocus angle, meaning the radius of the thin lens that sits on the pinhole eye, to 5 degrees. The focus distance is set to 10 units, in facts as you can see the red sphere is still blurry as it slightly further away from the focus point. If we were to put the red sphere at

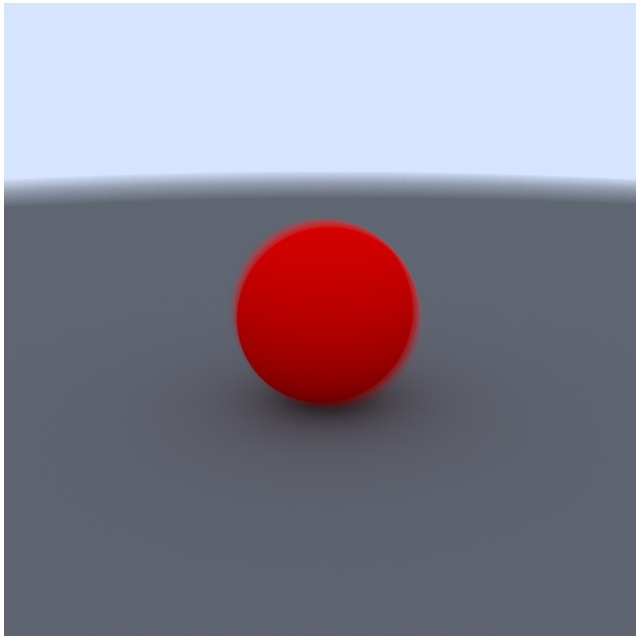


Figure 6.2. An example of Defocus Blur, where the subject is out of focus

the focus point, like in the following render, the red sphere will look much sharper but everything beyond that point will look blurry.

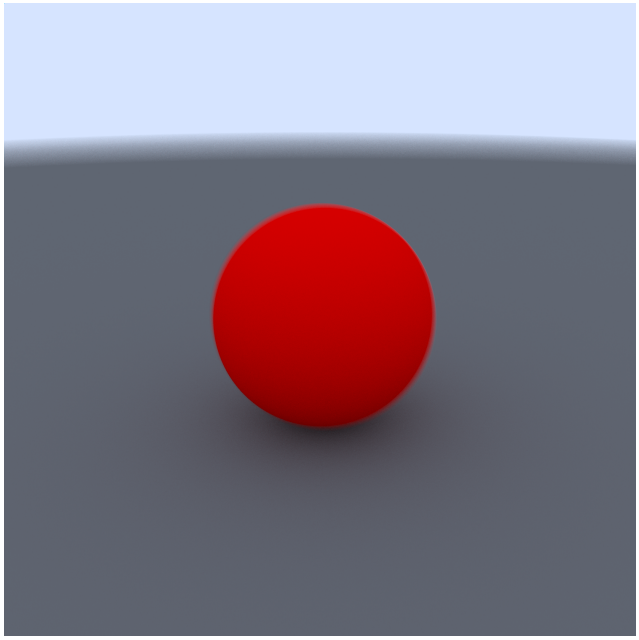


Figure 6.3. An example of Defocus Blur, where the subject is in focus

Chapter 7

Geometry

The point of a Ray Tracer is to render objects in a scene. In this chapter, we are going to analyze the process of intersecting and rendering a sphere, the simplest geometrical object to render, and then two bi-dimensional objects: Triangles and Quads, which are the base building block for any complex polygon mesh. In addition to that we are going to explain the importance of some common geometrical properties of surfaces like surface normals and UVs coordinates, and finally, we'll analyze the architecture of our render in more detail.

7.1 Data structures to describe Ray intersections

7.1.1 hittable Class

The hittable class represents a common interface for all hittable objects, from simple geometry to complex meshes. It exposes a basic list of functions that grants the compiler that every object that publically inherits this interface behaves like a hittable object. What a hittable object does is return a boolean if hit, provide a bounding box, and be able to apply a transformation (translation, rotation, and scaling). Then every object that inherits such a class implements how to execute these functions specifically for the kind of geometry they are.

```
class hittable{ // interface
public:
    // constructor
    virtual ~hittable() = default;
    // methods
    virtual bool hit(const ray& r, interval i, hit_record& record) const = 0;

    virtual axisAlignBbox bounding_box() const = 0;

    virtual void apply_w2o() const = 0;

    virtual void apply_o2w() const = 0;
}; // hittable interface
```

Figure 7.1. Hittable Class interface implementation

7.1.2 hit_record Class

The hit_record class is a simple, yet convenient way to collect and refer to data calculated at hit time by the ray. For each ray, there's a hit_record that will go with it. Furthermore, the hit_record class helps to adjust the normal surface vector after it has been calculated on hit, by checking if the dot product between the direction of the ray and the normal vector calculated on hit, is less than 0. This adjustment is necessary as we will see later in this chapter, all surface normals are calculated in the same direction of the intersecting ray.

```
class hit_record{
    /**
     * A record to save miscellaneous data at ray intersection
     */
public:
    // attributes
    vec3f position;                     // hit vector on the surface
    vec3f normal;                       // normal calculated on ray hit aka shading normal
    shared_ptr<material> material_ptr;  // pointer to material instance
    float t;                            // hit on geometry
    float u;                            // horizontal z, x texture component (u,v)
    float v;                            // vertical y texture component (u,v)
    bool front_face_hit;                // answers the question: Is the object being hit on the front?

    // constructors
    hit_record() {};

    // methods
    void set_face_normals(const ray& r, const vec3f& outward_normal){
        if (dot( u: r.direction(),  v: outward_normal) < 0.0){
            // ray hitting from inside
            normal = outward_normal;
            front_face_hit = true;
        } else {
            // ray hitting from outside
            normal = -outward_normal;
            front_face_hit = false;
        }
    } // set_face_normals method
}; // hit_record class
```

Figure 7.2. hit_record Class implementation

7.1.3 hittable_list Class

The hittable_list class is a container class for hittable objects, and as such it implements the hittable interface, implementing all the methods of said class, but acts as a caller looping over all instances of hittable objects that it contains and calling their respective hit() method.

```
class hittable_list : public hittable{
public:
    // public attributes
    vector<shared_ptr<hittable>> objects_list;
    vector<shared_ptr<material>> material_list;

    // constructors
    hittable_list() {}; // default constructor
    hittable_list(shared_ptr<hittable> obj) {
        objects_list.push_back(obj);
    };

    // methods
    void append(shared_ptr<hittable> obj) {
        objects_list.push_back(obj);
        bbox = axisAlignBbox( bbox1: bbox,  bbox2: obj->bounding_box());
    }

    bool hit(const ray& r, interval i, hit_record& record) const override{
        /**...*/
        hit_record temp;
        bool hit_detected = false;
        auto closest_hit_so_far : float  = i.max;

        for (const auto& obj : shared_ptr<...> const &  : objects_list){
            // update intersect interval to correctly calculate intersection order
            // of hittable objects
            auto updated_i = interval( m: i.min,  M: closest_hit_so_far);
            if (obj->hit(r,  i: updated_i,  &: temp)){
                hit_detected = true;

                if (temp.t < closest_hit_so_far){
                    closest_hit_so_far = temp.t; // update on closest so far
                }
                record = temp; // copies hit record from object to hit record of list.
            }
        }
        return hit_detected;
    } // hit method

    axisAlignBbox bounding_box() const override{
        return bbox;
    }

    void apply_w2o() const override {
        for (const auto& obj : shared_ptr<...> const &  : objects_list) {
            obj->apply_w2o();
        }
    }

    void apply_o2w() const override {
        for (const auto& obj : shared_ptr<...> const &  : objects_list) {
            obj->apply_o2w();
        }
    }

private:
    axisAlignBbox bbox;
}; // hittable_list class
}
```

Figure 7.3. hittable_list Class implementation

7.2 Spheres

The Sphere is the simplest geometrical object and is also considered one of the first objects to be rendered as its definition and intersection function are simple. To define a sphere the only information we need is the coordinates of the sphere in space and the radius of said sphere. While to intersect the sphere it's as simple as resolving the equation for the goniometric sphere rewritten to be solved through vectors. We then make algebraic simplifications to quicken the resolution of the intersection thanks to the quadratic formula, which has been algebraically simplified too.

But before the intersection point it's determined, the hit() function checks if the rays falls into the margins of the sphere. Once that's been found we can proceed with finding the exact intersection position, as well as calculating the surfaces' normals together with calling the sphere's material function and finding the UVs coordinates on the surface for any texture we could have applied.

Figure 7.4. Sphere render with color gradient applied on surface normals

```
class sphere : public hittable{
public:
    // constructors
    sphere() : center{ x: 0, y: 0, z: 0}, radius{1.0} {}; // default behavior
    sphere(const vec3f& c, const float& r, shared_ptr<material> mat) :
            center(c), radius(float (fmax( x: 0, y: r))), material_ptr(mat) {
        // build bvh
        auto temp_v = vec3f{ x: radius, y: radius, z: radius};
        bbox = axisAlignBbox{ a: center - temp_v, b: center + temp_v};
    };


    // methods
    bool hit(const ray& r, interval i, hit_record& hit_record) const override{
        auto c_q : vec3f  = center - r.origin();
        // optimized the dot function of a vector by noticing that the dot function of a
        // vector with itself is just the sqr_length of said vector.
        auto a : float  = r.direction().sqr_length(); // d^2
        // b in old func is now d * (C - Q)
        auto h : float  = dot( u: r.direction(), v: c_q); // d * (C-Q)
        // optimized the dot function of a vector to itself is, sqr_length of said vector
        auto c : float  = c_q.sqr_length() - (radius * radius); // (C - Q)^2 - r^2
        // delta optimized by algebraic simplification
        auto delta : float  = h * h - a * c;
        if (delta < 0) {
            return false;
        }
        auto sqrt_delta : float  = sqrt( x: delta);
        // roots of the quadratic formula
        auto roots : float  = (h - sqrt_delta) / a;
        if (i.excludes( x: roots)){
            roots = (h + sqrt_delta) / a;
            if (i.excludes( x: roots)) {
                return false;
            }
        }

        // adding tuple to save the hit to the sphere object
        hit_record.t = roots;
        hit_record.position = r.at(hit_record.t); // at what point the ray intersected the sphere
        auto normal : vec3f  = (hit_record.position - center) / radius; // calc normal on ray hit
        hit_record.set_face_normals(r, outward_normal: normal);
        hit_record.material_ptr = material_ptr;
        get_sphere_uv( &: hit_record.normal,  &: hit_record.u,  &: hit_record.v);
        return true;
    } // hit method

    axisAlignBbox bounding_box() const override{
        return bbox;
    }

    static void get_sphere_uv(vec3f& p, float& u, float& v){
        float phi = atan2( y: -p.z, p.x) + pi;
        float theta = acos( x: -p.y);
        u = phi / (2 * pi);
        v = theta / pi;
    }

    void apply_w2o() const override {
        transform_point( p_local: center,  m: *w2o);
    }

    void apply_o2w() const override {
        transform_point( p_local: center,  m: *o2w);
    }

private:
    // attributes
    vec3f center;
    float radius = 1.0;
    shared_ptr<material> material_ptr;
    axisAlignBbox bbox;
}; // sphere class
```

