Course Project
Solar System Raytracing

Name	ID
Rana Mohamed Ali	21010528
Noran Ashraf Youssef	21011492
Ranime Ahmed Elsayed Shehata	21010531

1. Introduction

This project aims to develop a ray tracer that renders a realistic scene of the Solar System at a specified simulation time `t`, viewed from a configurable camera position and orientation. The focus is on offline rendering to achieve physical realism rather than real-time performance.

---

2. Code Structure and Important Modules

- **main.py**: Entry point; handles scene setup, simulation time, camera, rendering loop, and integrates all modules.
- **raytracer.py**: Core ray tracing engine implementing ray generation, intersection tests, shading, and image output.
- **objects/planet.py**: Defines the Planet class, including geometry, texture loading, and atmosphere rendering.
- **sphere.py:** Generates sphere geometry for planets, moon, and sun.
- **orbit.py:** Handles orbit visualization.
- **transformation.py:** Handles the translation, and rotation of planets.
- **saturn_ring.py:** Creates saturn ring.
- **effects/skybox.py:** Renders the background starfield (skybox).
- **json_parser.py:** Loads simulation time and camera parameters from the provided JSON file.
- **utils/window_renderer.py:** Manages OpenGL window, matrix transformations and shader setup.
- **camera.py:** Implements camera movement, orientation, and view matrix calculation.
- **shaders/:** Contains GLSL vertex and fragment shaders for rendering, including support for solid color and alpha blending.
- **assets/textures/**: Folder containing planet and moon surface textures sourced from NASA archives.

---

3. Computing Transformations at Time t and Camera Setup

- **Simulation Time:** The simulation time t is read from the JSON input and used to compute each planet's and the moon's position along their orbits, as well as their rotation at the given time.
- **Orbital Transformations:** For each celestial body, the position is calculated using:

  *angle = orbit_speed * t*

  *position = [orbit_radius * cos(angle), 0, orbit_radius * sin(angle)]*
- **Rotational Transformations:** Each planet is rotated around its axis based on its rotation_speed and the simulation time.

- **Camera Setup:** The camera's position, look-at target, and up vector are set from the JSON file. The camera supports user movement and rotation via keyboard input.

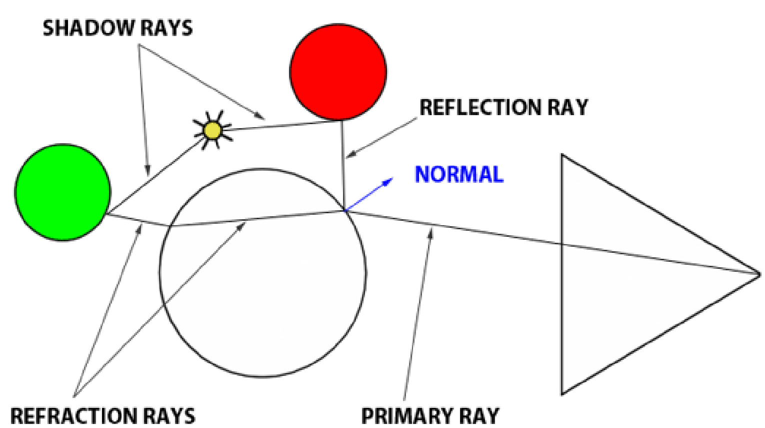
---

4. Materials and Lighting Models Used

- **Textures:** All planets and the moon use high-quality NASA textures mapped onto spheres.
- **Lighting:** The sun is modeled as an emissive object. Planets and the moon use Phong shading (ambient, diffuse, and specular components) for realism.
- **Atmosphere:** Earth's atmosphere is simulated by rendering a larger, semi-transparent blue sphere with alpha blending.
- **Sun Glow:** A billboard with additive blending simulates the sun's radiant glow.
- **Skybox:** A cube-mapped starfield provides a realistic background.

