CS488 Project Report

Title: Ray Tracer

Name: Yiduo Jing

Student ID: 20503408

User ID: y6jing

Purpose

This project extends the Assignment 4 ray tracer features and this report includes the manual and implementation for this project.

Statement

In this project, I added more features as the objective list shown at the end. I achieved cylinder/cone primitives adding, texture mapping, super sampling, mirror reflection, refraction and soft shadow, depth of field and phong shading. I did do the constructive solid geometry feature as well but it seems that there are some bugs/problems with my implementation which are not perfectly accomplished. I will talk about how I achieved those objectives and some guess or thought about why I was unable to achieve the CSG objective perfectly in the **Implementation** section.

Manual

The build instructions are simple as usual. Because this is an extension of assignment 4, I still keep the running program name as **'./A4'**. The Lua files that I made are all stored in the **Assets** folder, so please run **'./A4 Assets/xx.lua'** for testing the program with my scripts. The scripts for testing soft shadow require quite a lot of time due to distributed raytracing and it takes hours to finish running.

There are some new Lua commands along with extensions to the existing Lua commands are listed and explained in detail as below. The guide for the commands are important and should be strictly followed when making new scripts for testing my program. I assumed that only correct sets of arguments will be passed to each command since I did not do any argument number checking. I also assumed that no joint nodes are considered in this project. Implementations of features are discussed later in the **Implementation** section.

Primitives

- **gr.cylinder(name)** takes a string as the argument (code from line 224 in scene_lua.cpp) for creating a hierarchical cylinder with its center at (0, 0, 0), radius 1 and height 2.
- **gr.nh_cylinder(name, {x, y, z}, radius, height)** (code from line 322 in scene_lua.cpp) for creating a non-hierarchical cylinder with its center at (x, y, z) and user-defined radius and height in double value.
- **gr.cone(name)** takes a string as the argument (code from line 242 in scene_lua.cpp) for creating a hierarchical cone with its center at (0, 0, 0), radius 1 and height 2.
- **gr.nh_cone(name, {x, y, z}, radius, height)** (code from line 347 in scene_lua.cpp) for creating a non-hierarchical cone with its center at (x, y, z) and user-defined radius and height in double value.

- **gr.torus(name)** takes a string as the argument (code from line 206 in scene_lua.cpp) for creating a hierarchical torus with its center at (0, 0, 0), major radius 5, minor radius 2. The axis of revolution is default to be (0, 0, 1). This one is added as the Assignment 4 extra feature. I also added a command for creating a non-hierarchical torus.
- **gr.nh_torus(name, {x, y, z}, major radius, minor radius)** (code from line 297 in scene_lua.cpp) for creating a non-hierarchical torus with its center at (x, y, z) and user-defined major radius and minor radius in double value. The axis of revolution is also set to be (0, 0, 1) by default.

Material, Reflection, Refraction & Texture Mapping

- For realizing the mirror reflection and refraction, more arguments are added to the **gr.material** command (code from line 571 in scene_lua.cpp). Right now, the **gr.material** can accept up to 6 arguments.
  gr.material({dr, dg, db}, {sr, sg, sb}, p, reflectivity, outer index_of_refraction, inner_index_of_refraction) takes 6 arguments for creating a material. The outer index of refraction is the index of refraction for the medium(original medium) that the ray passes from and inner index of refraction is the index of refraction for the medium(new medium) that the ray passes to. Usually, I take 1 as the outer index of refraction and 1.53 for inner index of refraction if this is a glass material.
- For texture mapping, a new command **gr.texture(filename, {sr, sg, sb}, p, reflectivity, outer index_of_refraction, inner_index_of_refraction)** (code from line 610 in scene_lua.cpp) is added along with the **set_texture(texture)** (code from line 707 in scene_lua.cpp) command. The image file must be in PNG format. When setting a texture to a primitive, it replaces the material of the primitive to that texture. Please notice that setting a material uses **set_material** command while setting a texture uses **set_texture** command. Moreover, texture mapping only works for primitives and triangle mesh does not support this feature.

CSG (Not Perfectly Implemented)

- **gr.csg(node1, node2, csg_option)** command (code from line 169 in scene_lua.cpp) is added for constructing a CSG model and it will return a CSG node which is an inheritance of geometry node. The csg_option is a string with 3 options {‘union’ | ‘intersection’ | ‘diff’}. The node1 and node2 arguments should be geometry nodes **without parents.** It is important that when defining a CSG model, you always define the CSG node first then add the CSG node to a parent. For example, if I want to make a intersection model between a sphere and a cube, I firstly define a sphere and a cube **without** adding them as children of any node. Then I use gr.csg(sphere, cube, ‘intersection’) to create the CSG intersection model. This will return a new CSG node and add that CSG node to the parent. Some examples can be found in Assets/csg.lua.

Lighting

- For realizing the soft shadow, area light is needed. I extended the arguments for **gr.light** command (code from line 455 in scene_lua.cpp). Now the gr.light is used as follows:

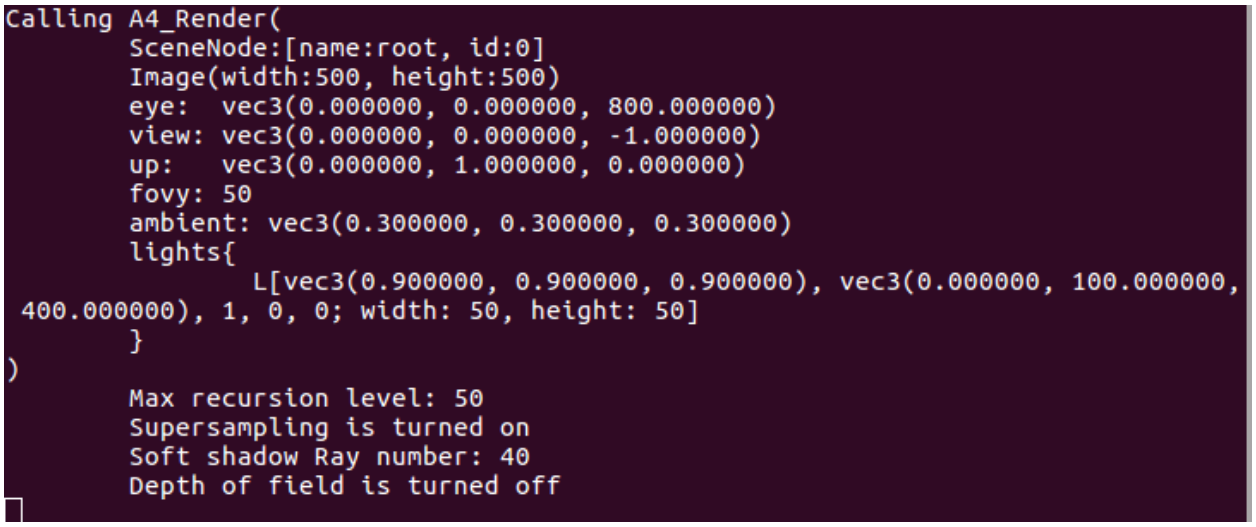
- **gr.light({x, y, z}, {r, g, b}, {c0, c1, c2})** creates a point light source located at (x, y, z) with intensity (r, g, b) and quadratic attenuation parameters c0, c1, c2.
- **gr.light({x, y, z}, {r, g, b}, {c0, c1, c2}, width, height)** creates an rectangular area light with its center located at (x, y, z) with user-defined width and height of the area along with the intensity (r, g, b) and quadratic attenuation parameters c0, c1 and c2.

Max Recursion level, Supersampling, Soft Shadow & Depth of Field

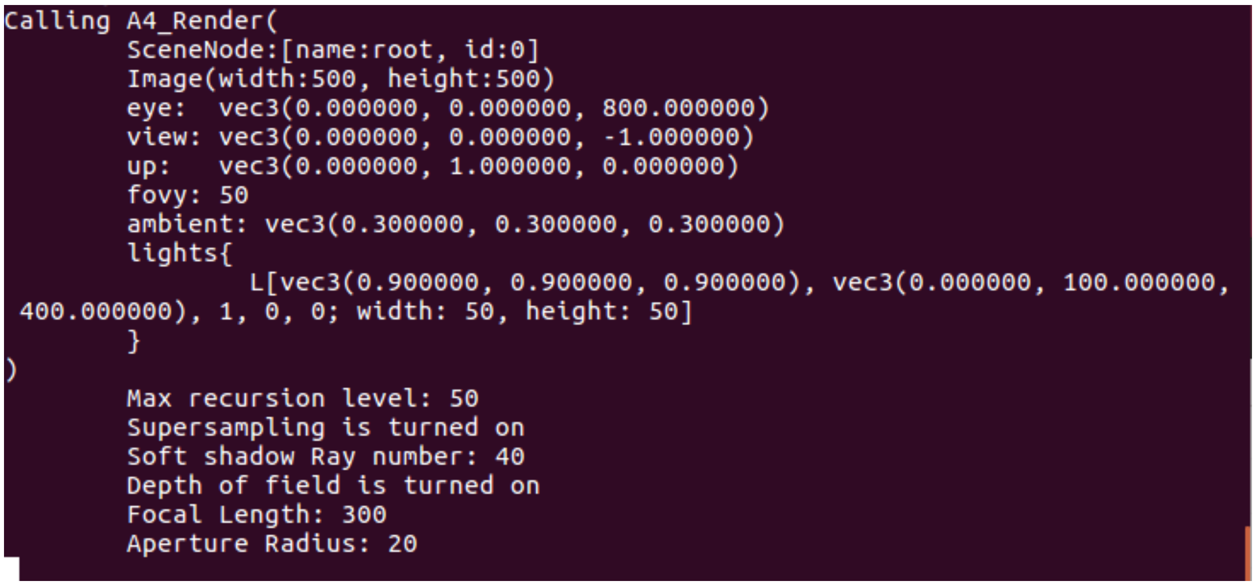
The **gr.render** command is extended. (code from line 496 in scene_lua.cpp)
Now the **gr.render** accepts up to 15 arguments as:

- **gr.render(node, filename, w, h, eye, view, up, fov, ambient, lights, max recursion level, supersampling flag, soft shadow ray number)**. The **max recursion level** defines a max integer for casting reflection and refraction rays. The supersampling flag is used for turning on/off the supersampling. By passing a number bigger than 0 will turn on the supersampling while passing 0 will turn off the supersampling and supersampling is set to be off by default. The **soft shadow ray number** defines the number of shadow rays that are needed to cast for achieving a soft shadow if there is an area light in the scene. By default, the soft shadow ray number is set to be 1.
- **gr.render(node, filename, w, h, eye, view, up, fov, ambient, lights, max recursion level, supersampling flag, soft shadow ray number, depth of field focal length, depth of field aperture radius)** will turn on the depth of field flag and render the scene using the passed in focal length and aperture radius.