Figure 7.5. sphere class implementation

7.3 Quads and Triangles

Quads and Triangles are the building blocks for every complex 3D model existing in the industry. All complex models, going from the model of hyper-realistic human beings to the simple Utah teapot, bases their geometry upon polygons, or as they are known in the industry, polygons meshes. These polygon meshes are simply a collection of points in space and a set of indexes to easily reference their edges and build the polygon. The unit of these polygon meshes is the triangle or the quad. Although, triangles and quads aren't the only way of building complex 3D models. Depending on the field or application of the model, complex models can also be represented through NURBS, Bezier patches or subdivision surfaces.

- The field of triangles and quads intersection has been deeply explored as a good routine to intersect polygonal models is of key importance when optimizing for rendering times. - The procedure itself is not overly complex, but optimizing can be challenging. In my Ray Tracer I take a geometrical approach to such problem, as I thought it was easier and clearer to understand.

But why Triangles? - Euclidian axioms come to the rescue! - If by using two points in space we can define a single line and infinite planes, then through 3 points we can define a single plane, the plane where our triangle is placed. This is not true for other figures like a quad, as one of the four points of a quad can reside on different planes, but the handy feature of triangles is that we can define any non-co-planar quad through two triangles.

Finding out if a ray intersects a triangle or a quad is simple and we can easily adapt the triangle intersection procedure to make it work for a co-planar quad. The only information we need is finding out if the ray intersects the same plane in which the triangle lays, and that said point of intersection is within the triangle boundaries. To find the intersection point to the plane, we need to find the scalar t value in the ray equation that defines the moment of intersection between it and the plane. To do so we just have to solve the ray-plane intersection formula, exposing the scalar t value as the unknown value.

The ray-plane intersection formula uses the origin of the plane and the component of the Normal to said plane to define a function, then replacing the x,y and z components on said formula with the very same components coming from the ray and exposing t as unknown variable we get our intersection point.

Normal coordinates: (A, B, C)

Point along the ray definition: $P = O + tR$

Plane function: $Ax + By + Cz + D = 0$

$$A(O_x + tR_x) + B(O_y + tR_y) + C(O_z + tR_z) + D = 0$$

$$t = -\frac{(A \cdot O_x + B \cdot O_y + C \cdot O_z + D)}{(A \cdot R_x + B \cdot R_y + C \cdot R_z)} =$$

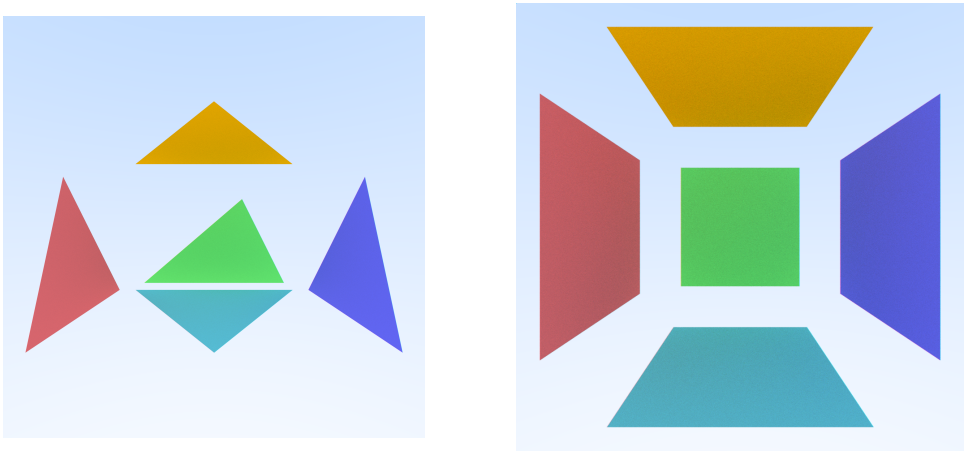
$$= -\frac{N \cdot O + D}{N \cdot R} = -\frac{\text{dot}(N, P.\text{origin}) + D}{\text{dot}(N, P.\text{direction})}$$

Figure 7.6. Plane intersection formula

But before that we need to check for the angle of intersection of the ray, because if the ray is parallel to the surface of the plane, the ray never hits the triangle. Finally before making sure that said point intersects into the boundaries of the triangle we need to take a look at which side of the triangle the ray is hitting. Clearly if the ray hits the triangle from behind (that could translate to intersecting a triangle not facing the camera) we don't have to render it.

To check if P sits inside the boundaries of the triangle we execute the inside-outside test, which encompasses checking if the dot product of the vector along one of the edges of the triangle and the vector defined by the first vertex of the tested edge and P is positive (meaning P is on the left side of the edge). If P is on the left of all three edges, then P is inside the triangle. We do this check by going in a counterclockwise order, and this winding is important to correctly render the face we want.

Note that the process of intersecting a co-planar quad is the same as the one hitting a triangle.



(a) rendering of a series of triangles arranged in a cube

(b) rendering a series of quads arranged in a cube

Figure 7.7. Triangles and Quad rendering

```cpp
class triangle : public hittable {
public:
    // constructors
//      triangle(const vec3f& v1, const vec3f& u, const vec3f& v, shared_ptr<material> material) :
//              v1(v1), u(u), v(v), material(material) {
//          // lower left + 2 vectors to find the other 2 vertices definition
//          v2 = v1 + u;     // lower right
//          v3 = v1 + v;     // upper left
//
//          set_bounding_box();
//      }
    triangle(const vec3f& v1, const vec3f& v2, const vec3f& v3, shared_ptr<material> material, bool ss = false) :
            v1(v1), v2(v2), v3(v3), material(material), single_side(ss) {
        u = v2 - v1;
        v = v3 - v1;

        set_bounding_box();
    }

    // methods
    friend inline ostream& operator<<(ostream& out, const triangle& t){
        return out <<"triangle_instance: (" << t.v1 << "," << t.v2 << "," << t.v3 << ")" << endl;
    }

    bool hit(const ray& r, interval i, hit_record& hit_record) const override {
        auto r_t : ray  = r;

        auto normal : vec3f  = cross(u,v);
        auto denominator : float  = dot( u: normal,  v: r_t.direction());
        auto D : float  = -dot( u: v1,  v: normal);
        if (fabs( x: denominator) < 1e-8) { // check if ray is parallel to the surface
            return false;
        }

        // check facing angle
        if (fabs( x: denominator) > 0 && single_side) {
            return false;
        }

        auto t : float  = -(D + dot( u: normal,  v: r_t.origin())) / denominator;
        if (!i.includes( x: t)) { // check if t intersection point is behind the surface
            return false;
        }

        if (dot( u: r_t.direction(),  v: normal) > 0 && single_side)
            return false;

        auto intersect : vec3f  = r_t.at(t);

        auto c1 : vec3f  = intersect - v1;
        auto c2 : vec3f  = intersect - v2;
        auto c3 : vec3f  = intersect - v3;
        if( dot( u: normal,  v: cross( u: v2 - v1,  v: c1)) < 0 ||
            dot( u: normal,  v: cross( u: v3 - v2,  v: c2)) < 0 ||
            dot( u: normal,  v: cross( u: v1 - v3,  v: c3)) < 0 ){
            return false;
        }

        // save data in hit_record
        hit_record.position = intersect;
        hit_record.t = t;
        hit_record.set_face_normals( r: r_t,  outward_normal: normal);
        hit_record.material_ptr = material;
        return true;
    }

    virtual void set_bounding_box() {
        auto bbox1 = axisAlignBbox( a: v1,  b: v3);
        auto bbox2 = axisAlignBbox( a: v1,  b: v2);
        bbox = axisAlignBbox(bbox1, bbox2);
    }

    axisAlignBbox bounding_box() const override{
        return bbox;
    }

    void apply_w2o() const override {
        transform_point( p_local: v1,  m: *w2o);
        transform_point( p_local: v2,  m: *w2o);
        transform_point( p_local: v3,  m: *w2o);
    }

    void apply_o2w() const override {
        transform_point( p_local: v1,  m: *o2w);
        transform_point( p_local: v2,  m: *o2w);
        transform_point( p_local: v3,  m: *o2w);
    }


private:
    // attributes
    vec3f u, v;                         // horizontal and vertical vectors respectfully
    vec3f v1, v2, v3;                   // coordinates of the vertices of the quad in counter clock-wise order;
    shared_ptr<material> material;
    axisAlignBbox bbox;
    const bool single_side;
}; // triangle class
```

Figure 7.8. Triangle Class implementation

7.4 Some simple primitives - Pyramids and Parallelepipeds

Primitives by definition are simple 3D geometrical shapes that can be composed by assembling pre-existing polygons, like quads and triangles or by defining shapes through geometric formulations. I'll show how to compose two simple primitives by using quads and triangles, which are a parallelepiped and a pyramid with a square base.

The following primitives are built upon the techniques developed for triangles and quads. And as we can see, they are a simple collection of polygons arranged in a hittable_list.

