# Implementation of Ray Tracing in C

The following describes the system architecture for a ray tracing application that has been implemented in C as a final project in COSC 3P98. It highlights the key aspects of organisation and interactions of the various components that achieves a realistic rendering of a scene containing objects with features including lighting, material effects/shading, shadows, and reflections through the backward tracing of representations of light rays from a light source to a viewer's eye.

## Components and Functions/Methods:

The major components of the system include:

Methods called to read in scene data from input file and allow the scene to be represented in memory and printing out the stored scene.

Methods to compute pre-rendering constants such as the normal and distance properties of triangles defined in the scene data, so that these do not need to be recomputed for every pixel/ray.

A Method sets up the scene/camera perspective, creates a file for image output and calls the ray tracing algorithm/function proper for each ray passing through a pixel in the image, writing image data as colours that are returned from the latter.

Functions are called within the ray trace function that return the direction vectors for the reflective rays that are then themselves traced through recursive calls.

The helper functions perform various calculations and/or modifications involving the constituent vectors of the traced rays or objects, the most important of which are for ray-object intersection checks. Many return material properties of objects to be used in shading or calculating direction vectors for recursive tracing. One function returns a sky gradient background depending on the direction of the ray passed to it.

## Data Structures:

The system utilizes numerous structs in C to represent colours, objects and their material properties, lights and their properties (apart from the ambient light coefficient, which is represented by a global constant double value), rays and their constituent vectors.

Vectors: three double values x, y, z for a point in 3D space. The camera position is defined with a vector.

Rays: an origin and direction vector specifying ray origin and direction of travel along 3D space.

Lights: direction vector, and vectors for specular and diffuse light colour intensities.

Colours: a vector containing the information for red, green, and blue channels. The background colour is specified using a colour vector, as well as the material properties of objects.

Objects: subdivided into several types including spheres, planes, and triangles, each having slightly different defining properties. These are stored in a dynamically allocated array at runtime:

> Spheres: a vector for centre in 3D space, a double for radius, followed by material properties (see below).
>
> Planes: a vector for unit normal and a double indicating distance from plane parallel with the origin, followed by material properties.
>
> Triangles: three vectors defining its vertices and edges, followed by its unit normal vector and a double distance value, and finally its material properties. Note that the normal and distance are computed at runtime and are given placeholder values of 0 by the user in the input file.
>
> Object material properties: Each object has the same definitions for its material properties: Colour vectors for diffuse, specular, and ambient colour components, followed by a double for the object's reflection coefficient, shininess (specular highlight coefficient), reflection coefficient, and index of refraction respectively.

Other elements of the system also use global variables, such as an integer for the number of objects in the scene, the type of background to be used, and a small value (epsilon) used for origin vector offsets.

Scene Description:

The scene is represented in terms of a cartesian coordinate system. The default view has the camera located at the origin facing towards the positive z-axis. The user can specify a different location of the camera in the scene data input file but changing the orientation of an axis would require changing the code to accommodate, otherwise the image will appear flipped along one or more axes. For example, the x and y axes would flip if the orientation along the z-axis was changed.

The image screen through which the rays are cast are mapped to the pixel coordinate system, taking in account the aspect ratio so that objects don't become distorted in the rendered images and the user can specify the image height and width.

The user can specify the objects in the scene including spheres, planes, and polygons (triangles) by entering the components that make up the data structures that describe the properties of these objects in the code. The scene data given or chosen by a user via text file is formatted as follows, where each line is space delineated to separate fields:

Line one gives the number of objects to be included in the scene, this will be used later to dynamically allocate space for objects in the objects array.

Line two gives the camera position in x, y, z coordinates.

Line three specifies the desired result image width and height.

Line four gives the light properties starting with the direction vector followed by the vectors for specular and diffuse colour intensities.

Line five gives the ambient light coefficient for the scene.

Line six defines whether to use a sky gradient background or the user supplied background colour. Relevant if the scene being rendered is meant to depict an indoor or outdoor environment.