*When running a Lua script, you will see the whole information for the passed in render arguments as the images below.



```
Calling A4_Render(
        SceneNode:[name:root, id:0]
        Image(width:500, height:500)
        eye:  vec3(0.000000, 0.000000, 800.000000)
        view: vec3(0.000000, 0.000000, -1.000000)
        up:   vec3(0.000000, 1.000000, 0.000000)
        fovy: 50
        ambient: vec3(0.300000, 0.300000, 0.300000)
        lights{
                L[vec3(0.900000, 0.900000, 0.900000), vec3(0.000000, 100.000000,
 400.000000), 1, 0, 0; width: 50, height: 50]
        }
)
        Max recursion level: 50
        Supersampling is turned on
        Soft shadow Ray number: 40
        Depth of field is turned off
```



```
Calling A4_Render(
        SceneNode:[name:root, id:0]
        Image(width:500, height:500)
        eye:  vec3(0.000000, 0.000000, 800.000000)
        view: vec3(0.000000, 0.000000, -1.000000)
        up:   vec3(0.000000, 1.000000, 0.000000)
        fovy: 50
        ambient: vec3(0.300000, 0.300000, 0.300000)
        lights{
                L[vec3(0.900000, 0.900000, 0.900000), vec3(0.000000, 100.000000,
 400.000000), 1, 0, 0; width: 50, height: 50]
        }
)
        Max recursion level: 50
        Supersampling is turned on
        Soft shadow Ray number: 40
        Depth of field is turned on
        Focal Length: 300
        Aperture Radius: 20
```

Phong Shading

Phong shading is turned on by default and if you want to turn it off, please change the macro value in line 10 of Mesh.cpp and rebuild the code.

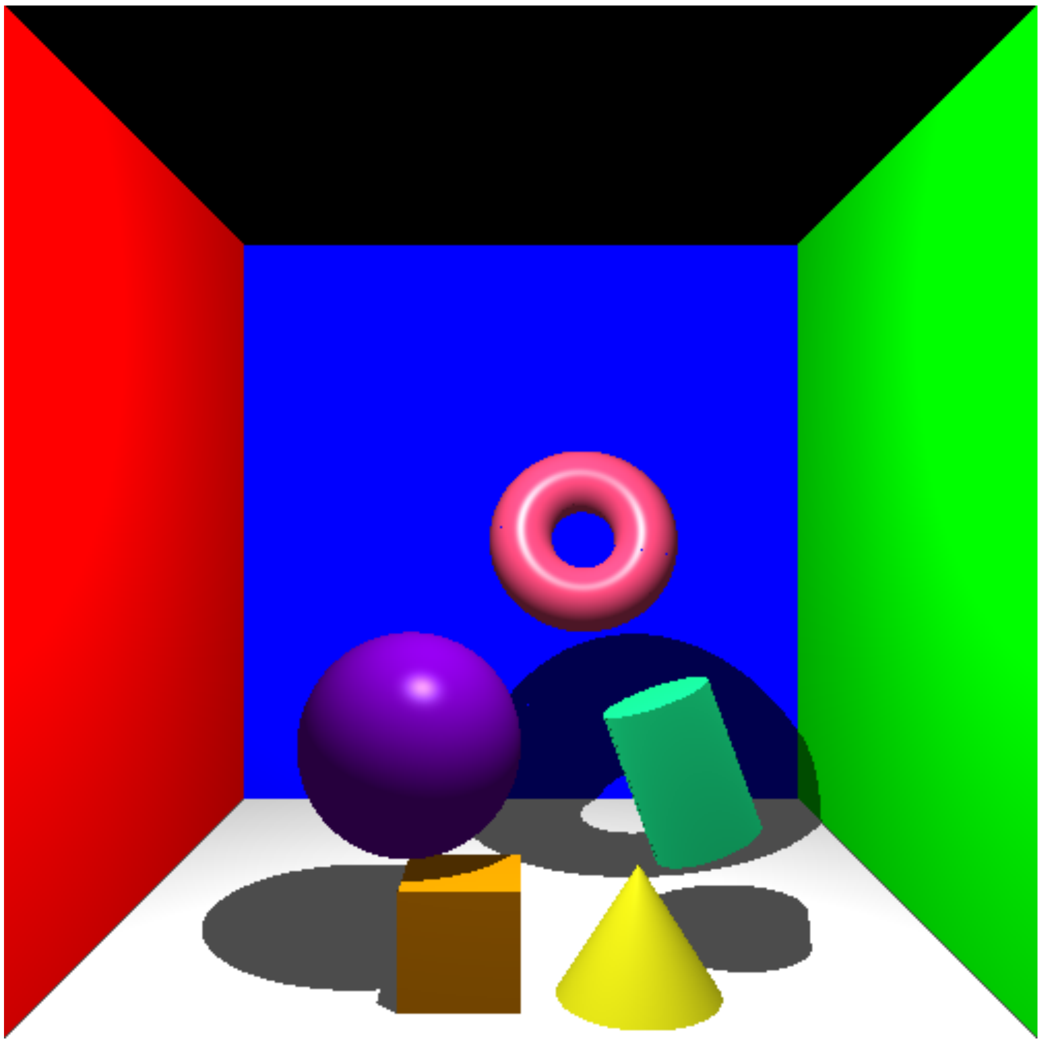
Implementation

Cylinder & Cone (code from line 154 to 449 in Primitive.cpp)

Both my cylinder and cone are **y-axis aligned**. New classes Cylinder, NonhierCylinder, Cone and NonhierCone are inherited from base class Primitive.

For cylinder intersection, I used the reference from https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html#SECTION00023200000000000000 for the quartic equation. For cylinder, it should be $\boldsymbol{x^2 + z^2 = radius^2}$. This only handles an infinite height cylinder. For a finite height cylinder, I also checked the y value of the intersection point. If it does hit the cap (y value bigger than ymax or less than ymin), I computed the new t value using $\boldsymbol{(yMin - yEye)/yRayDir}$. The normal of the cylinder is calculated at the same time with the intersection point. When hitting the cap, the normal is either (0, 1, 0) or (0, -1, 0). When hitting the cylinder bit, the normal is (x-intersection, 0, z-intersection).

For cone intersection , I used the reference from http://hugi.scene.org/online/hugi24/coding%20graphics%20chris%20dragan%20raytracing%20shapes.htm to come up with the quartic equation and the way for calculating the normal. The intersection equation is $\boldsymbol{x^2 + z^2 = tan * tan * y^2, where\ tan = radius/height}$. The y-value checking is similar to cylinder.


All the primitives implemented in A4 and Project

Texture Mapping (new Texture class defined in new files Texture.cpp/hpp)

All the primitives (sphere, cube, torus, cylinder and cone) support the texture mapping. The texture file must be PNG format since I used the lodepng for decoding the images and store its pixel data when creating a texture object.

The uv mapping are calculated depending on the kind of primitives and its intersection. (UV mapping calculation functions **computeTextCoord** are in **Primitive.cpp**)

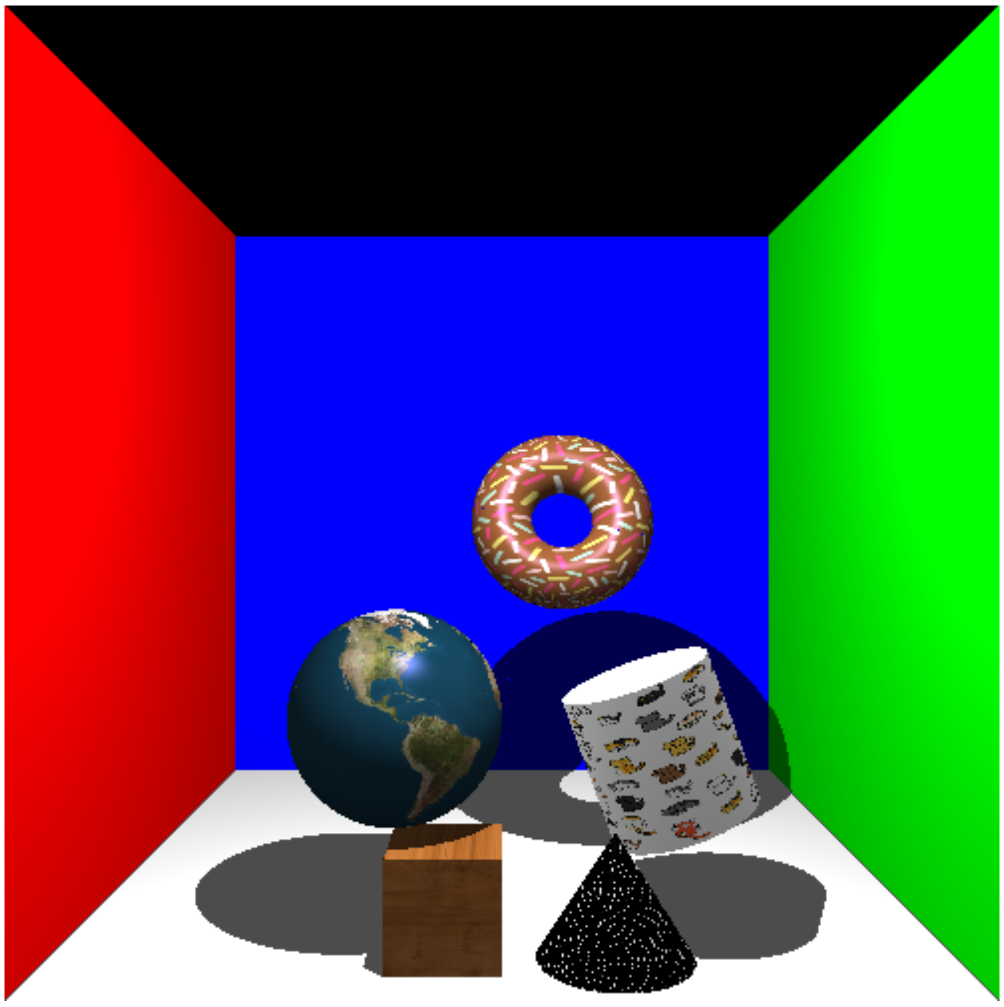
For sphere, I used $\boldsymbol{v} = \mathbf{acos}\ (\boldsymbol{y}/\boldsymbol{radius})$ and $\boldsymbol{u} = \mathbf{acos}\ (\boldsymbol{x}/(\boldsymbol{r}*\mathbf{sin}\ (\boldsymbol{pi}*\boldsymbol{v}))$ for calculating the uv.

For cube and torus, I also used the spherical mapping as the uv mapping function for sphere above but replacing the radius with the size for cube and sum of major radius and minor radius for torus.
For cylinder and cone, I used $\boldsymbol{v = y/height}$ and $\boldsymbol{u = \mathrm{acos}\,(x/radius)}$ to compute the uv coordinates.

The computed uv coordinates are firstly modulated by $\boldsymbol{v = v/pi}$ and $\boldsymbol{u = mod(u, 2 * pi)/(2{*}pi)}$ to make it between [0, 1] and then interpolated using bilinear interpolation to get the final color vector. The bilinear interpolation equation is referenced from http://www.cs.uu.nl/docs/vakken/gr/2011/Slides/06-texturing.pdf as

$$\boldsymbol{c(u,v) = (1-u')(1-v')cij + u'(1-v')c(i+1)j + (1-u')v'ci(j+1) + u'v'c(i+1)(j+1)\ where\ u' = u{*}width\text{-}floor(u{*}width), v' = v * height\text{-}floor(v * height)}$$

I was considering about using planar mapping ($u = x/size, v = y/size$) for cube, but the texture distorted a lot. I think I should improve the mapping for cylinder by considering the cap and cylinder bit mapping separately to get better texture mapping effects. Also, cube mapping for mapping different textures to different faces is also a nice feature for exploring later.