---

5. Ray Tracing Pseudocode



1. **First we check if we've bounced too many times (depth <=0):**
   - We return black as no more light is gathered.
2. **Check if the ray hits anything in the world:**
   - If not hit: return the background texture.
3. **If the ray hit an object:**
   - Get the hit point, surface normal and material properties.
   - Then we initialize the final color as black and we'll add the light contribution to it.
4. **Looping over each light source:**
   - We compute the direction from the hit point to the light.
   - Cast a shadow ray toward the light.

    - If the shadow ray hit something before reaching the light:
        - Attenuate the light intensity.
        - Then computing soft shadows by averaging multiple nearby shadow rays.
5. **If the light is visible:**
    - Compute the Phong lighting:
        - Ambient: small constant light contribution.
        - Diffuse: depends on how much the surface faces the light.
        - Specular: adds shiny highlights based on view and reflection direction.
    - Then we add these contributions to the final color.
6. **If the objective is reflective:**
    - We compute the reflected ray based on the normal and incoming ray.
    - We call the same algorithm recursively with the reflected ray and a reduced depth.
    - We add the reflected color to the final color.

---

6. How the ray is built:

1. **<u>Convert pixel to screen coordinates NDC:</u>**
    - Turns the pixel(x,y) into a number between -1 and 1.
    - This makes it easier to handle on different screen sizes.
    - We use dx and dy to help take samples inside the pixel for smoother images.
2. **<u>Map to the camera's viewing port</u>**
    - Uses the field of view to figure out how big the camera's image plane is.
    - Converts the screen position to a position on that image plane.
3. **<u>Create the ray direction in camera space</u>**
    - Imagining that the ray points forward from the camera
    - We set it to go through the point on the image plane
4. **<u>Rotating the ray to world space as:</u>**
    - The camera may not be facing forward in the world.
    - We adjust the ray direction using the camera's right, up and forward directions.
5. **<u>Returning the ray:</u>**
    - The final ray has:
        i. Origin: where the camera is.
        ii. Direction: where the ray should go to reach that pixel.

---

7. Challenges Faced

- **Accurate Transformations:** Ensuring all planets and the moon are positioned and rotated correctly at any simulation time required careful handling of transformation matrices.
- **Camera Controls:** Implementing a flexible camera system that correctly interprets arbitrary position, look-at, and up vectors.
- **Alpha Blending:** Making the atmosphere and sun glow visible and realistic required tuning blending modes and shader logic.
- **Shader Management:** Supporting both textured and solid color rendering in the same shader pipeline.
- **Aliasing in Ray Tracing:**
  Challenge: using a single ray per pixel causes visual artifacts because it undersampled fine details within the pixel.
  Solution: casting multiple rays per pixel at varied positions and averaging their results smooths edges and reduces aliasing by better capturing subpixel details.

---

8. Bonus Features

- **Skybox:** Implemented using a cube map for a realistic starfield.
- **Atmospheric Scattering:** Simulated by rendering a scaled, semi-transparent sphere around Earth with a blue color and alpha blending.
- **Saturn Ring:** Created the points with position relative to Saturn position, then added the texture.
- **Sun's Glow:**
  - Simulates a halo around the sun based on the angle between the view direction and the sun's direction.
  - Angular radius of the sun used to determine glow extent.
  - Smooth fade-out effect from inner glow to outer corona.
  - Adds sun rays (solar streaks) using sine-based angular modulation for cinematic flares.

These effects were implemented using additional shader code and blending techniques in the rendering pipeline and implemented also in the ray tracer algorithm.

---

9. Scene Input Format

- The renderer reads input from a JSON file specifying:
  - Simulation time `t` (in seconds)
  - Camera position `[x, y, z]`
  - Look-at target `[x, y, z]`
  - Camera up vector `[x, y, z]`

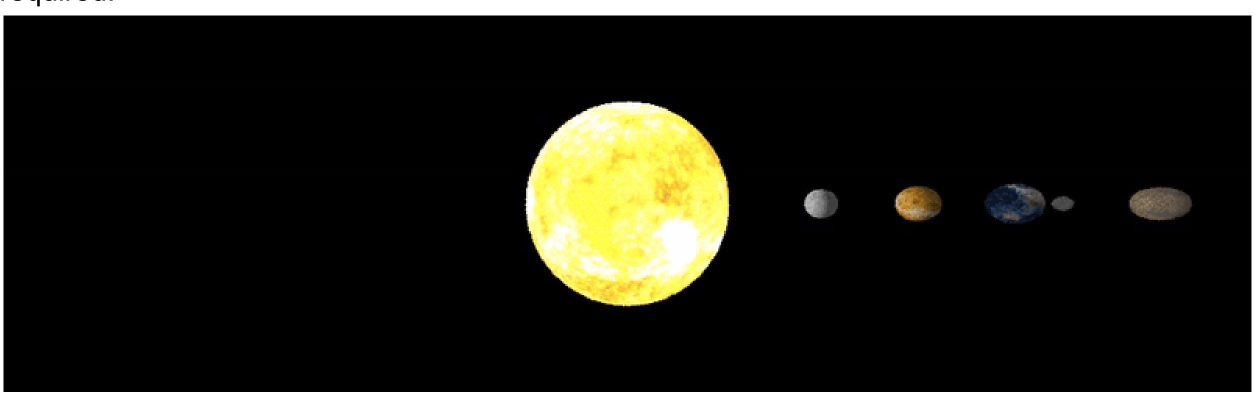
Example input:

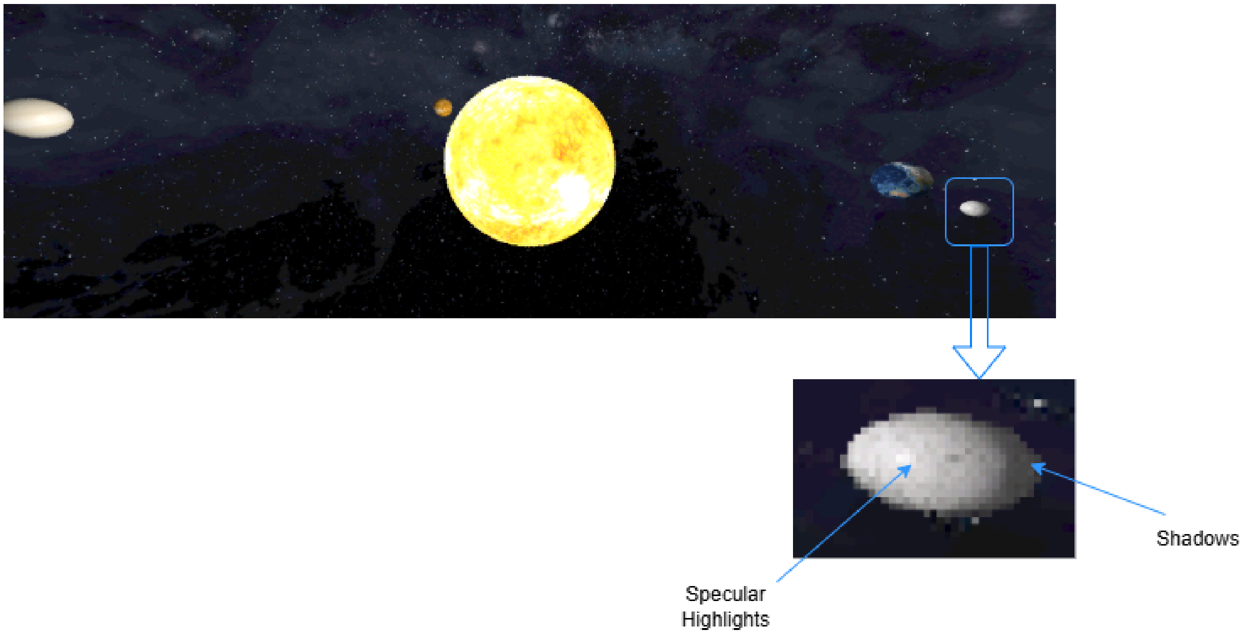
```
{
  "time": 86400,
  "camera": {
    "position": [0.0, 10.0, 50.0],
    "look_at": [0.0, 0.0, 0.0],
    "up": [0.0, 1.0, 0.0]
  }
}
```

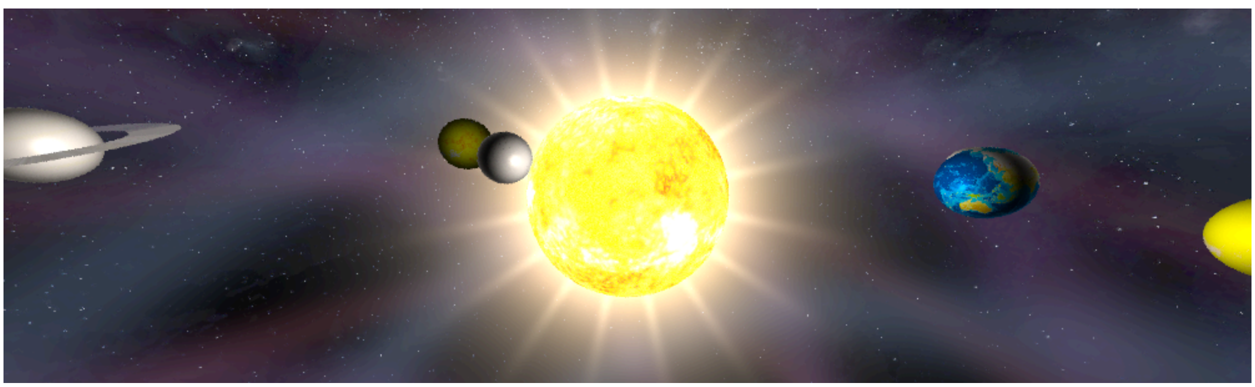
---

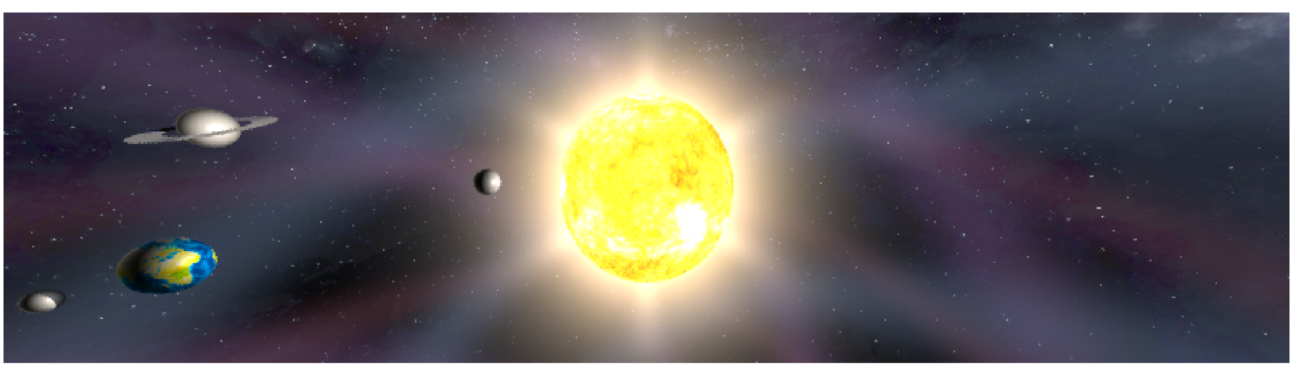
10. Rendering Results (Screenshots)

We rendered images of the Solar System at different simulation times and camera setups as required:









Note:

All code is original and does not use external raytracing engines.
All transformations, intersections, and shading are implemented from scratch.

Textures are sourced from NASA and are properly credited.

Resources:

https://planet-texture-maps.fandom.com/wiki/Mercury
https://www.solarsystemscope.com/textures/
https://www.songho.ca/opengl/gl_sphere.html
https://www.youtube.com/watch?v=0kPxilkCX_c&list=PL1P11yPQAo7opIg8r-4BMfh1Z_dCOfI0y&index=7
https://learnopengl.com/Lighting/Basic-Lighting
https://nasa3d.arc.nasa.gov/images
Ray Tracing in One Weekend by Peter Shirley
Scratchapixel's ray tracing
RGB to HSV conversion