The parallelepiped is built upon eight vertices and six faces that share the same eight vertices.

```
inline shared_ptr<hittable_list> box_quad(const vec3f& a, const vec3f& b, shared_ptr<material> mat){
    auto quad_list : shared_ptr<hittable_list>  = make_shared<hittable_list>();
    // lowest and highest vertices
    auto min = vec3f( x: fmin(a.x, b.x),  y: fmin( x: a.y,b.y),  z: fmin( x: a.z, y: b.z));
    auto max = vec3f( x: fmax(a.x,  y: b.x),  y: fmax( x: a.y, b.y),  z: fmax( x: a.z,  y: b.z));
    // find vectors
    auto dx = vec3f( x: max.x - min.x,  y: 0,  z: 0);
    auto dy = vec3f( x: 0,  y: max.y - min.y,  z: 0);
    auto dz = vec3f( x: 0,  y: 0,  z: max.z - min.z);
    // define quads
    quad_list->append( obj: make_shared<quad>( v1: vec3f(min.x, min.y, max.z),  u: dx,  v: dy,  material: mat)); // front
    quad_list->append( obj: make_shared<quad>( v1: vec3f(max.x, min.y, max.z),  u: -dz,  v: dy,  material: mat)); // right
    quad_list->append( obj: make_shared<quad>( v1: vec3f(max.x, min.y, min.z),  u: -dx,  v: dy,  material: mat)); // back
    quad_list->append( obj: make_shared<quad>( v1: vec3f(min.x, min.y, min.z),  u: dz,  v: dy,  material: mat)); // left
    quad_list->append( obj: make_shared<quad>( v1: vec3f(min.x, min.y, min.z),  u: dz,  v: dx,  material: mat)); // bottom
    quad_list->append( obj: make_shared<quad>( v1: vec3f(max.x, max.y, max.z),  u: -dx,  v: -dz,  material: mat)); // top

    return quad_list;
} // box_quad function
```

Figure 7.9. Parallelepiped Definition function

While the pyramid is built upon five vertices, four of which are used for the base and the fifth as the apex, positioned at the coordinates of the intersection of the two diagonal lines of the rectangular base.

```
inline shared_ptr<hittable_list> pyramid_sqr_base(const vec3f& v1, const vec3f& u, const vec3f& v,
                                                  const vec3f& h, shared_ptr<material> mat) {
    auto pyramid_list : shared_ptr<hittable_list>  = make_shared<hittable_list>();
    pyramid_list->append( obj: make_shared<quad>(v1, u, v,  material: mat)); // base
    // define vertex of base
    auto v2 : vec3f  = v1 + u;     // lower right     4_______3
    auto v3 : vec3f  = v1 + u + v; // upper right     /       /
    auto v4 : vec3f  = v1 + v;     // upper left    1/______/2

    // define x,y position vertex of the apex
    auto center = vec3f( x: (v1.x + v2.x + v3.x + v4.x)/4,  y: 0, z: (v1.z + v2.z + v3.z + v4.z)/4);
    auto apex : vec3f  = center + h;
    // (43, 124, 255)
    pyramid_list->append( obj: make_shared<triangle>(v1, v2,  v3: apex,  material: make_shared<lambertian>( albedo: pixel_f( r: 1,  g: 0,  b: 0)))); // front
    pyramid_list->append( obj: make_shared<triangle>( v1: v2,  v2: v3,  v3: apex,  material: make_shared<lambertian>( albedo: pixel_f( r: 0,  g: 1,  b: 0)))); // right
    pyramid_list->append( obj: make_shared<triangle>( v1: v3,  v2: v4,  v3: apex,  material: make_shared<lambertian>( albedo: pixel_f( r: 0,  g: 0,  b: 1)))); // back
    pyramid_list->append( obj: make_shared<triangle>( v1: v4,  v2: v1,  v3: apex,  material: make_shared<lambertian>( albedo: pixel_f( r: 0,  g: 1,  b: 1)))); // left

    return pyramid_list;
}
```

Figure 7.10. Pyramid Definition Function

Chapter 8

Bounding Volume Hierarchy

The Bounding Volume Hierarchy or BVH for short, is one of the many ways Computer graphics wizards accelerated the rendering process of Ray tracers/Path tracers and it represents one of the simplest ways to minimize rendering time. What it tries to achieve is to kill rays that won't hit any geometry as soon as possible, by Ray Tracing with a series of bounding volumes that covers a partitions of polygons of a model or primitive and intersecting those volumes with the rays we can identify as soon as possible those rays that will not intersect the actual geometry, by checking intersection with a bounding volume hierarchy first. The tighter the fit of the bounding volumes with the geometry is the more precise it will be, killing rays more precisely.

8.1 Axis Aligned Bounding Boxes

The first thing to do with BVH is to define the bounding volume class to divide sub-volumes and fit them into polygons. These Bounding Boxes need to implement the hittable class to grant the compiler the existence of the hit() function. Building a Bounding Volume can be very easy or very complex depending on how tight we want to make the fit. We can simply take two points in space, and by comparing each component from the two points and choosing the biggest one of the two We can build three intervals, one for each axis, that fit the two points. Alternatively, given two bounding volumes, by choosing the largest interval for each axis, We can merge the two volumes.

```
axisAlignBbox(const vec3f& a, const vec3f& b){
    // defines bbox from 2 points
    x = (a.x <= b.x) ? interval( m: a.x,  M: b.x) : interval( m: b.x,  M: a.x);
    y = (a.y <= b.y) ? interval( m: a.y,  M: b.y) : interval( m: b.y,  M: a.y);
    z = (a.z <= b.z) ? interval( m: a.z,  M: b.z) : interval( m: b.z,  M: a.z);

    minimum_padding();
}

axisAlignBbox(const axisAlignBbox& bbox1, const axisAlignBbox& bbox2){
    // define bbox from 2 bbox
    x = interval(bbox1.x,  y: bbox2.x);
    y = interval( x: bbox1.y,  bbox2.y);
    z = interval( x: bbox1.z,  y: bbox2.z);

    minimum_padding();
}
```

Figure 8.1. Axis aligned bounding boxes constructors

Finally, we need to be able to hit these bounding volumes with a ray to give the hit() method its proper implementation. We have to check if the ray hits the two boundaries defined by the intervals by finding the two t0, t1 of the ray function that make the ray function output the x,y,z values that's within the tree intervals of the bounding volume.

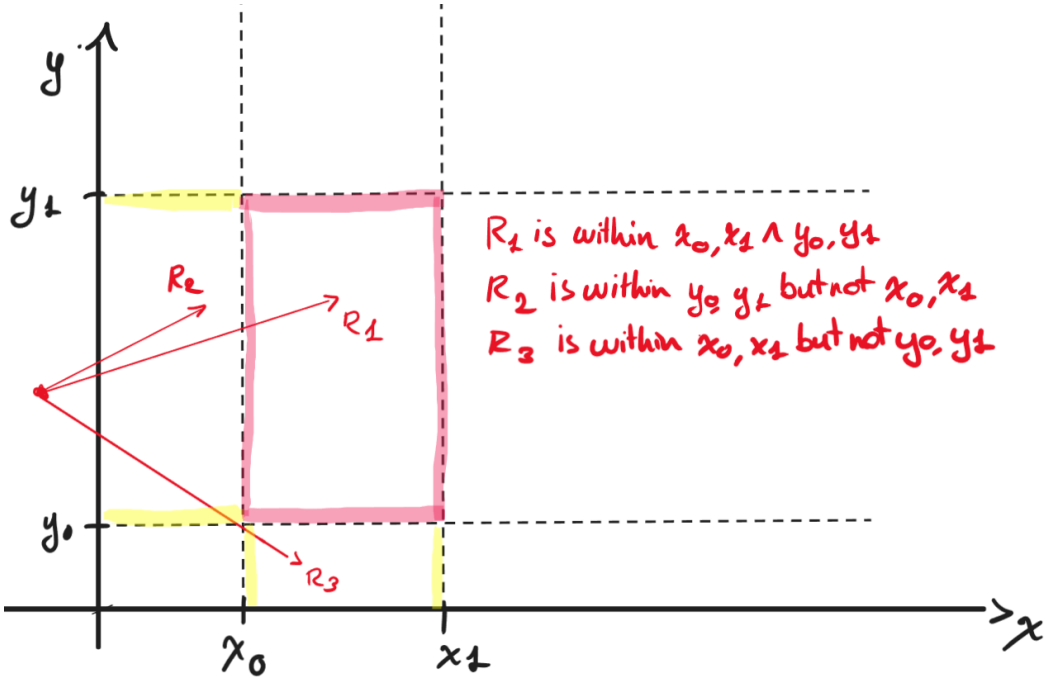


Figure 8.2. Ray to bounding interval intersect for two dimensions

```
bool hit(const ray& r, interval i) const{
    // check for hit, updates intervals
    const vec3f ray_origin = r.origin();
    const vec3f ray_dir = r.direction();

    for (int j=0; j < 3; j++){
        interval axis = (*this)[j];
        const double axis_div = 1.0f / ray_dir[j];
        // t0 and t1 are the intersects points with the bbox
        auto t0 : double  = (axis.min - ray_origin[j]) * axis_div;
        auto t1 : double  = (axis.max - ray_origin[j]) * axis_div;
        // once t0 and t1 are confirmed hits into bbox, check hit into object interval
        if (t0 < t1){
            if (t0 > i.min) i.min = t0;
            if (t1 < i.max) i.max = t1;
        } else {
            if (t1 > i.min) i.min = t1;
            if (t0 < i.max) i.max = t0;
        }

        if (i.max <= i.min) return false;
    }
    return true;
}
```

Figure 8.3. hit() function of the axisAlignBbox class

8.2 Hierarchy

Until now we have defined only the bounding boxes. The next step in the BVH is to define a hierarchy in the form of a binary tree of volumes. To do so we defined the BVH node as a hittableList where starting with an empty bounding volume, would loop over each object of the hittable-list, call the bounding box definition function of each object, and enlarge the empty bounding box with the new one, until we reach the last object of the list. Finally another simple optimization, we will sort the object in the bounding volume along the longest axis. This works because the longest axis is also the one with more objects aligned along it. Then we'll divide the longest axis in two and build recursively the two children of the node in the same manner.

```
bvh_node(vector<shared_ptr<hittable>>& hittableList, size_t start_index, size_t end_index) {
    bbox = axisAlignBbox::empty;
    for (size_t obj_index = start_index; obj_index < end_index; obj_index++){
        // for each object in hittableList define bbox from the previous bbox and the one of the current object
        bbox = axisAlignBbox( bbox1: bbox,  bbox2: hittableList[obj_index]->bounding_box());
    }

    int longestAxis = bbox.longest_axis();
    // comparison function selector used to sort the hittable list
    auto comp : bool (*)(shared_ptr<...>, shared_ptr<...>)  =
            (longestAxis == 0) ? box_x_compare: (longestAxis == 1) ? box_y_compare : box_z_compare;

    size_t delta = end_index - start_index;
    if (delta == 1){
        right = left = hittableList[start_index];

    } else if (delta == 2) {
        left = hittableList[start_index];
        right = hittableList[start_index + 1];

    } else {
        // sort hittableList by x,y,z axis randomly
        sort( first: begin( &: hittableList) + start_index,  last: begin( &: hittableList) + end_index, comp);

        auto m : size_t  = start_index + delta/2;
        left = make_shared<bvh_node>( &: hittableList, start_index,  end_index: m);
        right = make_shared<bvh_node>( &: hittableList,  start_index: m, end_index);
    }
}
```

Figure 8.4. BVH node constructor

Chapter 9

Materials and Texture

In computer graphics, a shading function is defined as a function that yields the intensity value of each point on an object's body, based on the characteristics of the light source, the object, and the position of the observer. - Phong, 1975.

