Abstract

This project focuses on implementing a ray tracing algorithm for rendering realistic images of scenes containing spheres. The program is written in C++ and utilizes classes and functions to organize the ray tracing logic effectively. Key components include sphere geometry, ray-sphere intersection, reflection, refraction, and color calculations.

The main functionality of the program involves reading sphere initialization values from a text file. Each line in the file represents the parameters for a single sphere, including its position, radius, material properties (e.g., reflectivity, roughness), color, transparency, and whether it acts as a light source. The program dynamically initializes Sphere objects based on the data read from the file.

Multithreading is employed to parallelize the rendering process, improving performance by distributing computations across multiple threads. The program divides the image into segments, with each segment processed concurrently by a separate thread.

After rendering, the program outputs the computed color data to a file for further analysis or visualization. Execution time is measured to provide insights into the performance of the ray tracing algorithm.

This Python script serves as a post-processing tool for the ray tracing program, converting the computed color data into a visual representation of the rendered scene. The ray tracing program outputs color data to a text file, which is read and processed by this script.

The script reads RGB values from the text file, ensuring that each value falls within the valid range of 0 to 255. Values outside this range are clipped to ensure image consistency. The RGB values are organized into a two-dimensional array, representing the pixel values of the image.

Using the NumPy library, the two-dimensional array is converted into a NumPy array of unsigned 8-bit integers, suitable for image representation. The NumPy array is then converted into a PIL Image object, facilitating further image manipulation and saving.

Finally, the PIL Image object is saved as a PNG file, providing a visual representation of the rendered scene. Additionally, the temporary text file containing the color data is removed to maintain cleanliness and organization.

Together, these components provide a comprehensive workflow for ray tracing, enabling users to efficiently generate and visualize complex scenes with realistic lighting and materials. The integration of the C++ ray tracing system and the Python post-processing script allows for a seamless transition from scene rendering to image generation, facilitating quick and effective visualization of ray traced scenes for analysis, presentation, or further processing.

Table of Contents

Problem Statement

The problem at hand revolves around the implementation of a basic ray-tracing program in C++, aiming to simulate the behavior of light rays within a defined scene to generate realistic images. Ray tracing, a foundational technique in computer graphics, plays a pivotal role in creating lifelike visual representations by tracing the paths of light rays as they interact with scene elements. This project seeks to address the need for a simple yet effective ray-tracing solution that can render basic scenes with minimal complexity, catering to various applications in computer graphics, virtual reality, and interactive systems.

Background Study

The concept of ray tracing dates back to the early days of computer graphics research in the 1960s and 1970s. Initially proposed by Arthur Appel in 1968, ray tracing algorithms have since evolved significantly, driven by advancements in computing power and graphics hardware. Traditional rendering techniques, such as rasterization, rely on approximations and simplifications to simulate lighting and shading effects. In contrast, ray tracing offers a more physically accurate approach by tracing individual light rays from the camera's viewpoint into the scene, accounting for reflections, refractions, and shadows.

Over the years, researchers and developers have explored various optimizations and enhancements to ray tracing algorithms to improve rendering performance and quality. Techniques such as bounding volume hierarchies (BVH), acceleration structures, and stochastic sampling have been introduced to accelerate ray-object intersection tests and reduce computational overhead. Additionally, advancements in parallel computing architectures, such as GPUs and multi-core CPUs, have enabled real-time or near-real-time ray tracing for interactive applications and high-fidelity rendering pipelines.

In the context of Human-Computer Interaction (HCI), ray tracing plays a crucial role in creating visually compelling user interfaces and immersive experiences. By incorporating realistic lighting, shadows, and reflections, ray-traced graphics enhance user engagement, comprehension, and enjoyment of interactive systems. From virtual reality environments to architectural walkthroughs, the integration of ray tracing enriches the visual quality and realism of HCI applications, thereby improving user satisfaction and usability.

Despite the widespread adoption and advancements in ray tracing technology, there remains a need for simple and accessible implementations suitable for educational purposes, prototyping, and research. Developing a basic ray-tracing program in C++ allows for a hands-on exploration of fundamental concepts in computer graphics, providing valuable insights into the underlying principles of light simulation and scene rendering. By addressing this need, the proposed work aims to bridge the gap between theoretical knowledge and practical implementation, empowering developers and researchers to explore the capabilities and limitations of ray tracing algorithms.