Line seven defines a background colour for the scene to be used when the sky gradient is not desired.

Lines eight onward give the properties for the objects in the scene, one per line. The first entry on a line gives the type of object with an integer that is mapped to the corresponding enumerated object type. This is followed by the object properties given in the order outlined above in the section on data structures for objects.

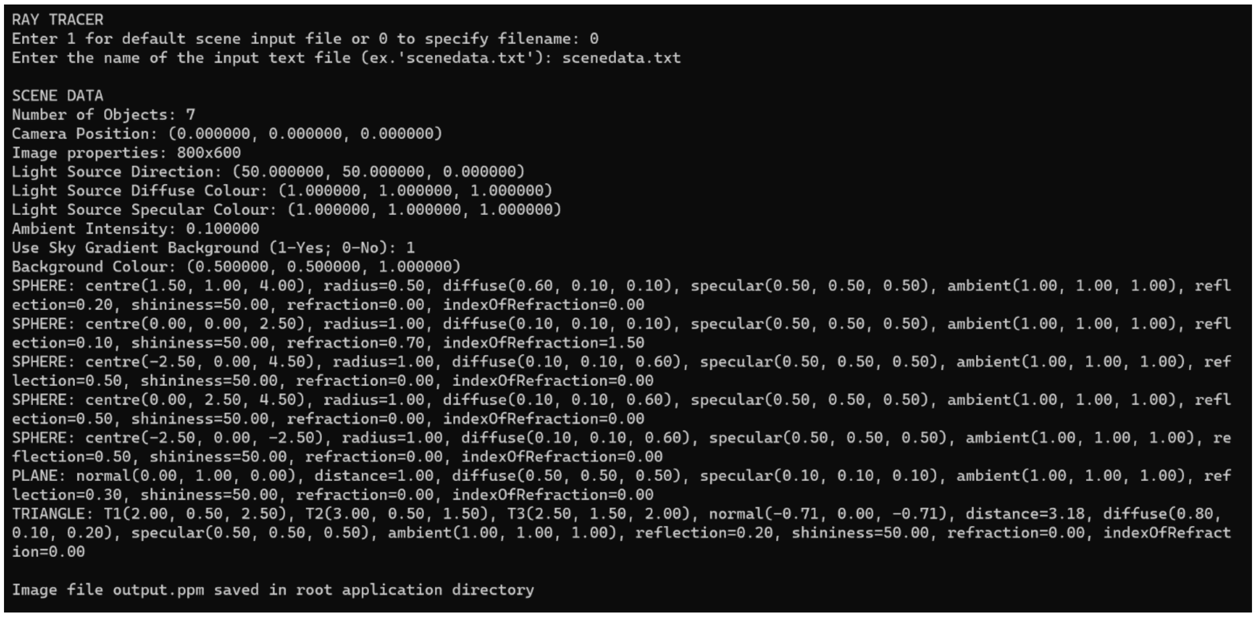
```
7
0.0 0.0 0.0
800 600
50.0 50.0 0.0 1.0 1.0 1.0 1.0 1.0 1.0
0.1
1
0.5 0.5 1.0
0 1.5 1.0 4.0 0.5 0.6 0.1 0.1 0.5 0.5 0.5 1.0 1.0 1.0 0.2 50.0 0.0 0.0
0 0.0 0.0 2.5 1.0 0.1 0.1 0.1 0.5 0.5 0.5 1.0 1.0 1.0 0.1 50.0 0.7 1.5
0 -2.5 0.0 4.5 1.0 0.1 0.1 0.6 0.5 0.5 0.5 1.0 1.0 1.0 0.5 50.0 0.0 0.0
0 0.0 2.5 4.5 1.0 0.1 0.1 0.6 0.5 0.5 0.5 1.0 1.0 1.0 0.5 50.0 0.0 0.0
0 -2.5 0.0 -2.5 1.0 0.1 0.1 0.6 0.5 0.5 0.5 1.0 1.0 1.0 0.5 50.0 0.0 0.0
1 0.0 1.0 0.0 1.0 0.5 0.5 0.5 0.1 0.1 0.1 1.0 1.0 1.0 0.3 50.0 0.0 0.0
2 2.0 0.5 2.5 3.0 0.5 1.5 2.5 1.5 2.0 0.0 0.0 0.0 0.0 0.8 0.1 0.2 0.5 0.5 0.5 1.0 1.0 1.0 0.2 50.0 0.0 0.0
```