Supersampling (implemented from line 94 A4.cpp)

As the command manual, supersampling flag is passed as an argument of gr.render command. I used 3x3 grid (9 samples) for each pixel. The (x, y) pixel is expanded to

(x – 1/3, y – 1/3),
(x – 1/3, y),
(x – 1/3, y + 1/3),
(x, y – 1/3),
(x, y),
(x, y + 1/3),
(x + 1/3, y – 1/3),
(x + 1/3, y) and
(x + 1/3, y + 1/3)

and cast rays for each. The color for each pixel is equal to the color sum of the samples divided by 9.

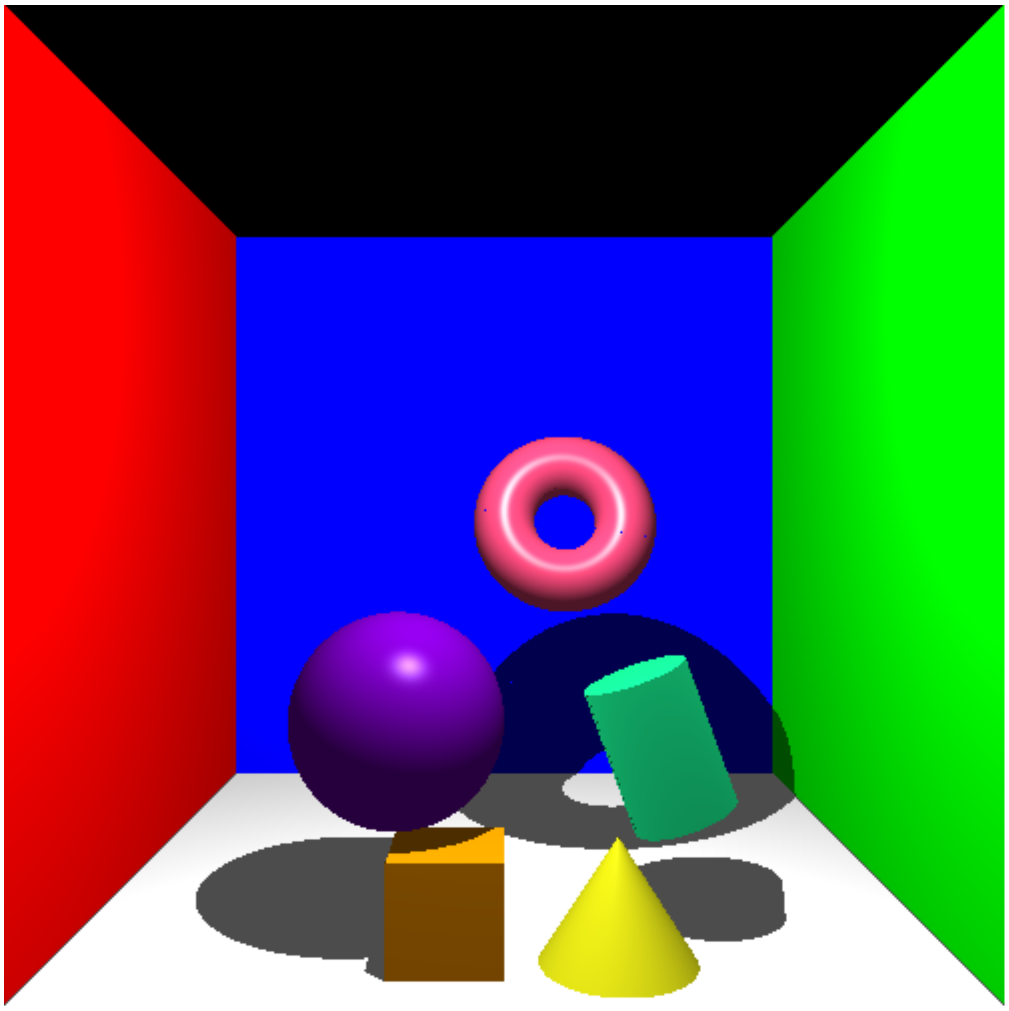


Figure rendered without supersampling (jaggies)

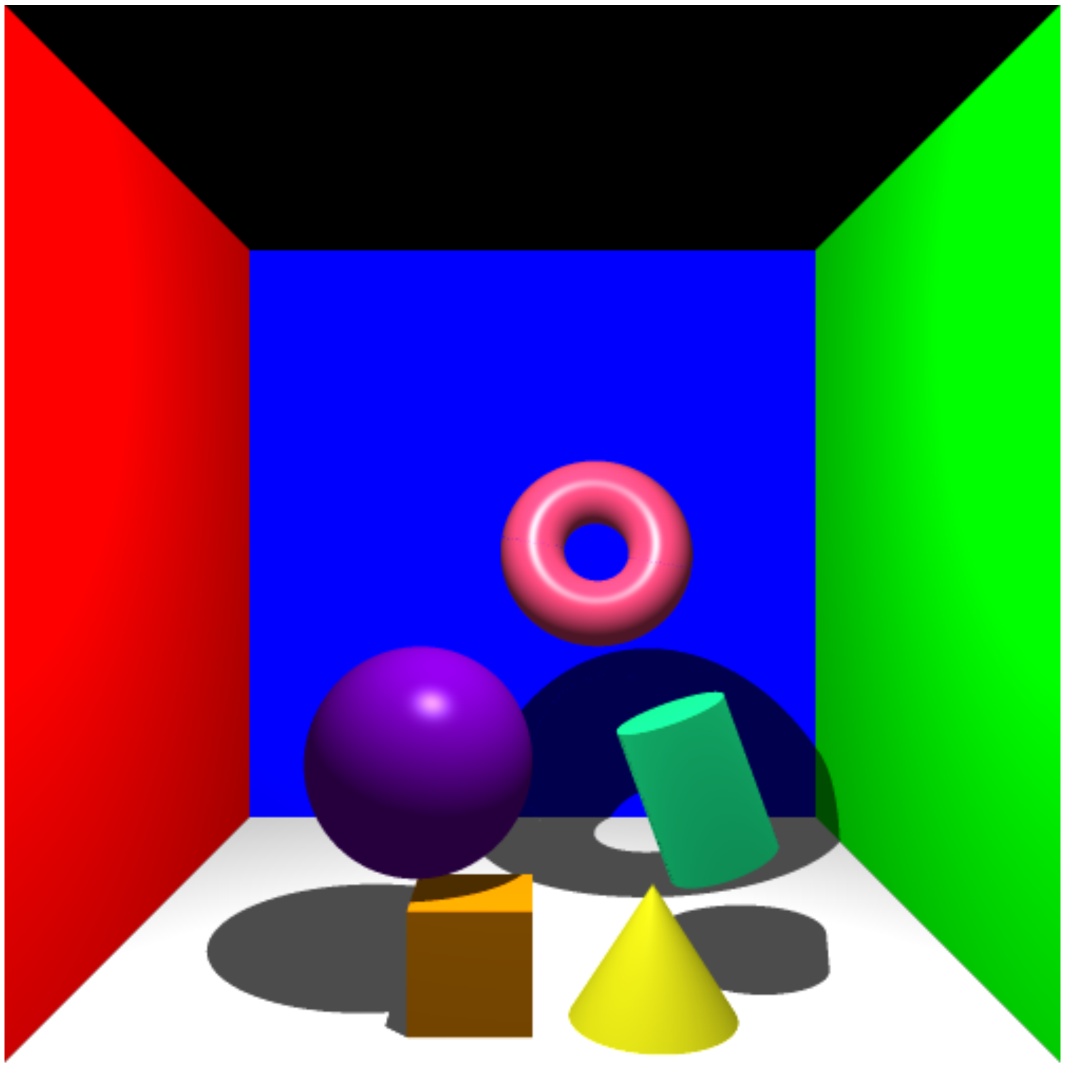


Figure rendered with supersampling (no jaggies)

Mirror Reflection & refraction (Implemented in A4.cpp)

I did this by casting secondary ray recursively and the max recursion number is passed as an argument to gr.render.
The reflected ray direction is calculated using

$$\boldsymbol{R = rayDir\text{-}2.0 * (rayDir * N)*N}$$

The reflection color is blended through multiplying with the passed in reflectivity argument from gr.material command.

The refraction ray direction is calculated using Snell's Law

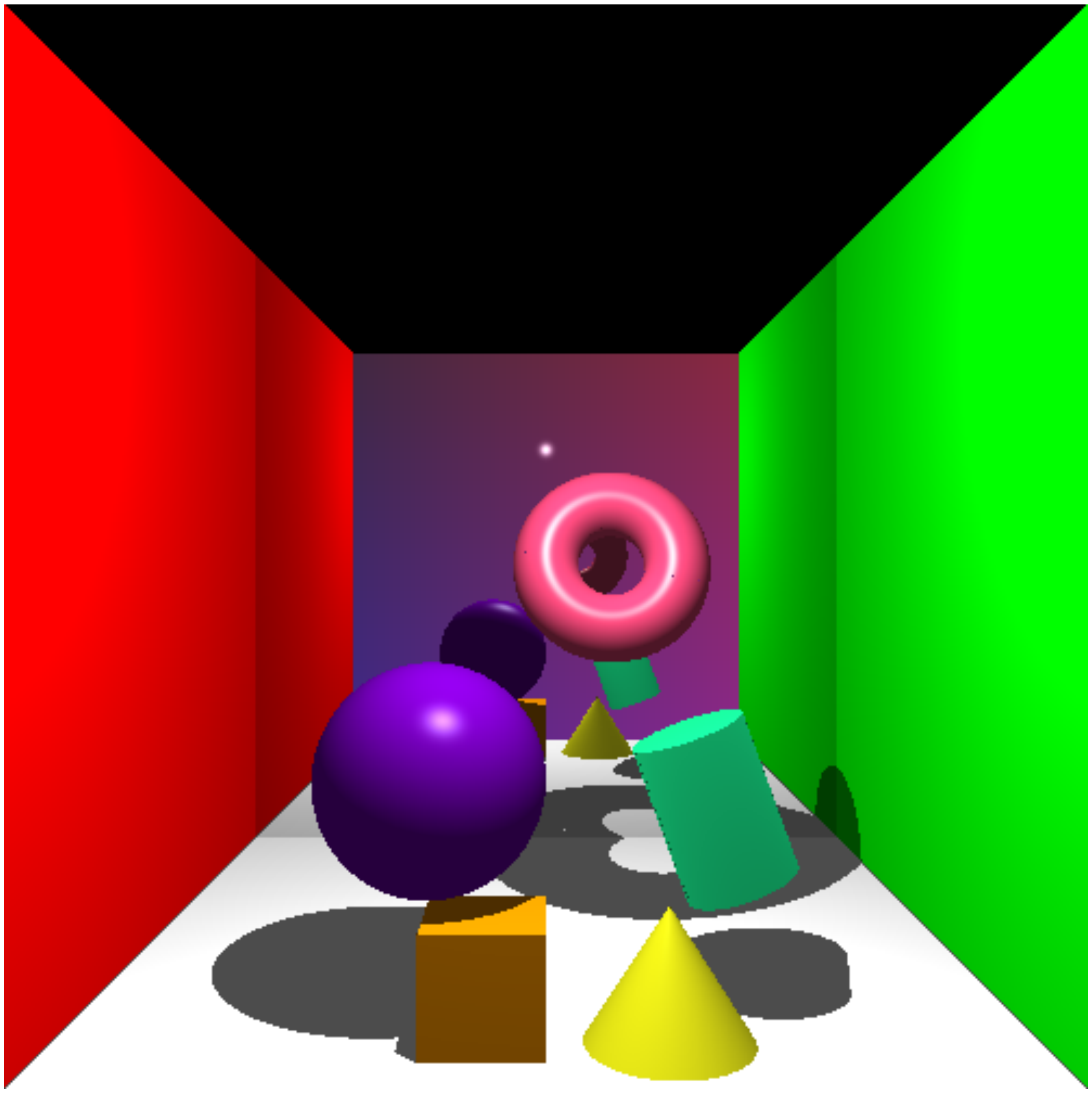
$$\boldsymbol{T = \frac{\eta 1}{\eta 2}(rayDir + \cos(theta_{in}) * N) \text{ - } N * \sqrt{1\text{-}(\frac{\eta 1}{\eta 2})^2(1\text{-}cos^2(theta_{in}))}}$$