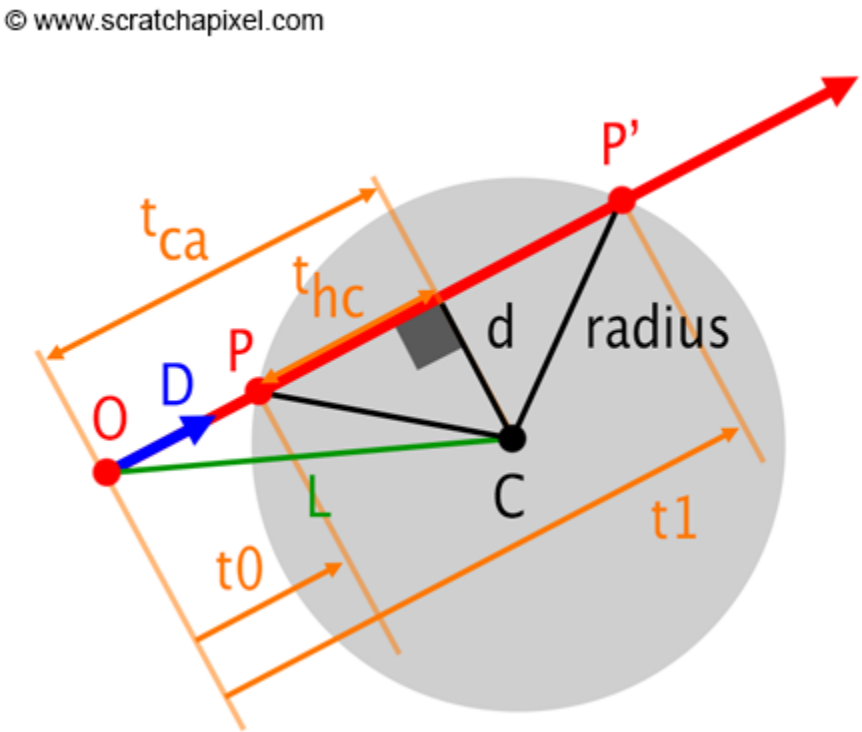
Approach to Solution - Link

1. Room and camera parameters

The room dimensions are set to a height of 350 units, width of 300 units, and depth of 240 units. The origin is established at the bottom-right corner nearest when facing into the room, providing a consistent reference point for spatial orientation.
For the camera, a standardized aspect ratio of 4:3 is adopted with a width of 40 units and a height of 30 units. To ensure optimal imaging, all rays originate from a distance of 20 units behind the center of the camera frame, enabling comprehensive coverage and perspective capture within the designated space.

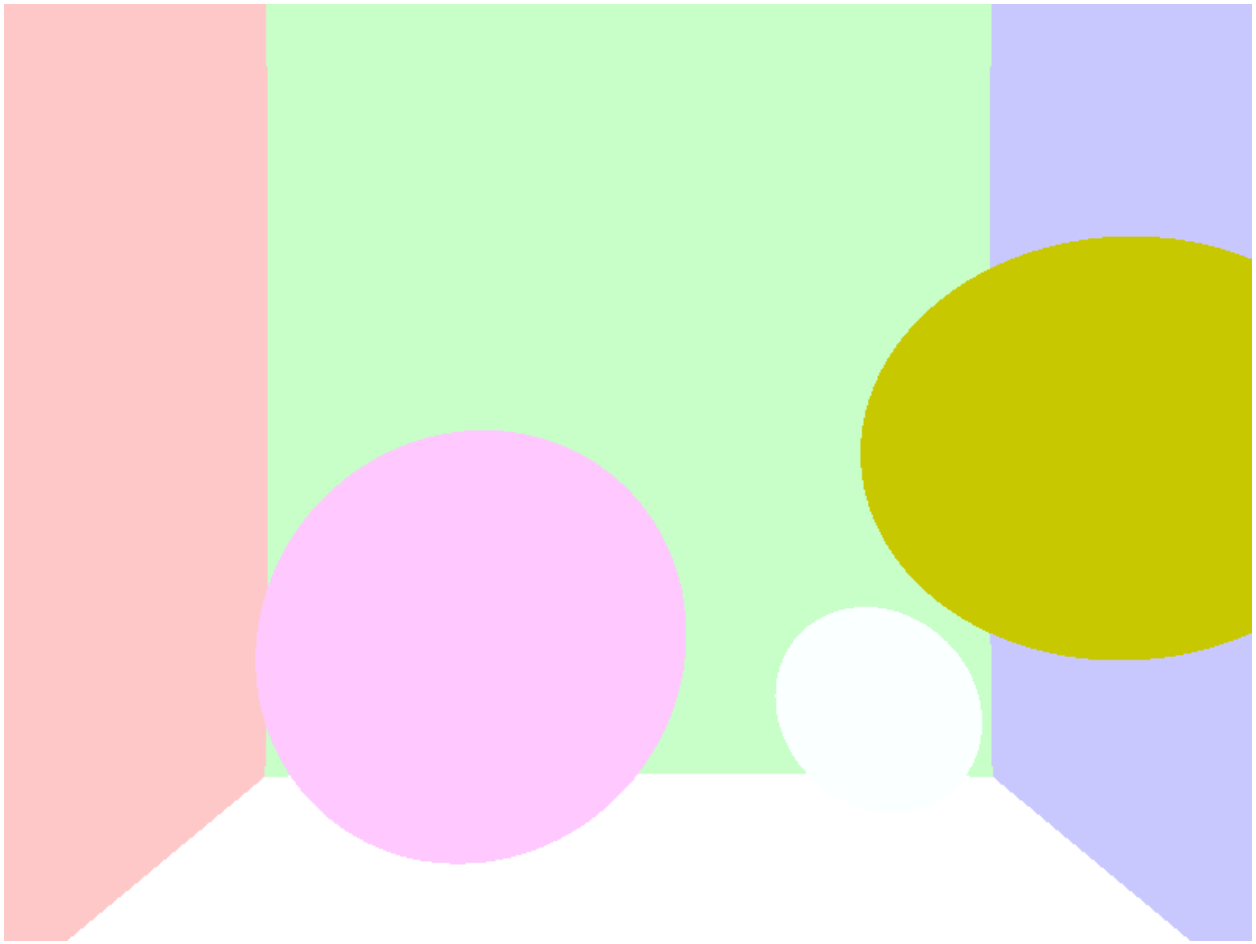
2. Basic intersection model

Initially, the basic code for a two point ray is written and spheres are introduced. Initially, the color representation was limited to boolean values, with true and false denoting white and black. Eventually, colour channels (RGB) were added. The intersection logic was implemented based on the principles illustrated in the provided figure, which outlines the intersection of rays with spheres.