*Figure 1: Example scene data input file*

<u>User Interface and Interaction:</u>

Upon running main.c the console will ask the user to choose between using the default scene data input file or specifying another input file. The application will then either use the default scenedata.txt file or prompt the user to enter the filename of the file with the scene data they wish to use for ray tracing. The files must be located in the root application directory.

The console will then print out the scene data that has been read from the input file for user confirmation and reference. When the application has finished generating the image file, a confirmation will be printed to the console to inform the user of the name of the output file and that it has been saved to the root application directory.



```
RAY TRACER
Enter 1 for default scene input file or 0 to specify filename: 0
Enter the name of the input text file (ex.'scenedata.txt'): scenedata.txt

SCENE DATA
Number of Objects: 7
Camera Position: (0.000000, 0.000000, 0.000000)
Image properties: 800x600
Light Source Direction: (50.000000, 50.000000, 0.000000)
Light Source Diffuse Colour: (1.000000, 1.000000, 1.000000)
Light Source Specular Colour: (1.000000, 1.000000, 1.000000)
Ambient Intensity: 0.100000
Use Sky Gradient Background (1-Yes; 0-No): 1
Background Colour: (0.500000, 0.500000, 1.000000)
SPHERE: centre(1.50, 1.00, 4.00), radius=0.50, diffuse(0.60, 0.10, 0.10), specular(0.50, 0.50, 0.50), ambient(1.00, 1.00, 1.00), refl
ection=0.20, shininess=50.00, refraction=0.00, indexOfRefraction=0.00
SPHERE: centre(0.00, 0.00, 2.50), radius=1.00, diffuse(0.10, 0.10, 0.10), specular(0.50, 0.50, 0.50), ambient(1.00, 1.00, 1.00), refl
ection=0.10, shininess=50.00, refraction=0.70, indexOfRefraction=1.50
SPHERE: centre(-2.50, 0.00, 4.50), radius=1.00, diffuse(0.10, 0.10, 0.60), specular(0.50, 0.50, 0.50), ambient(1.00, 1.00, 1.00), ref
lection=0.50, shininess=50.00, refraction=0.00, indexOfRefraction=0.00
SPHERE: centre(0.00, 2.50, 4.50), radius=1.00, diffuse(0.10, 0.10, 0.60), specular(0.50, 0.50, 0.50), ambient(1.00, 1.00, 1.00), refl
ection=0.50, shininess=50.00, refraction=0.00, indexOfRefraction=0.00
SPHERE: centre(-2.50, 0.00, -2.50), radius=1.00, diffuse(0.10, 0.10, 0.60), specular(0.50, 0.50, 0.50), ambient(1.00, 1.00, 1.00), re
flection=0.50, shininess=50.00, refraction=0.00, indexOfRefraction=0.00
PLANE: normal(0.00, 1.00, 0.00), distance=1.00, diffuse(0.50, 0.50, 0.50), specular(0.10, 0.10, 0.10), ambient(1.00, 1.00, 1.00), ref
lection=0.30, shininess=50.00, refraction=0.00, indexOfRefraction=0.00
TRIANGLE: T1(2.00, 0.50, 2.50), T2(3.00, 0.50, 1.50), T3(2.50, 1.50, 2.00), normal(-0.71, 0.00, -0.71), distance=3.18, diffuse(0.80,
0.10, 0.20), specular(0.50, 0.50, 0.50), ambient(1.00, 1.00, 1.00), reflection=0.20, shininess=50.00, refraction=0.00, indexOfRefract
ion=0.00

Image file output.ppm saved in root application directory
```

*Figure 2 Example application interface*

<u>Rendering Pipeline:</u>

1. Primary Ray Generation

In the main function, the scene data text file is specified, read, and the values are assigned to the corresponding data structures/global variables. Constants used in calculations such as triangle plane unit normal and distance are computed. The output image file is opened, its header written, and then the image is generated pixel by pixel based on the image size specified by the user.