The rendering process is essentially divided into two steps: visibility, shape definition, and geometry, solved through the ray intersection procedure, and shading which is the definition of the appearance of objects that depends on different parameters such as angle of vision, light conditions, and physical characteristics of both light and the object itself. In my renderer I've explored different types of basic techniques from the simplest matte materials to the more complex reflective and refractive materials, delving into texturing and procedural texturing.

The field of Shading has been expanded extensively, especially in the last twenty years as graphic hardware has improved tremendously. Currently, there are two kinds of rendering styles - Photorealistic and Non-Photorealistic. Photorealistic rendering aim to reproduce the look and feel of objects as presented in the real world, while Non-Photorealistic rendering aims to achieve a more concerned with more artistic and fantastic kind of look. The two of these serve a purpose both in industry and entertainment.

To summarize this chapter, we are going to analyze how light behaves when coming in contact with a surface, meaning how that surface will reduce luminosity or divert light, and also how light gets reflected or refracted through different algorithms and objects

We will make sure the renderer remembers the material each ray hits by recording the pointer to the material at the intersection, using the hit_record instance owned by each ray. The point is that when the ray_color() routine of the camera scatters the ray to define the color it can do so by using the pointer saved into the hit_record.

9.1 The material superclass

All of the classes that are considered materials will inherit the material superclass that exposes a basic constructor and two essential methods, scatter() and emitted(). The first one is necessary for all non-emitting materials, and the second one is needed for all kinds of lights.

```
class material{ // interface
public:
    // constructor
    virtual ~material() = default;

    // methods
    virtual bool scatter
            (const ray& ray_in, hit_record& hitRecord, ray& scatter_ray, pixel_f& attenuation)
    const{
        return false;
    }

    virtual pixel_f emitted(const float& u, const float& v, const vec3f& p) const {
        return { r: .0f, g: .0f, b: .0f};
    }

}; // material interface
```

Figure 9.1. The materials superclass implementation

9.2 True Lambertian reflection

The first material we will reproduce is a matte material that could be similar to a rough plastic surface that has no visible highlight. We'll do this by using a function called Lambertian distribution which randomly scatters secondary rays proportionally to the cosine of the angle between the intersection ray and the surface normal Alternatively it can also absorb rays, depending on the reflectance of the surface. To simplify our rendering process we'll choose to always scatter.

We can mimic this distribution by adding a random unit vector pulled from the unit sphere tangent to the outer surface of our object. We have to be careful that the random vector doesn't cancel our normal once it is added. The result is a nice uniform color. Even though we have no actual light point in the scene yet, color correctly bleeds onto other surfaces nearby, because of secondary rays. This is a desired result, as in the real world lights reflect off surfaces modifying the observer's perception of colors. As we can see, the gray surface close to the sphere has a slight red hue where the sphere sits, likewise the bottom side of the sphere is much darker with a gray hue from the floor's color

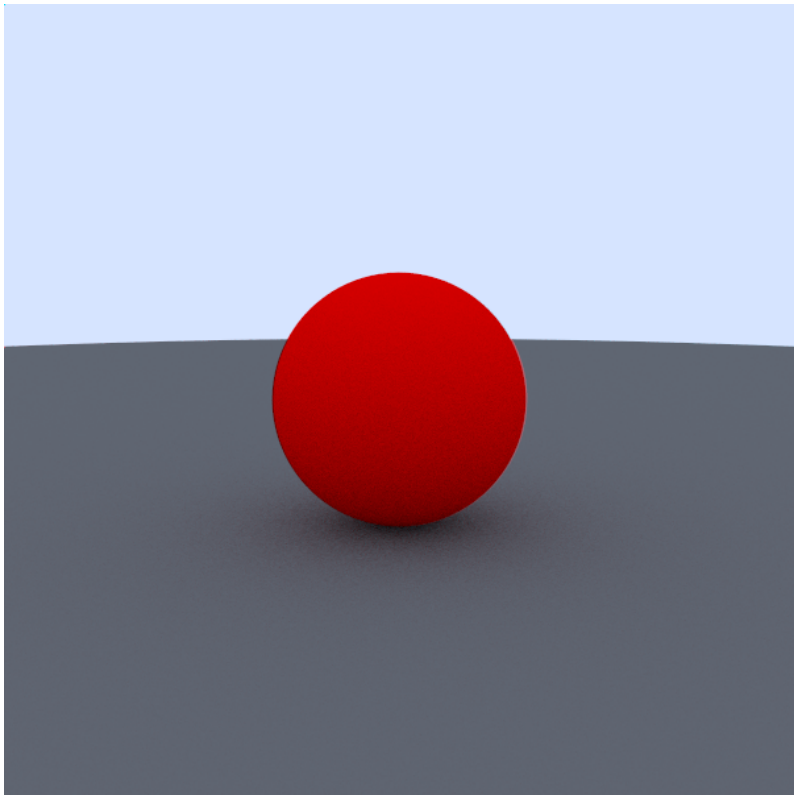


Figure 9.2. A simple sphere with Lambertian material, color bleeding artifacts are visible at the bottom of the sphere and on the floor

9.3 Metals and Mirrored Light reflections

Polished metals behave like a mirror, and as such the ray will not randomly scatter but will reflect bouncing off the surface with a well-defined direction. The direction can be determined using a bit of vectorial math. We can imagine the ray as a tennis ball that hits the ground at a certain angle, to the surface normal, and bounces off with an angle that is symmetrical to the incident direction. It follows what we know as the Law of reflection. After a few mathematical transformations, we end up with the following function. Not all metals are perfectly shiny. To give

```
inline vec3f reflect(const vec3f& v, const vec3f& n){
    // vect_reflect = v + 2b where b is one of the cathetus of the triangle between surface and v
    return v - 2 * dot( u: v, v: n) * n;
}
```

Figure 9.3. The Reflect function

them a brushed look, we just need to randomize the perfect reflection a bit by adding a fuzziness parameter, multiplied by a random unit vector, using the same function used for the Lambertian shading.

```
class metal : public  material {
public:
    // constructor
    metal(pixel_f albedo, float fuzz) : albedo(albedo), fuzziness(fuzz <= 1 ? fuzz : 1)  {}

    // method
    bool scatter
        /**
         * Fuzzy metal, fuzz values are real an in the interval [0,1] where 0 is mirrorball like finish
         * and 1 is as fuzzy as it can get, almost looks plasticky
         */
        (const ray& ray_in, hit_record& hitRecord, ray& scatter_ray, pixel_f& attenuation)
        const override{
        auto reflected_ray :vec3f  = reflect( v: ray_in.direction(),  n: hitRecord.normal);
        auto reflected_fuzzy_ray :vec3f  = unit( v: reflected_ray) + (fuzziness * random_unit_vector());
        scatter_ray = ray( origin: hitRecord.position,  direction: reflected_fuzzy_ray);
        attenuation = albedo;
        return dot( u: reflected_fuzzy_ray,  v: hitRecord.normal) > 0;
        // if dot(N,R) < 0 the ray would be clipping into the ball
    }
private:
    pixel_f albedo;
    float fuzziness;
}; // metal class
```

Figure 9.4. The metal class definition

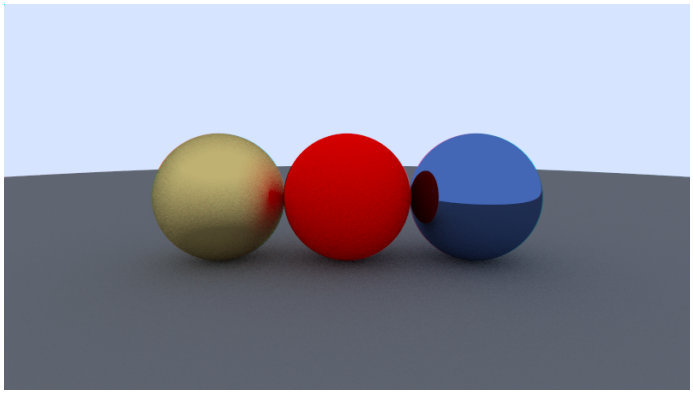


Figure 9.5. From left to right, fuzzy metal, Lambertian, shiny metal

9.4 Dielectrics - Refraction and Snell's law

Dielectrics materials in physics, are partially electrically conductive materials, in this field it refers to the ability of reflecting light. As we saw Metals are able to reflect light perfectly, and for that reason they can also be referred as conductors.

Dielectrics materials like water are only partially conductors but mostly dielectrics, meaning that they will both refract and reflect light rays. The amount of rays that a material refracts depend on the proper refractive index, unique for each material.

When a dielectric material is immersed in another dielectric material, the refractive index becomes the the refractive index of the object's material divided by the refractive index of the surrounding material.

9.4.1 Snell's law

The refraction phenomenon is described by Snell's law: That dictates that the refractive index of both materials and the incident angle of the ray, returns the outward refracted angle. To find a valid direction for the refracted ray, we need to solve it for the sine of the incident and outward rays. Furthermore, by separating the outward ray by its components in respect to the surface normal, we will have the parallel component alongside the perpendicular component. What we are interested in, is solving for both components separately, so we can treat them as two separate vectors into the code.