I also used the Fresnel equation for calculating the ratio between reflection and refraction.

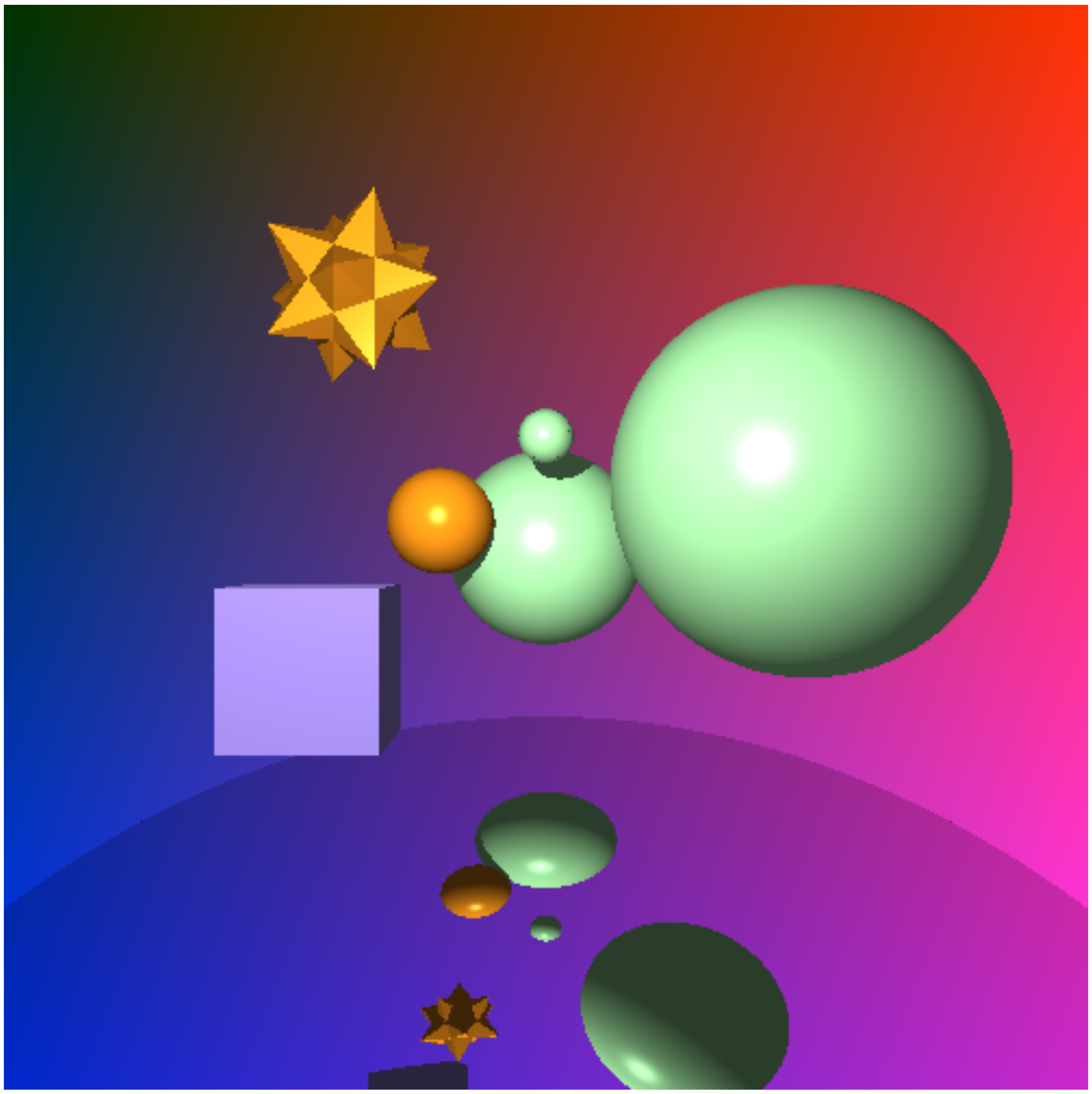
$$\boldsymbol{Fr = \frac{1}{2}\left(\left(\frac{\eta 2 cos(theta_{in}) - \eta 1 cos(theta_{out})}{\eta 2 cos(theta_{in}) + \eta 1 cos(theta_{out})}\right)^2 + \left(\frac{\eta 1 cos(theta_{out}) - \eta 2 cos(theta_{in})}{\eta 1 cos(theta_{oug}) + \eta 2 cos(theta_{in})}\right)^2\right)}$$

The $\boldsymbol{\eta 1}$ is the index of refraction of the medium that the ray transmits from (original medium) and $\boldsymbol{\eta 2}$ is the index of refraction of the medium that the ray transmits to (new medium).

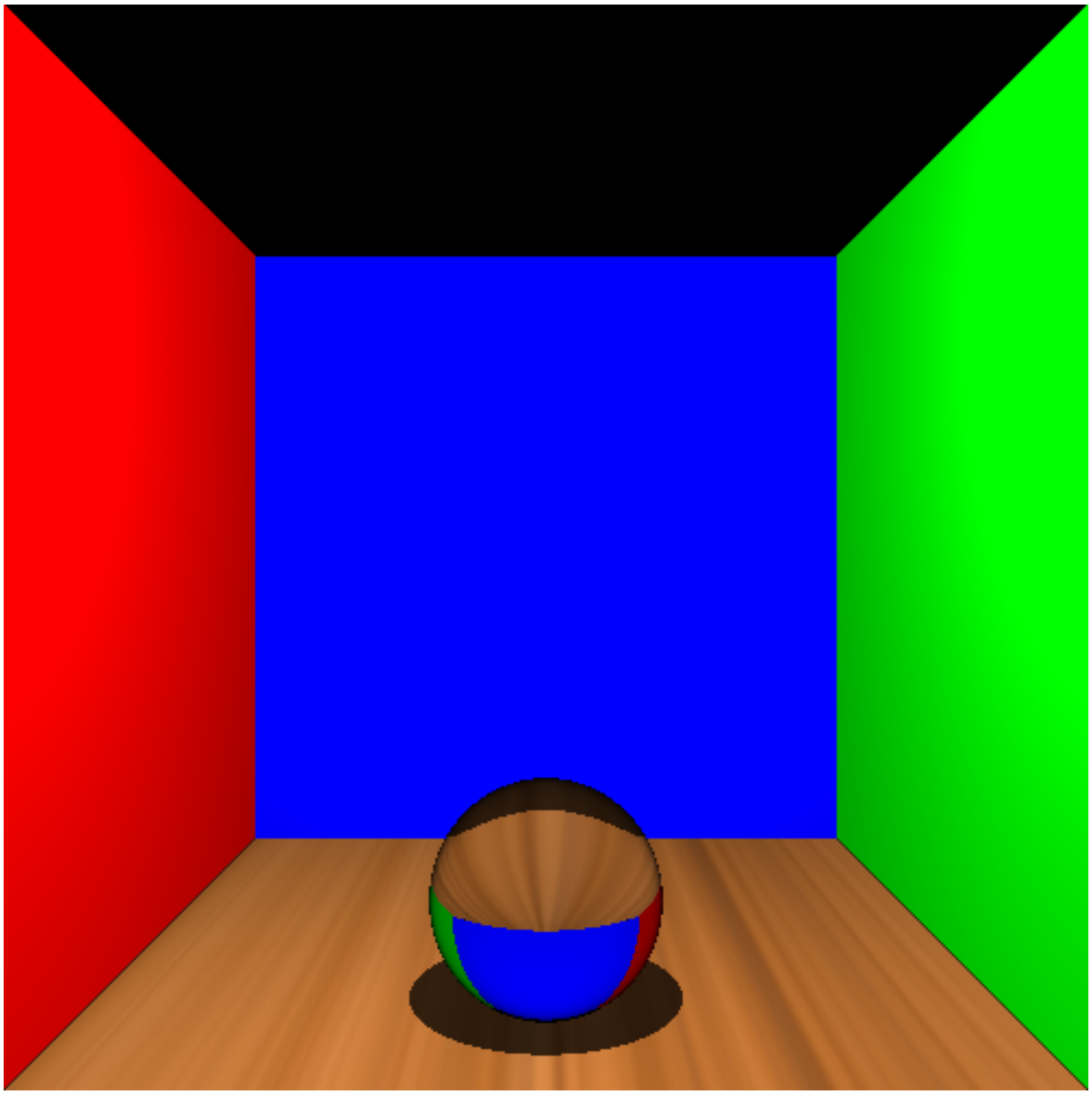
An parameter called recursionDepth is passed to castRay function and it counts from 0 and increments every time when the ray hit a reflective or refractive object and halts the recursion when it exceeds the max recursion number.



