Ludwig-Maximilians-Universität München

Your Own Ray Tracer In C++

Thesis by Kasimir Rothbauer

Study Programme: Advanced Topics in Computer Graphics

Ludwig-Maximilians-Universität München, Kommunikationssysteme und Systemprogramierung

Munich, 2026-02-08

supervised by
Dr. Rubén Jesús García-Hernández

Contents

List of Figures

Chapter 1

Introduction

The book series are a tutorial to create a ray tracer from scratch in C++. My work implemented this tutorial and documented the progress with graphics of the rendered images.

1.1 Overview

I present simple explanations of algorithms implemented. The main code will be in C++.

- remark
- ppm image format
- viewport
  - bottom top left to bottom right
  - u, v helper vectors

1.2 Ray tracing from Eye

- take ray and calculate where it will hit something
- if hit calculate color
  - depending on material the color will have a different value
  - either bounce or apply color
- if ray goes to infinity then use background color
- Linear interpolation
  - Remark

1.3 Object Creation

1.3.1 Adding a Sphere

- spheres
- remark: sphere formula

1.3.2

1.4 Materials

1.4.1 Lambertian Surfaces

- any incoming ray will either be absorbed or reflected
- if reflected new ray in random direction with cosine
- if absorbed color at that point
- more bounce = more dark
- depending on material

1.4.2 Shadow Acne

When a ray bounces of the material it might happen, that rounding errors in the floating point arithmetic cause a ray to bounce inside the material. This may cause the

1.4.3 Metals

- either absorb the ray or reflect
- this time reflection is always incoming ray angle = outgoing ray angle
- when creating fuzziness we adjust the outgoing ray angle
- create a new random vector from outgoing vector and add them figi
- more fuzz = larger vector

1.4.4 Glass

- if ray hits glass ray will be refracted
- depends on angle of refraction
- Snells law
- if putting air bubble inside glass there is even inter reflection

Chapter 2

Algorithms implemented

I present simple explanations of algorithms implemented. The main code will be in C++.

2.1 Ray tracing from Eye

2.2 Materials

2.2.1 Lambertian Surfaces

2.2.2 Glass

2.2.3 Metals

2.2.4 Fuzziness

2.3 The student should prepare a written report about his topic following the following schema:

- Start with the introduction: **What is your topic and what have other researchers done in that topic, with references**
- Add a 2nd section with **In detail description of the algorithms you want to implement**
- Implement your algorithms in your engine of choice
- Start testing and document the issues you have **this will become section 3**
- Fix the issues and document your solution **this will become section 4**
- Document further possible work which could enhance your results **This will become section 5: Future work.**

Chapter 3

Ray Tracing in a Weekend

by S. H. Peter Shirley Trevor David Black [1] on the website https://raytracing.github.io

The book series are a tutorial to create a ray tracer from scratch in C++ for acceptable performance.

3.1 C++ References

- https://makefiletutorial.com/
- https://www.learncpp.com/cpp-tutorial

3.2 Ray calculation

(i) Calculate the ray from the "eye" through the pixel,
(ii) Determine which objects the ray intersects, and
(iii) Compute a color for the closest intersection point.

- **Viewport**: The canvas of the camera view(Gray plane in Figure 45)
- To make the image scalable, keep **aspect ratio** fixed, and calculate the height when the width changes. The **aspect ratio** can't always be achieved as the height can't be *real valued* but an integer rounded down.
- **lerp**: short for *linear interpolation*

$$\text{blendedValue} = (1 - a) \cdot \text{startValue} + a \cdot \text{endValue}$$

Please **note** that the coordinate system between the *viewport* and the *image* differ (Figure 2 and Figure 45). A way to calculate our way out of this is presented in the code below.

The ray formula has the form

$$\boldsymbol{P}(t) = \boldsymbol{A} \cdot t\boldsymbol{b} \tag{1}$$

and can be seen in Figure 1.

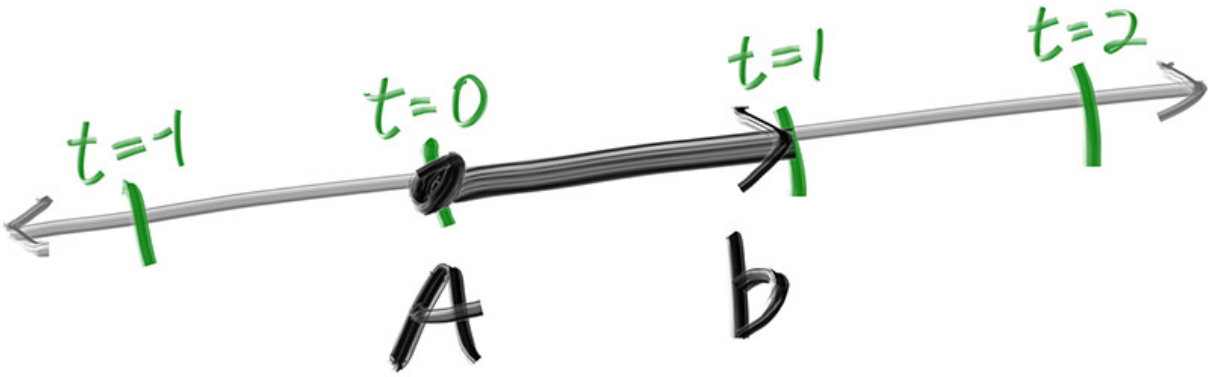


Figure 1: Ray formula: A is the origin position and b is the direction. t is the scalar which scales b [1]

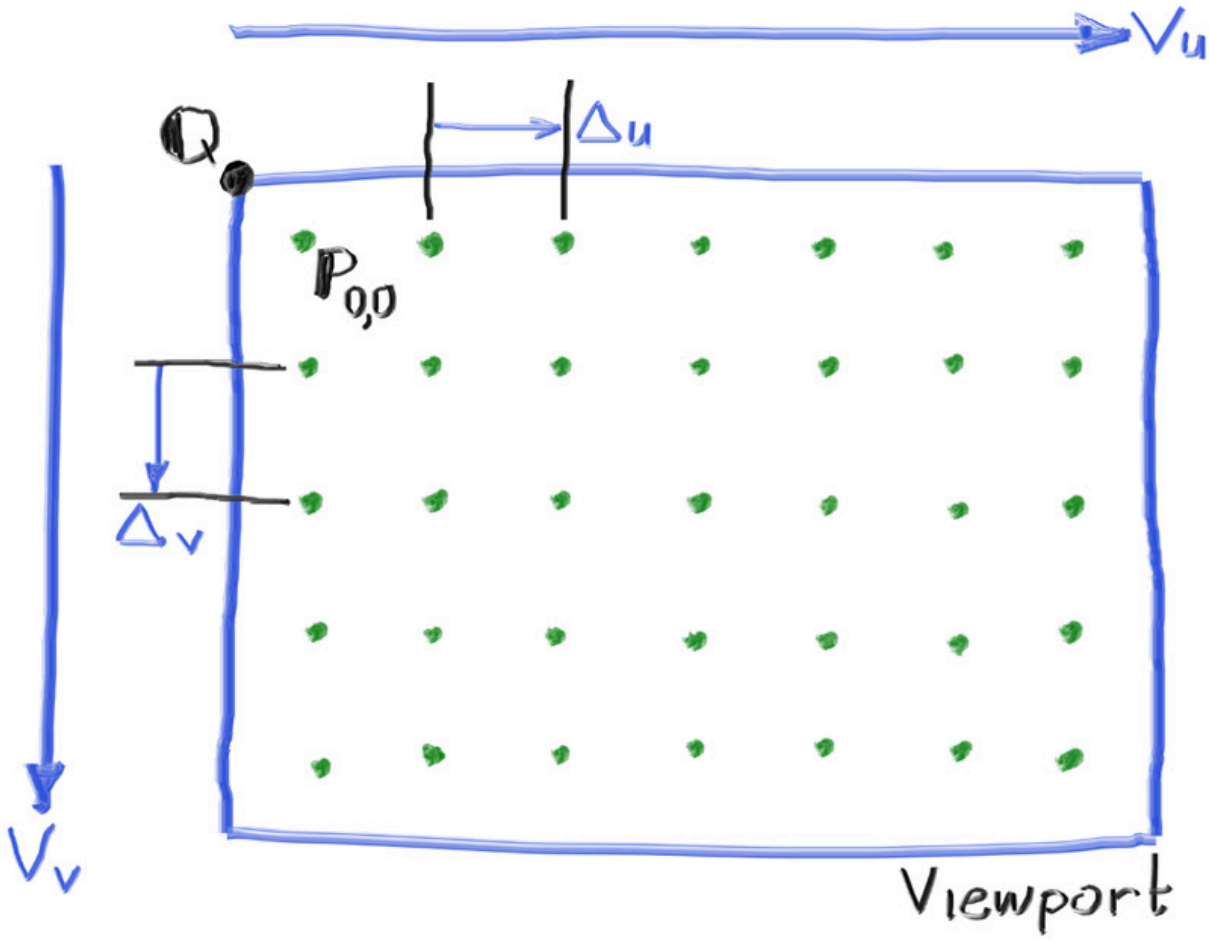


Figure 2: Viewport as seen from the *origin* or *eye*. The pixels are the green dots. v_u and v_v are helper vectors to navigate the viewport better [1].

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

#include <iostream>

color ray_color(const ray& r) {
    vec3 unit_direction = unit_vector(r.direction());
    auto a = 0.5*(unit_direction.y() + 1.0);
    // linear interpolation between max color value of eg. red: 1.0 = full red and
0.0 = white
    return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
}

int main() {

    // Image
    auto aspect_ratio = 16.0 / 9.0;
    int image_width = 400;

    // Calculate the image height, and ensure that it's at least 1.
    int image_height = int(image_width / aspect_ratio);
    image_height = (image_height < 1) ? 1 : image_height;
```

```
    // Camera
    auto focal_length = 1.0;
    auto viewport_height = 2.0;
    auto viewport_width = viewport_height * (double(image_width)/image_height);
    auto camera_center = point3(0, 0, 0);

    // Calculate the vectors across the horizontal and down the vertical viewport
edges.
    auto viewport_u = vec3(viewport_width, 0, 0);
    auto viewport_v = vec3(0, -viewport_height, 0);

    // Calculate the horizontal and vertical delta vectors from pixel to pixel.
    auto pixel_delta_u = viewport_u / image_width;
    auto pixel_delta_v = viewport_v / image_height;

    // Calculate the location of the upper left pixel.
    auto viewport_upper_left = camera_center
                             - vec3(0, 0, focal_length) - viewport_u/2 -
viewport_v/2;
    auto pixel00_loc = viewport_upper_left + 0.5 * (pixel_delta_u +
pixel_delta_v);

    // Render

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

    for (int j = 0; j < image_height; j++) {
        std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' <<
std::flush;
        for (int i = 0; i < image_width; i++) {
            auto pixel_center = pixel00_loc + (i * pixel_delta_u) + (j *
pixel_delta_v);
            auto ray_direction = pixel_center - camera_center;
            ray r(camera_center, ray_direction);

            color pixel_color = ray_color(r);
            write_color(std::cout, pixel_color);
        }
    }