Two for loops iterate through the pixels. Pixels are normalised between the values of 0 and 1, and then these are mapped to the screen space with x and y values between -1 and 1. The z value is always set to 1 for casting rays towards the positive z axis. The x value is scaled based on the aspect ratio of the image to prevent distortion.

A ray is initialized for each pixel with the camera position as origin and the pixel vector as direction. The direction vector is then normalised, and this primary ray is sent through the ray tracing function along with a recursion depth value of 0 to receive the colour to be rendered for that pixel.

2. Ray-Object Intersection

The primary ray is first tested for any object intersections from its origin at the camera position in its direction path. This is accomplished with a call to the ray intersection function from within the ray tracing function. The ray intersection function initialises the parameter of t at infinity. T represents the distance along the ray scaled to the direction vector.

As the objects of the scene are being checked for intersections with the ray any that are closer than the numerical representation of infinity will become the new value of t, until all the objects in the scene have been checked and the closest object to the origin of the ray will correspond to the closest t value.

The ray object intersection function calls other functions that specialise in calculating intersections for each of the object types in the scene as it iterates through the scene object list. Each time t is updated with a closer object, the values of the variables in the ray tracing function for the index of the intersected object, the hit point of the intersection, and surface normal of the object at the hit point that were sent as arguments to the ray object intersection function are all updated by the specific object type intersection function that was called.

The return type of the ray object intersection function is an integer, so it simply returns whether an intersection was found. If none were found, the ray tracing function returns the specified background colour, or a sky gradient value computed using the primary ray's direction.

3. Material Shading and Shadow Rays

When a ray-object intersection has been found and the details have been returned to the ray tracer function, the shading function is then called to return the local colour for that pixel with the index of the object, hit point, and hit point surface normal as arguments.

The first step of shading is to calculate the ambient light at the point using the global variable for ambient light intensity of the scene multiplied by the ambient light coefficient of the object.

At this point the shadow ray is cast from the hit point to see whether the point is in shadow. The shadow ray uses the hit point as origin (with a small offset in the direction of the hit point surface normal to prevent shadow acne) and the normalised vector from the hit point vector subtracted from the light direction vector, which is passed to the shadow ray intersection function which iterates through the object list looking for an intersection with any object including the hit point object. It uses a switch statement to differentiate the object types. If an object is transparent (by having a refraction coefficient >0 and an index of refraction of 1) it's ignored. You can also specify an opaque object not cast shadows by leaving its refraction coefficient at 0 but setting its index of refraction to 1. The function can also accept an index for an object to prevent self-intersections, if needed. As with the other intersection functions, it returns an integer indicating whether an intersection was found. If so, the shading function only returns the ambient light colour components for the pixel, as diffuse and specular light are only computed if the cosine of the angle between the light source and the surface normal are greater than zero.

If there is no intersection found for the shadow ray and cosine is greater than zero, the diffuse and specular colour components are computed for the local lighting at the pixel. The object's diffuse and specular material properties are fetched by the helper functions for these purposes and used in the appropriate calculations for these lighting components along with other values that must be computed by the shading function. The final colour returned for the local lighting in this case is the sum of the three components. If the cosine condition earlier wasn't met, again only the ambient lighting component is returned to the ray tracer function.

4. Reflection and Refraction Rays

With the local colour computed, the ray tracing function initialises the reflective colour vector and calls a function to check if the hit point object has a reflective surface, indicated by the object's reflective coefficient, a part of its material properties. This function merely checks that the former is above 0 and returns an integer to indicate. The reflective vector is declared with an assignment from the result of the reflection vector calculation function, which takes the primary ray, hit object index, hit point, and surface normal as arguments.