Mirror reflection 1



Mirror Reflection 2



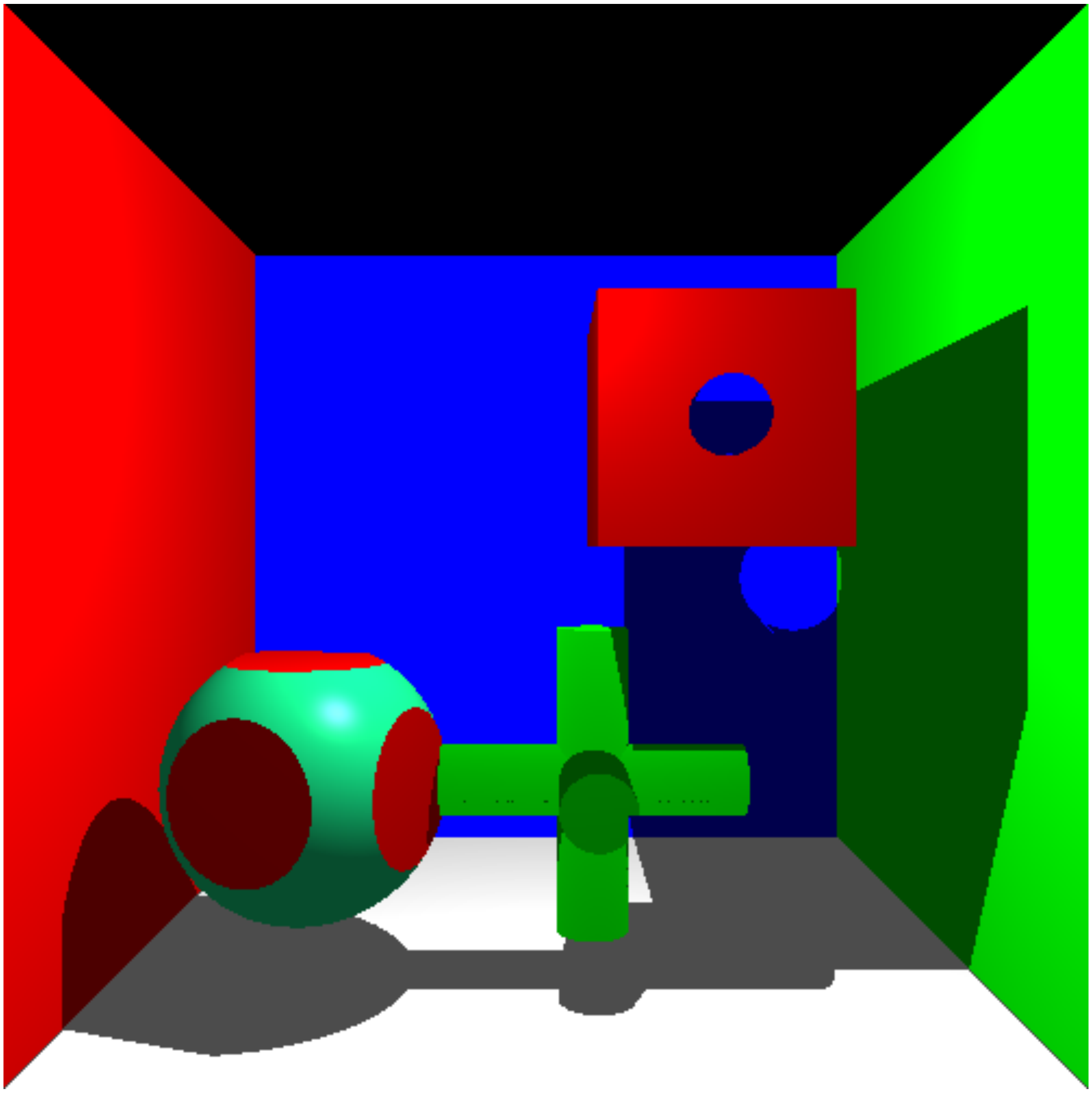
Refraction

This picture is rendered by passing outer index of refraction as 1 and inner index of refraction as 1.53 (Like a glass).

Constructive Solid Geometry (implemented in GeometryNode.cpp)

I defined a new class called CSGNode which inherits from GeometryNode. It has three members, the left GeometryNode, right GeometryNode and the CSG option string. For each ray, I traversed down both the left tree and right tree to get two interval lists and combine them into a new interval list according to the option. Eventually, I took the nearest t value as the intersection t-value from the final interval list. I used the reference from http://slideplayer.com/slide/680289/ when deciding 'no overlap', 'partial overlap' and 'full overlap' of intervals for each option. My

CSG works fine for sequential uses of only one option. The combination of options and transformations distort the model. I think there are still cases that I did not notice or made mistakes of. Also, I was quite confused about the hierarchy stuff when implementing the CSG. Finally, I decided to assume that always doing the CSG modelling first then add the final CSG node to its parent like I emphasized in the **Manual** section for using gr.csg command. But I think the hierarchy is still a big problem in my implementation. When casting secondary rays, it also distorts both the model itself and other models. Also, my implementation is not very efficient, which requires O(n^2) to merge interval lists. I will figure out the bugs and improve the tree traversal algorithm later.

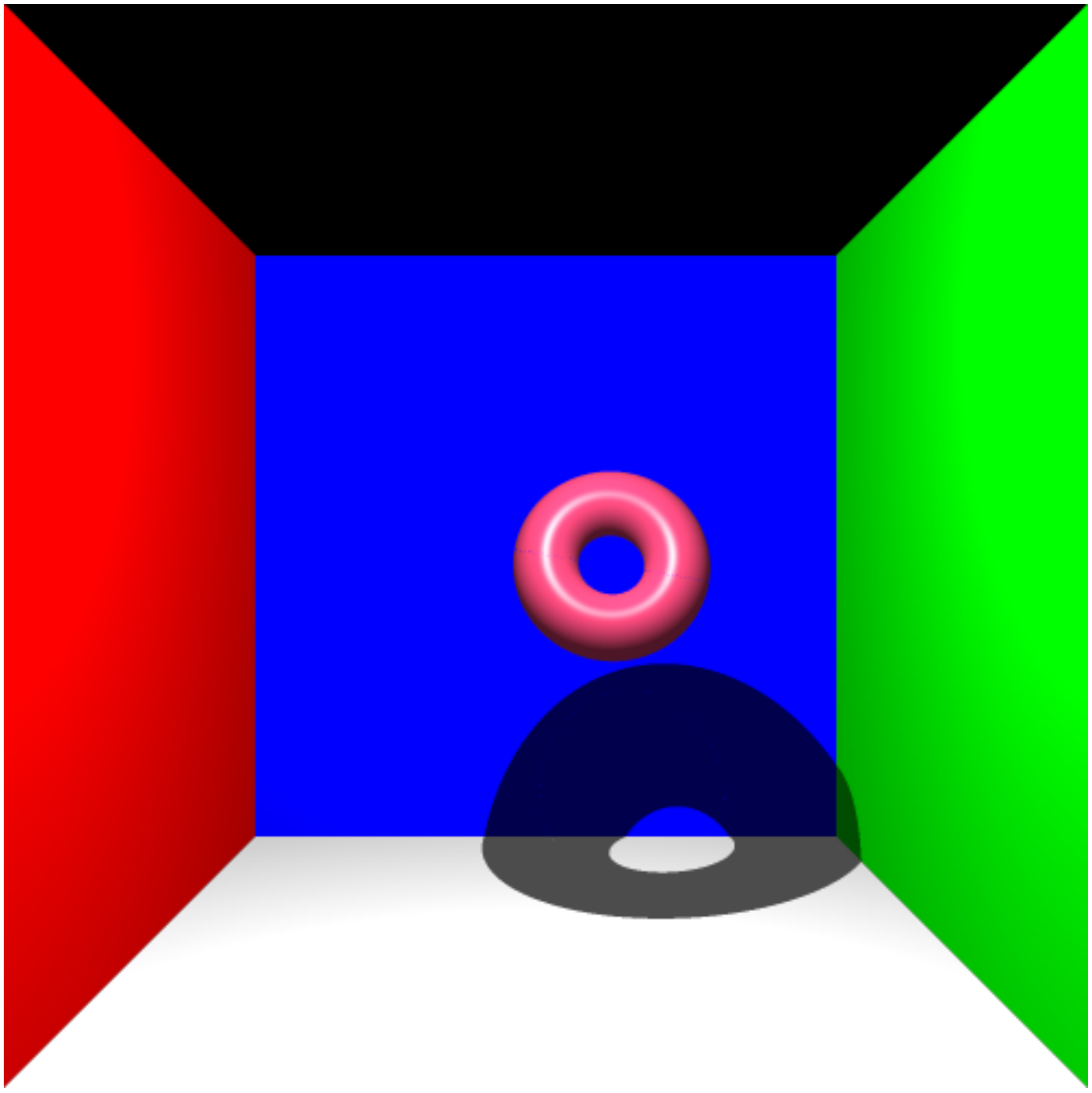


Left: Intersection between one sphere and one cube
Middle: Union of three cylinders

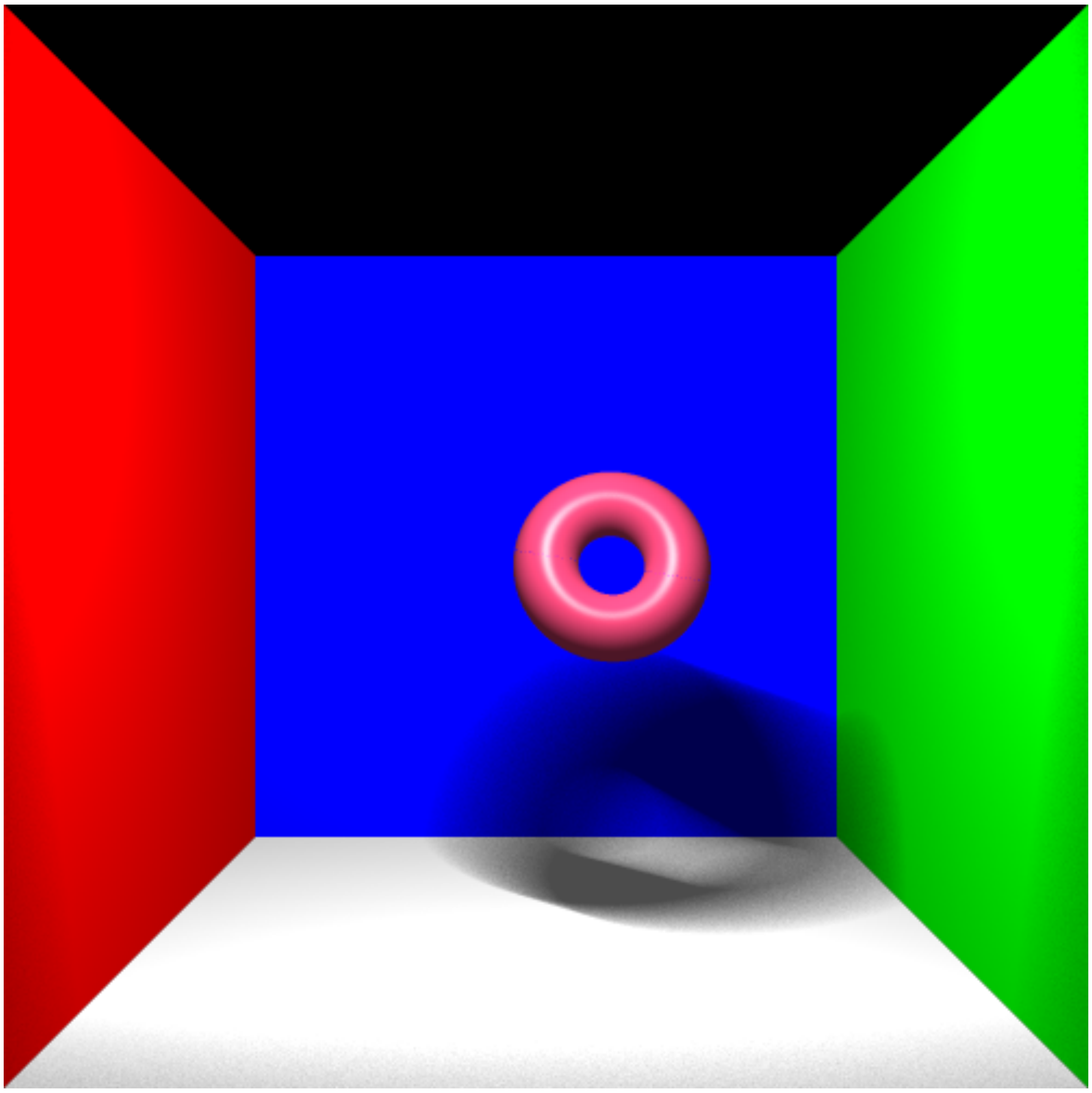
Right: One cube minus one cylinder

Soft Shadow (implemented in A4.cpp and new AreaLight class in Light.cpp)

Firstly, I added a new class AreaLight for defining a rectangular light. It has two more members than the usual point light, width and height passed from the gr.light command. When casting shadow rays for an area light, I used the rand() function to choose a random position within the range of the light area and repeat this for several times. The shadow ray casting number is an argument passed into gr.render command. The final color is calculated by summing up the color and dividing the sum by the shadow ray casting number.