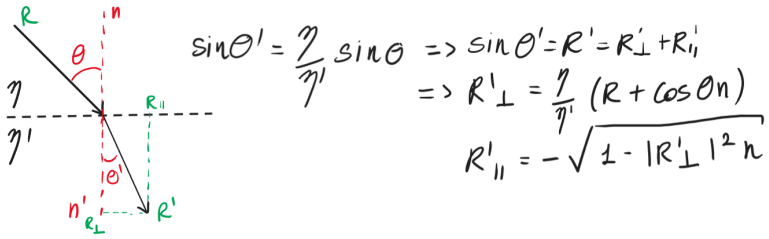


Figure 9.6. Snell's Law and refraction law explained

```
inline vec3f refract(const vec3f& unit_vec_R, const vec3f& normal, float etai_over_etat){
    // refract function to simulate water refraction
    // etai_over_etat being the fraction between the two coefficients
    auto cos_theta : float  = fmin( x: dot( u: -unit_vec_R,  v: normal),  y: 1.0f);
    auto r_y : vec3f  = etai_over_etat * (cos_theta * normal + unit_vec_R); // r parallel component
    auto r_x : vec3f  = -sqrt( x: fabs( x: 1.0f - r_y.sqr_length())) * normal; // r perpendicular component
    return r_y + r_x;
}
```

Figure 9.7. Enter Caption

What we have to consider too is that dielectrics behave like a perfect mirror in some cases, like when we try to look up toward the sky from underwater. This happens because glancing rays entering a medium of lower refractive index from one of higher index gets diffracted at an angle higher then 90 degrees (in radians $sin(\theta^i) > 1.0$). In these cases, Snell's law brakes and the solution doesn't exist. So thanks to this condition we can choose when to reflect and when to refract.

These kind of behavior usually happens with solid transparent materials, for this reason the phenomenon is called total internal reflection.

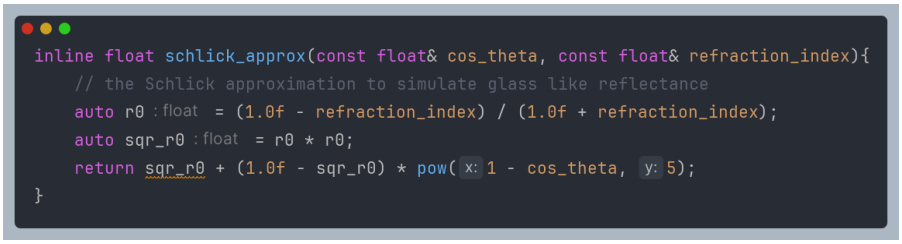
```
class dielectric : public material{
public:
    dielectric(float rf_index) : refraction_index(rf_index) {}

    bool scatter
    (const ray& ray_in, hit_record& hitRecord, ray& scatter_ray, pixel_f& attenuation)
    const override{
        attenuation = pixel_f{ r: 1.0, g: 1.0, b: 1.0};
        float ri = refraction_index;
        if (hitRecord.front_face_hit){
            // invert coefficients if ray hits from the inverse direction
            ri = 1.0f/refraction_index;
        }
        vec3f uv_ray_in = unit( v: ray_in.direction());
        auto cos_theta : float  = fmin( x: dot( u: -uv_ray_in, v: hitRecord.normal), y: 1.0f);
        auto sin_theta : float  = sqrt( x: 1.0f - cos_theta * cos_theta);
        vec3f direction;
        // you want glass behavior? throw in there the schlick approximation
        if (ri * sin_theta > 1.0f|| schlick_approx(cos_theta, refraction_index: ri) > random_float()){
            direction = reflect( v: uv_ray_in, n: hitRecord.normal);
        } else {
            direction = refract( unit_vec_R: uv_ray_in, hitRecord.normal, etai_over_etat: ri);
        }

        scatter_ray = ray( origin: hitRecord.position, direction);
        return true;
    }
private:
    /**
     * Refraction indexes List
     * 1.0f transparent
     * higher then 1.0f, eg 1.5f make refraction apparent, but you can't see the Total initial reflection
     * to see the Total initial reflection use the air ball
     * 1.0f/1.33f
     */
    float refraction_index; // eta on eta1
}; // dielectric class
```

Figure 9.8. Enter Caption

Finally to simulate glass in a accurate manner we would have to use a big and mathematically complex equation that has been effectively approximated through an hack that makes reflective materials behaves like glass. Such hack is a polynomial approximation invented by Christophe Schlick, known as Schlick Approximation. What it does is it changes reflectivity according to the observer angle of view just like glass would do.



```
inline float schlick_approx(const float& cos_theta, const float& refraction_index){
    // the Schlick approximation to simulate glass like reflectance
    auto r0 : float  = (1.0f - refraction_index) / (1.0f + refraction_index);
    auto sqr_r0 : float  = r0 * r0;
    return sqr_r0 + (1.0f - sqr_r0) * pow( x: 1 - cos_theta, y: 5);
}
```

Figure 9.9. Schlick approximation polynomial

And these are the results:

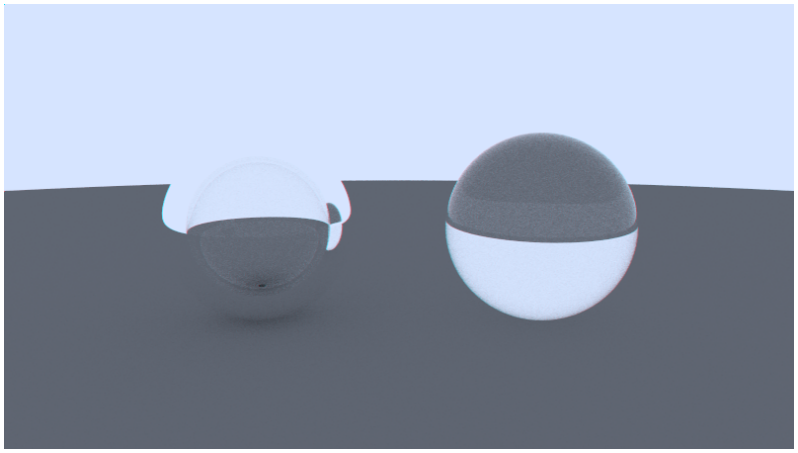


Figure 9.10. A hollow glass sphere that sometimes reflects, sometimes refracts on the left and a glass sphere that always refracts

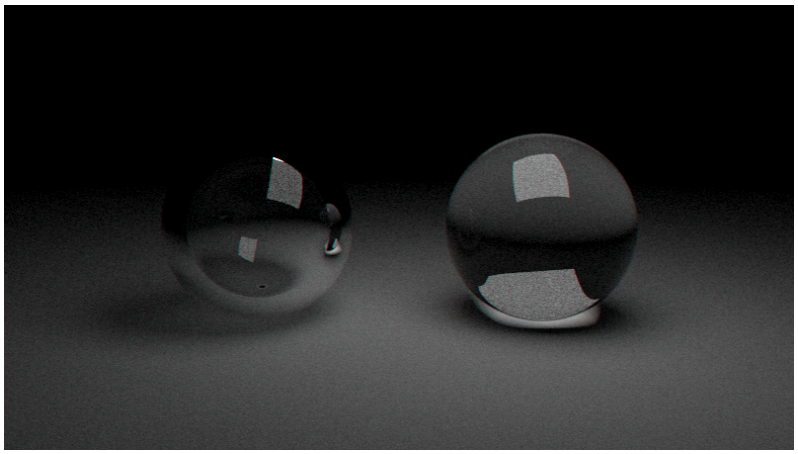


Figure 9.11. A hollow glass sphere that sometimes reflects, sometimes refracts on the left and a glass sphere that always refracts, with a light behind the camera

9.5 Emissive materials and Lights

Emissive materials makes diffuse light surfaces possible and given enough bounces per ray it will also generate realistic looking shadows. Without any heavy modification to our code, we will implement the emitted() function for this specific material, so that it doesn't scatter any ray, but it simply return the texture color. All the other non-emitting material will return the color black because they do not emit light, but they will still reflect it. If a ray doesn't hit a light in its path then it will return a default background color set as a camera parameter. Let's see why this simple light emitting material works. If we observe the ray_color() function of the camera: We'll notice that if the ray hits an emissive material, meaning a light

```
pixel_f ray_color(const ray& r, hittable_list& worldList, const int& recursive_depth){
    // keeps track of light
    if (recursive_depth <= 0){ // end of recursion check
        return background;
    }

    hit_record hitRecord;                // initialize hit record
    interval i( m: 0.01f, M: infinity);    // clamp rays too close to object surface
    if (!worldList.hit(r, i, & hitRecord)) { // if ray doesn't hit geometry
        return background;                // return background color
    }
    // else
    ray scatter;
    pixel_f attenuation;

    // calculate material emission
    auto color_emission : pixel_f  = hitRecord.material_ptr->emitted(hitRecord.u, hitRecord.v, p: hitRecord.position);
    if (!hitRecord.material_ptr->scatter( ray_in: r, & hitRecord, & scatter, & attenuation)){
        // object didn't scatter ray, so it emits light: pixel takes color of emission
        return color_emission;
    }

    // attenuation is used to reduce the light reflected by said object, related to albedo
    pixel_f color_scatter = attenuation * ray_color( r: scatter, & worldList, recursive_depth: recursive_depth-1);
    return color_scatter + color_emission;
}
```

Figure 9.12. ray_color() method of the camera class

surface it will return the color of the emission. But if the ray scatters, the value returned in the color_scatter value it's going to be added with the color_emission value, that for all non-emitting material it's going to return black. The color_scatter value saves the color of the bounce and the color_emission will boost the luminosity of its value. The emitter material works like a color booster, all the ray that trips along an emitting material will gain a boost in luminosity that will be reflected on the rendered color with a bright spot, or a spot in shadow.

In this first render we can see the result of the ray_color() routine and the behavior of light. As we can see light rays are correctly calculated on both sides of the quad. Even the back side of the quad, that is not exposed to the camera, correctly casts light. The adjacent Lambertian sphere shows a very accurate gradient, as the brighter spot is the closest to the light, shading towards darker colors the further we move from the light. More interestingly is the faint shadow visible right under the sphere. We can see the sharp line that grows from the point of contact of the sphere and runs towards the dark.