Credit: A Minimal Ray-Tracer

Here is the output of simple intersection code:

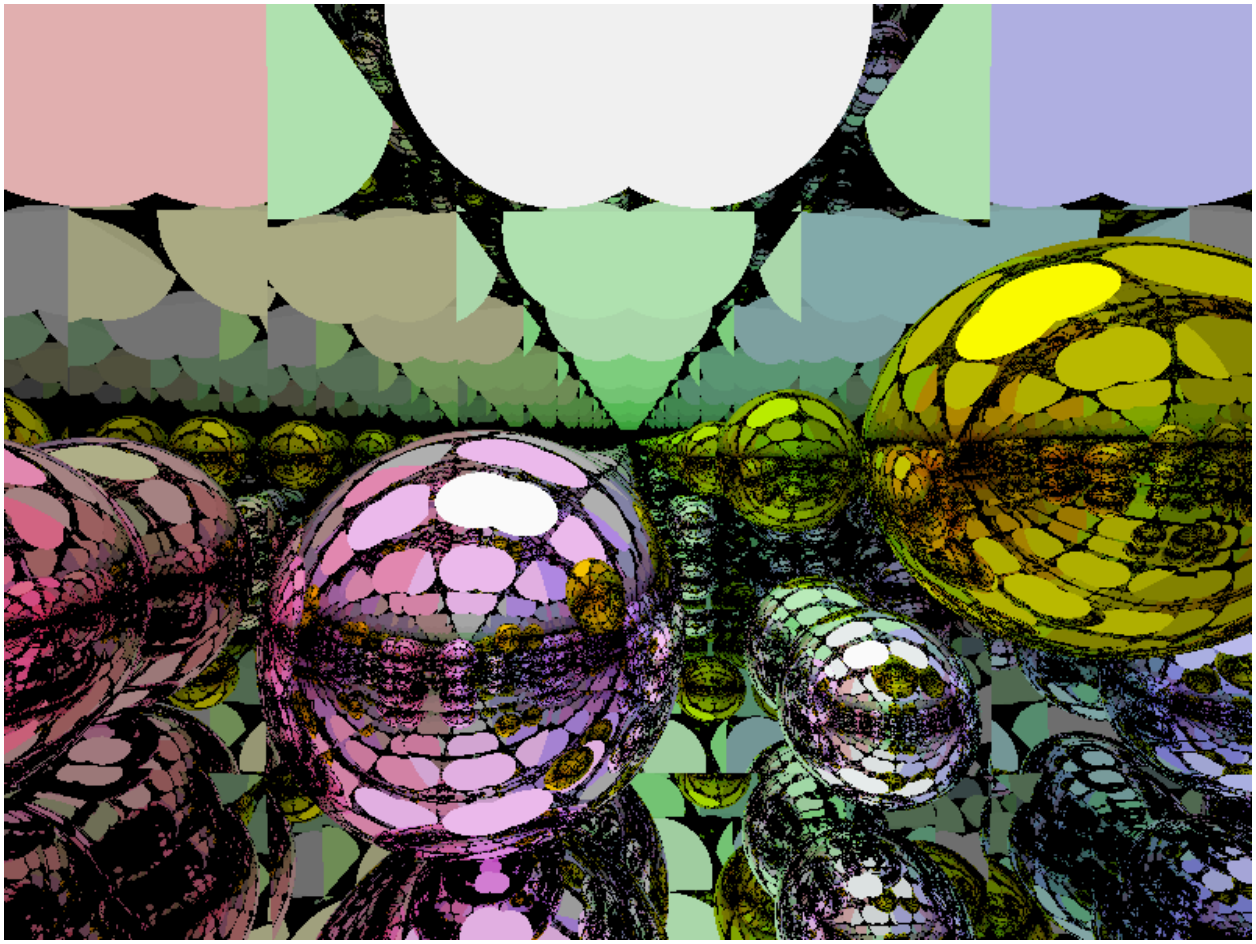


3. Reflection model

The reflection model implemented in the ray tracer aims to simulate the behavior of light as it interacts with objects in the scene. Unlike simpler models, which may only consider direct illumination, this realistic tracer incorporates reflections, mimicking real-world light propagation. At its core is a recursive function designed to handle light reflection upon hitting the nearest object.

When a ray intersects an object, the recursive function calculates the reflection of light based on the object's surface properties. If the intersected object is a light source, the function returns the color of that light. However, if the number of bounces, representing the depth of recursion, exceeds a predefined limit ('MAXBOUNCE'), the function terminates and returns black, preventing infinite recursion. Additionally, if the ray does not intersect any object in the scene, the function also returns black.