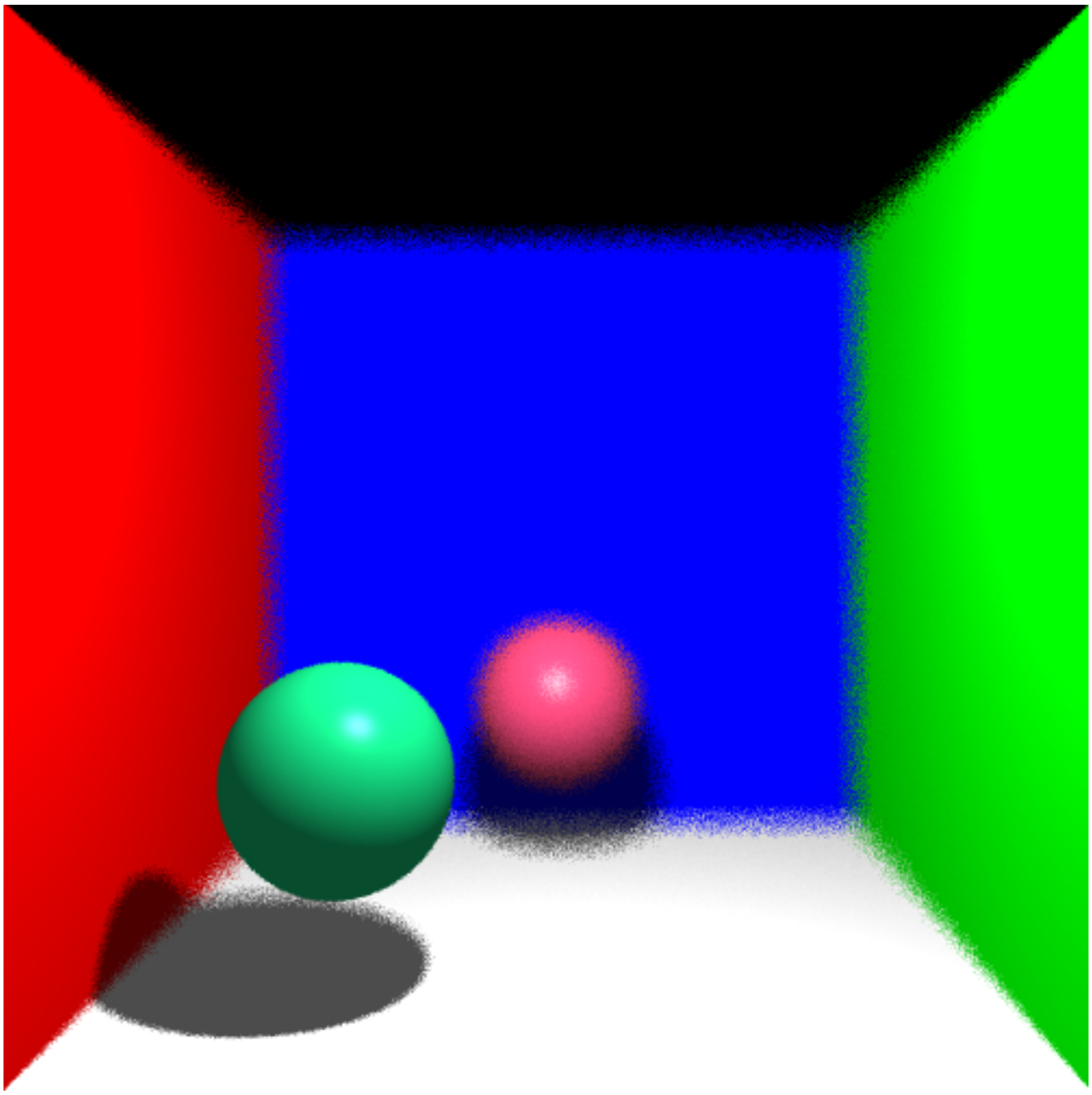
Torus shadow without soft shadow



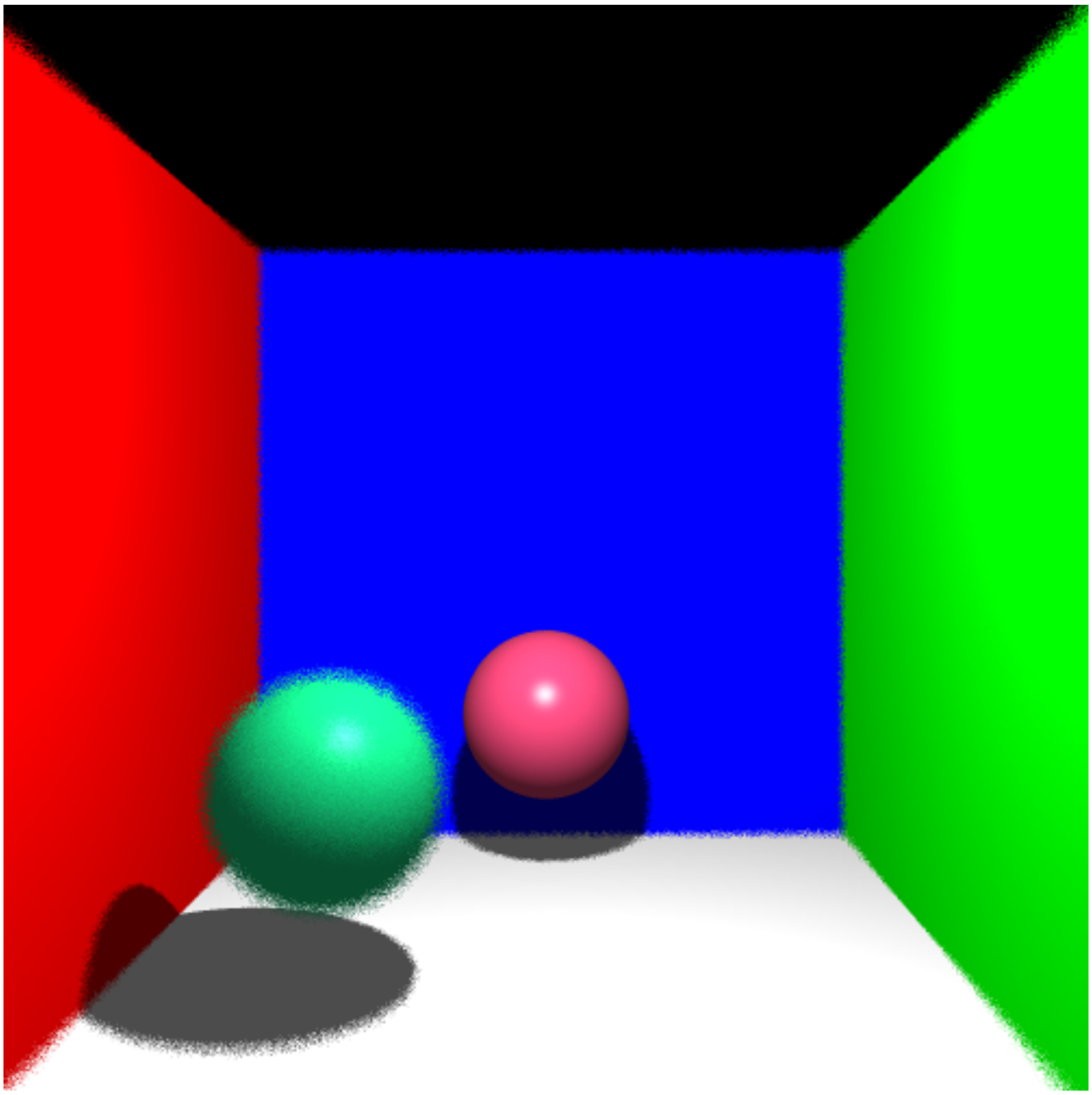
Torus shadow with casting 60 shadow rays (area light width 50, height 50)

Depth of Field (implemented in A4.cpp)