This function determines the reflective direction vector by calculating the cosine of the incidence direction vector and the hit point surface normal using the dot product helper function and then returns the normalised result of the reflective ray calculation using this cosine value along with calling the scalar, normalise, and subtract vector helper functions.

The ray tracing function now initialises the reflective ray using a calculated offset added to the hit point as origin and this returned direction vector and along with an incremented recursion depth uses these to recursively call the ray tracer function as an assignment value for the reflective colour to be computed for this pixel. This ray constitutes a secondary ray. When the result is returned, the reflection coefficient is fetched from the hit point object and the colour vector is scaled by it. This feature allows for the simulation of reflective surfaces on the objects in the scene.

Virtually the same process is used for the secondary refractive ray (if applicable, as of time of writing, this feature, though implemented, is being debugged and this process is bypassed by lack of positive fields in the scene data input file for refractive coefficients in object material properties). The function to calculate the refractive ray requires more computations and requires keeping track of which material is being entered/exited, and this is accomplished by computing the cosine of the angle of the incidence ray and the object hit point surface normal. This normal might need to be inverted depending on the result, utilising the invert vector helper function. The index of refraction for the refractive object is fetched by helper function. After another calculation, whether to compute the refractive ray is determined by whether the root term of the refractive ray direction is negative. If computed, the add vector helper function is used. The refractive direction vector is then returned to the ray tracing function and used for computing the refraction colour by recursive call to the ray tracer function. The refraction colour is then attenuated in the same manner as the reflection colour by the refraction coefficient of the hit object.

5. Recursive Ray Tracing

The secondary rays that call the ray tracer function for the reflective and refractive colours are recursively repeat the same steps outlined above until the recursive depth reaches a defined constant, a non-reflective/refractive surface is encountered, or no intersections are found. The results of the successive recursive calls are computed and eventually returned to the primary ray's function call.

The final step in the case when an intersection has been found is to combine the results of the local, reflective, and refractive colours, which is done by summing and clamping the sum within the range of zero and one.

6. Output Image

As the ray tracing function returns final colours with their red, green, and blue channels to the main function, the latter is casting and scaling these as integers within the range of 0 and 255 and writing them for each pixel that is to make up the output ppm image file. When all the pixels have been iterated through, the file is closed, and the user is notified of completion.

The dynamically allocated memory for the objects list and the user specified input file are freed up and the application exits.

<u>Extensibility and Modularity:</u>

There are obviously many more features that could be added to the system but were not due to lack of time for implementation and/or debugging. These include additional object types, multiple lights, GPU accelerated computing, textures, anti-aliasing, emissive lighting, and runtime optimizations. Unfortunately, most of these will have to be added in the future as a personal interest project.

Multiple lighting could be implemented by simply defining additional light objects and rather than making considerations for a single light in shading/shadowing, iterate through each light during these stages of rendering.

As of the time of this writing, refraction is in the process of implementation/debugging and all the required functions, recursive calls, and material properties are present in the code but are simply bypassed at execution by having the refraction coefficient for all objects in the scene set to zero. However, objects can be defined as transparent by using some reflection coefficient and setting the index of refraction to 1.

<u>Error Handling and Debugging:</u>

Error logs are printed to console for the user when input/output operations or dynamic memory allocation fails, and the application attempts to close gracefully.

The system has a debug flag that is used to toggle printing statements to console as rays move through the algorithm for the purposes of debugging in a pixel-wise fashion. One or several pixel locations in the scene can be sent to the ray tracer individually for this purpose.

Integration with External Libraries:

The system makes use of the standard C input/output library (stdio.h) for getting user input from the console and opening and reading in scene data from the input file and creating and writing to a ppm file for rendered image output. As well as the standard C math library (math.h) for use in checking ray-object intersections.