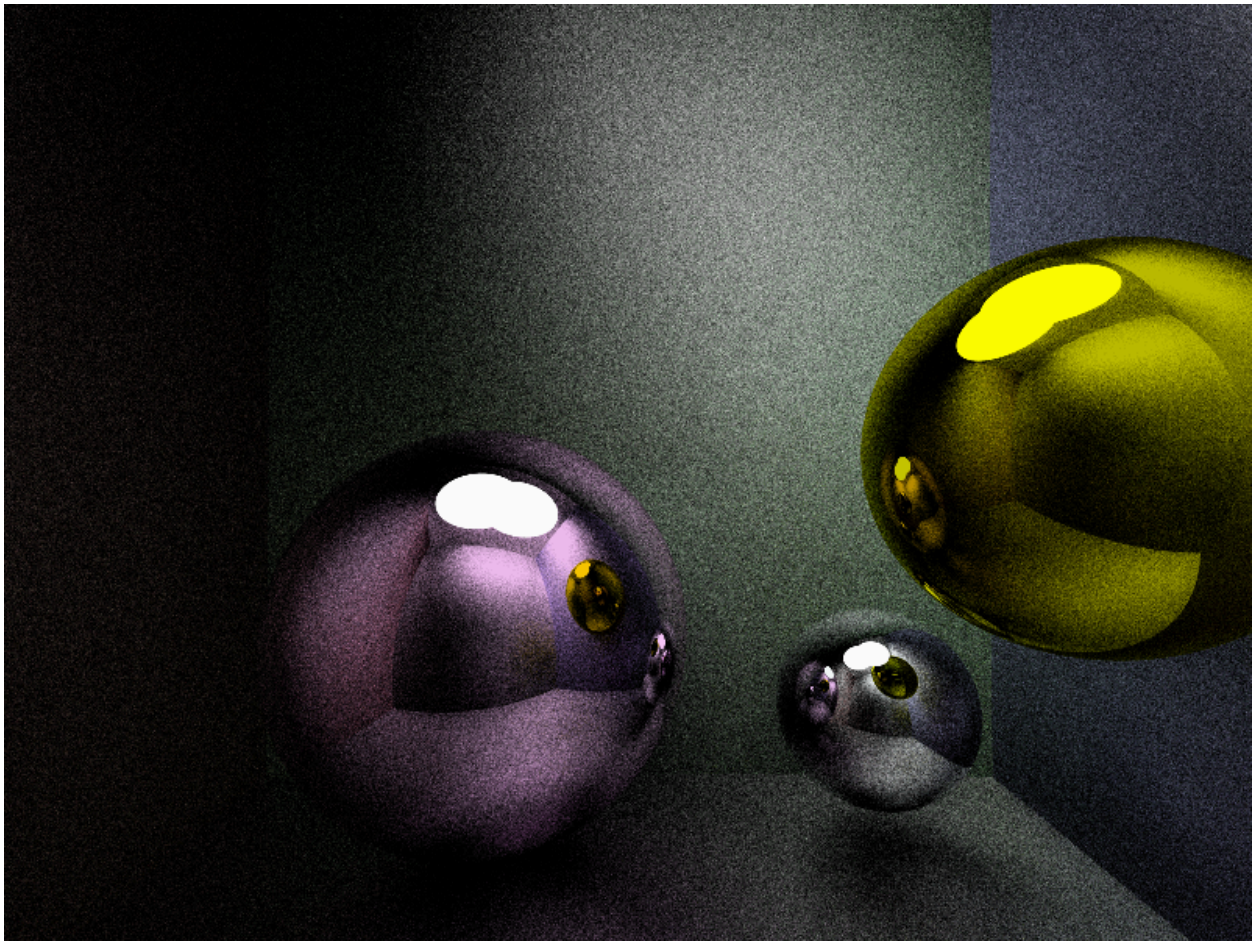
The implementation of the reflection logic was guided by a reference, linked here, which provided insights into handling reflections in a ray tracing context. The resulting code output demonstrates the effects of purely reflective surfaces within the rendered scene.



4. Adding diffusion property

In the latest iteration of the ray tracer, an important enhancement has been introduced to simulate the real-life behavior of surfaces: the diffusion property. Unlike the previous version, where walls exhibited perfect mirror-like behavior, this version incorporates roughness to surfaces, like the natural variability observed in real-world objects.
The diffusion effect is achieved by modifying the reflection vector with a random 3D vector if the surface is assigned a roughness parameter value greater than 0. By adding this random vector to the perfect reflection vector, the rays bounce off the surface in a more randomized manner, mimicking the irregular scattering of light on rough surfaces.
Furthermore, to enhance the visual realism of rendered images, anti-aliasing has been implemented to reduce image granularity. In the earlier version, each ray traced only a single path, leading to sharp and sometimes pixelated images. However, in this iteration, for every ray, multiple rays are traced in the vicinity (determined by the 'neighbour_per_pixel' parameter) and their color values are averaged. This averaging process smoothens the image, resulting in a more visually appealing output with reduced pixelation.

The output of this current model version showcases the combined effect of diffusion property implementation and image post-processing, resulting in rendered images that closely resemble real-world scenes with smoother surfaces and reduced granularity.



