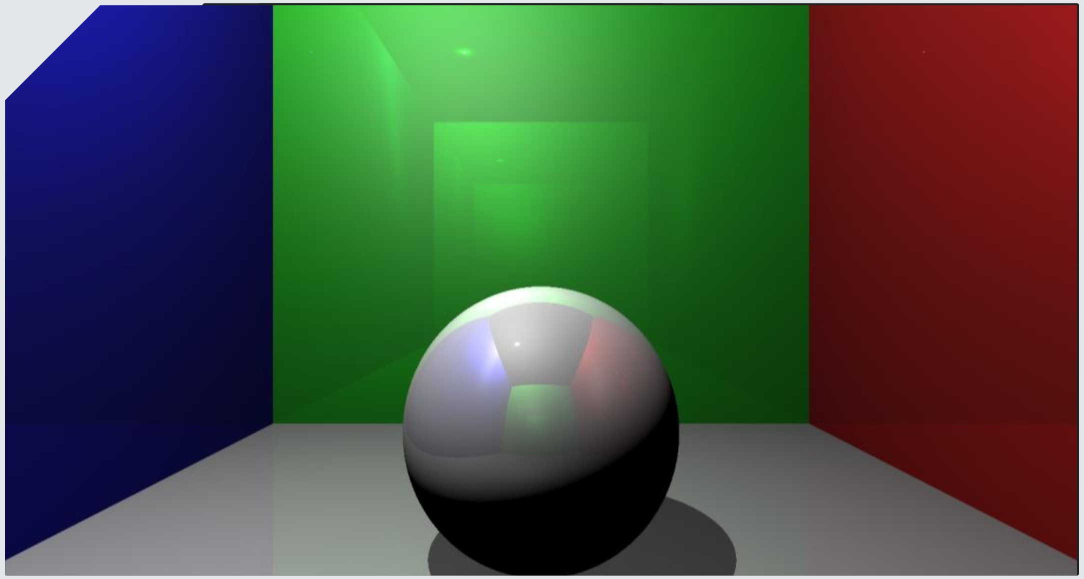
SOMAIYA
VIDYAVIHAR

Somaiya
TRUST

Python Ray Tracer

By Farzan, Rhythm and Eeshanya



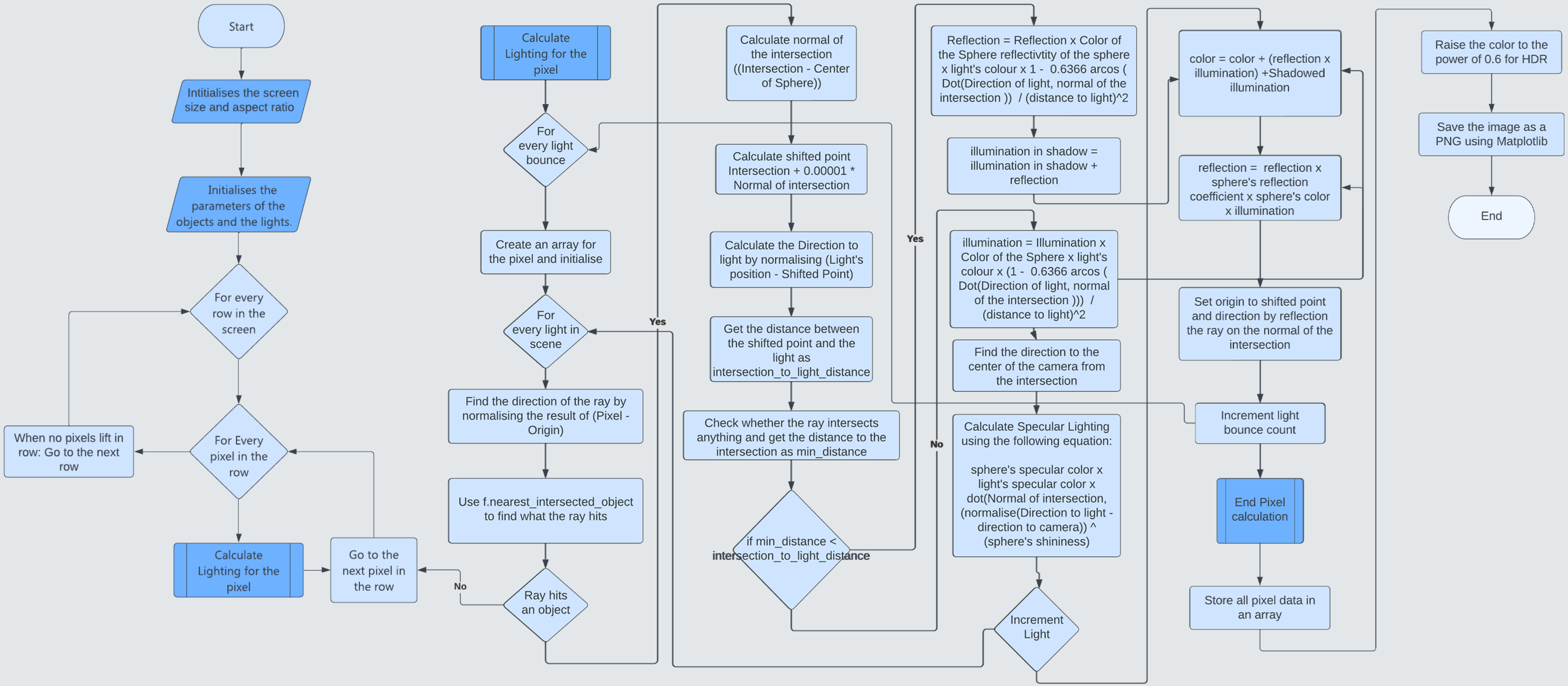
Problem Statement

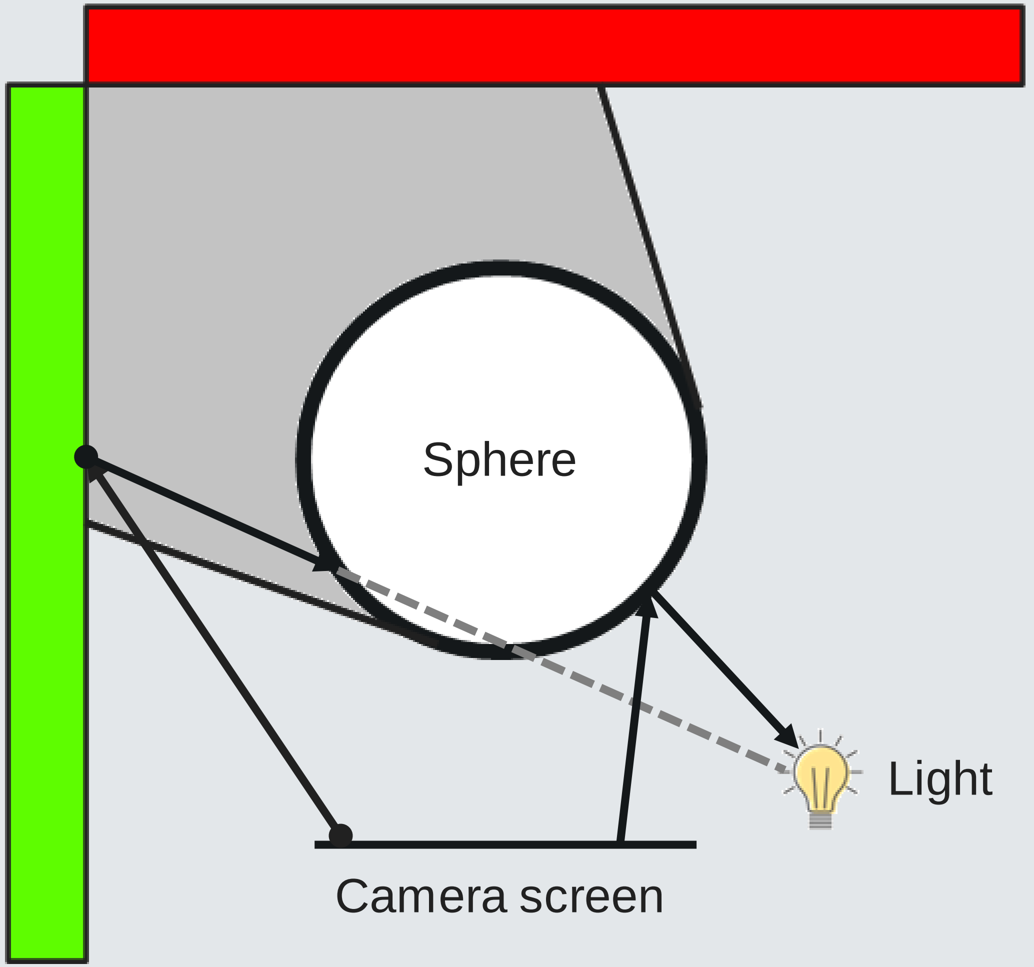
Simulation of light is an invaluable technique employed in animations and real-time applications, such as games. However, many existing methods lack physical accuracy and necessitate compromises in lighting precision. Ray tracing, on the other hand, strives to offer the utmost realism in lighting at the expense of performance.