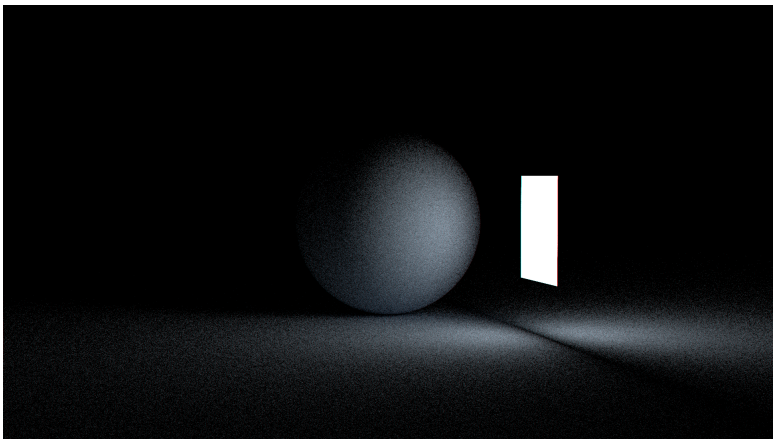


Figure 9.13. Diffiuse Emitting quad with a Lambertian sphere.

For this render we decided to change the material of the sphere to a fuzzy metal and to add another light on top to better analyze the shadow rendering. We can notice how the round shadow casted by the sphere on the ground fades with a gradient, and it's not a sharp cut. That happens because some rays, coming from the top sphere, scrape the surface of the sphere at the apex and they get scattered down at an angle. That will make the shadow softer as it gets colored by those scraping rays. Another interesting feature is the interaction of the fuzzy metal material with the light sources. Firstly they correctly appear on the sphere as bright white spots, as it does the ground. Secondly the shadow reflects into the sphere with a faint ring at the base of the object.

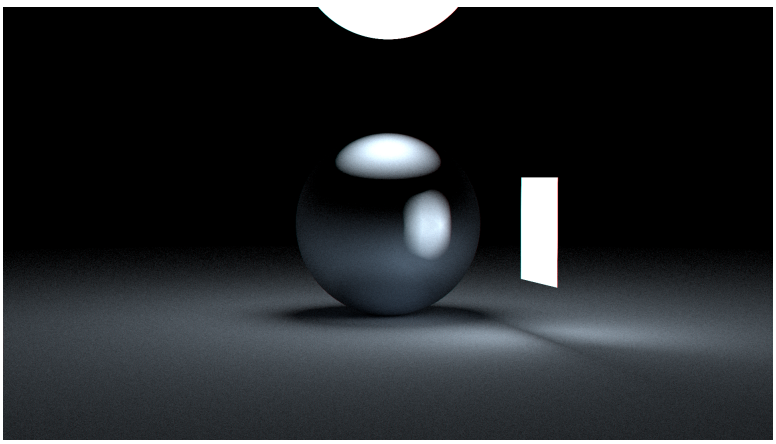


Figure 9.14. Fuzzy metal with a sphere emitter and a quad emitter.

In this render we removed the quad emitter on the side and changed the color of the ground to analyze how the color bleeds. Also the sphere is now a mirror ball now. This scene, beside looking dramatic, it shows that the mirror correctly boosts the brightness of the ground where light from the sphere reflects. As we can see there's a faint red ring very close on the base of the sphere. And the shadow ring is also smaller because of all the reflected light coming from the mirror ball.

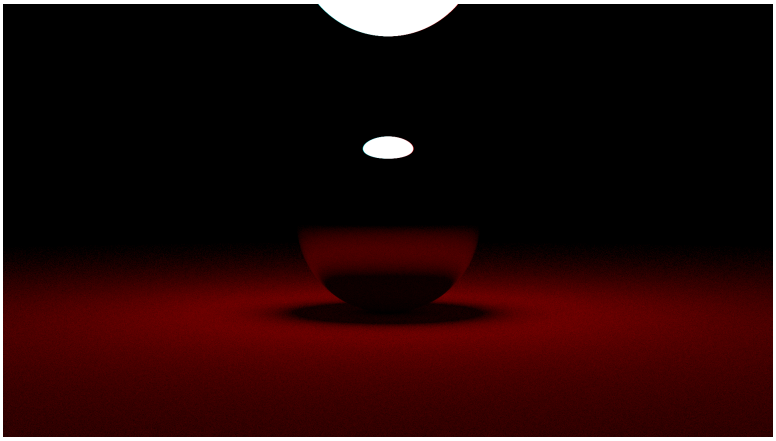


Figure 9.15. Mirror metal sphere with a diffuse sphere and a red colored ground

Finally, in this scene we returned to a Lambertian material, but we inserted in the scene a big quad behind the sphere that acts like a wall. Furthermore, we also hid the emitter outside of view. The point of this scene is to show how matte materials interact with each other. First thing we can notice by looking at the sphere is the slight lighter hue right under the middle longitudinal line, that's because of the rays bouncing off the ground and hitting the sphere. Then, looking at the shadow of the sphere, we can notice that it's bigger and that the fading soft shadow is smaller, that's because the light source sits further away from the sphere then in the other scene shown until now. Finally looking down at the shadow and specifically at the side closer to the wall, we can see that it's clearly of a lighter color instead of a solid black. That's because all the rays bouncing off the wall are illuminating that side.

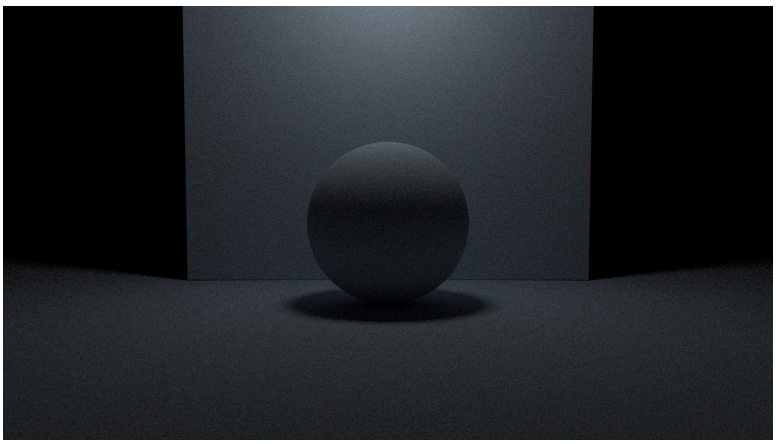


Figure 9.16. Faraway sphere emitter (out of view) with Lambertian sphere, and a Lambertian quad acting as backwall

9.6 Texture mapping

Texture mapping is the process of applying an effect to geometry like bump mapping, or special effects like wet masks or dirt masks. We need a means to map a 3D surface to a 2D texture. What usually happens is that the 3D model comes baked in with texture coordinates, also known as UVs. The name UVs comes from a convention of calling the x and y axes on a texture u and v axes. However our models are procedurally generated and they can get stretched, scaled or rotated. So we need to find a procedural way to convert 3D coordinates to bi-dimensional UVs for any kind of geometry.

9.6.1 Texture Coordinates for Spheres

Texture Coordinates for sphere can follow the convention of spherical coordinates, longitude and latitude. We compute two angles θ and ϕ, with theta having 180 degrees of freedom going from -y to +y, while phi has 360 degrees, going around the y axis (from -x to +z to +x to -z and back to -x). Converting these angles to points on the sphere follows these formulas defined in the get_sphere_uv() function.

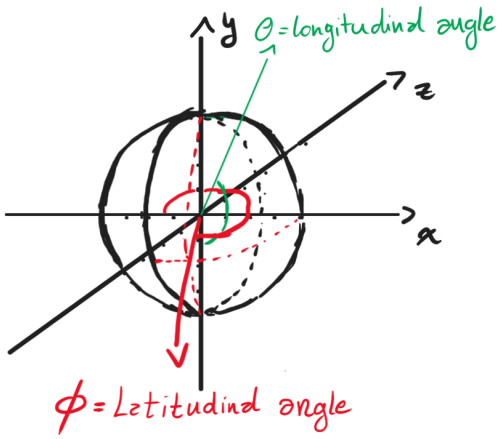


Figure 9.17. Logitudidinal and latitudinal angles

```
static void get_sphere_uv(vec3f& p, float& u, float& v){
    float phi = atan2( y: -p.z, p.x) + pi;
    float theta = acos( x: -p.y);
    u = phi / (2 * pi);
    v = theta / pi;
}
```

Figure 9.18. get_sphere_uv() function form the sphere class

9.6.2 Image mapping and Rendering

Now to render an image on a sphere we firstly have to load it into memory in a handy way. We built a wrapper function that loads raw pixel data into a vector matrix of r,g,b tuples (the I/O process is handled by the stbi library).

```
bool  load_img(vector<vector<pixel_f>>& tuple_img, const string& filename, int& width, int& height){
    //    width;          -- width of the loaded image
    //    height;         -- height of the loaded image
    int bytes_pixel;      // number of bytes per pixel [1,8]
    int bytes_scanline;   // number of bytes per row in image
    int bytes_total;      // number of bytes in the entire image

    auto dir_fname : string  = R"(C:\Raytracing_developing_a_3D_renderer_from_zero\resources\textures\)" + filename;
    auto env : char *  = getenv( VarName: "RTW_IMAGES");

    // FILL INFO
    int query_result = stbi_info( filename: dir_fname.c_str(),  x: &width,  y: &height,  comp: &bytes_pixel);

    // LOAD FILE
    auto file_data : stbi_uc *  = stbi_load( filename: dir_fname.c_str(),  x: &width,  y: &height,  comp: &bytes_pixel,  req_comp: 3);
    if (file_data == nullptr){
        cout << "\nCall to load_img(" << dir_fname << "): Failed to open file" << endl;
        return false;
    };

    // CONVERSION TO VEC3F
    bytes_scanline = width * bytes_pixel;
    bytes_total = bytes_scanline * height;

    int r = 0;       // row counter
    for (int i = 0; i < bytes_total; i+=bytes_scanline, r++){
        vector<pixel_f> scanline = {};
        tuple_img.push_back(scanline);
        int c = 0;   // column counter
        for (int j = 0; j < bytes_scanline; j+=3, c++){
            // stbi_load already loads data as bytes!
            auto color_comp_r = int (*file_data++);
            auto color_comp_g = int (*file_data++);
            auto color_comp_b = int (*file_data++);
            pixel_f pixel = { r: int_to_float( n: color_comp_r),
                              g: int_to_float( n: color_comp_g),
                              b: int_to_float( n: color_comp_b)};
            tuple_img[r].push_back(pixel); // write pixel tuple
        }
    }
    return true;
} // load_png
```