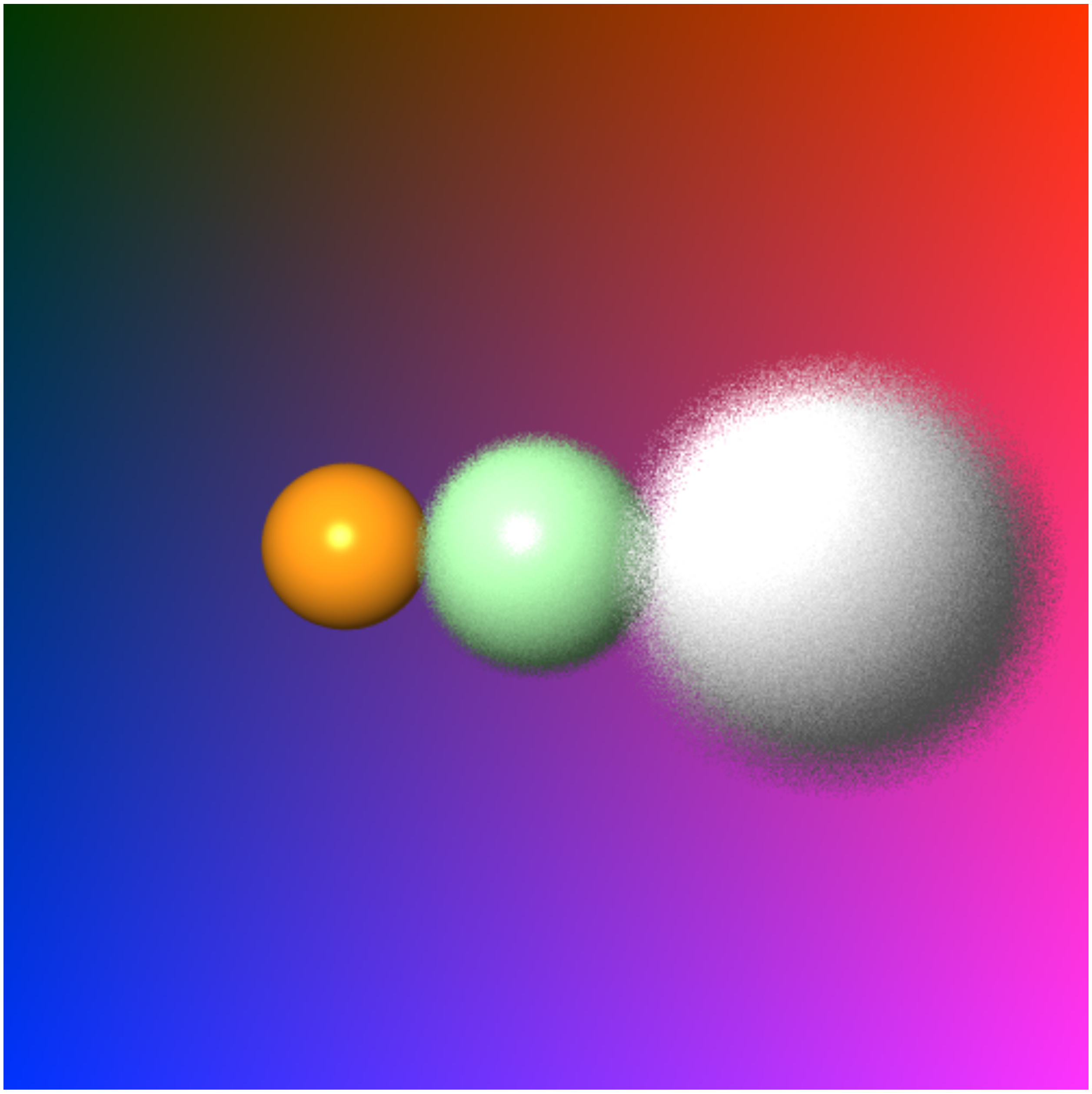
The focal length and aperture radius are passed as arguments for gr.render command.
The focal point is calculated as **focalPoint = eye + focalLength * rayDir**. The new eye position is that **newEye = oldEye + (x_aperture, y_aperture, 0)**. The x_aperture and y_aperture are chosen randomly within the range of the passed in aperture radius. Then the new ray direction can be calculated as **newDirection = normalize(focalPoint - newEye)**. Depth of field requires several samples for good rendering effects. I only used 9 samples which is not sufficient and needs improvements in the future.



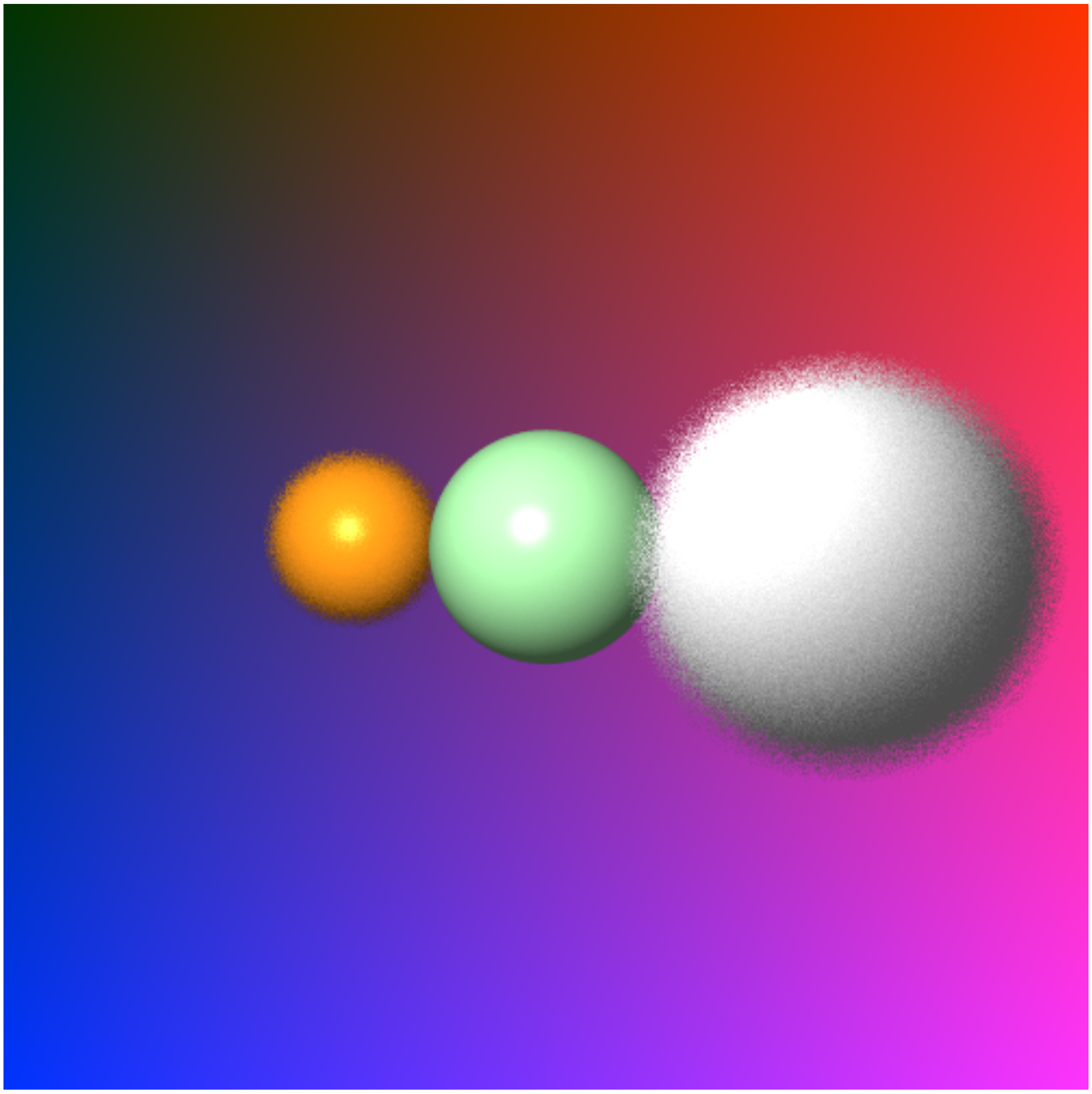
Depth of Field 1 focusing on the front sphere (aperture radius 35)



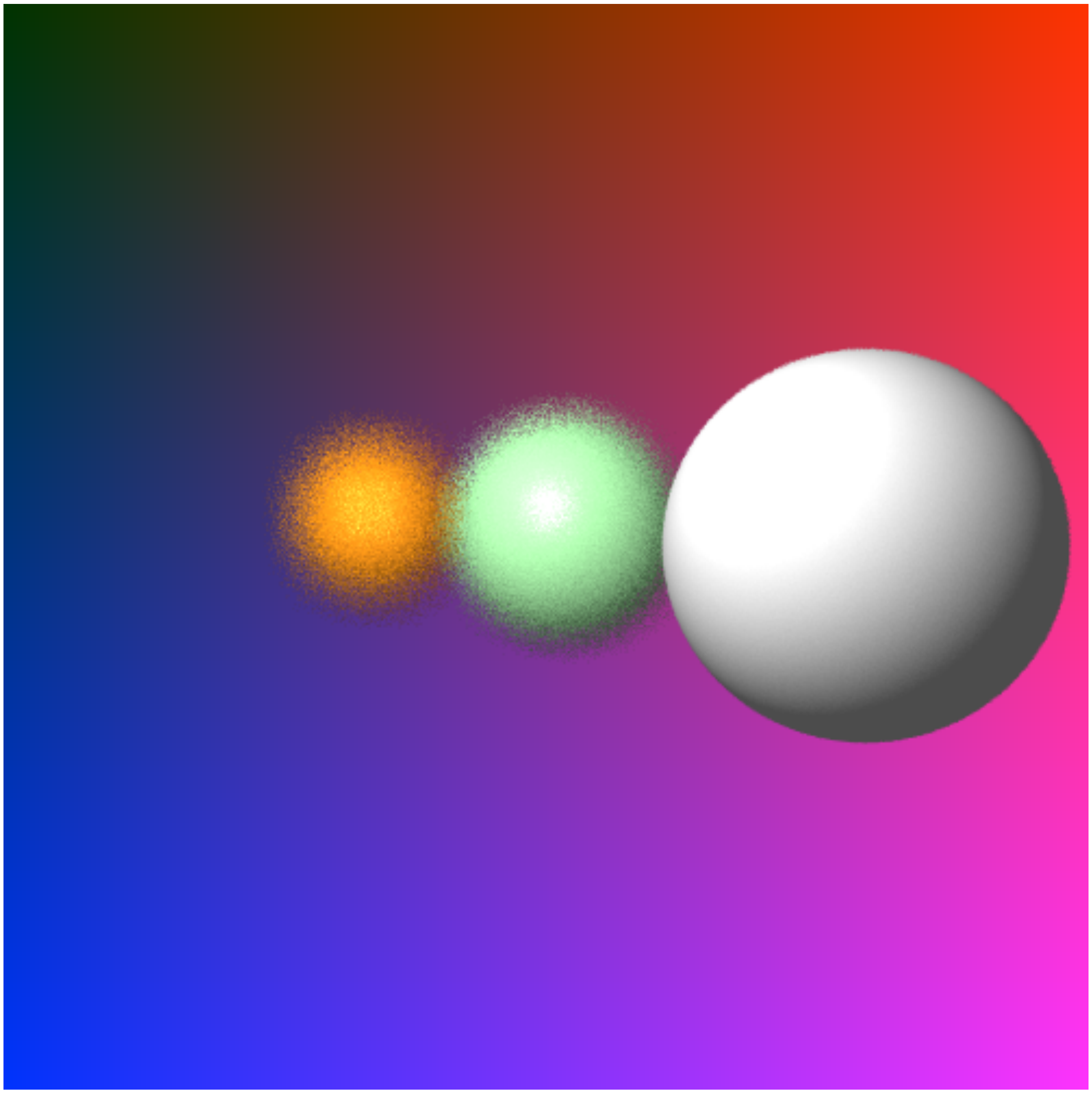
Depth of Field 1 focusing on the back sphere (aperture radius 35)



Depth of Field 2 focusing on the back sphere (aperture radius 25)



Depth of Field 2 focusing on the middle sphere (aperture radius 25)

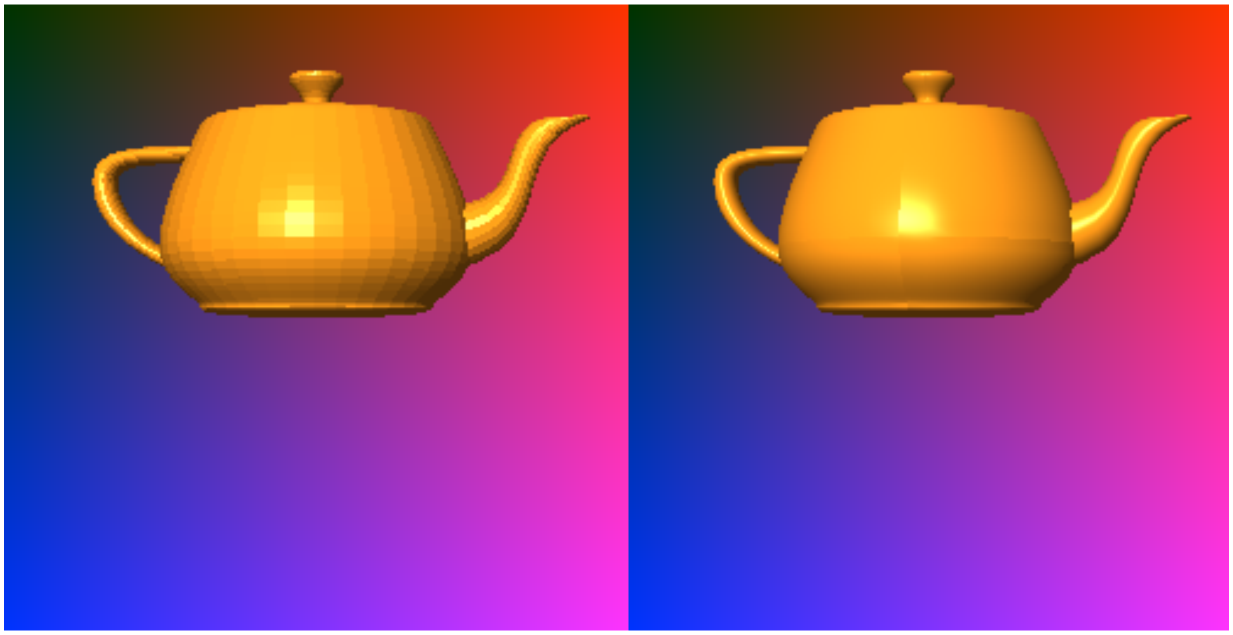


Depth of Field 2 focusing on the front sphere (aperture radius 25)

Phong Shading (implemented in Mesh.cpp)

Firstly, I calculated and stored the vertex normal for one mesh after loading its obj file. The vertex normal is calculated by initializing and storing all the vertex normal as (0, 0, 0) at first, then iterating each triangle faces and adding the face normal to the stored vertex normal of each vertex. When calculating the hit point normal, interpolating with the barycentric coordinates through

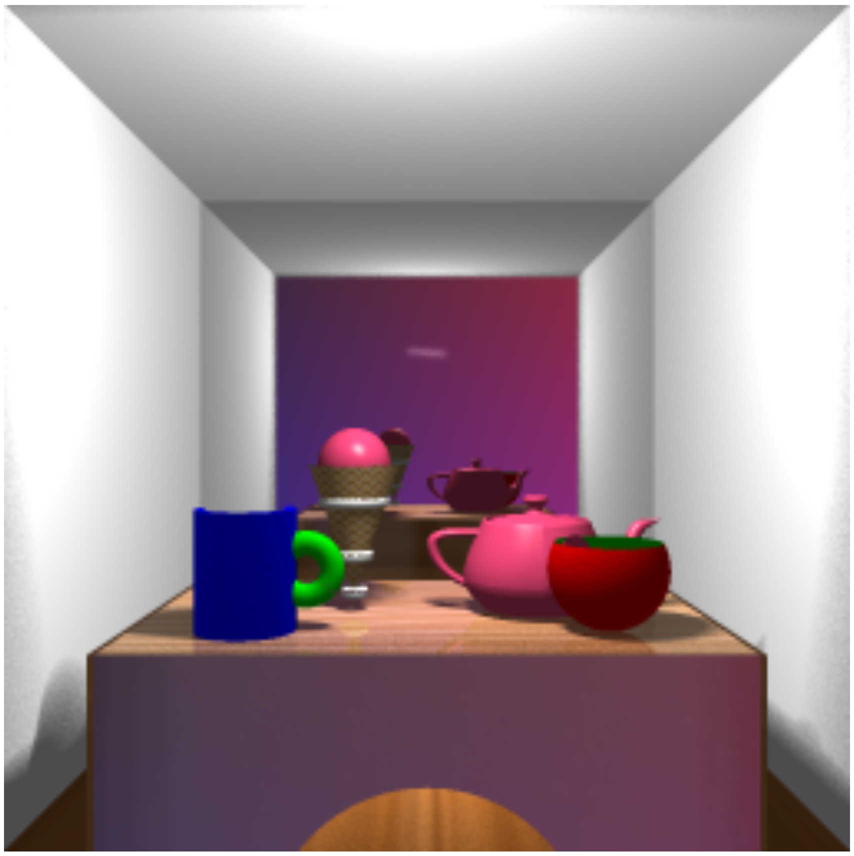
finalNormal = (1 – barycentricCoord.u – barycentricCoord.v) * v1.vertexNormal + barycentricCoord.u * v2.vertexNormal + barycentricCoord.v * v3.vertexNormal



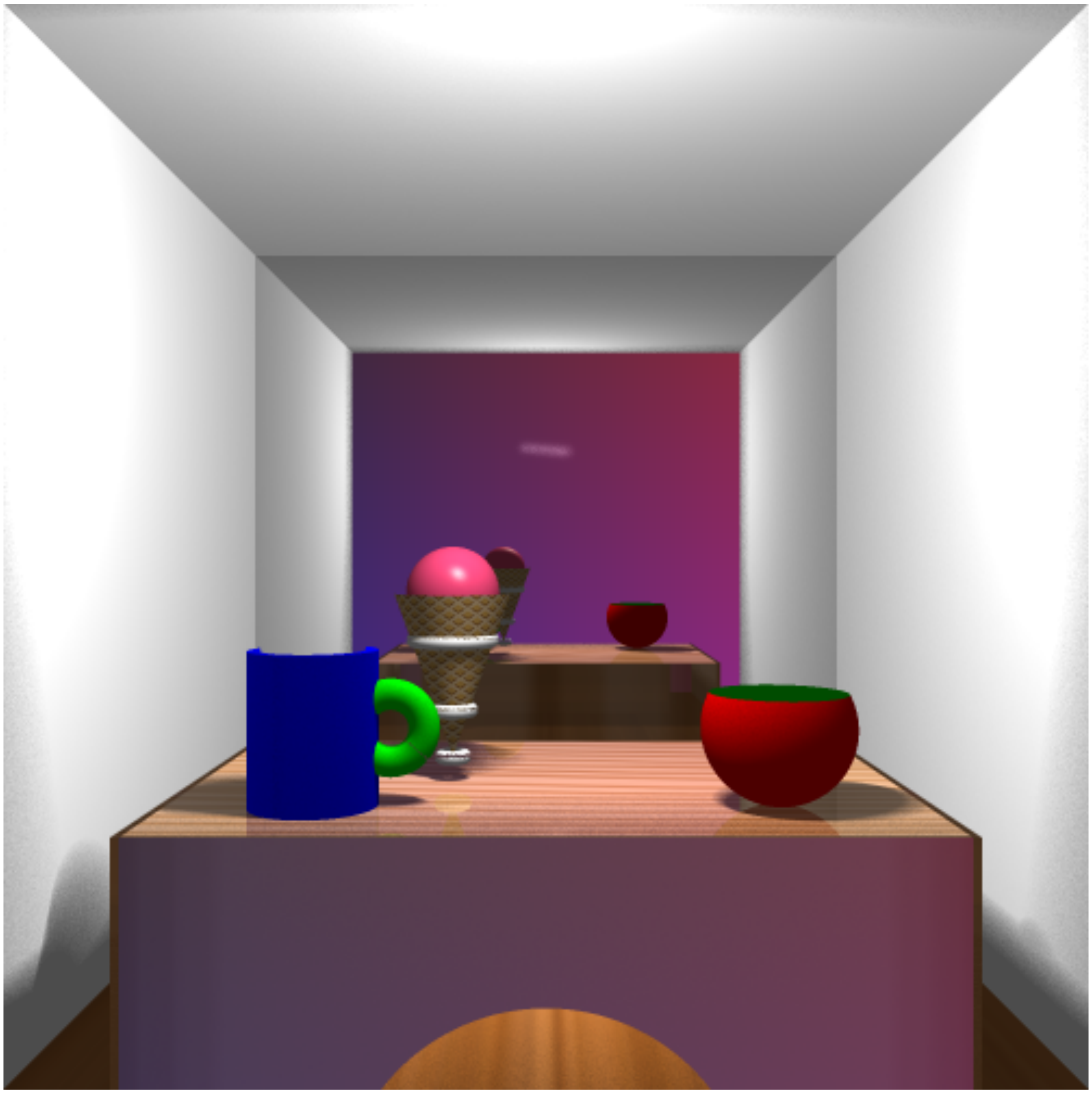
Left: Teapot without Phong shading
Right: Teapot with Phong shading

Final Scene

I finally got my final scene in 256 x 256 size which is so bad. I should manage my time more efficiently. The mug, bowl and table were made with csg difference. The mug was made by subtracting a smaller cylinder from a larger one. The bowl was created in the same way. The table was created by subtracting a sphere from a cube. An ice cream was made by the sphere and cone. The cone was applied a waffle cone texture. The white torus's around the cone are ice cream standing base. The teapot is phong shaded. Wood textures were applied both to the table and the floor. The back wall is set to be a mirror so that you can see the mirror reflection effect behind. There is an area light with width 50, height 50 in the scene and 40 shadow rays supposed to be casted.



Final scene (256 x 256) finished after running almost the whole night



Final scene (500 x 500) without teapot

More on future possibilities

It took too much time to render a scene with distributed ray tracing like soft shadows. My final scene which is 500x500 did not finish even running the whole night. Multithreading is a big must and I regret that I did not do that and wasted a lot of time.

Reference & Bibliography

- https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html Ray tracing primitives intersection
- http://hugi.scene.org/online/hugi24/coding%20graphics%20chris%20dragan%20raytracing%20shapes.htm Cone intersection
- http://www.cs.uu.nl/docs/vakken/gr/2011/Slides/06-texturing.pdf Bilinear Interpolation
- https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf Reflection & refraction
- http://slideplayer.com/slide/680289/ Constructive Solid Geometry
- https://graphics.stanford.edu/courses/cs148-10-summer/as3/code/as3/teapot.obj Teapot obj file
- http://www.inf.ed.ac.uk/teaching/courses/cg/lectures/cg7_2012.pdf Vertex Normal Calculations & Phong Shading
- https://www.cs.unc.edu/~dm/UNC/COMP236/LECTURES/SoftShadows.pdf Soft Shadow
- http://woo4.me/raytracer/depth-of-field/ Depth of Field

Objective

- Cylinder and cone primitives are added along with the modelling language
- Texture mapping
- Supersampling
- Mirror Reflection
- Constructive Solid Geometry
- Refraction
- Soft Shadow
- Depth of Field
- Phong Shading
- Final Scene