Ray tracer in DrRacket

Project by
120050041 Nishanth N. Koushik
120050007 Devdeep Ray

Description of the problem statement:

To take a mathematical description of a set of objects in 3D space and render the image as seen through a given camera model. Also, to incorporate surface properties like reflection and refraction, light properties like diffuse shadows and specular reflections and camera properties like depth of field blur.

Program Design:

The program is divided into five files.

1. The ray tracing algorithm -> tracer.ss

```
(require "vectors.ss"
        "colors.ss"
        "otherstructs.ss"
        "scene.ss"
        "objects.ss"
        images/flomap
        racket/flonum
        racket/gui)
```

2. Vector struct and associated helper functions -> vectors.ss
3. Color struct and associated helper functions -> colors.ss

```
(require picturing-programs)
```

4. Light struct, screen struct, etc. -> otherstructs.ss

```
(require "vectors.ss"
        "colors.ss")
```

5. Object definitions -> objects.ss

```
(require picturing-programs
        "vectors.ss"
        "otherstructs.ss"
        "colors.ss")
```

Algorithm outline:

```
1. Generate a ray from the camera to a pixel on screen.
2. Calculate its closest intersection with an object in the scene.
3. If it intersects any object at any point;
        Send a shadow ray to the light and check if the light is visible;
        If it is visible, calculate colour due to direct light, else use ambient light.
        Calculate reflected ray and refracted ray for the ray if recursion level is not yet reached.
        Go to step 2 for both rays, and recurse.
4. If it doesn't intersect, return a colour.
5. Go to the next pixel.
```

In this way, we generate the colour of every pixel on the screen, creating the final image.

We also have features like soft (diffuse) shadows (implemented using a random ray generator followed by an averaging process), total internal reflection (as seen in sample case 4) and a partial (see limitations below) implementation of Constructive Solid Geometry, allowing us to combine objects to create new compound ones.

Colour of a point is calculated using the following model:
C -> colour
K- > coefficient of
    d- > diffusivity
    rfr -> refraction
    rfl- > reflection
spc- > specular highlighting
l -> light

1. Diffuse shading is cos (theta) shading.
    $Cd = Kd * Cl * \cos(\theta)$
    where theta is the angle between the normal vector and the point -> light vector.

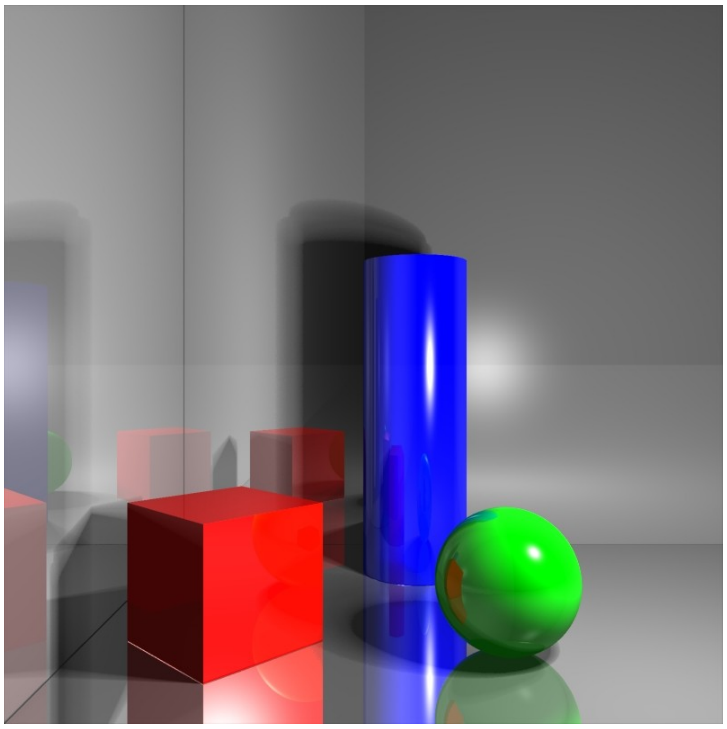
2. Specular highlighting
    Cspc = max Kd (r, g, b) * Cl * (cos (theta) ^ smoothness)
    where theta is the angle between the reflected ray and the point -> light vector.

3. Reflection component
    Crfl = color (reflected ray) * Krfl

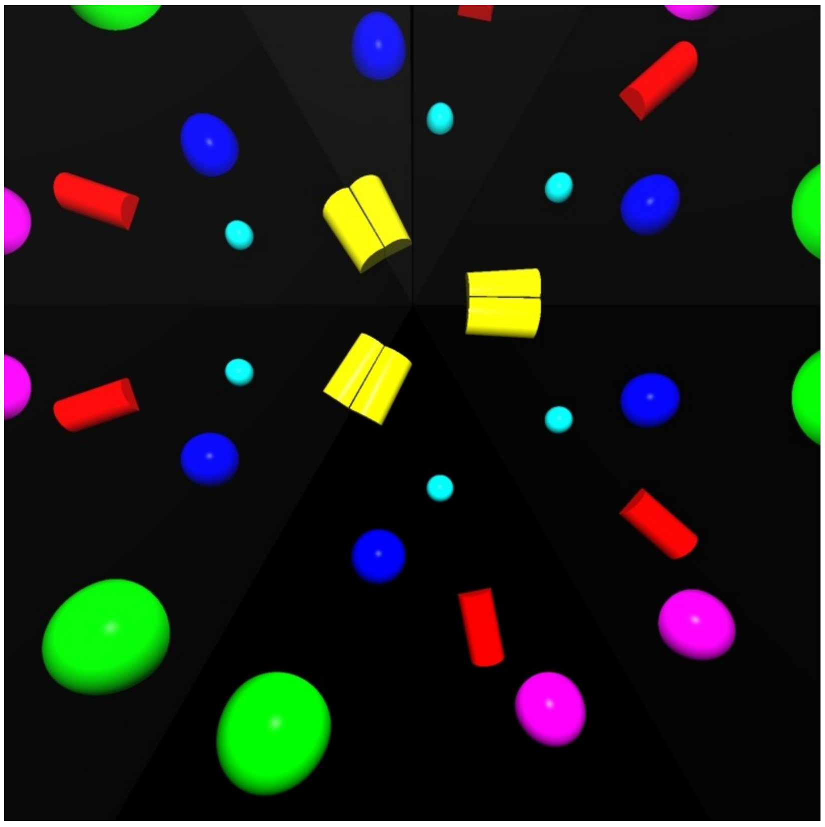
4. Refraction component
    Crfr = color (refractedray) * Krfr

Sample input-output

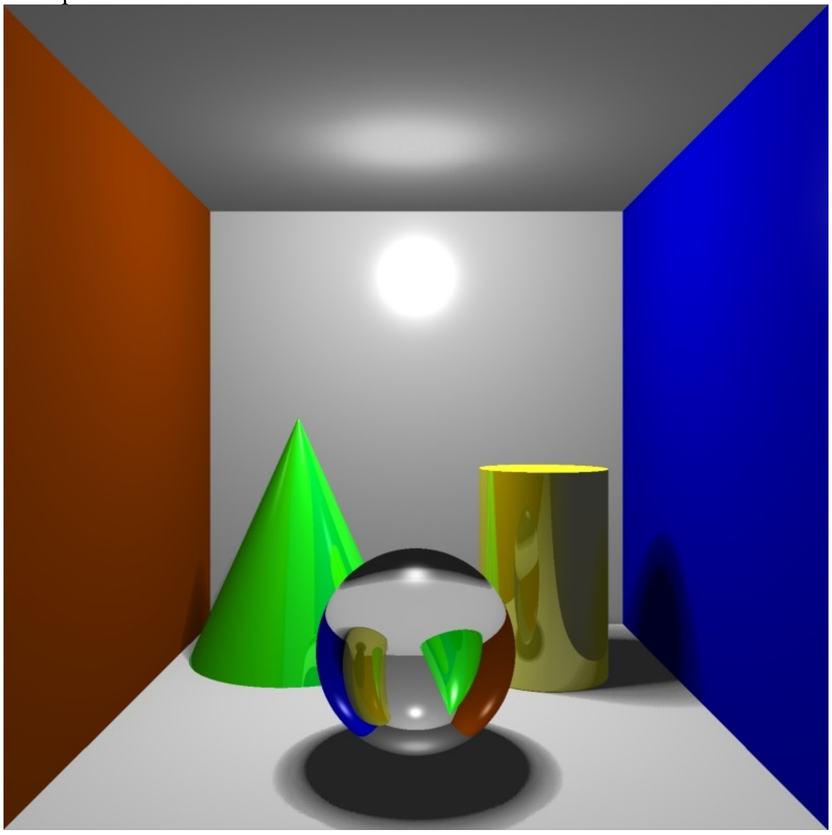
Sample output for scene1.ss



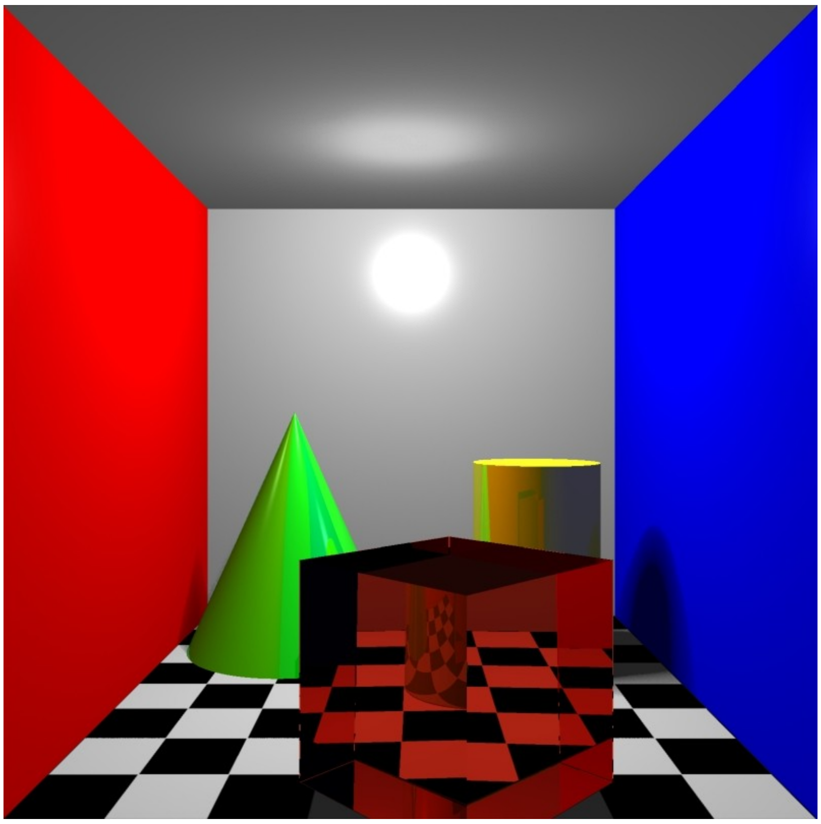
Sample output for scene2.ss



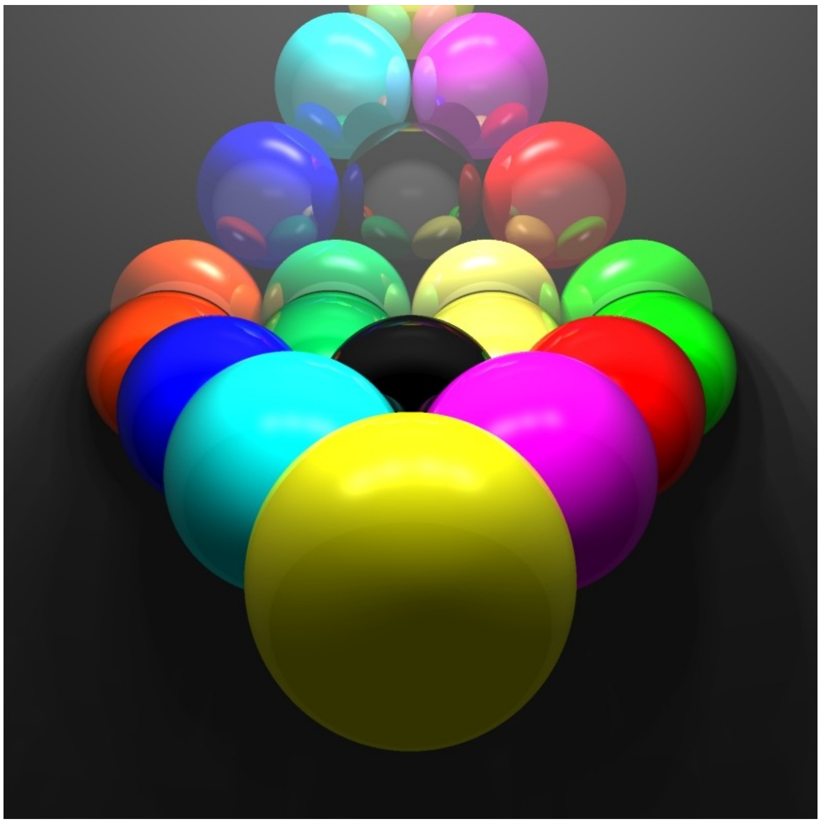
Sample output for scene3.ss



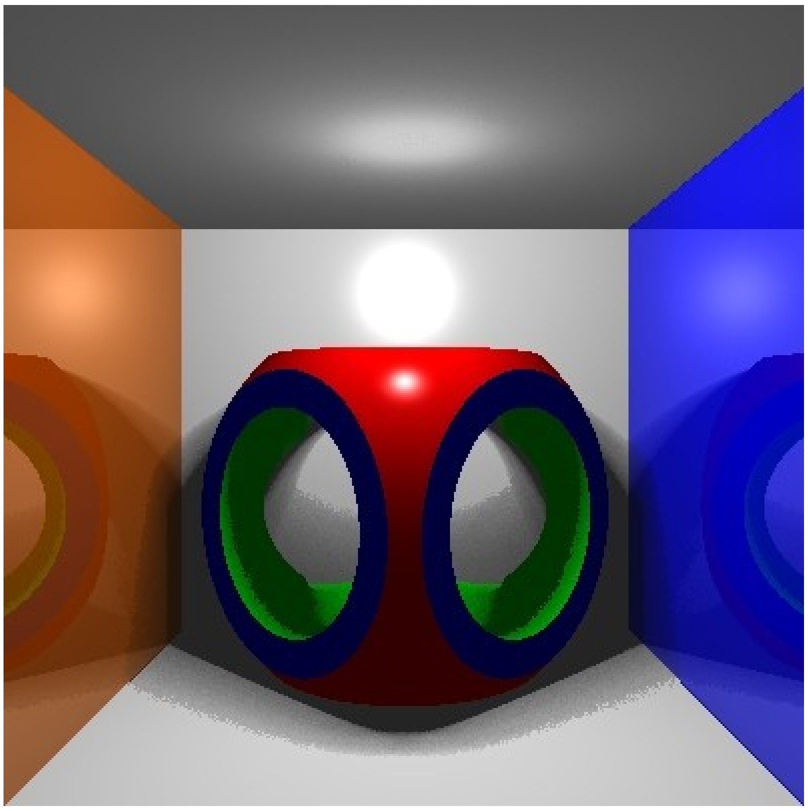
Sample output for scene4.ss



Sample output for scene5.ss



Sample output for scene6.ss



Limitations and bugs:

1. Cannot be used for real-time ray tracing due to single core limitations.
2. Scene definitions are difficult to describe as there is no graphical interfaced scene builder.
3. Glass caustic cannot be implemented under standard ray tracing (Requires photon mapping).
4. Diffuse reflections not implemented.

Points of interest:

1. We had introduced type-checking into our code, but, unfortunately, it only ended up slowing the program down.
2. We had also tried to speed up our object detection routine by using trees, but it yielded no or negative benefit for such small inputs, so we ended up abandoning the idea.
3. We have implemented CSG with union, difference and intersection operations. They work quite well.
4. We have sped up depth of field and soft shadow calculation by sampling a few points first to see whether multisampling is necessary or not.