Figure 9.19. load_image() wrapper function, called on image_texture constructor call

Regarding the actual rendering process, we define a unified interface for all texture type objects that instances of class material will require to render color on them.

```
class texture {  // interface
    /**
     *  A common interface for all texture related classes
     */
public:
    virtual ~texture() = default;

    virtual pixel_f color(float u, float v, const vec3f& p) const = 0;
};  // texture interface
```

Figure 9.20. Texture super-class

Finally to render the right pixel on the texture, once the ray trips over the geometry, we call the get_sphere_uv() function and save the UVs in the ray's hit_record. Then the camera will call the material's function to scatter or emit color from the surface handling the texture's color() function call. The call will clamp the UV values to make sure they aren't of the image bounds and then by multiplying the UV coordinates by the width and height of the image it can reference the pixel on the image matrix. And if for some reason a pixel can't be referenced, the color function will return a bright fuchsia color.

```
pixel_f color(float u, float v, const vec3f& p) const override{
    // check h and w if they are set correctly
    //
    if (height <= 0 || width <= 0 || tuple_img.empty()) {
        return BRIGHT_PURPLE;
    }
    // clamping in 0,1 could hold values that equals to w and h
    u = interval( m: 0.0f, M: 1.0f).clamp( x: u);
    v = 1.0f - interval( m: 0.0f, M: 1.0f).clamp( x: v); // v coords are flipped

    auto x = int(u * width);
    auto y = int(v * height);

    if (x < width && y < height) {
        return tuple_img[y][x];
    } else {
        return BRIGHT_PURPLE;
    }
}
```

Figure 9.21. color method implementation of the image_texture class

Figure 9.22. A world map applied as a Texture to a sphere, 1000x1000 texture resolution

9.7 Procedurally Generated Textures

9.7.1 Spacial Texture

Spacial Textures or Constant Color Textures are simple color tints. Whatever the UV coordinate given to the color() function, it will always return the color value set at the creation of the instance.

```
class spacial_texture : public texture {
public:
    // constructor
    spacial_texture(const pixel_f& albedo) : albedo(albedo) {}
    spacial_texture(float r, float g, float b) : albedo(pixel_f{r,g,b}) {}

    // methods
    pixel_f color(float u, float v, const vec3f& p) const override{
        return albedo;
    }

private:
    pixel_f albedo;
};
```

Figure 9.23. simple spacial texture class

9.7.2 Checked Texture

To define a checked texture the approach is pretty simple. We firstly define the two colors we want to render in our checker surface by using spacial texture for definition. Then we take the intersection point, we find the floor of the float value of each component and we scale it to make the squares bigger or smaller depending on the parameter. Finally to decide which color we'll render, we check if the sum of the components is odd or even returning the color assigned to either of the two values.

```
class checker_texture : public texture {
public:
    // constructors
    checker_texture(float scale, shared_ptr<texture> even_tex, shared_ptr<texture> odd_tex) :
                    scale(1.f/scale), even(even_tex), odd(odd_tex) {};
    checker_texture(float scale, const pixel_f& even_color, const pixel_f& odd_color) :
    checker_texture(scale,  even_tex: make_shared<spacial_texture>( albedo: even_color),  odd_tex: make_shared<spacial_texture>( albedo: odd_color)) {};

    // methods
    pixel_f color(float u, float v, const vec3f& p) const override{
        int _x = int(floor(p.x) * scale);
        int _y = int(floor( x: p.y) * scale);
        int _z = int(floor( x: p.z) * scale);
        return (_x + _y + _z) % 2 == 0 ? even->color(u,v,p) : odd->color(u,v,p);
    }

private:
    float scale;
    shared_ptr<texture> even;
    shared_ptr<texture> odd;
}; // checker_texture class
```

Figure 9.24. Checked texture class

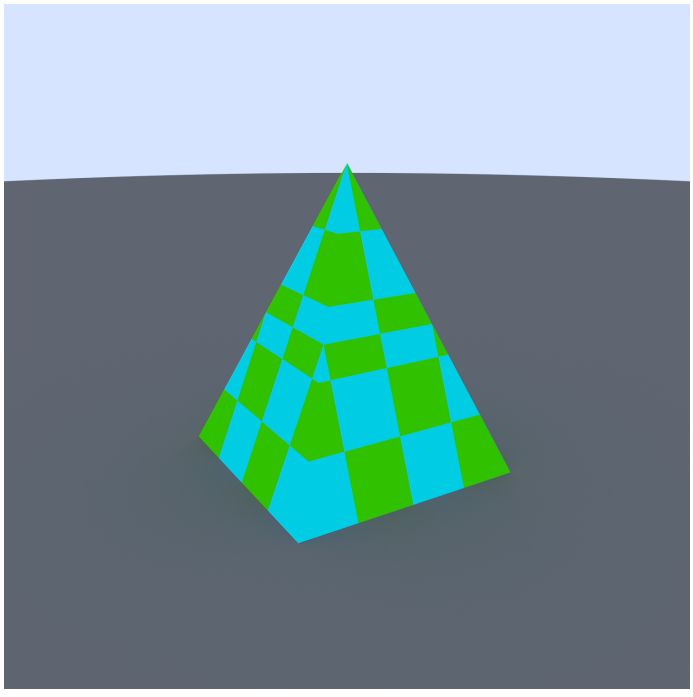


Figure 9.25. The Checked texture applied to a pyramid

9.7.3 Perlin Noise

The Perlin Noise is a technique invented by Ken Perlin in 1985 and presented at SIGGRAPH in a paper called "An Image Synthesizer". This function generates a lattice-based noise, meaning that it defines a 3D grid in the world, where for each corner of the lattice cell a random unit vector is generated, also called gradient. The problem with keeping a tri-dimensional lattice of 3D vectors is that it can quickly become expensive in therms of memory. So to solve this problem, we'll save a big enough array of 3D vector and then to pick a random vector we will pass through an hash function. The trick to get a uniform blurred noise is to generate the gradient vectors uniformly in the unit sphere without too many repetitions or obvious repetitive schemes. Finally we have to interpolate these vectors so that we can return a float value to use as color on our object. To do so we'll pass the 8 vectors of the lattice cell through a tri-planar interpolation function that will do the dot product between each couple of gradient vector, returning a float value at the end.

In this render is clearly visible how my implementation of the Perlin noise isn't perfect, as the lattice steps are clearly visible, my guess is that the tir-planar interpolation is not completely correct. But I still think is worth showing the results.

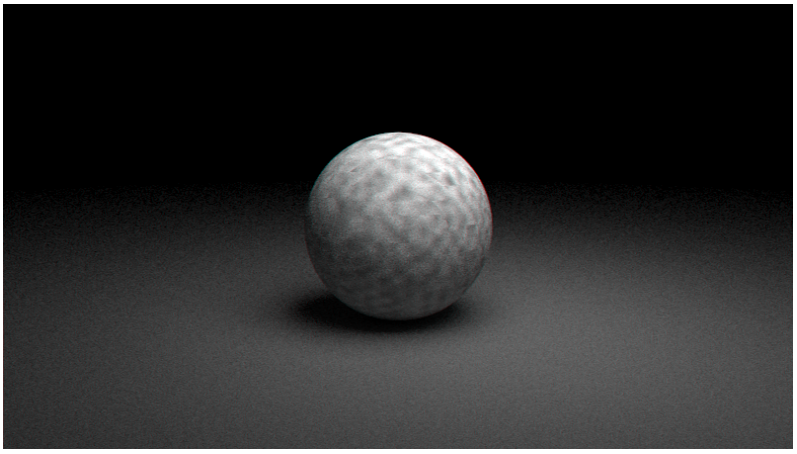


Figure 9.26. sphere with a Perlin noise texture applied with smoothstep interpolation

Chapter 10

The Cornell Box

The Cornell Box developed by the department of Compute Graphics of Cornell University, is a test aimed at determining the accuracy of any rendering software. It was a controlled physical environment where lighting, geometry, and material reflectance properties where measured with specialized equipment. The point of it, was to have a reference point to compare the accuracy of physically based renderings with reality. Believing that Computer Graphics simulation would never become close to reality without modeling the physics of light reflection and propagation correctly. The original Cornell Box was first simulated in 1984 by Cindy M. Goral, Kenneth E. Torrance, and Donald P. Greenberg and it was originally unobstructed by objects.

The Box is designed like this to show the diffuse interreflection, meaning the scattering of light from a surface to another. For example some light from the red wall should reflect on the green or bounce on the white wall. So part of the back wall should look red or green.

10.1 Analysis and results

The following is the Cornell Box has been rendered trough the current iteration of our Ray Tracer. This render took approximately 1h to render on a single core with a resolution is 1000x1000, each pixel we sampled had 500 rays with 100 bounces each.

While the image immediately underneath has been rendered through Blender's appleseed plugin at 500x500 pixels and with the same amount of rays and bounces. This render took just a few milliseconds to render on my GPU with tensor cores through the DirectX12 API.

Putting aside the obvious lack of integration of our render let's compare the image graphic render Let's start from the overall look. Our render less brighter and more noisy, that's because of a lack of shadow rays and Importance Sampling. But the light, shadows and scattering are accurate. Now let's scan the image top to bottom. Starting from the light we can notice how our render bleeds light directly at the sides of the quads, just like the other render. If we look at the side and back walls, soft lights interact accurately with colored walls, making brighter spots at the center of the walls closer to the light. Also the corners of the box are darker. Moving down to the taller box, we can see how the green wall reflects it's tint to the exposed left side of the tall box, as well as coloring the shadow at the bottom green, just like the other render. Another beautiful feature is to notice how the shadow

of the tall box has lighter spots were the back wall illuminates that side. A visible defect is that the shadows are too soft but that could be related to the overall noise of our render. Finally bringing our attention to the front cube, ad for the other box, it correctly shows a tint on its right side, as well as its shadow. Notice also how along the left front edge is brighter then the right edge because of light bleeding. Overall the our Cornell box looks fairly accurate.