    std::clog << "\rDone.                 \n";
}
```

Figure 3: Resulting image of the above code

3.3 Adding a Sphere

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

Is the equation for a sphere. If this equation is fulfilled, a *point* $\boldsymbol{P} = (x, y, z)$ lies on the sphere with radius r. A point is inside the sphere if $x^2 + y^2 + z^2 \leq r^2$ and outside the sphere if $x^2 + y^2 + z^2 \geq r^2$.

Using arbitrary Point $\boldsymbol{C}$ (and not $(0, 0, 0)$) for the center of the sphere we get:

$$(\boldsymbol{C}_x - x)^2 + (\boldsymbol{C}_y - y)^2 + (\boldsymbol{C}_z - z)^2 = r^2$$

Checking whether a ray hits our sphere, we need to check if the ray equation Figure 1 holds for any t.

So for a point $\boldsymbol{P}$ the formula is:

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

Filling in the Formula (1)

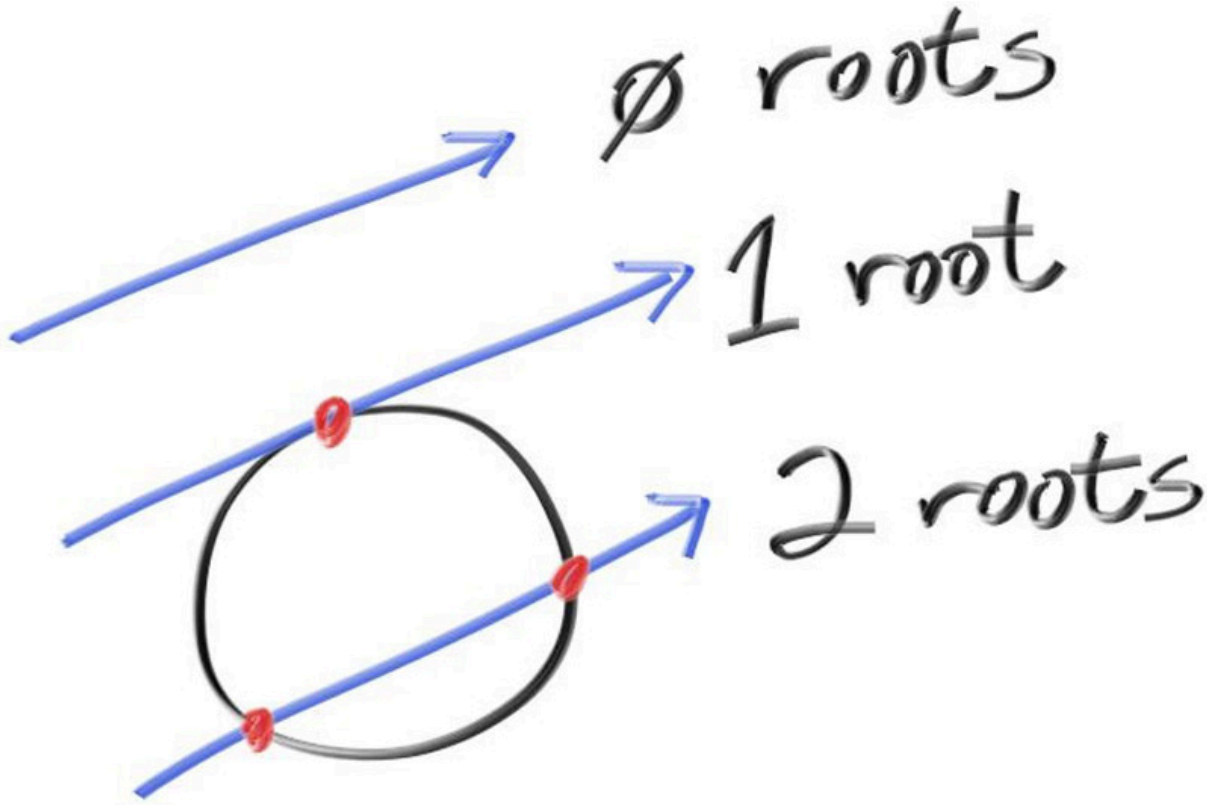


Figure 4: Three solutions to the quadratic solution of a ray passing a sphere [1]

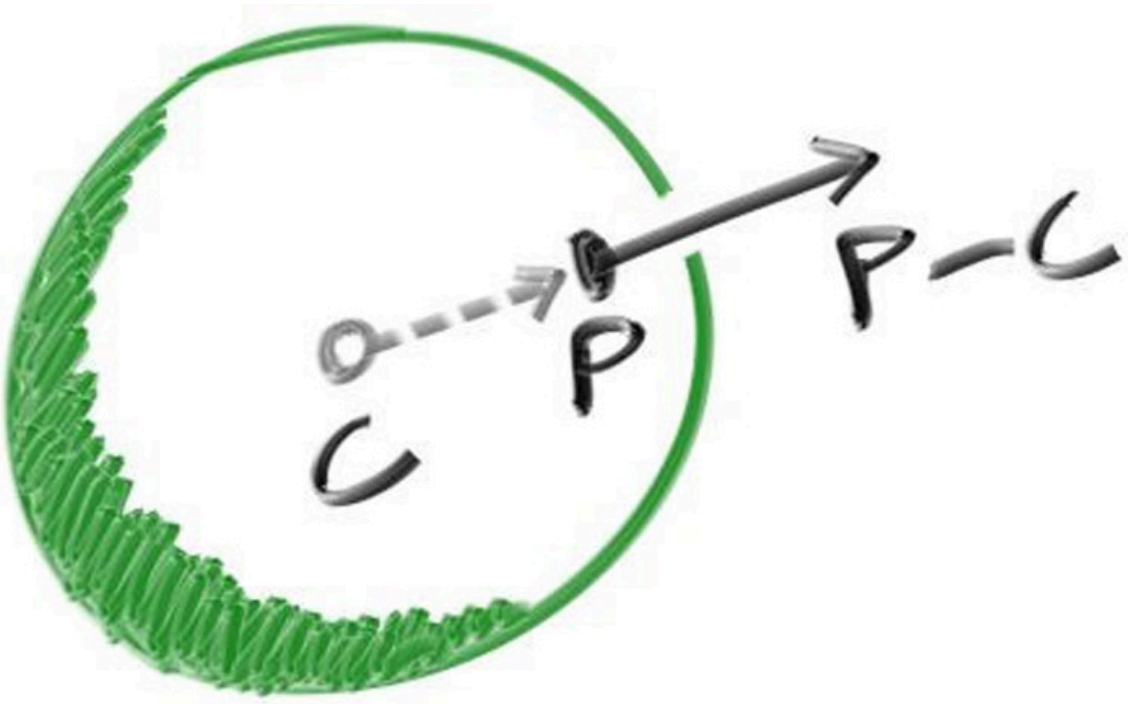


Figure 5: The normal of a sphere [1].

Figure 6: Using rays and calculating the normals from the hitting rays on the sphere we can color in the surface according to the normalized surface vector.

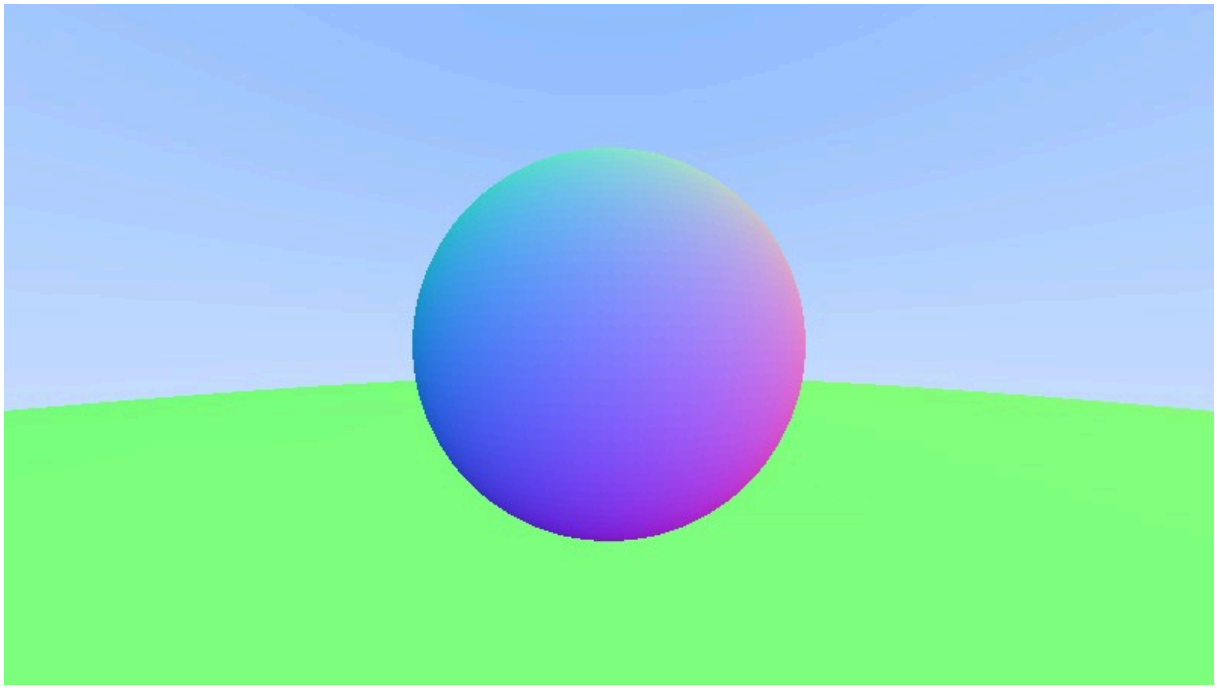


Figure 7: Adding a 100 unit radius sphere at 100.5 units below. The blueish sphere is sitting exactly on top of the green sphere (radius 0.5)

Figure 8: Antialiasing at work.

3.4 Diffuse Materials

Creating a random point on a unit sphere is surprisingly complicated. The method used for our approach is to generate a unit square, and then discard any random vector that lies outside the sphere (see Figure 9).

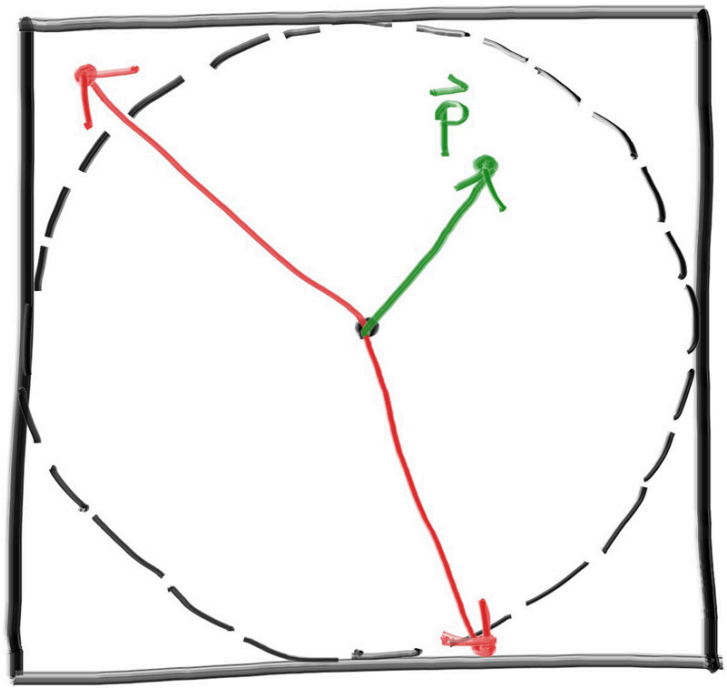


Figure 9: Two vectors were rejected before finding a good one (pre-normalization) [1].

We then normalize the random vector (pointing on the surface of the unit sphere) and check whether it points inside the material (using dot product pos or negative)

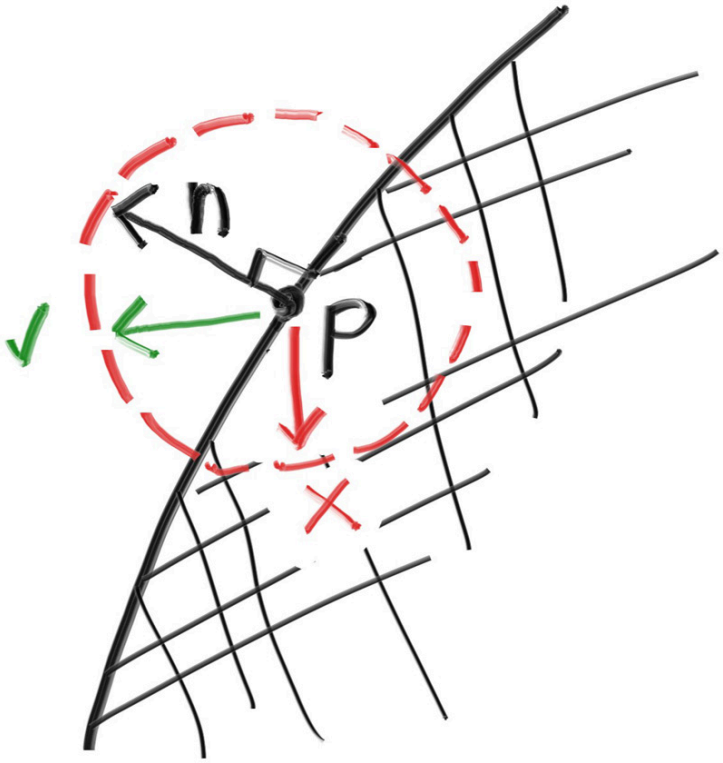


Figure 10: Check if random vector points inside the surface.

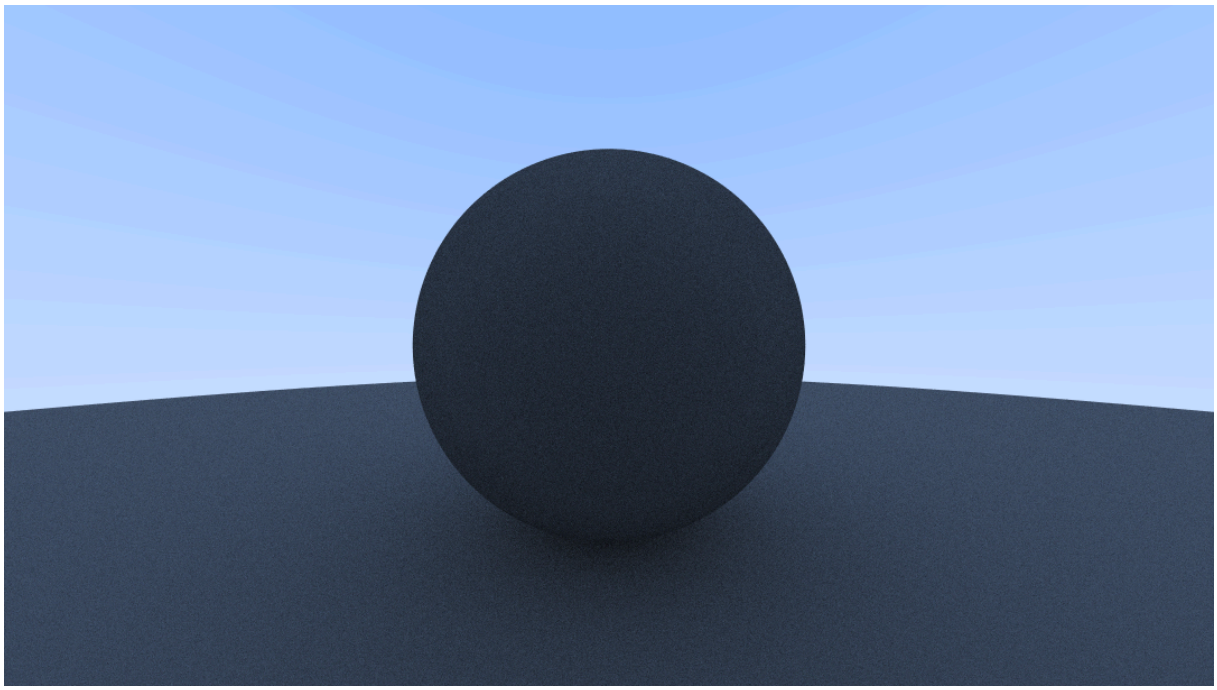


Figure 11: Diffuse sphere using a normal distribution

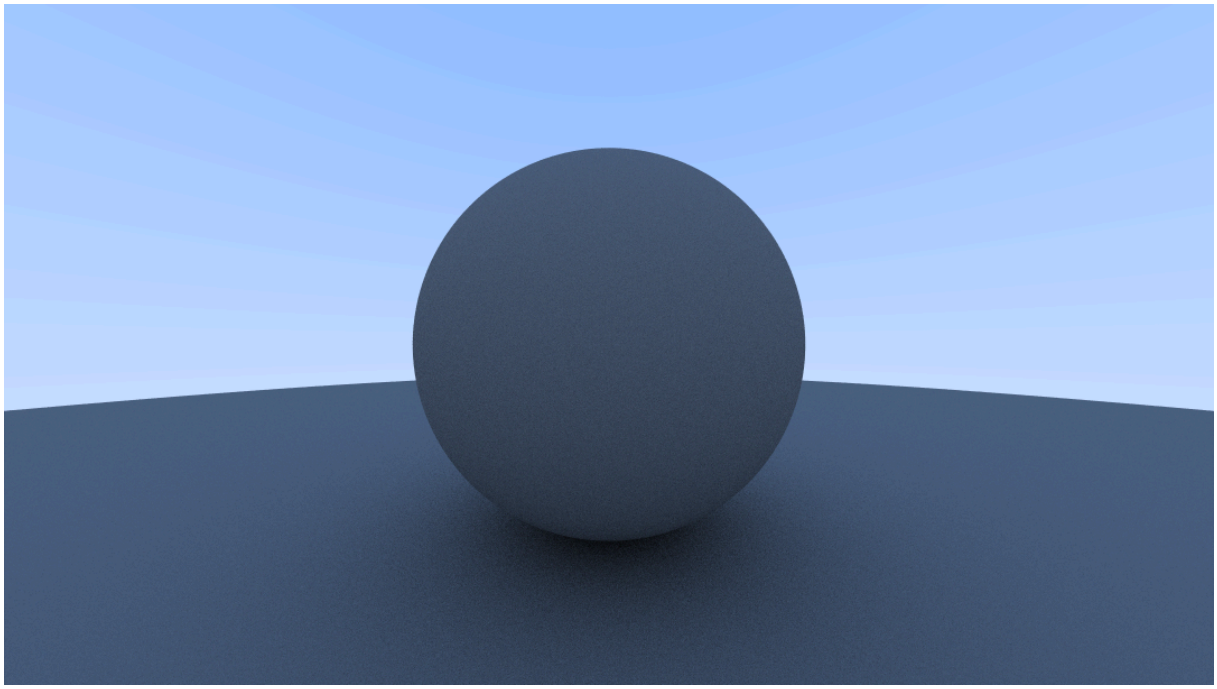


Figure 12: Rendered sphere using lambertian distribution (2). Note the now blueish tainted spehere surfaces from the sky and the more pronounced shadows.

Remark 3.1. (Lambertian Distribution):

$$I = I_0 \cos(\theta) \tag{2}$$

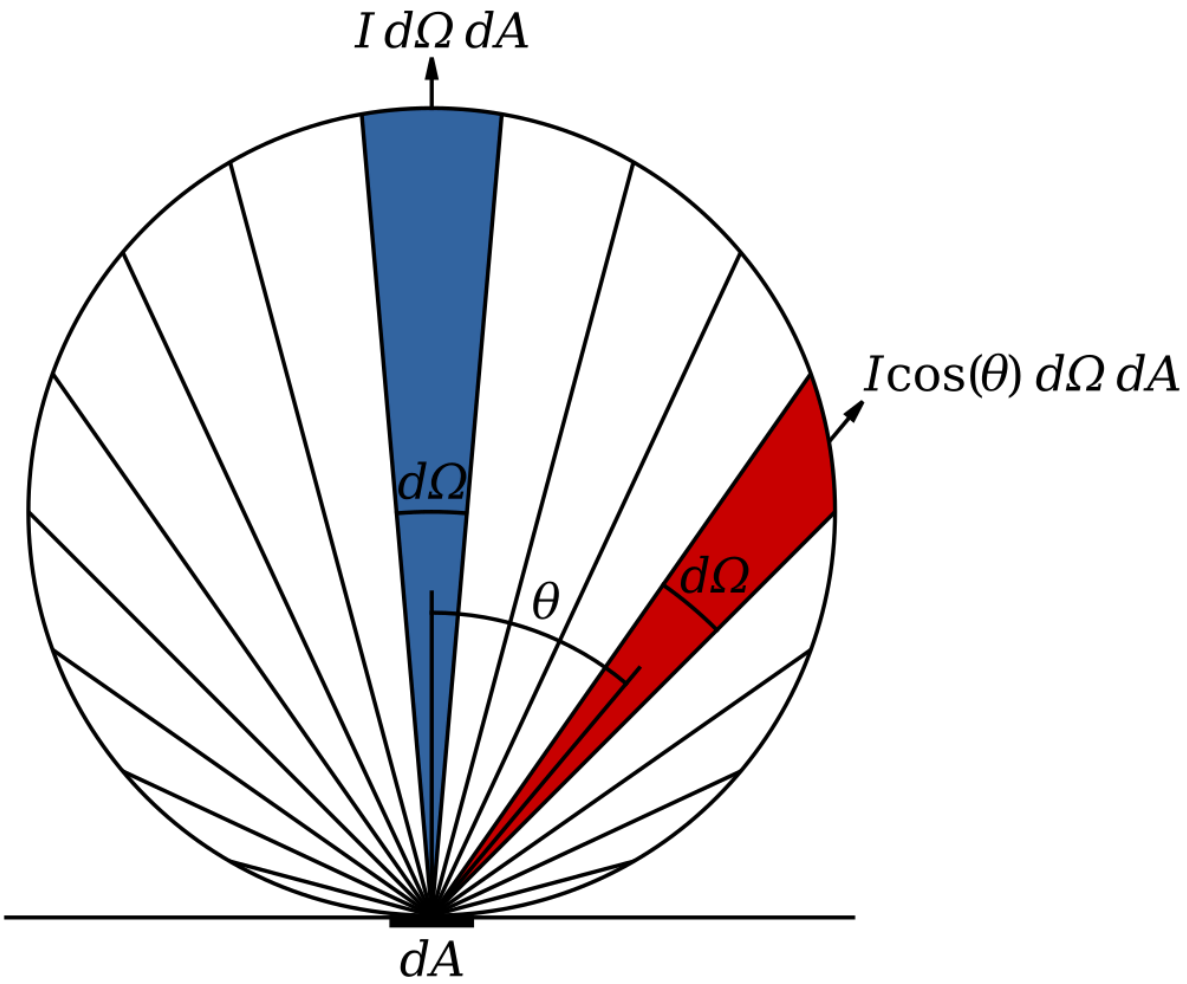


Figure 13: Lambertian Distribution as in (2) by Inductiveload - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=5290437

3.5 Metals

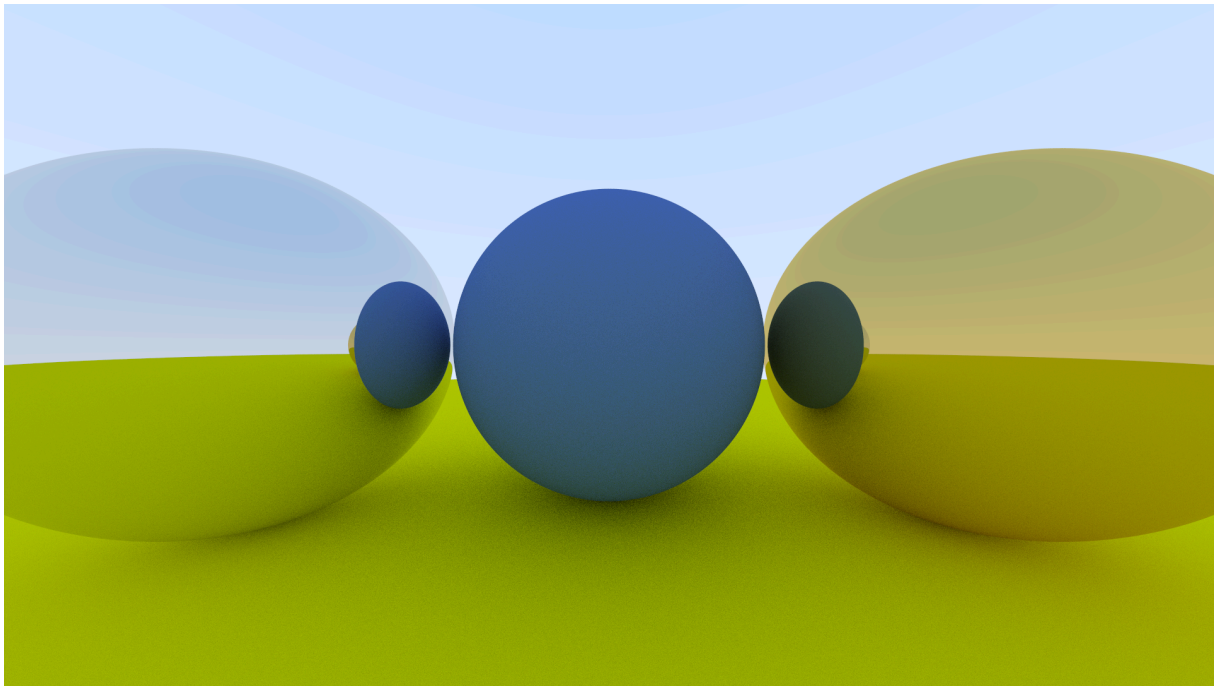


Figure 14: Adding a reflective sphere.

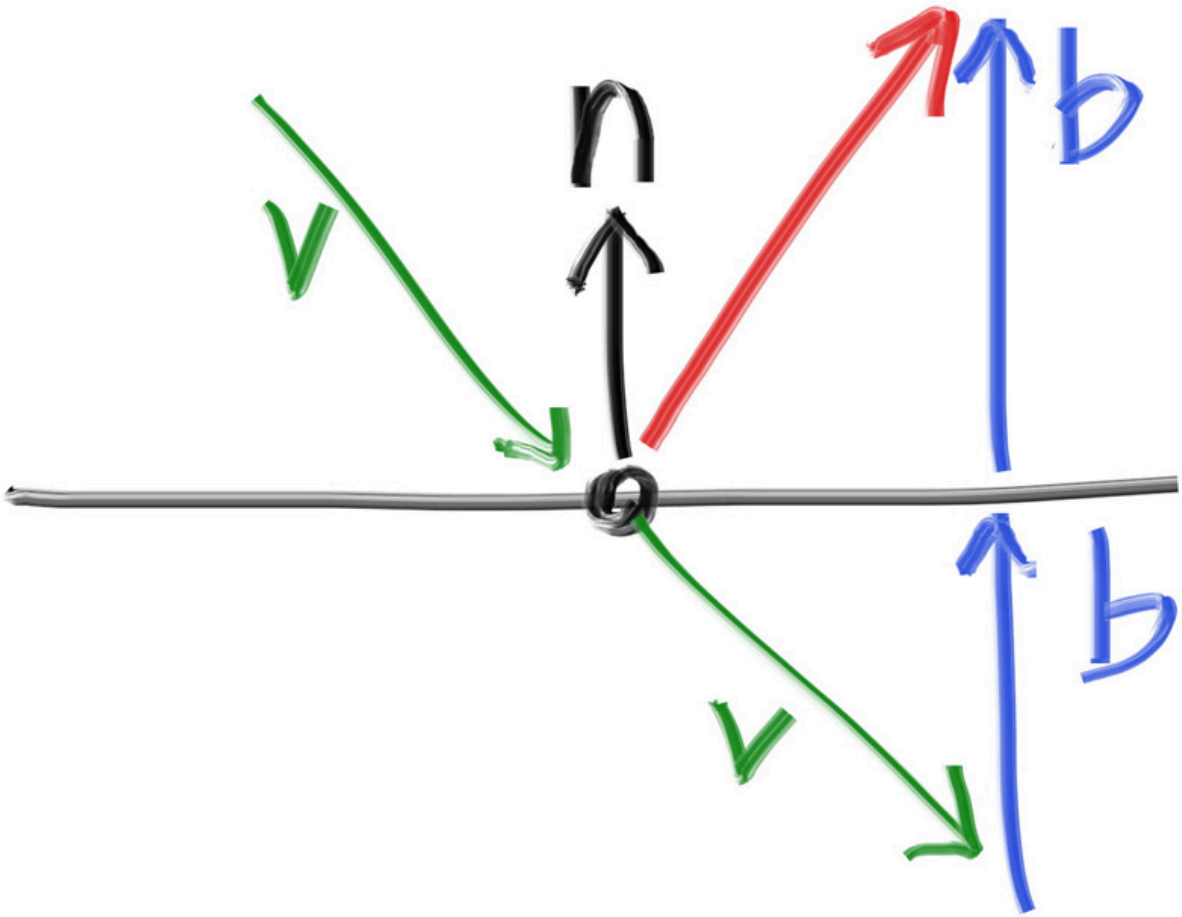


Figure 15: Reflection vector for metallic surfaces by S. H. Peter Shirley Trevor David Black [1]

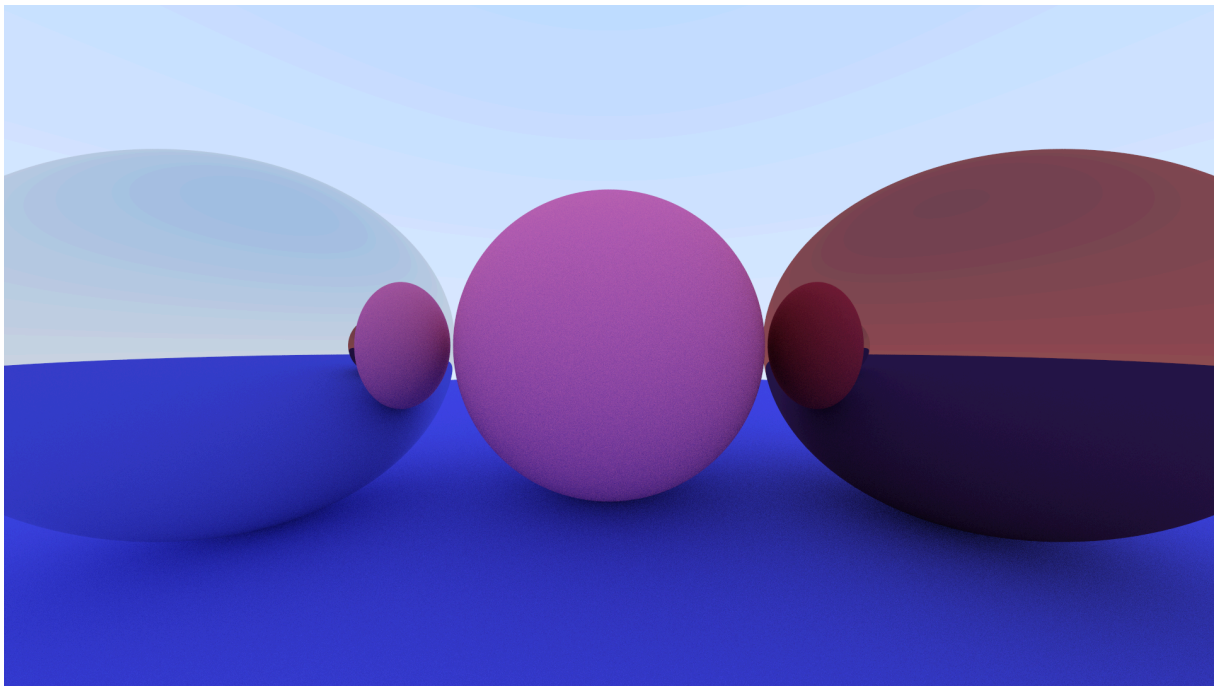


Figure 16: Prettier spheres

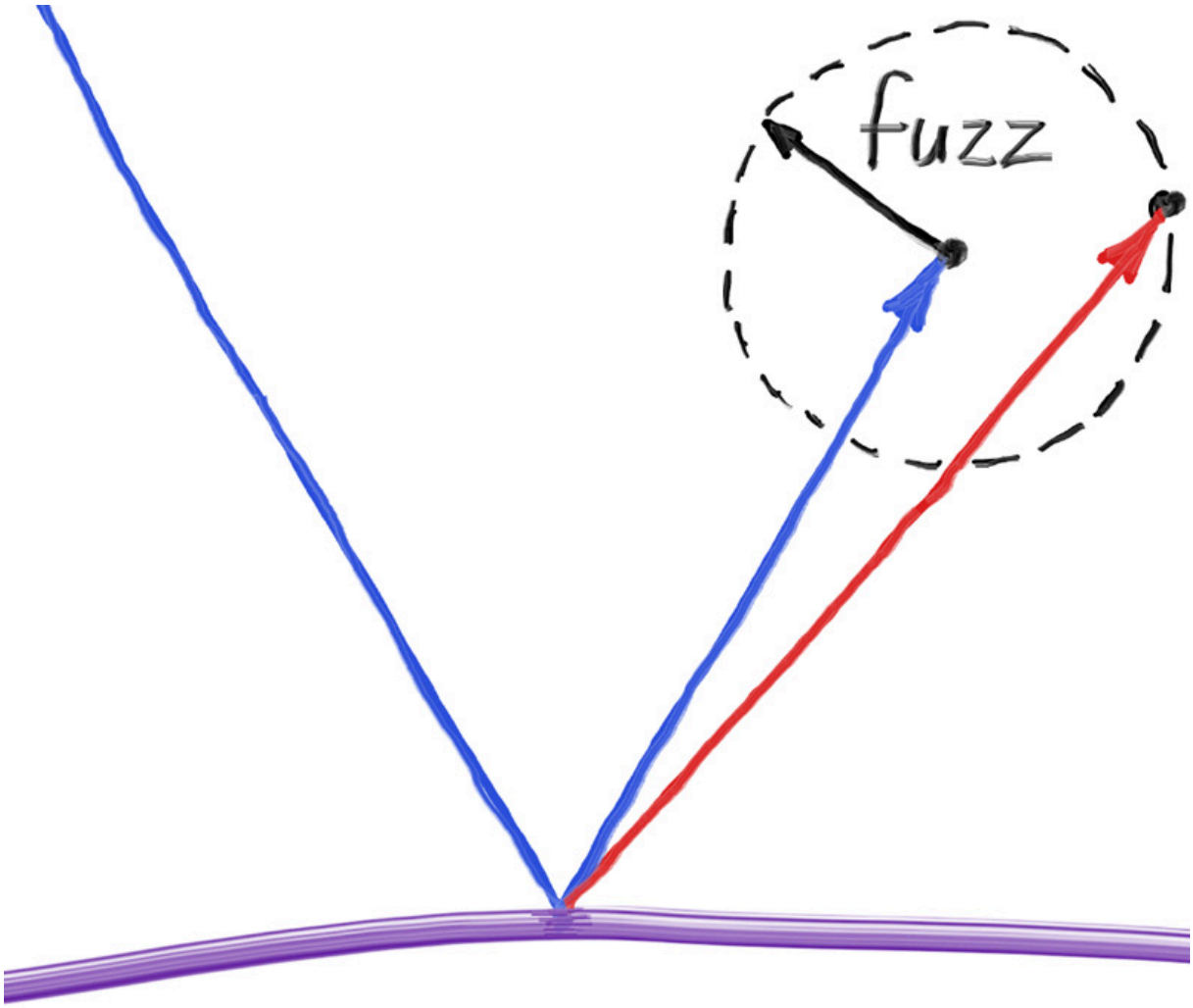


Figure 17: Creating fuzziness by adding a random vector to our reflected ray.

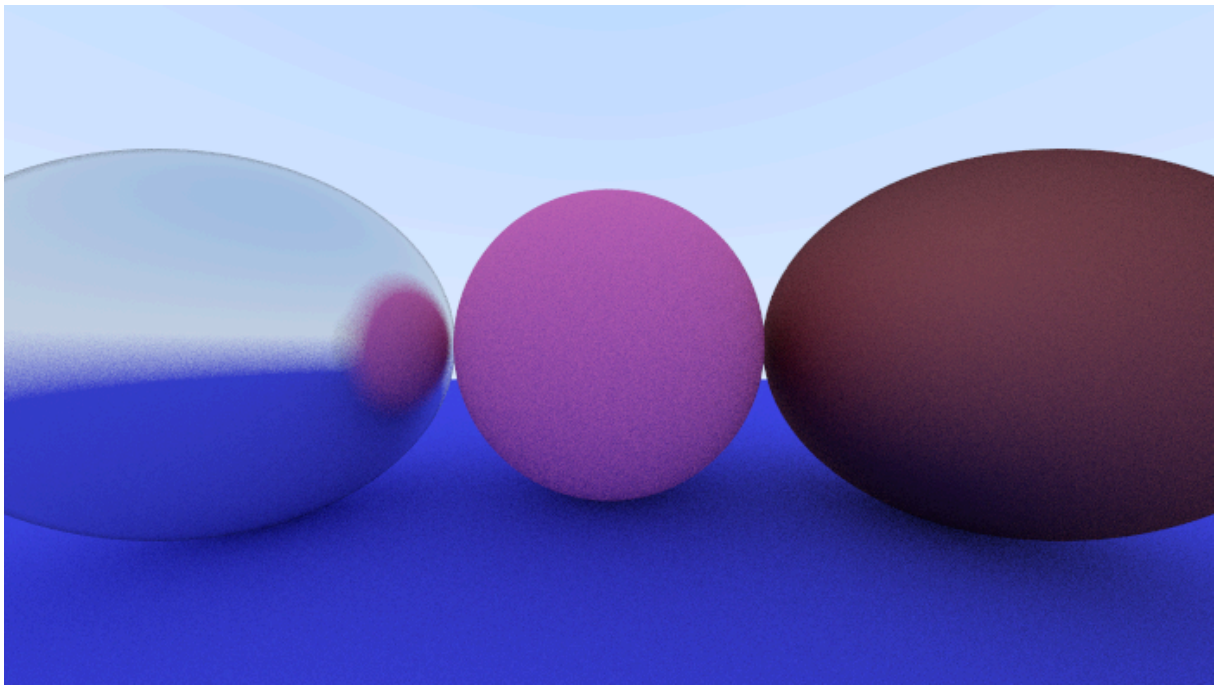


Figure 18: Fuzzed spheres

3.6 Dielectrics

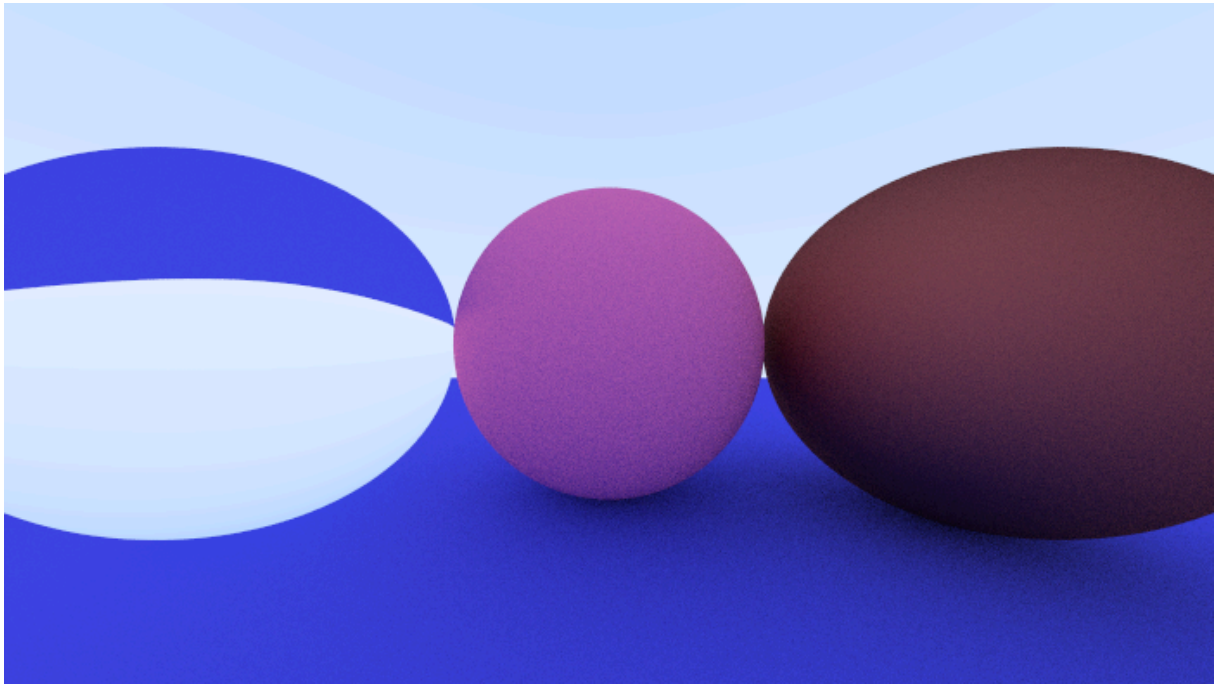


Figure 19: Fully reflective glass sphere with $\eta = 1.4$

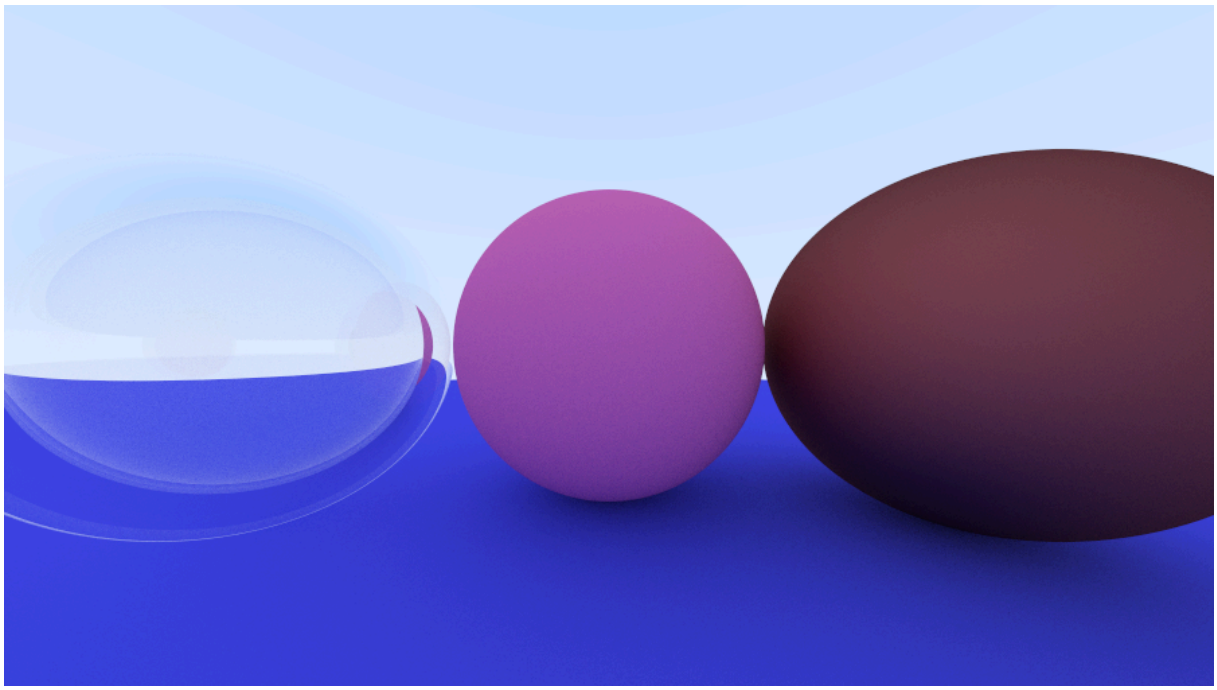


Figure 20: Glass sphere with an air bubble inside. Using refraction and internal reflection.

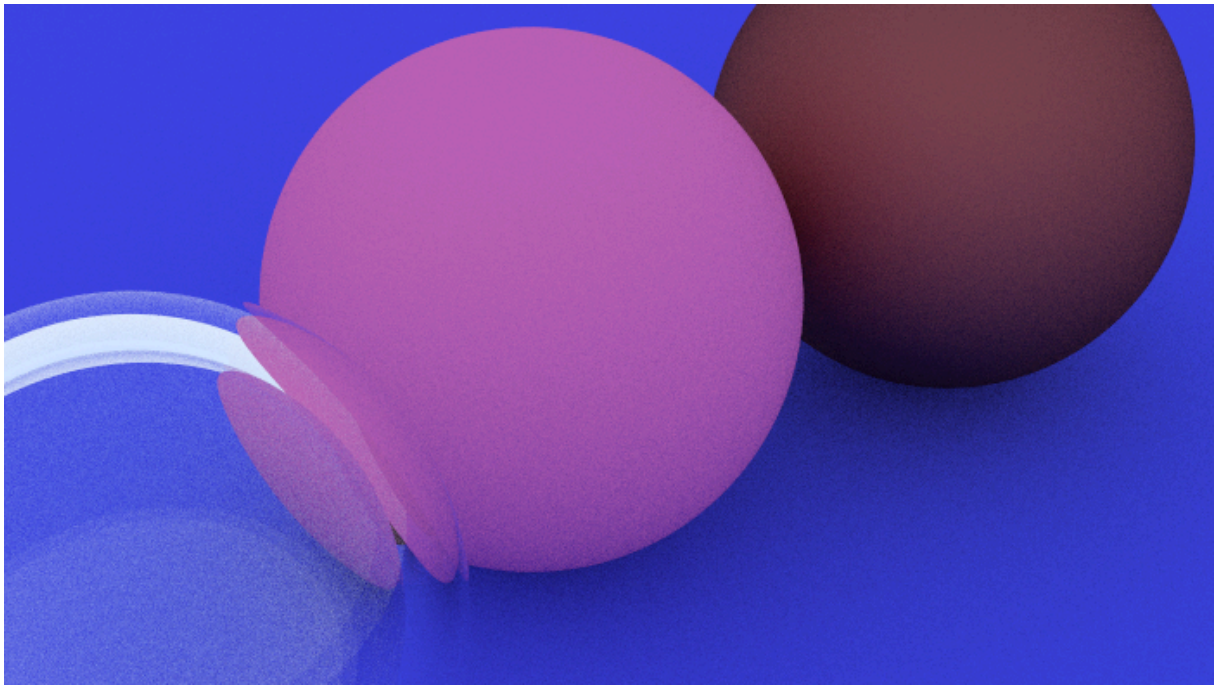


Figure 21: Changing the fov to 20 instead of 60

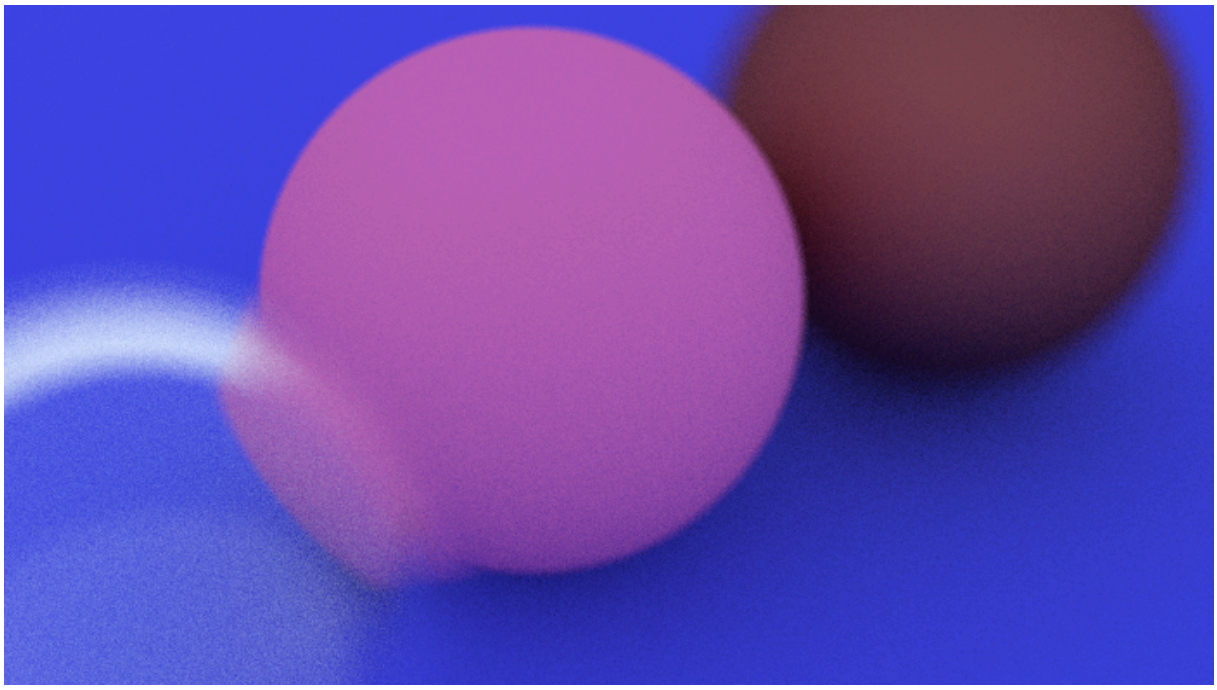


Figure 22: Using *defocus blur* to simulate *depth of field*

3.7 Using own Camera Class

After moving the camera to an own class, we can use different angles and focal points for our spheres.

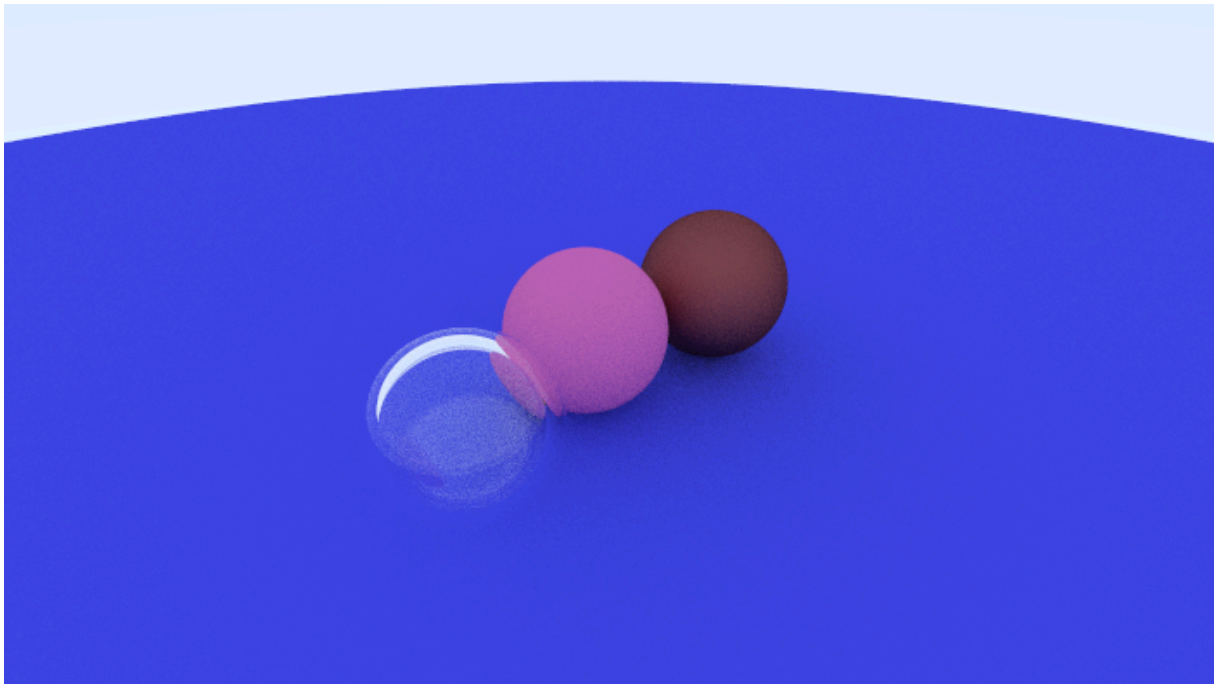


Figure 23: Different view of the spheres

3.8 Final Image

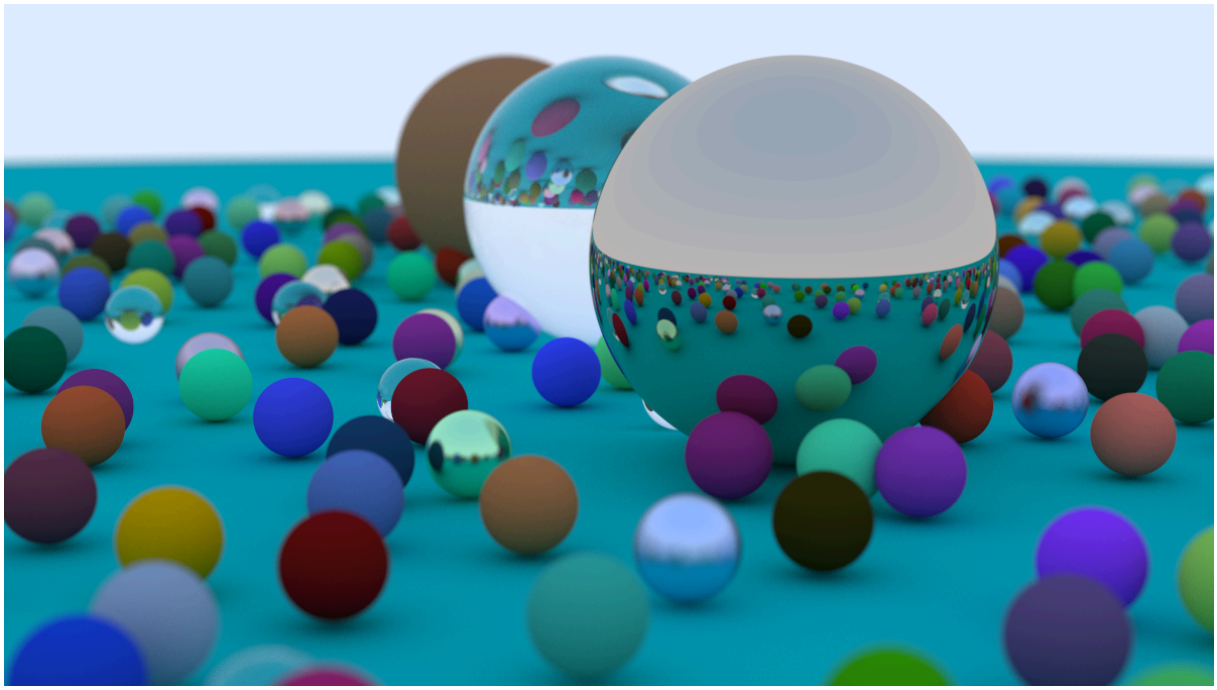


Figure 24: Final Image

Chapter 4

Ray Tracing the Next Week

4.1 Adding Movement

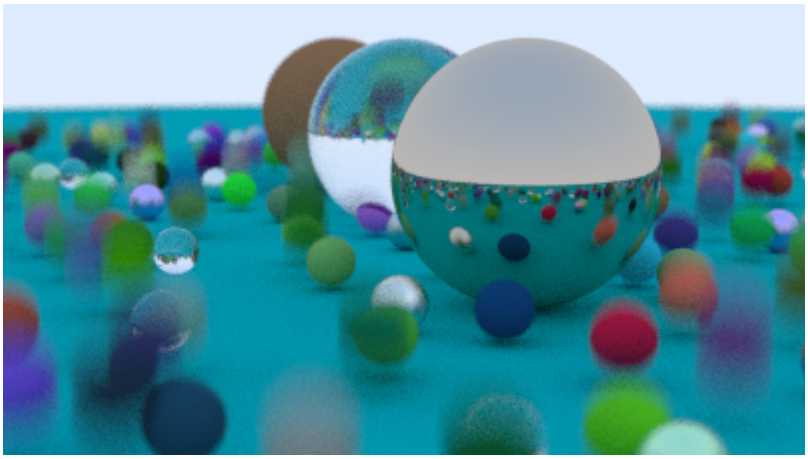