Example Images:

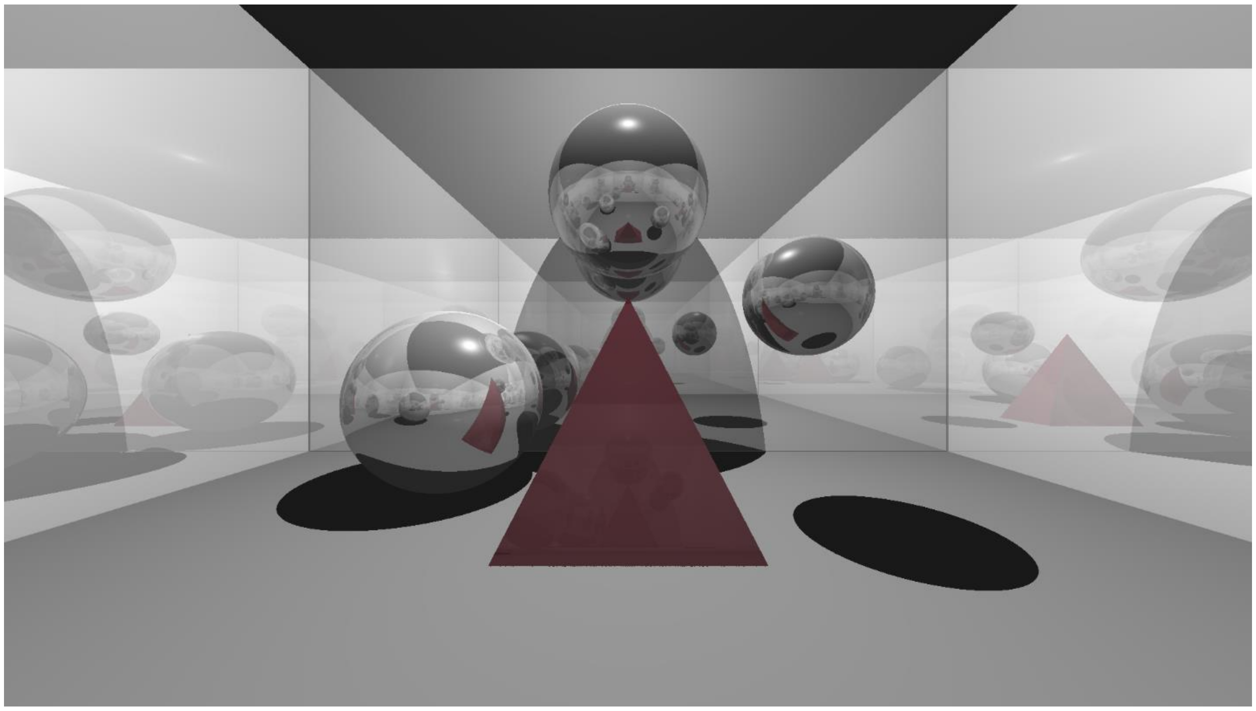
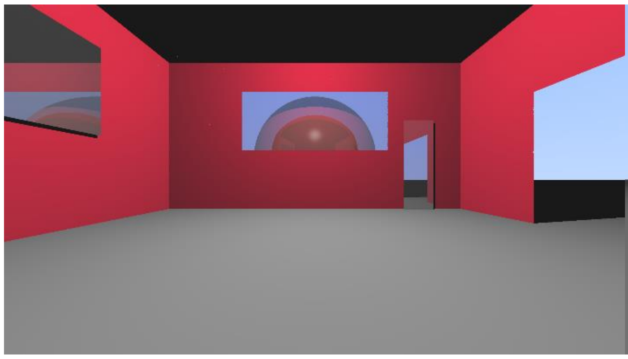