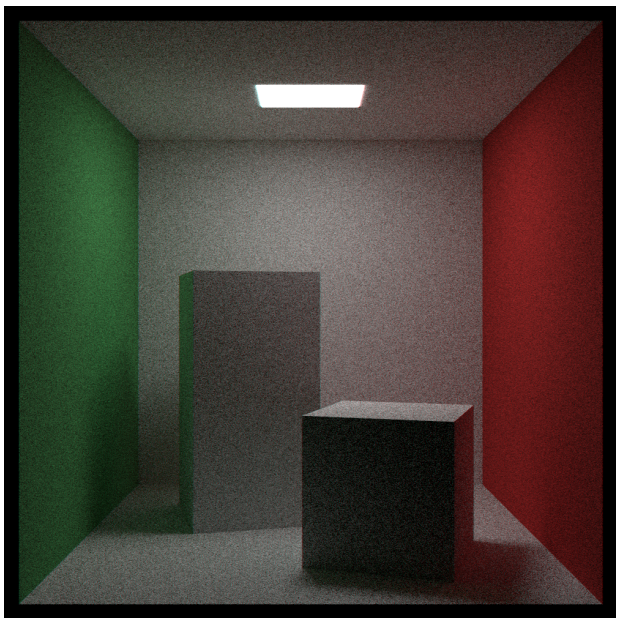


Figure 10.1. The Cornell box with the distinctive two slightly rotated boxes rendered by our render

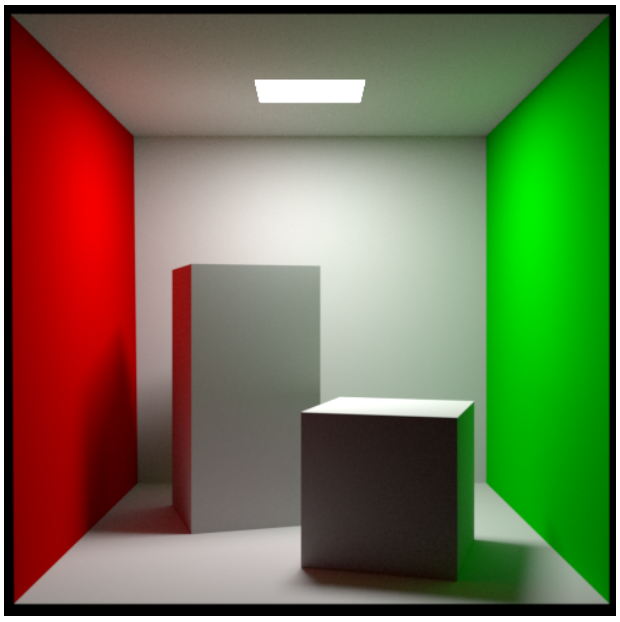


Figure 10.2. The Cornell box reproduced and rendered through blender's appleseed plugin

Chapter 11

Some complex scenes.

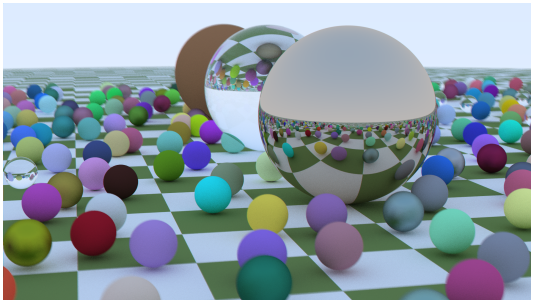


Figure 11.1. Randomly positioned spheres in space, with checked floor texture, water refraction, and metal reflection. Defocus blur, No light.

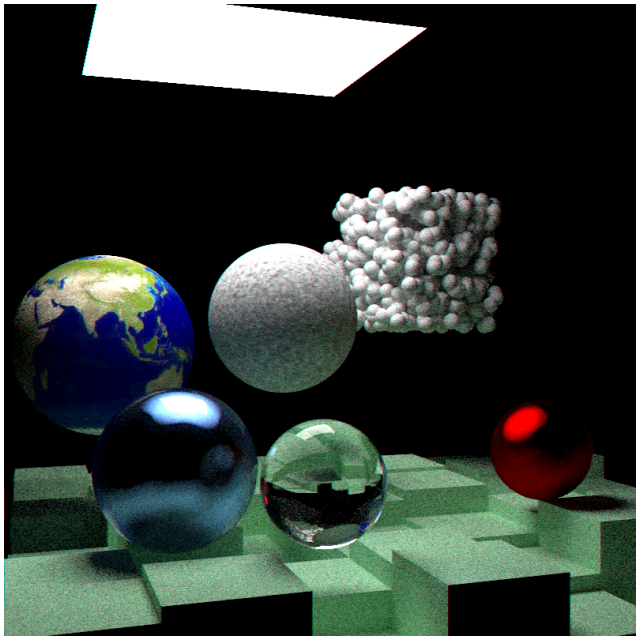


Figure 11.2. The most complex scene I could render, with all the components I've described until now, it took approximately 8 hours to render.

Chapter 12

Next possible steps and conclusions

The next obvious steps necessary to increase the quality and speed of the rendering would be:

- Better probability math library: Most artifacts seen in these renders, from the Perlin noise to the grid clearly visible in the Cornell Box are related to the basic implementation of probability routines.
- Monte Carlo Integration: Implementing the Monte Carlo integration in the camera render() method loop to remove square like artifacts in renders with low lights, like in the Cornell Box.
- Shadow Rays: By Implementing Shadow rays We should be able to substantially reduce noise in the image, but also generate sharper shadows in scenes with multiple intense light sources.
- Importance Sampling: Alongside Shadow rays We could implement a method to send more rays towards the light source, again to reduce noise and increase visual clarity.
- Glossy surfaces, The Phong Model: A very accurate material model to render glossy objects like shiny plastic.
- BSDF or Bidirectional Scattering Distribution Function: A function that returns the amount of light reflected by any surface to the viewing direction for any incident light, it can also be used to better fit and define scattering or reflecting behaviors for objects or textures.
- Better triangle intersection routine: to quicken the render time for complex scenes We need to hack the geometrical solution used to make triangles and quad intersection possible.
- Complex Mesh rendering: with a quicker renderer We can render models with high triangle mesh count, currently a wrapper to load .ply models in the renderer is present, but the render times are too long.
- Better fitting Bounding Volumes: With more complex triangle meshes it's necessary to upgrade the Bounding volume class to better fit a complex model.

- Tri-planar UV mapping: a simple technique to place a bi-dimensional texture to any kind of model.
- Multi-threading: Better hardware integration is necessary to take advantage of any modern machine completely, either by using all the CPU cores or by integrating Open/GL to render through a GPU.

As it stands the current iteration of this render is a competent attempt, that shows remarkable accuracy for how simple the code is. And personally I believe that with light emitting materials, scene renders are rather beautiful and photorealistic.

At the end of this journey I was able to learn a lot, and understand the massive scale of this field better. I feel like I only scraped the surface, and it got me even more curious to delve deeply into this field. I hope to be able to go through with these improvements and to cultivate this passion so much so that it could become a job in this field. This experience has been deeply humbling. Every step of the way I was doubtful of the quality of my work. But, in the end, with the time I was given I managed to do quite a lot. At the end of this journey I'm even more hungry to do and learn more.

Bibliography

[1] "Ray Tracing (Graphics)", Wikipedia, Wikimedia Foundation, 2024-10-04, https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

[2] "Cornell box", Wikipedia, Wikimedia Foundation, 2024-08-19, https://en.wikipedia.org/wiki/Cornell_box

[3] "Visibility (geometry)", Wikipedia, Wikimedia Foundation, 2024-08-18, https://en.wikipedia.org/wiki/Visibility_(geometry)

[4] "What is Computer Graphics", Cornell University, 1998-04-15, https://www.graphics.cornell.edu/online/tutorial/

[5] "The Cornell Box - History", Cornell Univeristy, 1998-01-02, https://www.graphics.cornell.edu/online/box/ https://www.graphics.cornell.edu/online/box/history.html

[6] Peter Shirley, Trevor David Black, Steve Hollasch, "*Ray Tracing in One Weekend Series - Ray Tracing: The Next Week*", v4.0.1, 2024-08-30, https://raytracing.github.io/books/RayTracingTheNextWeek.html

[7] Peter Shirley, Trevor David Black, Steve Hollasch, "*Ray Tracing in One Weekend Series - Ray Tracing In One Week*", v4.0.1, 2024-08-30, https://raytracing.github.io/books/raytracinginoneweekend.html

[8] "*ScratchPixel 4.0 - Rendering an Image of a 3D Scene*" www.scratchapixel.com https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/computer-discrete-raster.html

[9] "*ScratchPixel 4.0 - The Pinhole Camera Model*" www.scratchapixel.com https://www.scratchapixel.com/lessons/3d-basic-rendering/3d-viewing-pinhole-camera/how-pinhole-camera-works-part-1.html

[10] "*ScratchPixel 4.0 - Ray-Tracing: Rendering a Triangle*" www.scratchapixel.com https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful.html

[11] "*ScratchPixel 4.0 - Introduction to Acceleration Structures*" www.scratchapixel.com https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-acceleration-structure/introduction.html

[12] "*ScratchPixel 4.0 - Procedural Generation of virtual worlds*" www.scratchapixel.com https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise.html

[13] Haines, Akeninie-Moller "*Ray Tracing Gems High-Quality and Real-time rendering with DXR and other APIs*", Apress, 2019 https://link.springer.com/book/10.1007/978-1-4842-4427-2

[14] Alacron, McGuire, "*Ray Tracing From the 1980's to Today An Interview with Morgan McGuire, NVIDIA*", Nvidia Developer, 2019-03-19, https://developer.nvidia.com/blog/ray-tracing-from-the-1980s-to-today-an-interview-with-morgan-mcguire-nvidia/

[15] François Beaune et al. "*a*ppleseedh1q/blenderseed", 2018, https://github.com/appleseedhq/blenderseed