Figure 25: Moving spheres.

4.2 Bounding Volume Hierarchies

To speed things up we introduce bounding volumes. With these the ray-object intersection calculations. We can reduce the time to search for an object by introducing these bounding volumes. With this we can bring down the search to a logarithmic time.

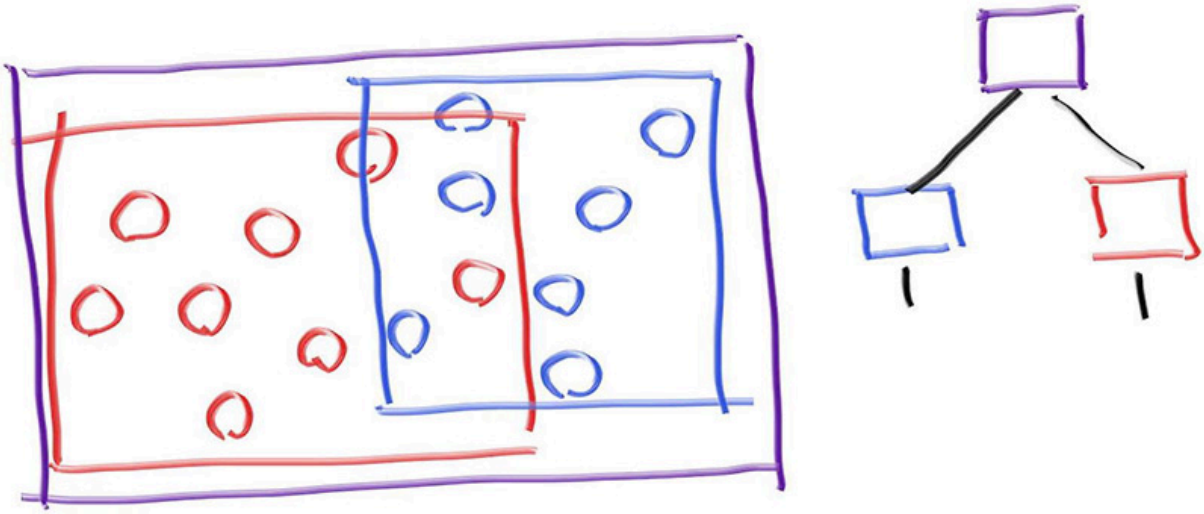


Figure 26: Bounding Volumes Hierarchy. Note that the bounding volumes can overlap.

4.3 Axis-Aligned Bounding Boxes (AABBs)

For this scene we will use AABBs. They split up the scene nicely in Depending on the scene, we can decide how to efficiently split up the scene. For example when we want to render a world where there are sticks just pointing vertically in the sky, we might want to split the scene in vertical stripes, without a horizontal component.

Appendix: Lecture Notes

Remark 1.1. (What to do): For each material, write a small comment (e.g. one paragraph) regarding
- Most interesting parts for you
- Parts you did not understand
- References you'd like to explore in more depth
- Any further observations

A.1 Rainbows

[3]

While being around for a long time, rainbows are still not fully understood, especially the rare *twinned rainbow* which can be seen in Figure 29. However, advancements have been made by I. Sadeghi *et al.* [3], by ray tracing non-spherical water drops. The provided Figure 28 is helpful to understand the rainbow creation process.

The work extends Lorenz-Mie theory by ray tracing non-spherical water drops. A simulation needs to include the following aspects:

- *Diffraction* (scattering of light, white band)
- *Geometric optics* (primary and secondary arc)
- *Interference* (supernumerary arcs)
- *Polarization* (polarization of primary and secondary arc)

A.1.1 Supernumerary arcs

Because of different path length there are additional rays which result in supernumerary arcs (See Figure 28 (c)). This can not be explained with purely geometric optics.

In Figure 27 are the primary and secondary arc clearly visible, together with the Alexander band (dark area between rainbows).

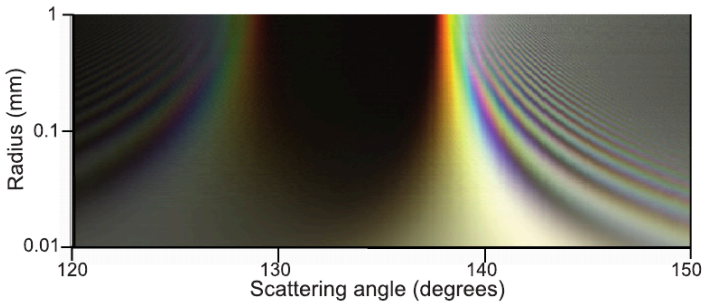


Figure 27: Lee diagram showing the variation in appearance of primary and secondary rainbows caused by scattering of sunlight by a spherical water drop as a function of radius (Lorenz-Mie theory calculations).

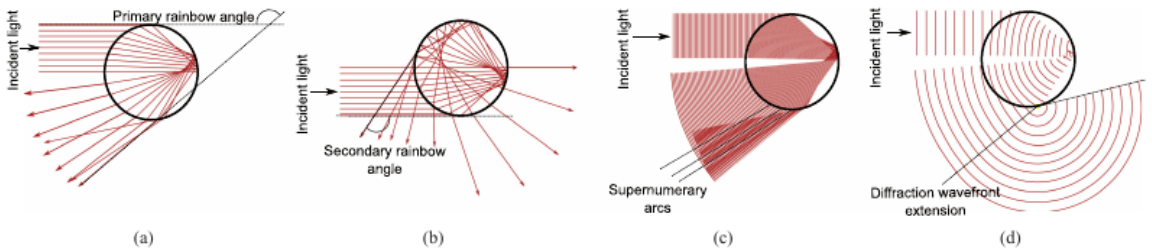


Figure 28: Generation of rainbows from the point of view of geometric optics and wave optics: (a) primary rainbow angle, after a single internal reflection; (b) secondary rainbow angle, after two internal reflections; (c) supernumerary rainbows are generated from constructive and destructive interference patterns (inspired by R. L. Lee and A. B. Fraser [2] (d) diffraction extends the wavefront and avoids abrupt intensity changes.

Figure 29: Twinned rainbows

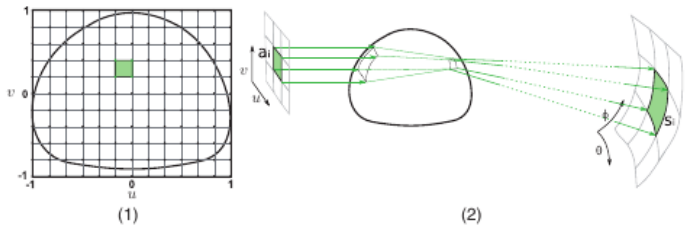


Figure 30: Steps of the Algorithm

A.2 Birefringence and Iridescence

by S. Steinberg [4]

Remark 1.2. (Note):
- Math is way above me
- I should look into "Electrodynamics and Optics" by

W. Demtröder [5] again...

When polarized light waves have different paths through an *anisotropic* optically transmissive medium (transparent) depending on their polarization it is called *birefringence*. Color changes based on viewing angle is called *iridescence*. Light creates iridescence when it interferes which each other inside the medium. Iridescence is most drastic when seen through a polarization filter.

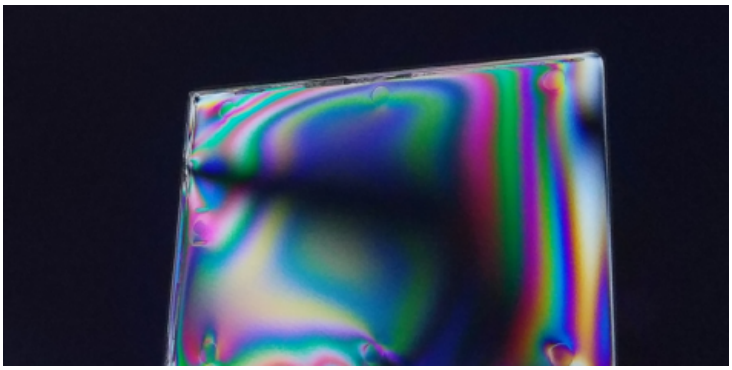


Figure 31: Birefringence-induced iridescence due to deformations under non-uniform mechanical stress

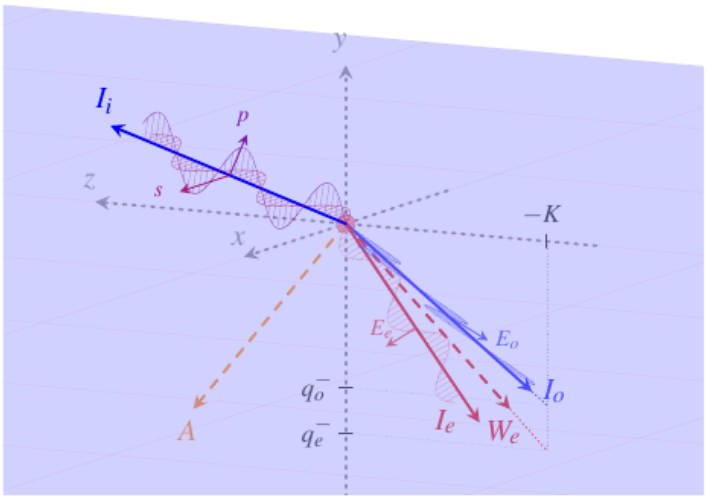


Figure 32: Randomly polarized light wave, incident to an anisotropic material, refracted into linearly-polarized ordinary and extraordinary rays. Note that the extraordinary Poynting vector leaves the plane of incidence (Y Z) and is detached from its wave's direction of propagation. The incident parameter K and the normal modes q_o^-, q_e^- for the (not normalized) ordinary and extraordinary waves are marked. The incidence parameter remains constant across surface boundaries for all participating waves.