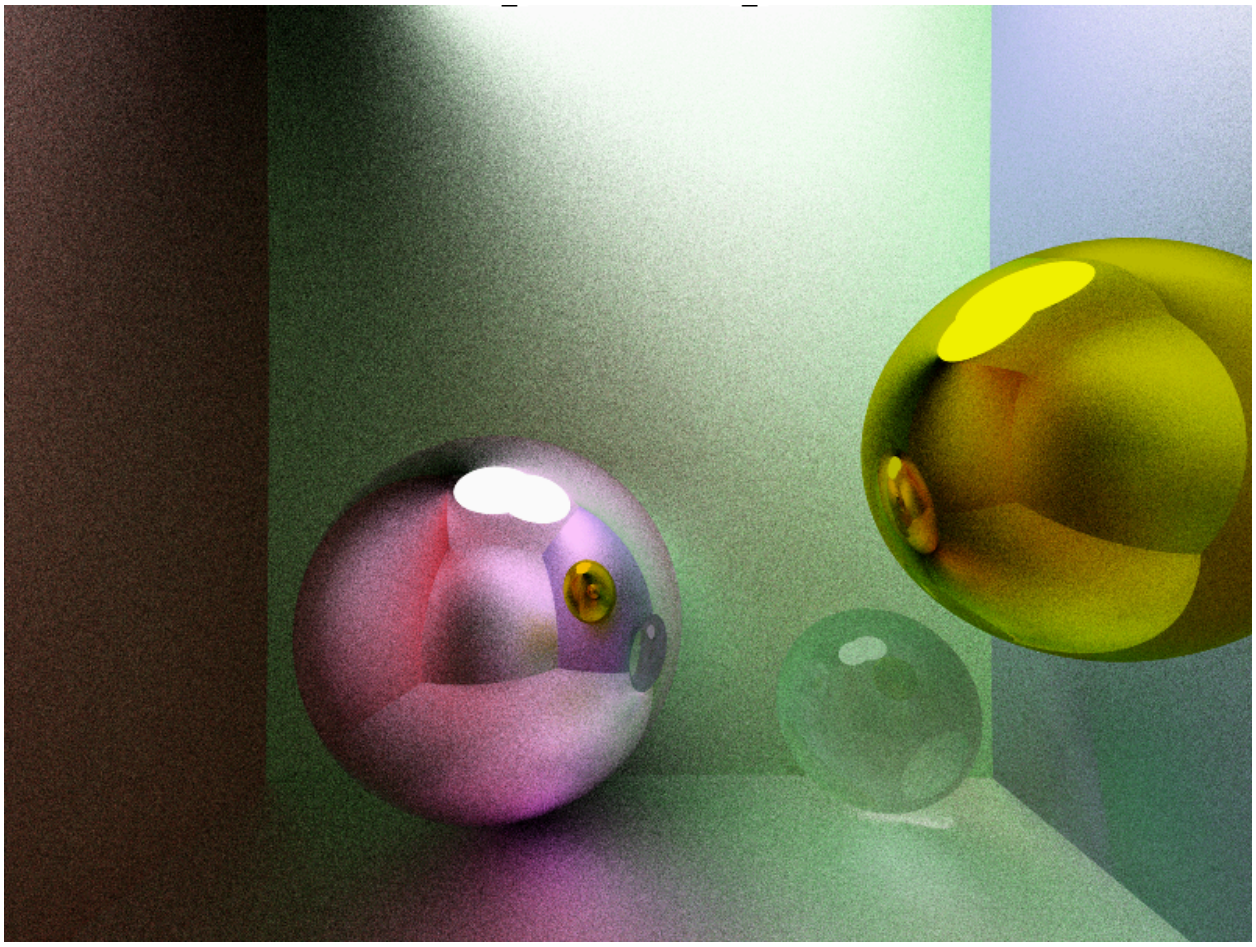
5. Adding transparent objects

In this latest iteration of the ray tracer, significant advancements have been made to introduce transparency to objects within the scene. Unlike the previous versions where objects behaved solely as opaque surfaces, transparent objects are now supported, adding a new level of realism to the rendered images.

Transparent objects in the scene now cause rays to split into two distinct paths upon interaction: a reflected ray and a refracted ray. The proportion of the incident light that is refracted versus reflected can be adjusted using parameters specified in the sphere constructor. Additionally, the refractive index parameter influences the path taken by the refracted ray, simulating the bending of light as it passes through different mediums.

To maintain physical accuracy and realism, total internal reflection is automatically handled within the tracer. This means that when light encounters a surface at a certain angle beyond which total internal reflection occurs, the ray is reflected back rather than refracted through the

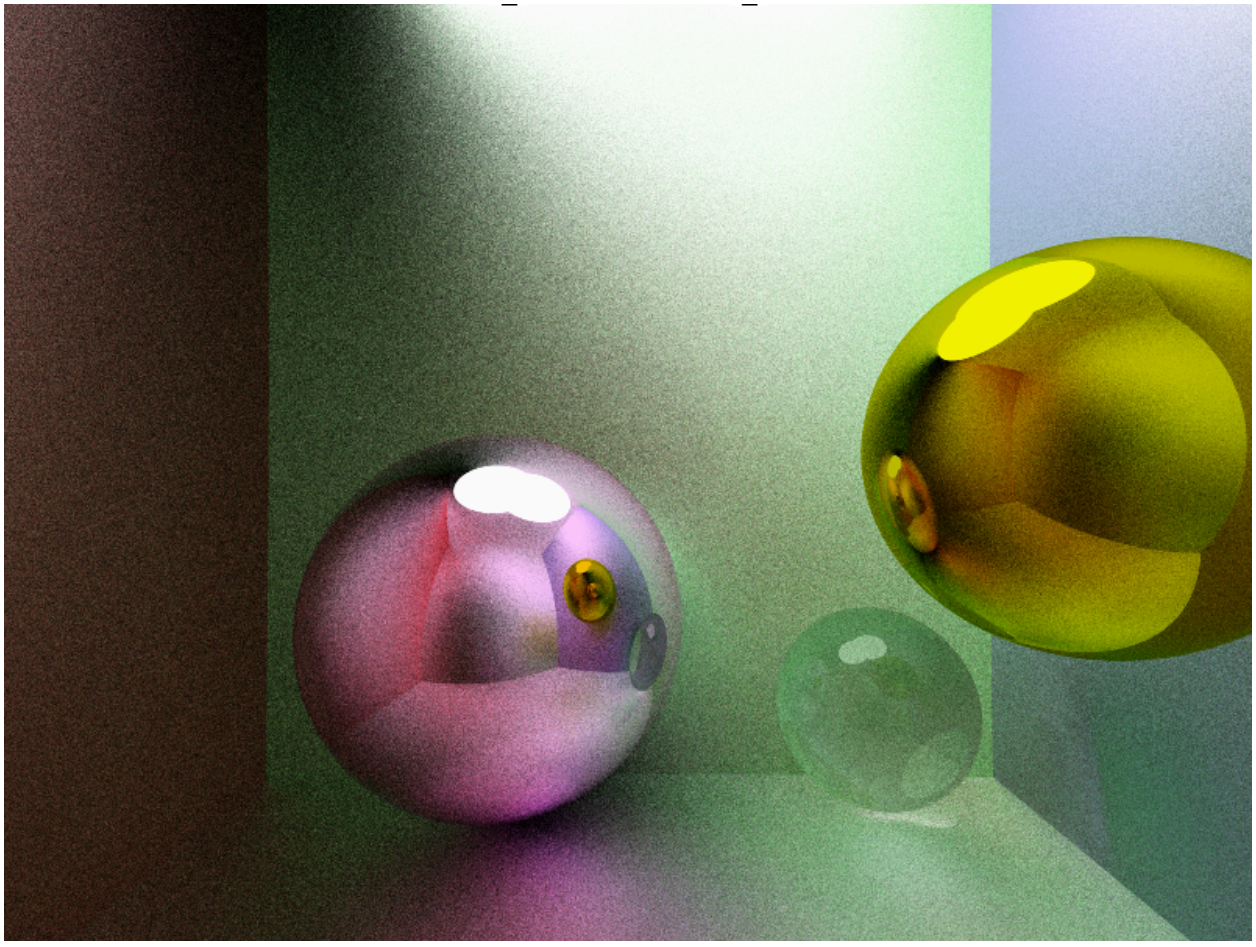
surface. This behavior is crucial for maintaining the integrity of the rendered scenes and ensuring they adhere to real-world physics principles.
However, it's important to note that the introduction of transparent objects has led to a notable increase in computational complexity and rendering time. This is primarily due to the doubling of recursive calls at each step, as both reflection and refraction paths need to be computed. As a result, this version may exhibit slower performance compared to previous iterations of the tracer. Despite the increased computational overhead, the output of this current model version demonstrates the capability of the ray tracer to accurately simulate the behavior of transparent objects within the scene, leading to visually compelling and realistic rendered images.