*Figure 3 Mirror room showing reflections with all object types.*



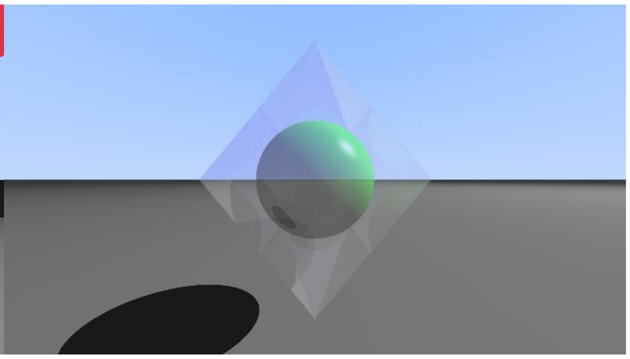


*Figure 4 Indoor room with mirrors and transparent window*

*Figure 5 Sphere in transparent pyramids*

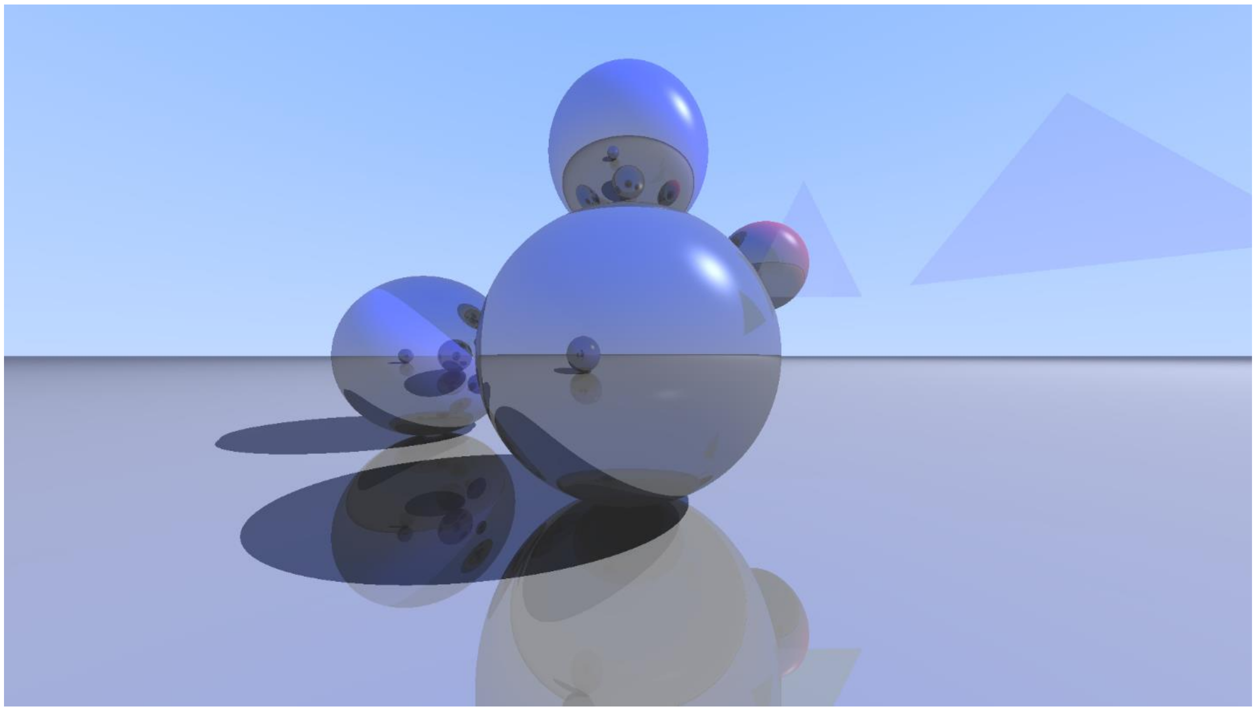


*Figure 6 Reflective spheres and plane, transparent triangles and sky gradient background*

References:

[1] COSC 3P98: Ray Tracing Basics, Brian Ross, 2023

[2] Wikipedia. Ray tracing. https://en.wikipedia.org/wiki/Ray_tracing_(graphics), 2023. Last accessed December 27, 2023.

[3] Point in triangle test. https://blackpawn.com/texts/pointinpoly/, 2023. Last accessed December 27, 2023.