Figure 33: Crystal where two different polarized waves split up on different paths.

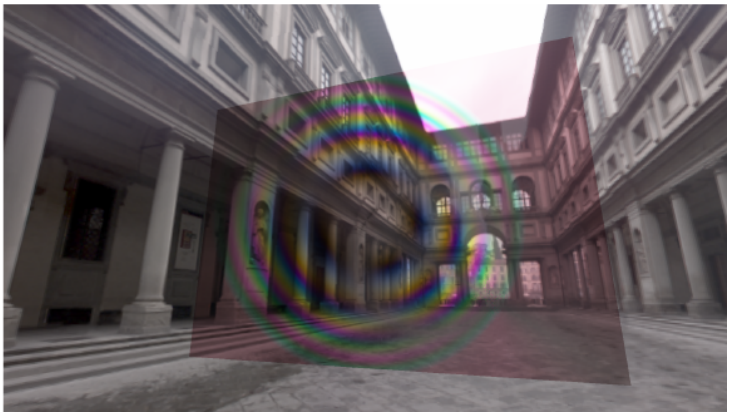


Figure 34: Coherence of $200\mu m$. Interference with itself?

A.3 Ray Tracing Special Relativity

by T. Müller, S. Grottel, and D. Weiskopf [6]. The methods in the paper render polygons while taking special relativistic effects into account. Vectors in 3D have one additional vector in the time dimension (Minkowsky spacetime). $(ct, x, y, z)^T$ and $(ct, x', y', z')^T$. The time is multiplied by c to have the same units in both systems.

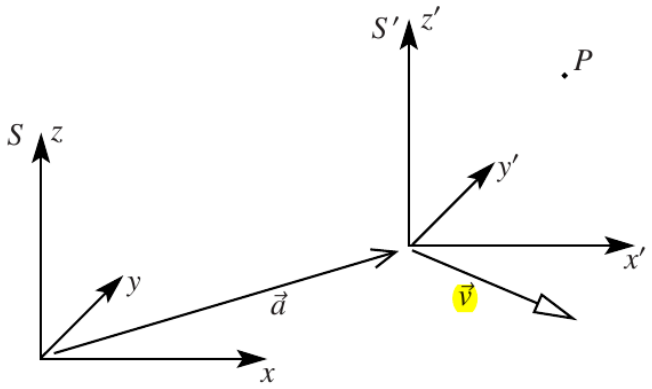


Figure 35: Two coordinate system S and S' with relative to each other. S travels with velocity v.

When using Polygons instead of ray tracing, some parts of a model render incorrectly (see Figure 36). The method of this paper can render the *Doppler effect*, *color shifts* and the *searchlight effect*.

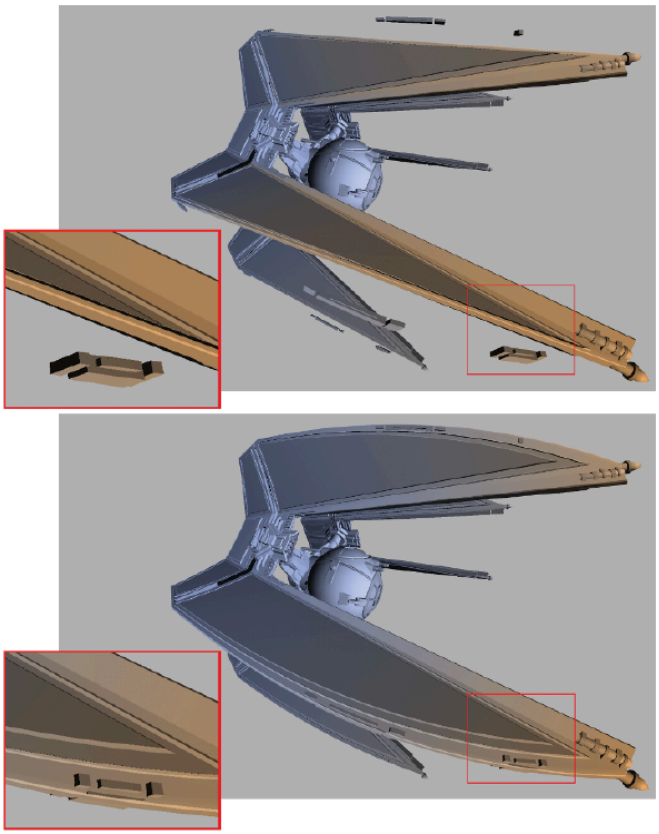


Figure 36: Clipped part because of polygons (Top). Correctly ray traced (Bottom)

Cannot be applied to moving objects why?

A.4 General Relativity

by O. James, E. v. Tunzelmann, P. Franklin, and K. S. Thorne [7].

Remark 1.3. (Website with Black Hole Simulations): https://jila.colorado.edu/~ajsh/insidebh/schw.html

Previous work has been done on Black holes that are far away. This paper is interested in a more close up and moving camera setting. I did not notice before, that the black hole in the movie *Interstellar* is a spinning black hole, and is therefore not symmetric (see Figure 37).

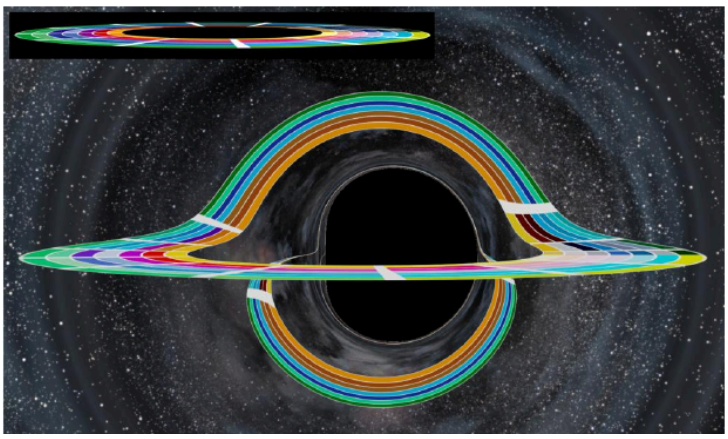


Figure 37: Paint-swatch

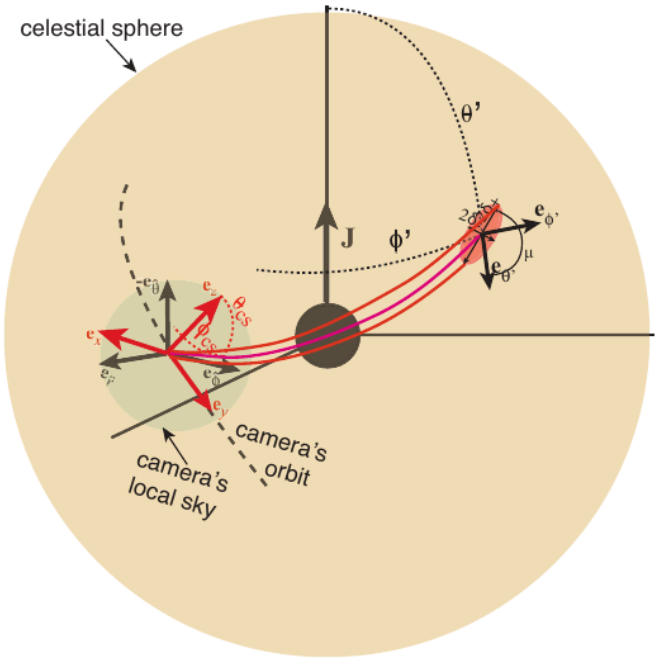


Figure 38: The mapping of the camera's local sky $(\theta_{cs}, \varphi_{cs})$ onto the celestial sphere (θ', φ') via a backward directed light ray; and the evolution of a ray bundle, that is circular at the camera, backward along the ray to its origin, an ellipse on the celestial sphere.

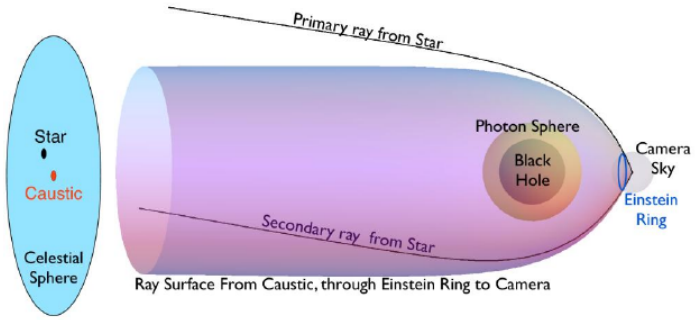


Figure 39: Two light rays from a star hitting the camera's sky.

A.5 Tetrachromatic Vision

The human eye has two types of photoreceptors. While rods are used for low light scenarios and are used mostly for our peripheral vision, cones are used for color vision (phototropic vision) and are mostly concentrated in the middle of our FOV.

There exist three cones the human vision responsible for respectively blue, green and red light (see Figure 40). Human vision is so called trichromatic vision due to the three cones.

Tetrachromatic vision extends the trichromatic vision. While humans generally don't have this type of vision, there has been one case, where another cone sat between the green and red cone.

Remark 1.4. (Mesopic Vision): In low light situations the human brain uses both rods and cones to see better. While rods are not used for color vision per se, due to their sensitivity being highest at the blue/green frequency, blue and green color is perceived stronger in low light situations, as for example red light.

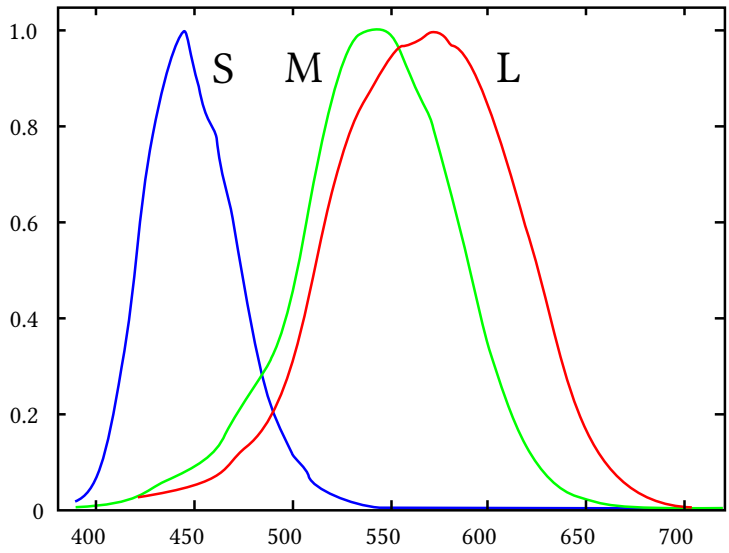


Figure 40: Normalized responsivity spectra of human cone cells, S, M, and L types. By Vanessaezekowitz at en.wikipedia, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=10514373

A.6 ScratchAPixel's Ray Tracing Introduction

- Path tracing: Advanced ray tracing including *global illumination* and *monte carlo* rays with multiple bounces. **Very accurate**. And better for offline simulation.

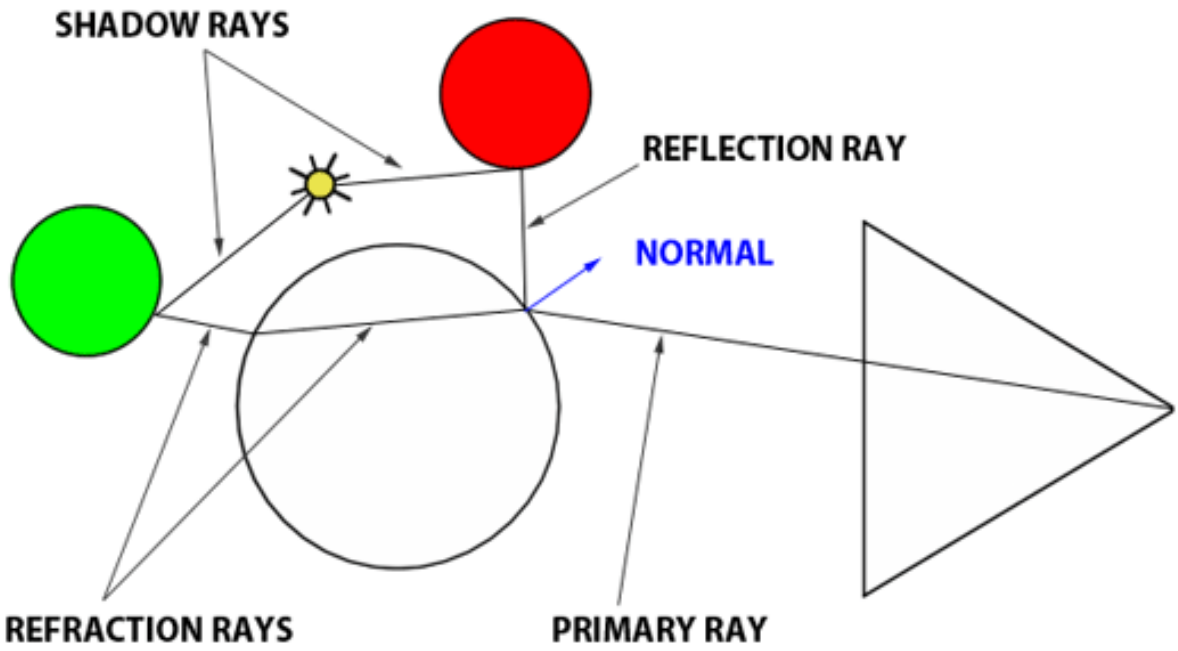


Figure 41: Reflection and refraction on a sphere.

A.7 Setup VNC

Install `tigervnc` client: `pacman -Syu tigervnc`

Same on the server: `pacman -Syu tigervnc`

Open the necessary ports on the server:

```
firewall-cmd --add-service=vnc-server
firewall-cmd --add-port=5900
```

A.8 Error Correction in Photon Mapping

For some reason there exists an error in literature, where the k-th photon is included in the irradiance calculation. By ignoring this last photon the bias disappears [8].

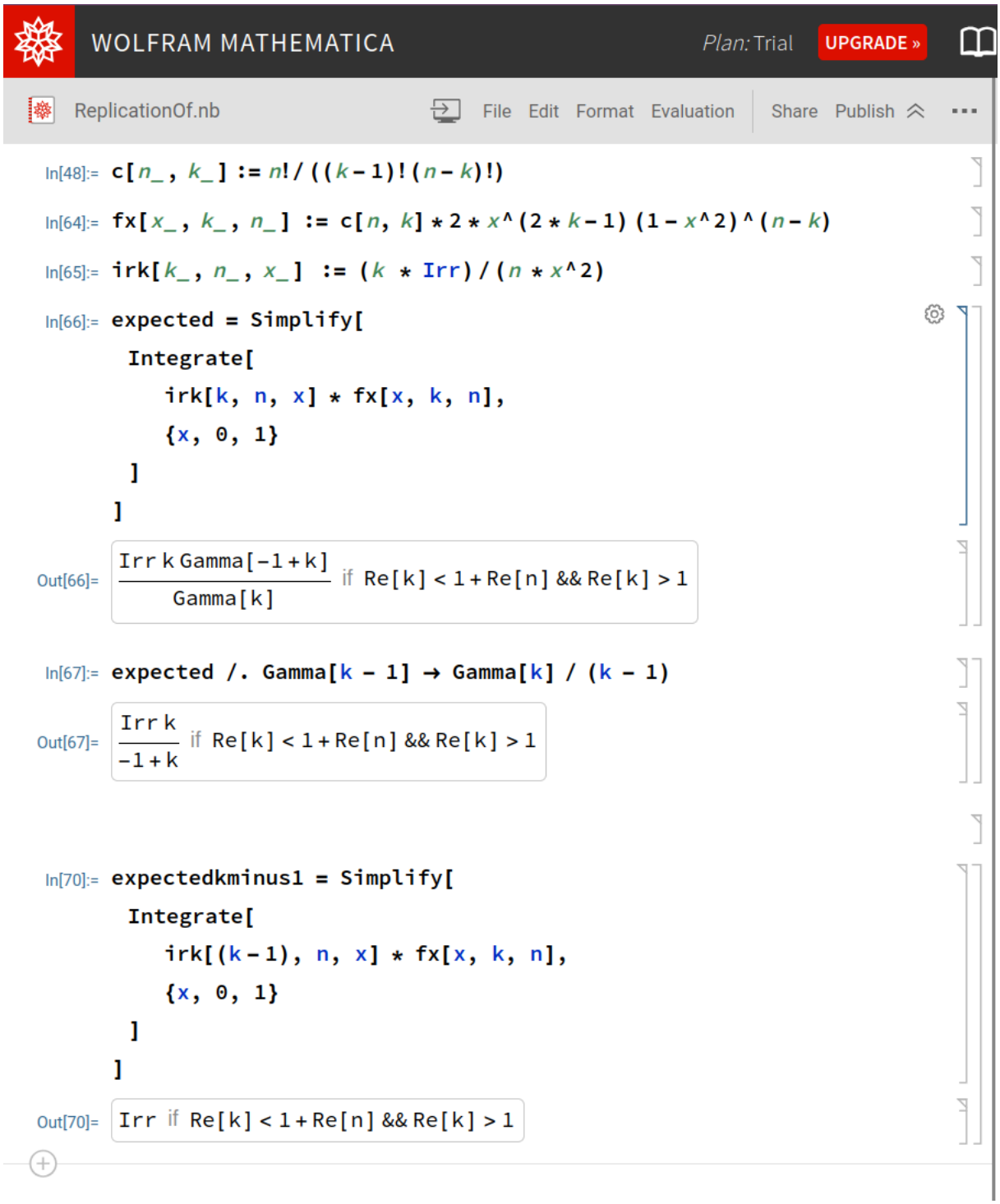


Figure 42: Mathematica code calculating the bias of the k-th photon. Second term uses the k-1-th photon for the correct result.

A.9 Estimates for Ray Tracing Runtimes

As ray tracing uses recursive functions for each ray, computation of a full picture can be quite expensive with naive algorithms.

A.9.1 Ray Tracing (from Eye)

Remark 1.5. (Pseudocode):

```
for x in width:
  for y in height:
    sendRay(direction)
    if intersectWithObject:
      if depth < maxDepth:
        color = updateColor(point, intensity)
        sendRay(direction * distribution)
      else return color
```

A.10 Text 2 Image with ComfyUI

Install the UI from the ComfyUI GitHub page. When choosing a model template, you need to install the downloaded `.safetensor` files in the `/models/checkpoints/` directory.

The model used was only 2GB in download size. When trying out the bigger models, my system ran out of RAM and crashed.

Figure 43: Prompt: "Chalet in Swiss". The right image additionally contains the word *snow* in the *negative prompt*

Figure 44: Prompt: “Two birds in the sky with a hut in the woods”.

Chapter 2

Notes from Computer Graphics 1

Course from summer semester 2025: https://moodle.lmu.de/course/view.php?id=39021

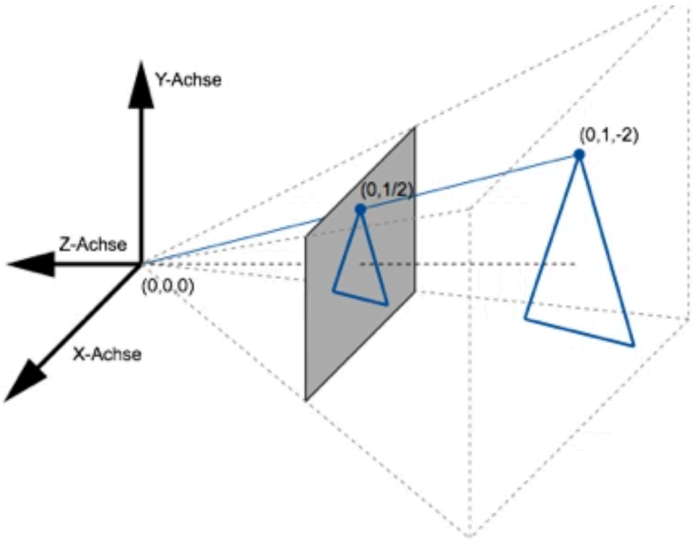


Figure 45: Convention for z-axis to be negative to keepx and y positive.

$$\begin{pmatrix} x_{sicht} \\ y_{sicht} \\ z_{sicht} \\ w_{sicht} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z \end{pmatrix}$$

Figure 46: One possibility for a projection matrix. Where w_{sicht} is the negative z axis from Figure 45.

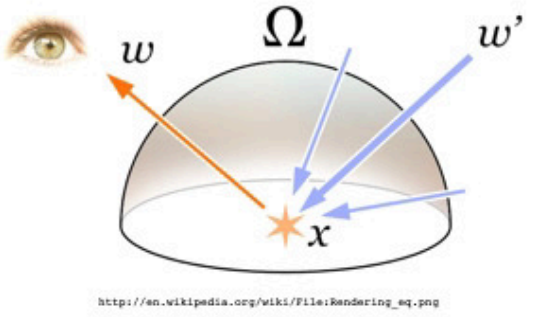
To get perspective with a matrix we use *perspective division* which divides each point by its 4th coordinate w

B.1 Rendering Equation

The Rendering Equation [Kajiya '86]

$$I_o(x,\vec{\omega}) = I_e(x,\vec{\omega}) + \int_\Omega f_r(x,\vec{\omega}',\vec{\omega}) I_i(x,\vec{\omega}') (\vec{\omega}' \cdot \vec{n}) d\vec{\omega}'$$

- I_o = outgoing light
- I_e = emitted light
- Reflectance Function
- I_i = incoming light
- Angle of incoming light
- Describes all flow of light in a scene in an abstract way
- Doesn't describe some effects of light:
  -
  -



http://en.wikipedia.org/wiki/File:Rendering_eq.png

LMU München – Medieninformatik – Andreas Butz – Computergrafik 1 – SS2025 27

Chapter 3

Vulkan Tutorial

Vulkan can be very fast because everything is done implicit. This means to do basic things you also have to be implicit about the whole thing. Meaning setup of graphic cards and so on. This leads to a lot of boilerplate code.

C.1 Triangle Tutorial

From https://vulkan-tutorial.com/

Remark 3.1. (Steps to draw a triangle:):
- Create a VkInstance
- Select a supported graphics card (VkPhysicalDevice)
- Create a VkDevice and VkQueue for drawing and presentation
- Create a window, window surface and swap chain
- Wrap the swap chain images into VkImageView
- Create a render pass that specifies the render targets and usage
- Create frame buffers for the render pass
- Set up the graphics pipeline
- Allocate and record a command buffer with the draw commands for every possible swap chain image
- Draw frames by acquiring images, submitting the right draw command buffer and returning the images back to the swap chain

Shell Commands 3.2. (Prerequisites Vulkan for OpenSUSE Linux): `$ sudo zypper in glm-devel vulkan-devel shaderc libXi-devel libglfw-devel libXxf86vm-devel`

C.1.1 Coding Conventions

Functions are written with lowercase `vk`, while structs and enums are `Vk` and enumerations have a `VK_` prefix.

Structs are very heavily used in Vulkan. To create a C++ struct first initialize an empty struct with `VkCreateInfo createInfo{};` (note the `Vk` prefix).

C.1.2 Validation Layers

Vulkan does not check for any errors to be as fast as possible. Validation Layers provide access to some debug functionality. When in debug mode one can activate these layers with a small performance impact. But they can be ignored otherwise.

```
const std::vector<const char*> validationLayers = {
    "VK_LAYER_KHRONOS_validation"
};

#ifdef NDEBUG
    const bool enableValidationLayers = falsespecific // As Vulkan uses queues,
    we need to check whether our GPU supports a queues
```

```
#else
Using optional as the indices could be anything, and we need to provide a way to
say "Nothing found!"const bool enableValidationLayers = true;
#endif
```

Chapter 4

OpenGL Tutorial

OpenGL is the predecessor of Vulkan also by Khronos Group.

Remark 4.1. (Dependencies needed):

Similar to Vulkan, OpenGL also uses glfw3. So for window creation we need to compile our C++Code with

```
-lglfw3 -lGL -lX11 -lpthread -lXrandr -lXi -ldl
```

D.1 Triangle Tutorial

From https://learnopengl.com/Getting-started/Hello-Triangle

The rendering pipeline can be used to create 2D object from a 3D system. With a shader we can modify objects (e.g. color).

(i) Vertex shader:

Takes a single vertex as input and performs transformations. (Maybe used for rotation? Together with a matrix?) OpenGL supports at least 16 vertex attributes.

You can't modify shaders from another shader perse, they always take an input and give out an output. These are the primary ways to pass information between shaders. There is however a way of accessing shader variables and so on with the `uniform` keyword.

Remark 4.2. (C Programming Note): There is no function overloading in native c (OpenGL shader language).

VBO	Vertex Buffer Object	Stores raw vertex data
VAO	Vertex Array Object	Stores state and configuration in an array on how to interpret the VBOs
EBO	Element Buffer Object	Stores list of indices for vertexes that can be reused (two triangle give a rectangle, they share 2 vertices)

Buffer objects load objects in the GPUs RAM.

(i) Generate VBOs and EBOs
(ii) Configure how they should be read in the VAO
(iii) Bind VAO in render loop to draw

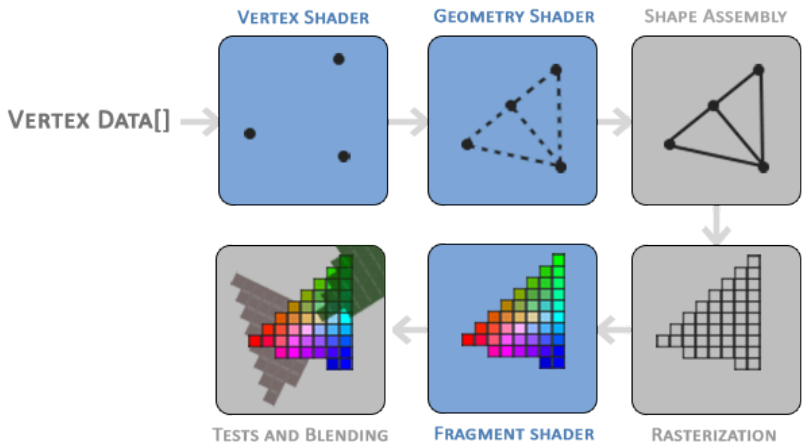


Figure 48: OpenGL Pipeline

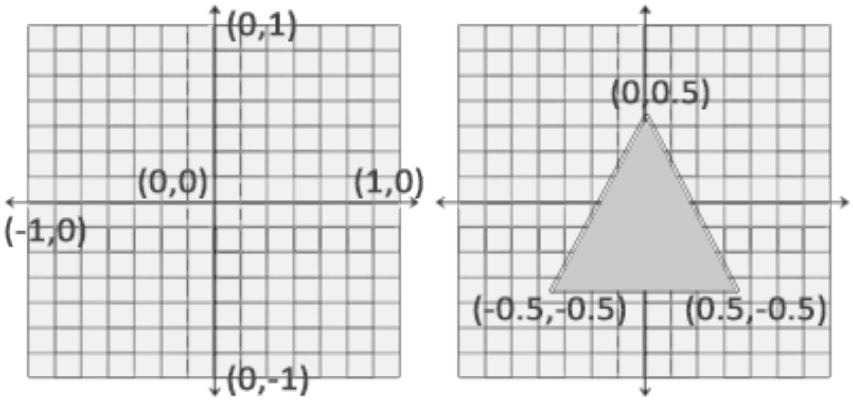


Figure 49: Normalized Device Coordinates (NDC). Note that everything outside of [(-1,-1), (1,1)] will be clipped

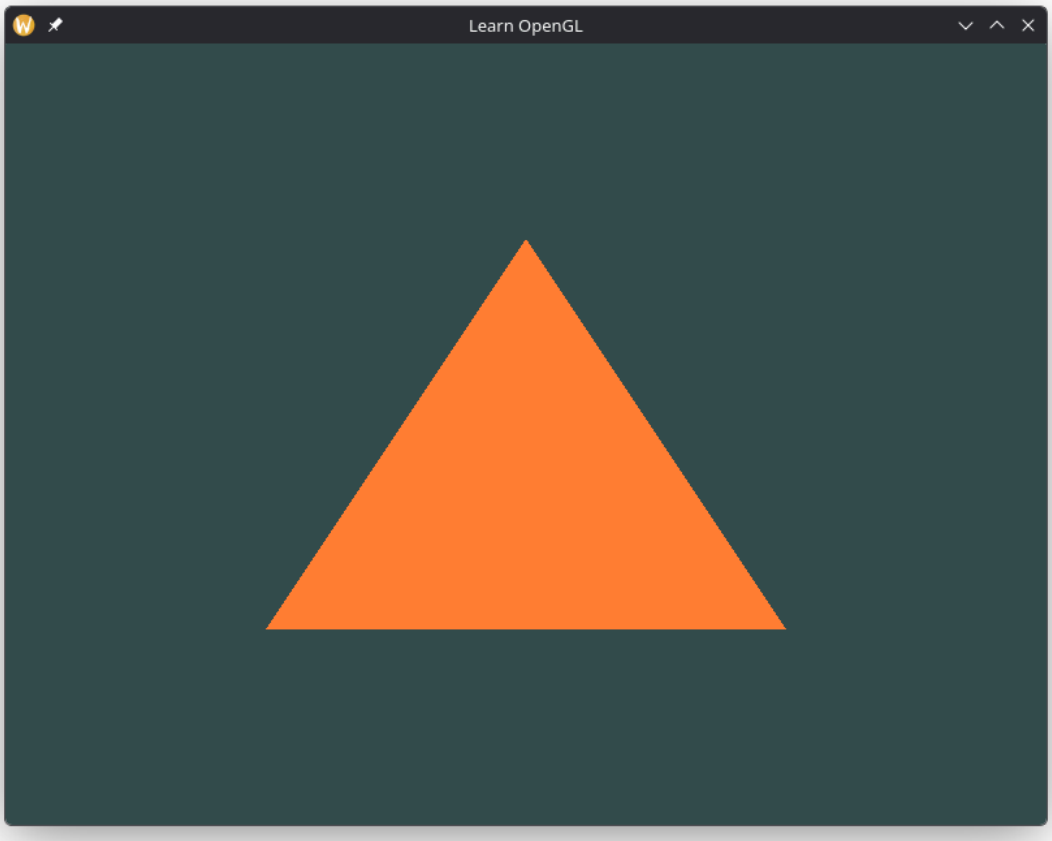


Figure 50: My first triangle!

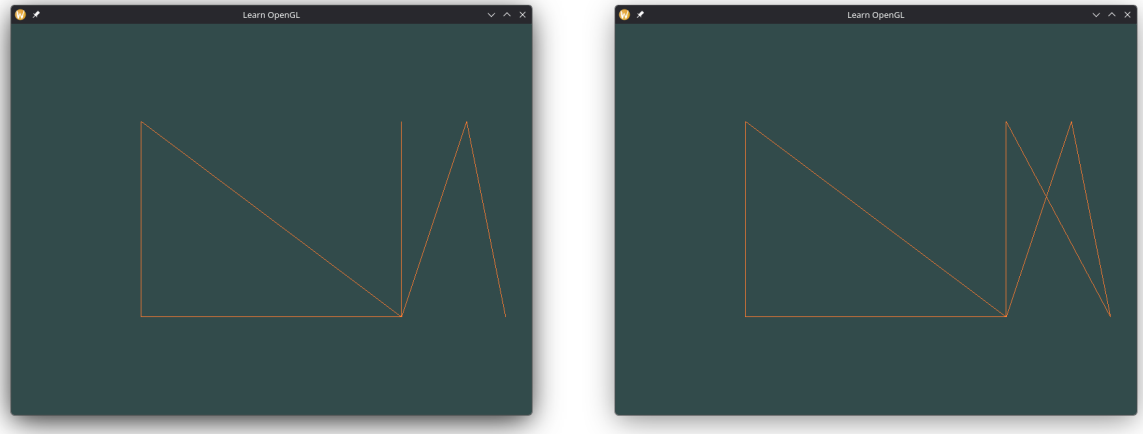


Figure 51: Using GL_LINE_STRIP and GL_LINE_LOOP instead of GL_TRIANGLE as a parameter in glDrawElements

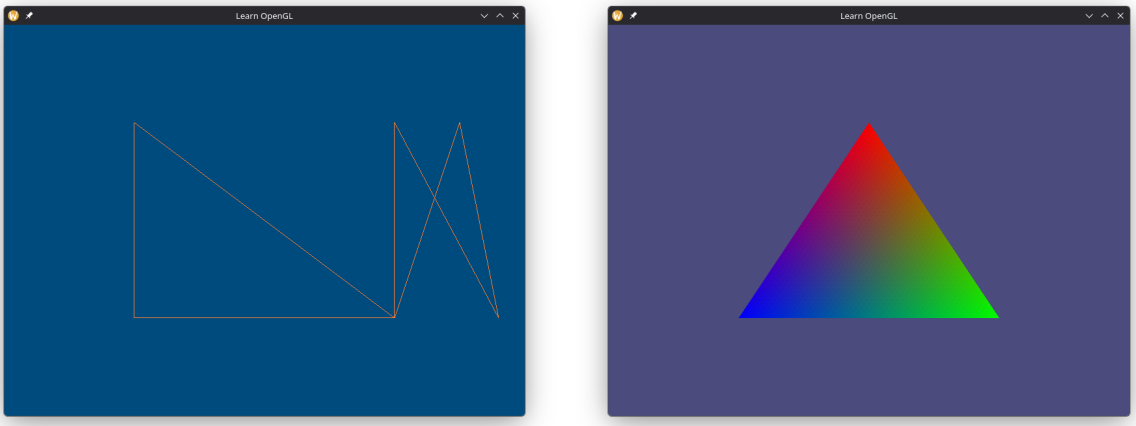


Figure 52: Prettier colors and RGB triangle using shaders

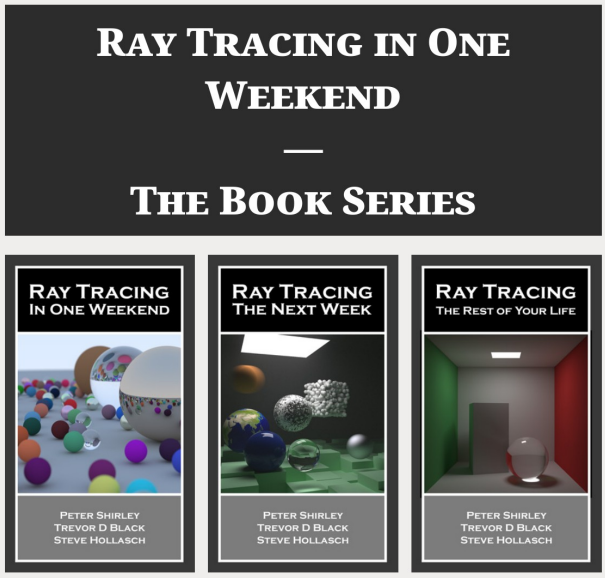
Chapter 5

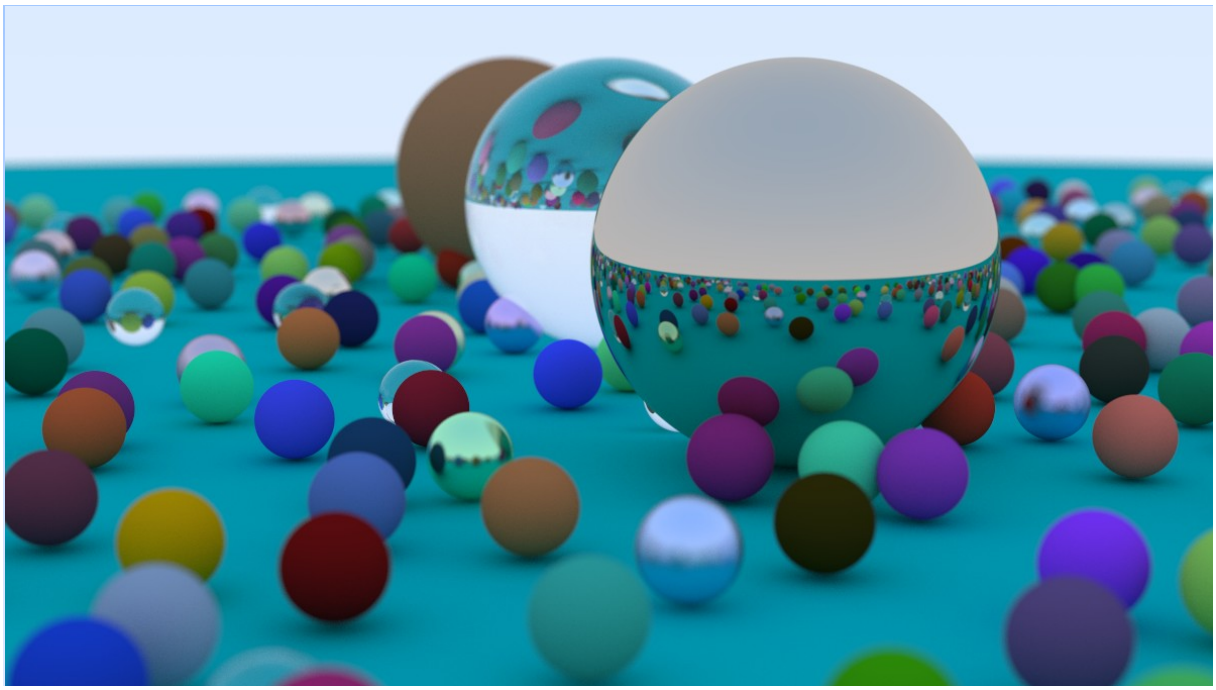
Presentation

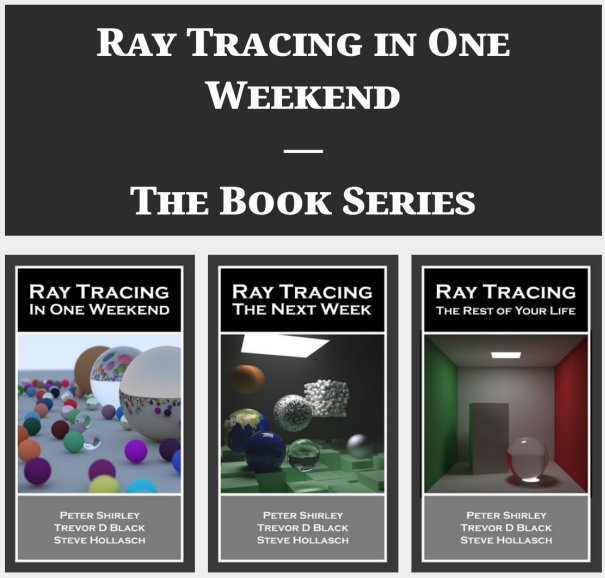
Writing Your Own Ray Tracer From Scratch Is Wasting Your Time

Kasimir Rothbauer

Advanced Topics in Computer Graphics






RAY TRACING IN ONE WEEKEND
—
THE BOOK SERIES
RAY TRACING
IN ONE WEEKEND
PETER SHIRLEY
TREVOR D BLACK
STEVE HOLLASCH
RAY TRACING
THE NEXT WEEK
PETER SHIRLEY
TREVOR D BLACK
STEVE HOLLASCH
RAY TRACING
THE REST OF YOUR LIFE
PETER SHIRLEY
TREVOR D BLACK
STEVE HOLLASCH

WOW

Roadmap

- Write all in C++
- Get Image
- Get Rays
- Get Colors
- Get Objects
- Profit?

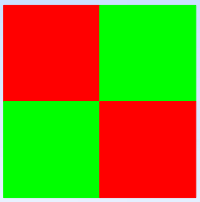
Write C++

```
// Vector Utility Functions

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

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

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

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

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


Get Image – Image.ppm

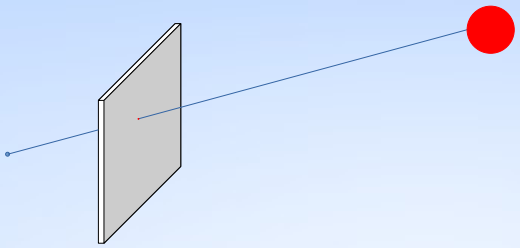
- Header
- RGB Values for each pixel

```
P3
2 2
255
255 0 0    0 255 0
0 255 0    255 0 0
```



Get Rays – Vectors with Distance

- Shoot Rays Through Viewport
- If Distance Infinity: Return Color of Background
- If Hit: Return Color of Object

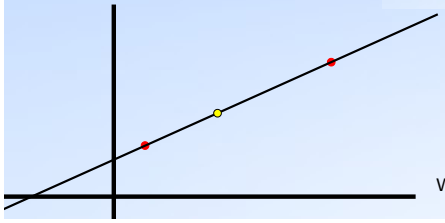


Write C++

```
// Vector Utility Functions

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

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

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

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

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

First Gradien Image

Lerp
Linear Int**erp**olation



Get Object – Spheres

- Create Spheres as Objects
- “Simple” Formula: $x^2 + y^2 + z^2 = r^2$
- On Hit Return Color

Write C++

```
// Vector Utility Functions

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

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

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

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

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


First Sphere(?)

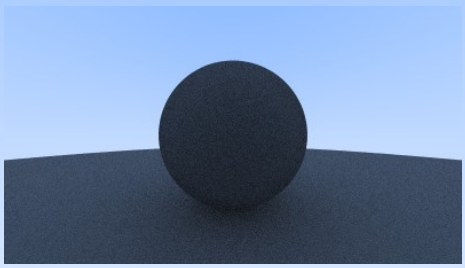
Write C++

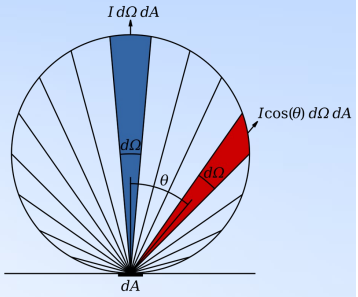
```
// Vector Utility Functions

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

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

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

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

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


Include Surface Normals

Spheres: Diffuse Materials

- Lambertian Reflection



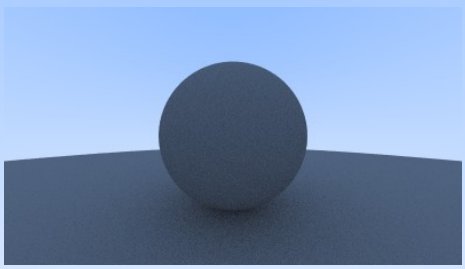


Kasimir Rothbauer

Writing Your Own Ray Tracer From Scratch Is Wasting Your Time

Sphere: Shadow Acne

- Floating Point Errors
- Ray Intersects Inside Sphere?
- Insert Padding
- 0.000001

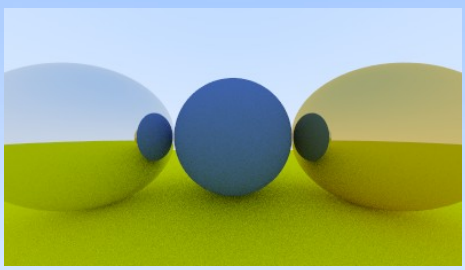


Kasimir Rothbauer

Writing Your Own Ray Tracer From Scratch Is Wasting Your Time

Spheres: Shiny Materials

- Incoming Angle = Outgoing Angle

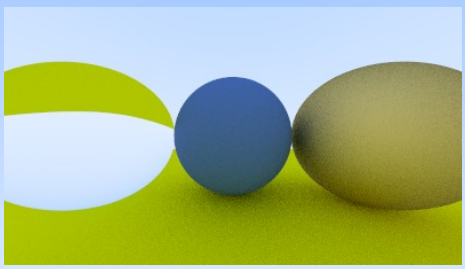


Kasimir Rothbauer

Writing Your Own Ray Tracer From Scratch Is Wasting Your Time

Spheres: Glass Materials

- Snells Law
- Ideal Refraction

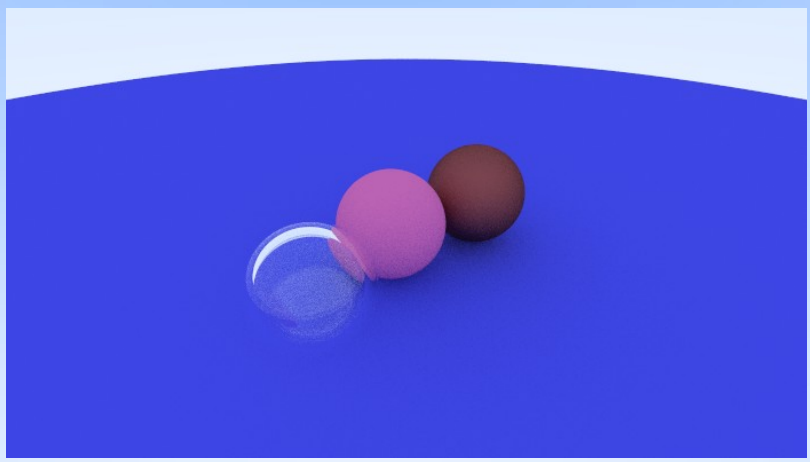


$$\frac{\sin\theta_1}{\sin\theta_2} = n_{2,1} = \frac{n_2}{n_1} = \frac{v_1}{v_2}$$

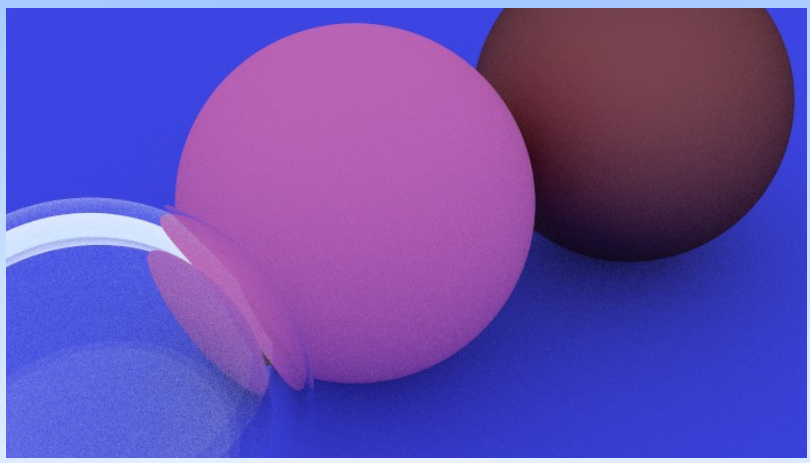
Kasimir Rothbauer

Writing Your Own Ray Tracer From Scratch Is Wasting Your Time

Changing The View

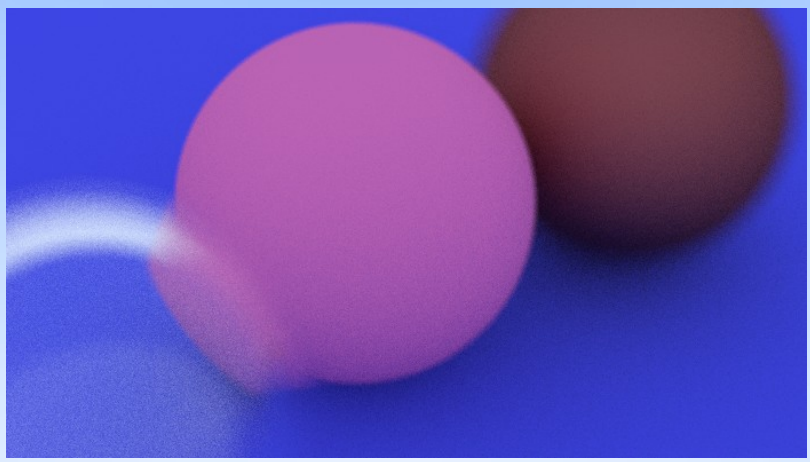


Zooming In



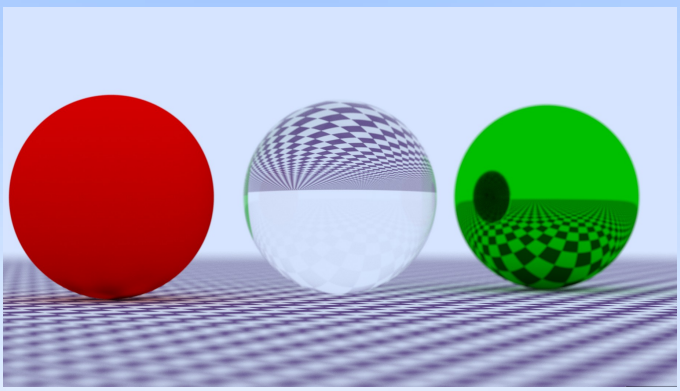
Defocus Blur

- Create Fuzz
- Fuzz by Creating Random Values On Unit Sphere

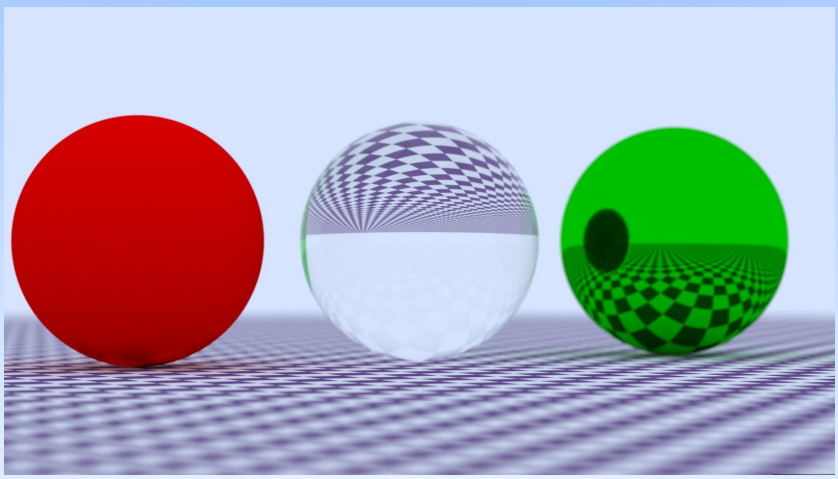


Defocus Blur - Issues

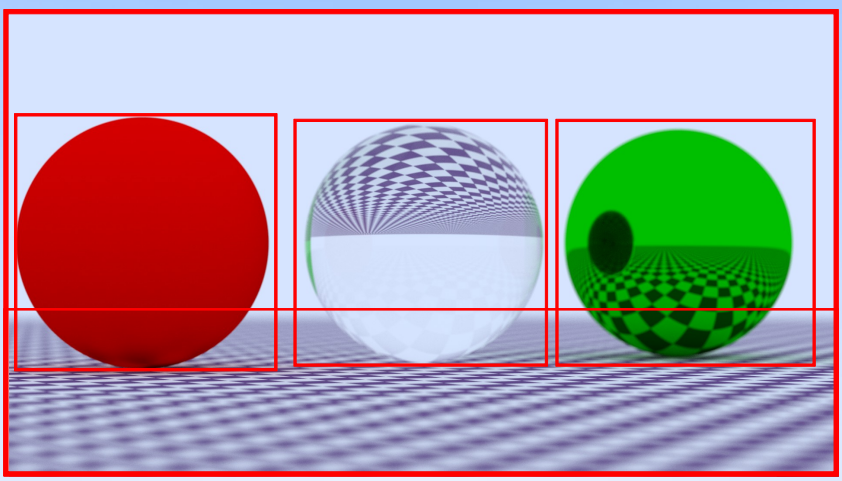
- Create Fuzz at given Distance
- Glass Behind Sphere Still Sharp



Why Wasting Time



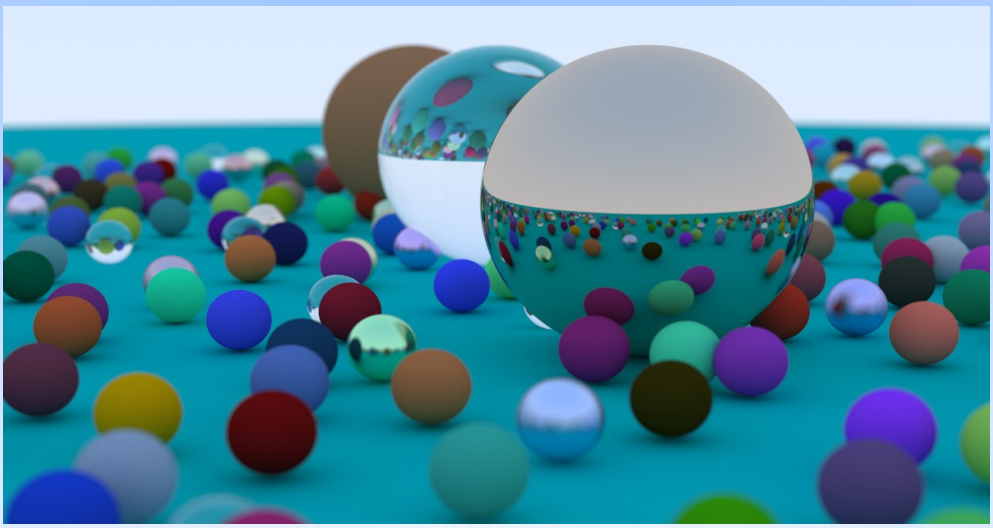
Bounding Volume Hierarchies

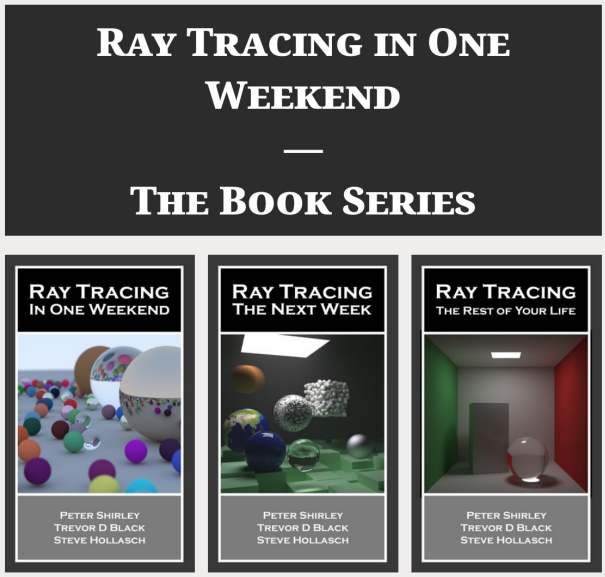


Wasting Time - CPU

- Only Using CPU
- Single Threaded
- Use GPU for Parallelization

Final Image





Wasting Time – Stop While You Can

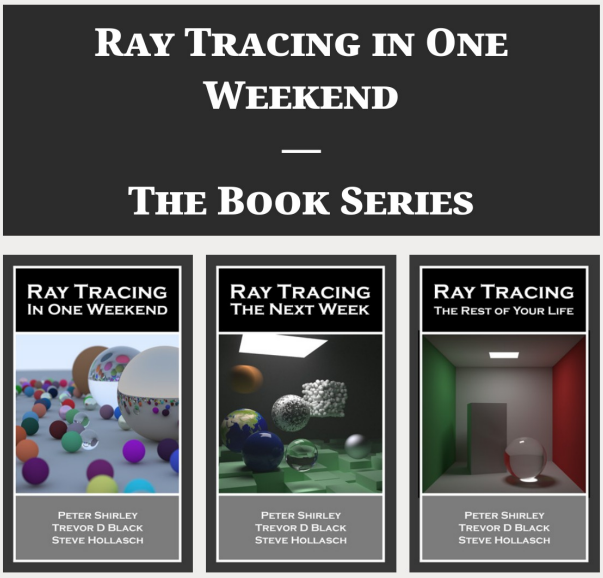
- No “Advanced Computer Graphics”
- Never Perfect
- What About Light Sources?
- Caustics?
- Never Ending
- Simply Use An Existing Ray Tracer

Kasimir Rothbauer

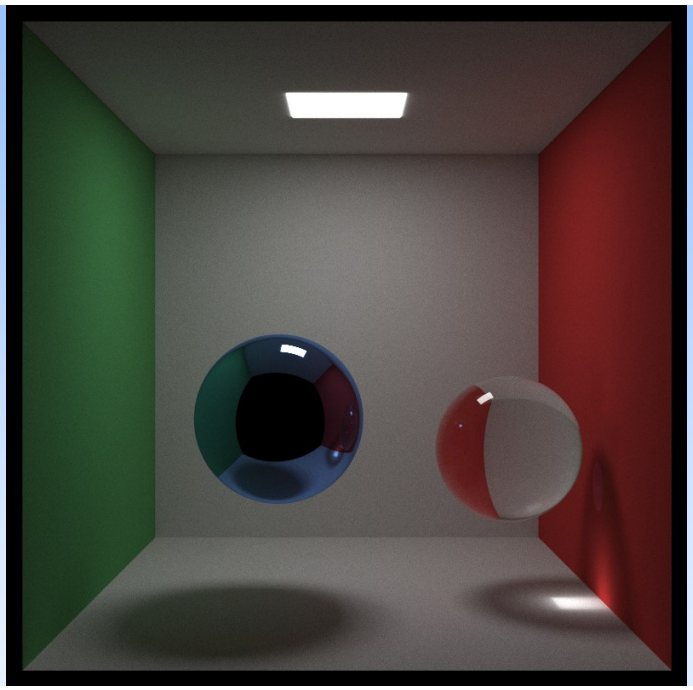
Writing Your Own Ray Tracer From Scratch Is Wasting Your Time

Benefits

- Deepen C++ Knowledge
- Own Makefile Creation
- Pretty Renders
- Playing Around With The Code
- Enhance Ray Tracer Further



1.5h



Bibliography

[1] S. H. Peter Shirley Trevor David Black, "Ray Tracing in One Weekend." [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html

[2] R. L. Lee and A. B. Fraser, *The rainbow bridge: rainbows in art, myth, and science*. Penn State Press, 2001.

[3] I. Sadeghi *et al.*, "Physically-based simulation of rainbows," *ACM Trans. Graph.*, vol. 31, no. 1, pp. 3:1–3:12, 2012, doi: 10.1145/2077341.2077344.

[4] S. Steinberg, "Analytic Spectral Integration of Birefringence-Induced Iridescence," *Comput. Graph. Forum*, vol. 38, no. 4, pp. 97–110, 2019, doi: 10.1111/CGF.13774.

[5] W. Demtröder, *Electrodynamics and optics*. Springer Nature, 2019.

[6] T. Müller, S. Grottel, and D. Weiskopf, "Special Relativistic Visualization by Local Ray Tracing," *IEEE Trans. Vis. Comput. Graph.*, vol. 16, no. 6, pp. 1243–1250, 2010, doi: 10.1109/TVCG.2010.196.

[7] O. James, E. v. Tunzelmann, P. Franklin, and K. S. Thorne, "Gravitational lensing by spinning black holes in astrophysics, and in the movie Interstellar," *Classical and Quantum Gravity*, vol. 32, no. 6, p. 65001, Feb. 2015, doi: 10.1088/0264-9381/32/6/065001.

[8] R. J. García, C. Ureña, and M. Sbert, "Description and Solution of an Unreported Intrinsic Bias in Photon Mapping Density Estimation with Constant Kernel," *Comput. Graph. Forum*, vol. 31, no. 1, pp. 33–41, 2012, doi: 10.1111/J.1467-8659.2011.02081.X.