6. Increasing speed

In this latest version of the ray tracer, the focus is on speeding up the rendering process through parallelization. By utilizing multiple threads, the tracer can perform ray-tracing calculations concurrently, significantly reducing the time required for image generation. Initially, a set number of threads are created, with each assigned a portion of the image grid to process simultaneously. This parallel approach distributes the computational workload efficiently, resulting in faster rendering times. To avoid conflicts between threads, a temporary 2D array is employed to store RGB values computed by each thread for individual pixels. This method has

resulted in a decrease in the time taken for image generation from 50 minutes (for the previous image) to 2.5 minutes (the image below) - almost a 20 fold decrease in execution time, while maintaining rendering quality.



7. Final tweaks

This model conforms to the assignment input format by reading an input file, and generates the spheres accordingly. Other minor changes include

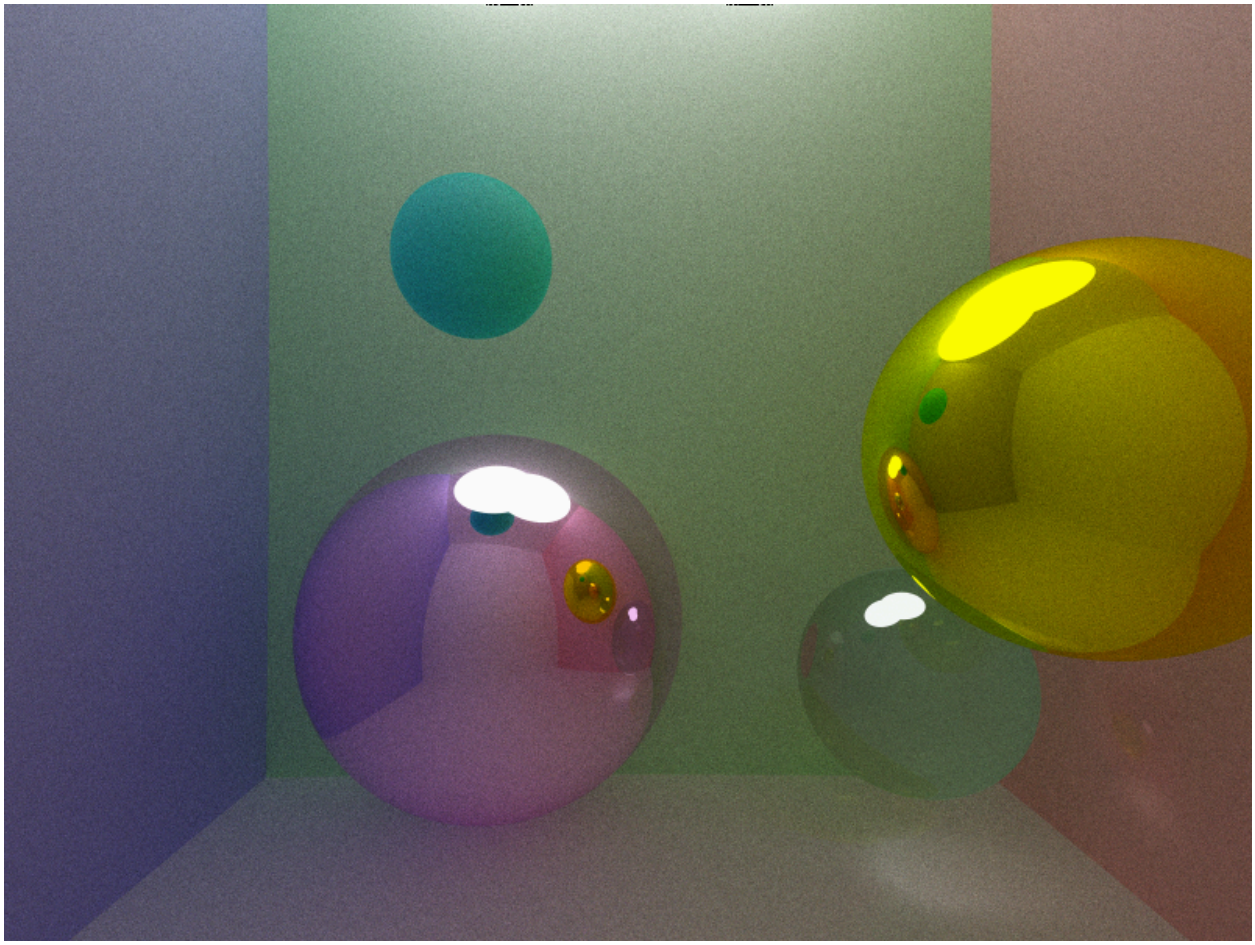
1. Changing the random vector function to allow generation of vectors with negative values (which was why the above images were illuminated less in the left hand side of images).
2. Execution time prediction code has been added. These predictions come with healthy margins to accommodate variations in computing environments. On average, the current implementation achieves rendering times between half to one-third of the expected duration on both Macbook Air M1 and HP Victus 16 - AMD Ryzen 5 5600H systems.
3. Finally, a pleasant non-speech sound has been introduced to signify the completion of image generation, adding a touch of user-friendly feedback to the rendering process.