Features

1. **Pixel Perfect Shadows:** Ray tracing calculates lighting for every pixel resulting in sharp and crisp shadows.
2. **Multi-Bounce Reflections:** Light rays bounce around the scene multiple times allowing reflections within reflections. The maximum bounces can be set by the user on cost of performance.
3. **Physically Based model light:** Light follows physical laws such as Inverse Square law and Angular law.
4. **Multiple Lights:** Unlike many software ray tracers, it supports shadows and reflections from multiple light sources.
5. **High Dynamic Range:** The renderer preserves detail in very bright areas and very dark areas despite the limited brightness range that can only have 256 values.

System Architecture




Sphere
Light
Camera screen

Main Ray Tracing Loop

```
image = np.zeros((height, width, 3))
for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
    for j, x in enumerate(np.linspace(screen[0], screen[2], width)):
        pixel = np.array([x, y, 0])
        origin = camera
        direction = f.normalise(pixel - origin)
        color = np.zeros((3))
        reflection = 1
        for k in range(max_depth):
            nearest_object : Sphere
            nearest_object, min_distance = f.nearest_intersected_object(objects, origin, direction)
            if nearest_object is None:
                break
            intersection = origin + min_distance * direction
            normal_to_surface = f.normalise(intersection - nearest_object.center)
            shifted_point = intersection + 1e-5 * normal_to_surface
            illumination = np.zeros((3))
            shadowed_illumination = np.zeros((3))
            for light in get_lights():
                intersection_to_light = f.normalise(light.position - shifted_point)
                _, min_distance = f.nearest_intersected_object(objects, shifted_point, intersection_to_light)
                intersection_to_light_distance = np.linalg.norm(light.position - intersection)
                is_shadowed = min_distance < intersection_to_light_distance
                if is_shadowed:
                    reflection *= 0.25 * (nearest_object.diffuse * nearest_object.reflection) * (light.diffuse * (1 - 0.63661977236759*np.arccos(np.dot(intersection_to_light, normal_to_surface))) / intersecti
                    shadowed_illumination += reflection
                illumination += nearest_object.diffuse * light.diffuse * (1 - 0.63661977236759*np.arccos(np.dot(intersection_to_light, normal_to_surface))) / intersection_to_light_distance ** 2
                intersection_to_camera = f.normalise(camera - intersection)
                H = f.normalise(intersection_to_light + intersection_to_camera)
                illumination += nearest_object.specular * light.specular * np.dot(normal_to_surface, H) ** (nearest_object.shininess / 4)
            color += (reflection * illumination) + (shadowed_illumination)
            reflection *= nearest_object.reflection * nearest_object.diffuse * illumination
            origin = shifted_point
            direction = f.reflected(direction, normal_to_surface)

        color = color * exposure + ((gamma*-1)+2.2)
        if hdr:
            color = color**0.6
        image[i, j] = np.clip(color, 0, 1)
    print(" %d / %d , %d" % (i + 1, height, (i+1)/height*100),"%")
```

Output