Sample Input and Outputs

1. Example 1

Input:
60, 60, 180, 35, 0.999, 0, 250, 255, 255, 0.9, 1.5, 0
50, 130, 120, 45, 0.999, 0, 255, 255, 0, 0, 0, 0
200, 70, 220, 70, 0.999, 0, 255, 200, 255, 0, 0, 0
190, 180, 140, 20, 0.999, 1.4, 55, 250, 230, 0, 0, 0

Output:

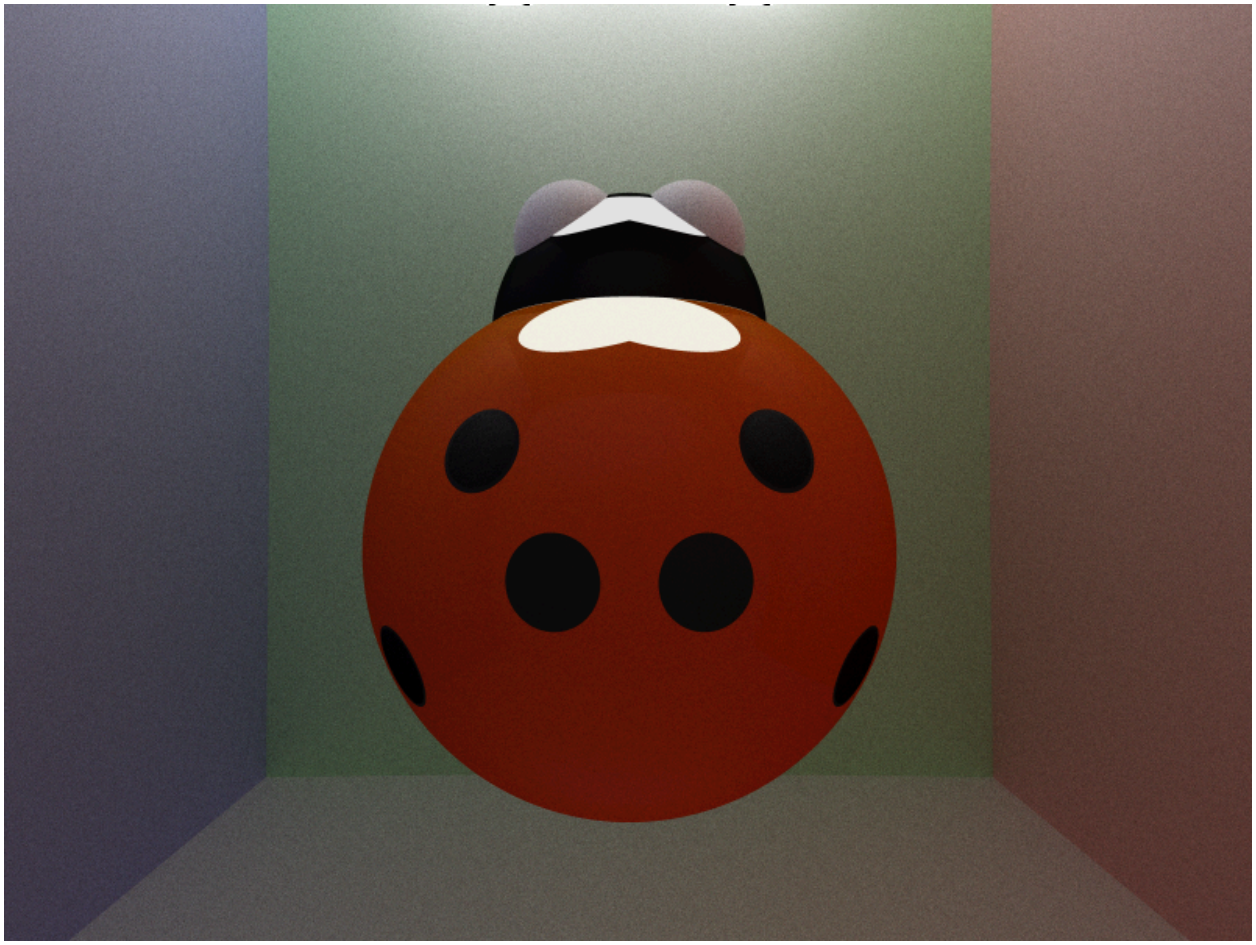


2. Example 2 - ladybug

Input:
150, 110, 120, 55, 0.999, 1.4, 255, 50, 0, 0, 0, 0
150, 110, 120, 55.1, 0.999, 0, 255, 255, 255, 0.9, 1.5, 0
150, 155, 120, 30, 0.999, 1.4, 0, 0, 0, 0, 0, 0

150, 155, 120, 30.1, 0.999, 0, 255, 255, 255, 0.9, 1.5, 0
162, 173, 110, 12, 0.999, 1, 255, 255, 255, 0, 0, 0
138, 173, 110, 12, 0.999, 1, 255, 255, 255, 0, 0, 0
162, 120, 95, 26, 0.999, 1, 20, 20, 20, 0, 0, 0
138, 120, 95, 26, 0.999, 1, 20, 20, 20, 0, 0, 0
172, 100, 100, 24.3, 0.999, 1, 20, 20, 20, 0, 0, 0
128, 100, 100, 24.3, 0.999, 1, 20, 20, 20, 0, 0, 0
152, 110, 110, 44.9, 0.999, 1, 20, 20, 20, 0, 0, 0
148, 110, 110, 44.9, 0.999, 1, 20, 20, 20, 0, 0, 0

Output:



How to Run Code

Input file format:

The Input file format takes as input 12 numbers per line. The numbers denote:

1. Sphere's center - x: Typically between 0 to 300 due to room's width
2. Sphere's center - y: Typically between 0 to 350 due to room's height
3. Sphere's center - z: Typically between 0 to 240 due to room's depth
4. Sphere's radius
5. Sphere's light loss: Percentage of light lost by interacting with the sphere, also used by lights for representing illumination intensity (typically greater than 1 for lights)
6. Sphere's roughness: 0 for perfectly smooth spheres and 1.4 works great for walls.
7. Sphere's colour - red: Between 0 and 255
8. Sphere's colour - green: Between 0 and 255
9. Sphere's colour - blue: Between 0 and 255
10. Sphere's transparency: Restricted between 0 to 1. 0 for solid spheres, 0.9-0.99 for transparent objects. 1 indicated transparent object that does not reflect any light (no real object does that)
11. Sphere's refractive index: Value does not matter for solid spheres, should be greater than 1 for transparent objects. 1 means that it does not affect light - meaning it is invisible. Value less than 1 also does not occur in nature.
12. isLight: 0 if normal reflecting and/or refracting sphere and 1 if it is a light source.

Note:

1. Each line with 12 inputs will be counted as a sphere.
2. Lights are already added to roof with illumination intensity 10.
3. Room's walls are added according to dimensions specified above.

How to execute:

1. Open terminal / command prompt / powershell
2. Type "pip install pillow" (to be done once if pillow is not already installed)
3. Type "pip install numpy" (to be done once if numpy is not already installed)
4. Add input numbers to "sphere_initialization.txt"
5. Type "sh v3.2.sh" in terminal / command prompt / powershell
6. Done! See output in folder.

Power Using:

The default parameters of the ray-tracer are optimized to give decent renders in less amount time. However, these parameters can be changed in a code editor, to suit your needs:

1. Change image resolution: This keeps the image in 40:30 ratio but increases/decreased the resolution (default 800:600). Eg: 1080:720 or 1440:1080 or 3840:2880.
   Adjust the “ray_per_pixel” variable to change the resolution, shown in the image below at line number 427:

```
426
427     int ray_per_pixel = 20; // Adjust the number of samples per pixel
428     int neighbour_per_pixel = 7; // Adjust to smoothen image
```

2. Change reflected / refracted light depth: Increase in depth leads to increase the detail in refractions and reflection, while also increasing overall illumination.
   Adjust the “MAXBOUNCE” variable shown in the image below at line number 25:

```
24
25    #define MAXBOUNCE 15
26
```

3. Change render quality: Increasing iterations leads to better image quality.
   Adjust “iterations” variable shown in the image below at line number 429:

```
428     int neighbour_per_pixel = 7; // Adjust to smoothen image
429     int iterations = 2; // Number of times to iterate over all pixels
430
```

4. Change image granularity: Increasing neighbouring rays per pixel (amount of anti-aliasing) leads to smoother and better image quality.
   Adjust “neighbour_per_pixel” variable shown in the image below at line number 428:

```
427     int ray_per_pixel = 20; // Adjust the number of samples per pixel
428     int neighbour_per_pixel = 7; // Adjust to smoothen image
429     int iterations = 2; // Number of times to iterate over all pixels
```

5. Change field of view of camera: This is equivalent to changing the focus length in a camera, but without the blurring effects. Values closer to 0 lead to a wider field of view.
   Adjust the third number in vect orig (here -20) at line number 355:

```
354   void renderImagePart(int startRow, int endRow,
355       vect orig(150, 125, -20);
356       float divider = 1.0/(float)iter;
```

References:

1. Steven Parker, William Martin, Peter-Pike J. Sloan, Brian Smits, Peter Shirley, Charles Hansen; Interactive Ray Tracing; ACM; 1999
2. Odom, C.N.S., Shetty, N.J., Reiners, D.; Ray Traced Virtual Reality; Springer; 2009
3. James Arvo; Backward Ray Tracing; ACM; Chelmsford, MA; 1986
4. Paul S. Heckbert, Pat Hanrahan; Computer Graphics; Old Westbury, NY ; Volume 18, 1984, 119-127
5. https://my.eng.utah.edu/~cs6958/
6. https://www.scratchapixel.com/index.html
7. https://omaraflak.medium.com/ray-tracing-from-scratch-in-python-41670e6a96f9
8. https://raytracing.github.io/books/RayTracingInOneWeekend.html
9. https://math.stackexchange.com/questions/2334939/reflection-of-line-on-a-sphere

Additional Outputs: