Department of Computer Science
Technical University of Cluj-Napoca

Solar System Simulator

OpenGL Application

Developed By: Borbála Fazakas
GitHub Repository: OpenGL_SolarSystem
Group: 30432

Laboratory Assistant: Adrian Sabou

Contents

Chapter 1

Subject Specification

The goal of this project is to simulate the **Solar System**'s main components' movement.

The user can **observe** how

- how the planets, the moons and the Sun are positioned and how large they are relative to each other
- how the planets, the moons and the Sun rotate around their own axis
- how the planets orbit around the Sun
- how the moons orbit around a planet
- how the Sun illuminates all the objects in the Solar System
- how the objects in the Solar System cast shadows on each other

Additionally, the user can also **interact** with the system: if the "user" (i.e. the camera) collides with one of the planets/the moons, it is considered that the viewer **landed on that planet**, so they get redirected to a different scene, representing the surface of a planet.

Chapter 2

Scenario

2.1 Scenes

Based on the specification from the previous chapter, it is clear that the project requires two scenes: one for the entire Solar System, and one for the surface of the planet that the viewer landed on.

2.1.1 The Solar System Scene

The main scene of the project is the Solar System, which is also modeled with far more details.

2.1.1.1 The objects

- the Sun
- the 8 planets of the Solar System: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
- the Moon of the Earth
- 8 randomized moons of Jupiter

All of these objects were modeled in **Blender** based on **spheres** with different sizes. Additionally, the Moon of the Earth also has some deformations, ditches for its **craters**, created with Blender's **sculpt mode**.

2.1.1.2 Arranging the Objects in the Scene: sizes and distances

While modelling the Solar System scene, I always prioritized realism as much as possible. However, if all the distances and object sizes would have been kept realistic relative to each other, then hardly could the user see more than two planets on the screen at the same time,

planets would be hard to observe next to the Sun, etc. To overcome this issue, I set **distances between objects to be 250-times smaller relative to the the object sizes**.

To be specific, for the sizes and distances of the planets and the Sun

- the radius of the spheres representing the planets and the moons was set to be the real radius in km/1000 (in Blender)
- the radius of the sphere representing the Sun was set to be the real radius in km/4000 (in Blender)
- the distance between two objects is set to be the real distance in km/250000 (from the code)

However, since this setup would have caused many of the moons to fall inside their planet, the moons' sizes and distances from their planets were randomly assigned.

References:

- Planet Sizes
- Planets' distances from the Sun

2.1.1.3 Textures

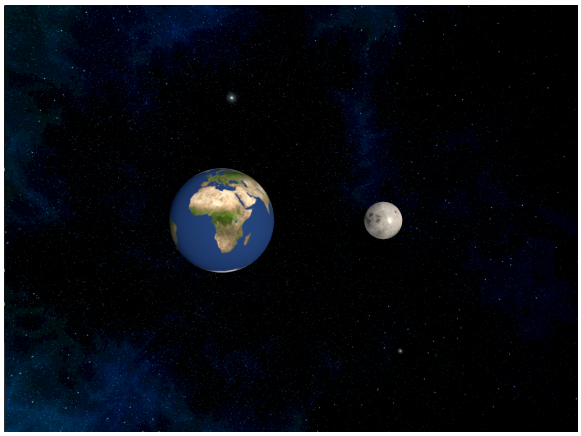
All the planets, the Sun and the Earth's Moon have a realistic texture, taken from solarsystemscope.com.

The moons of the Jupiter were given the same texture, taken from www.space.com.

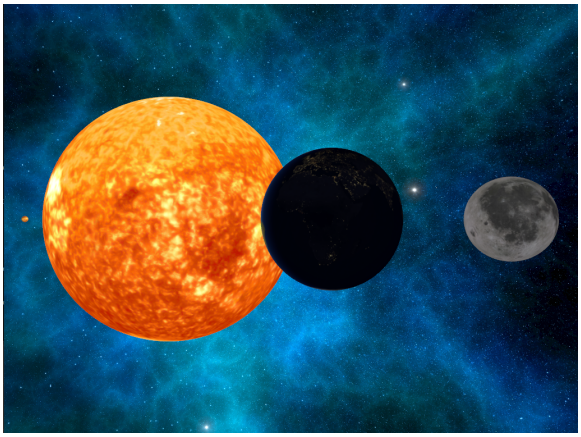
2.1.1.4 The SkyBox

The origin of the SkyBox images is this website.

2.1.1.5 Pictures of the final look

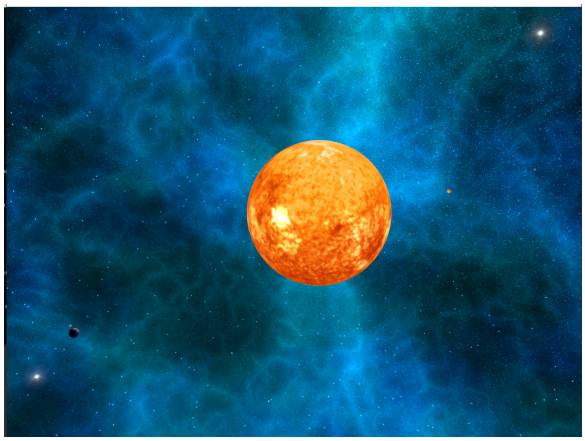


(a) The Earth and the Moon, when illuminated by the Sun

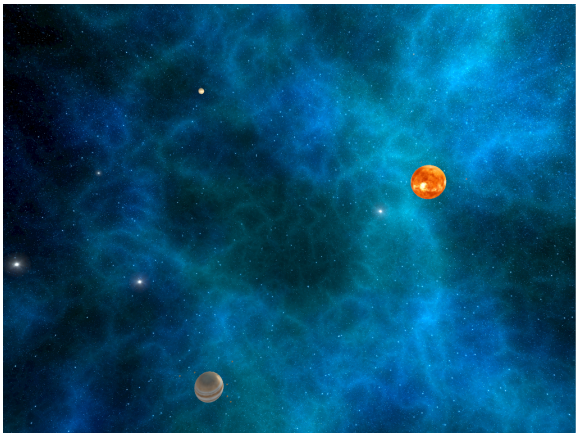


(b) The Earth and Moon, as seen from the other side, when they are not illuminated by the Sun

Figure 2.1: The Earth, the Moon and the Sun



(a) a view from above, with the Sun, the Earth, the Venus and Mars being visible



(b) a view from above, with the Sun, Jupiter and Saturn being visible

Figure 2.2: Views from above

2.1.2 The Planet Surface Scene

The second, and less detailed scene of the project is the Project Surface scene, This scene was in fact introduced to help demonstrate the effect of directional lights. additionally, the apparition of this scene also proves that collisions were correctly detected.

2.1.3 The objects

- a planet surface object, taken from free3d.com
- Curiosity rover, taken from free3d.com

The scene was arranged in Blender and exported such that the relative coordinates of the objects do not need to be modified from the code (the rover is standing on the surface, slightly away from the scene's origin).

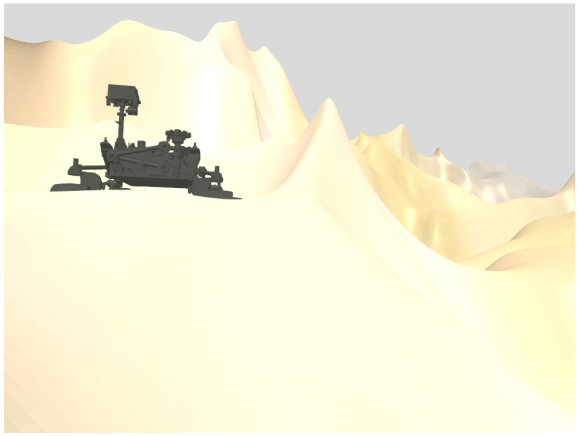


Figure 2.3: The Planet Surface scene

2.2 Functionalities

2.2.1 Animation: simulating the movement of the Sun, the planets and the moons

The primary functionality of the application is simulating the movement of the components of the solar system. All these animations are automatic, meaning that they do not require any user input to take place. The time intervals of the different movements are realistic relatively to each other: in the demo, 1 second in the real world corresponds to 100 seconds in the simulation, but this conversion can be easily changed by changing a global parameter in the code.

The movements, that I took into account are:

- the **rotation of the planets around their axis**. The axis and the rotation speed can be varied. In the demo, the rotation speeds are set based on spaceplace.nasa.gov, and the axes are all set to the Y axis in OpenGL.
- the **orbital movement of the planets around the Sun**. The axis of the orbit and the orbital period can be varied. In the demo, the orbital periods are based on nssdc.gsfc.nasa.gov and the axes are all set to the Y axis in OpenGL. Moreover, as an approximation, the orbits are considered to be circular, not elliptic.
- the **rotation of the Sun around its own axis**
- the **rotation of the moons around their own axis**. For the Earth's Moon, the speed is based on Wikipedia. For other moons, it was randomly set. The axis of the orbit is set to the Y axis for the Earth's Moon, and to a random value for Jupiter's moons.
- the **rotation of the moons around a planet**. For the Earth's Moon, the speed is based on Wikipedia. For other moons, it was randomly set. The axes of the orbits are all set to the Y vector in OpenGL. The orbits are all considered to be circular.

Observing the Solar System is made more please by allowing the user to move around the camera and direct it to different directions.

2.2.2 Interaction with the Solar System: Landing on Planets.

Additionally to observing the Solar System, an interactive functionality was added too: when the viewer's camera (positioned by the viewer from the keyboard) gets close enough to any planet/moon, the viewer can land on the planet/moon. This means, that the viewer is redirected to a different scene, representing a planet's surface, as described above. The viewer can also take off from the planet later, simply by pressing the "UP" button. This functionality is based on collision detection, as described in the next sections.

Chapter 3

Implementation details

3.1 Class Hierarchy

3.1.1 The Model Layer

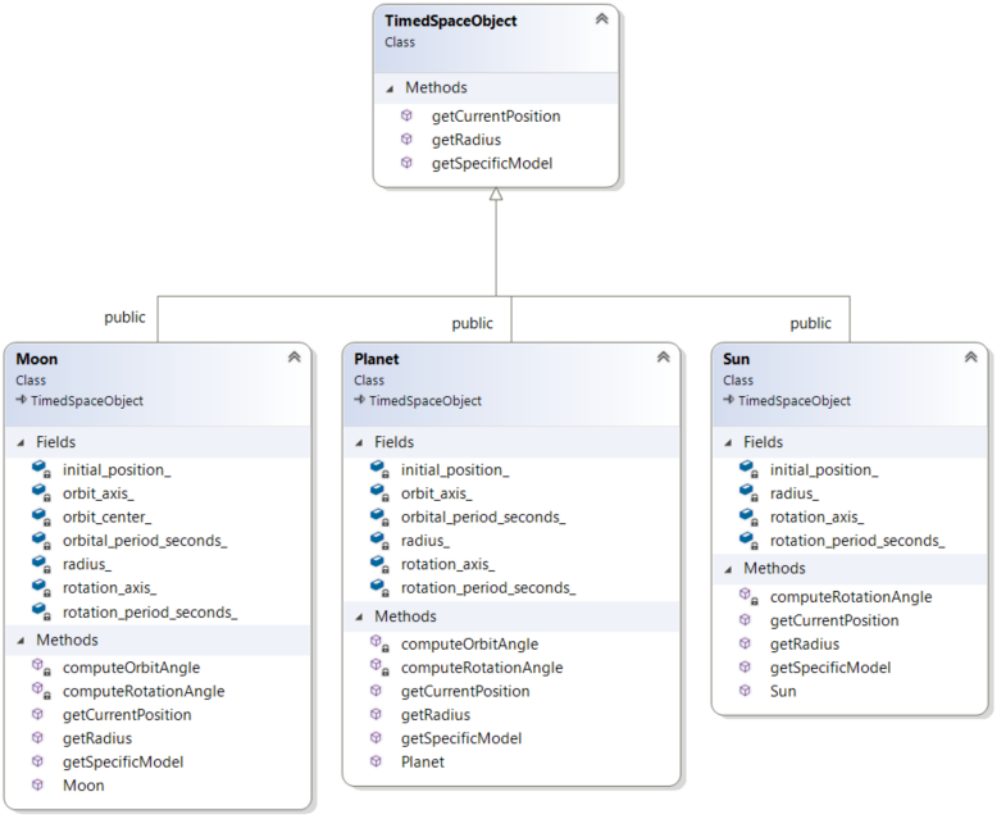


Figure 3.1: The Class Hierarchy of the Model Layer

The model classes are in charge of performing the mathematical computations for an object's position. By a TimedSpaceObject abstract class represents objects whose position depends on time.

Note that, the three types of TimedSpaceObjects are defined as:

- the Sun is an object placed in the system' origin, rotating around its own axis only
- a Planet is an object orbiting around the Sun (the origin) and rotating around its own axis
- a Moon is an object orbiting around a planet and rotating around its own axis. Thus, a Moon must have a reference to a planet (see the orbit_center_ field)

3.1.2 The View Layer

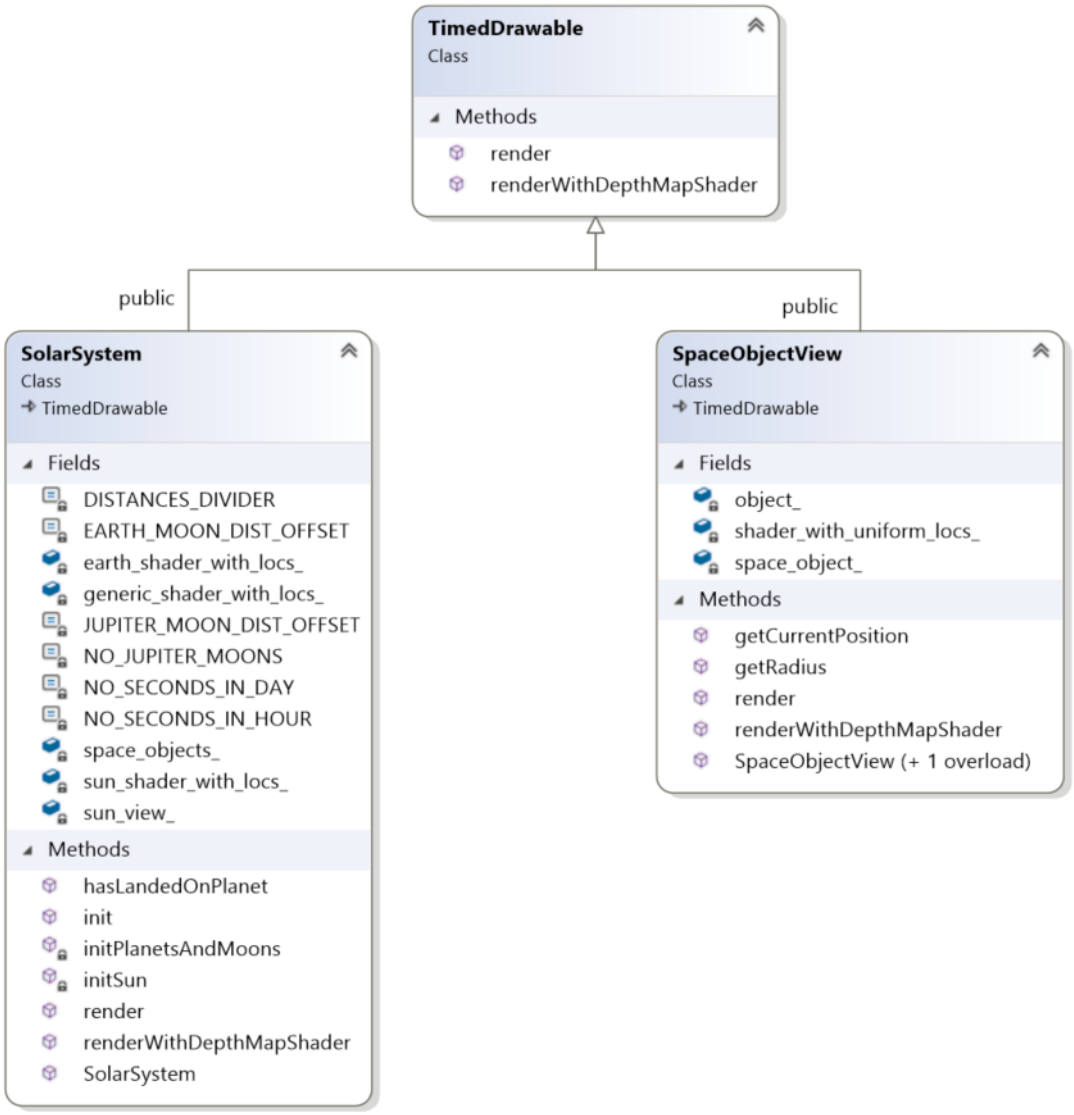


Figure 3.2: The Class Hierarchy of the View Layer

For displaying the Planets, Moons and the Sun, every TimedSpaceObject is wrapped in a SpaceObjectView, which is the class in charge of displaying the object: setting the active shader, sending it the required uniforms, and finally drawing the object. In this sense, a SolarSystem can be represented as a collection of SpaceObjectView objects.

3.1.3 Dependencies between the View and the Model

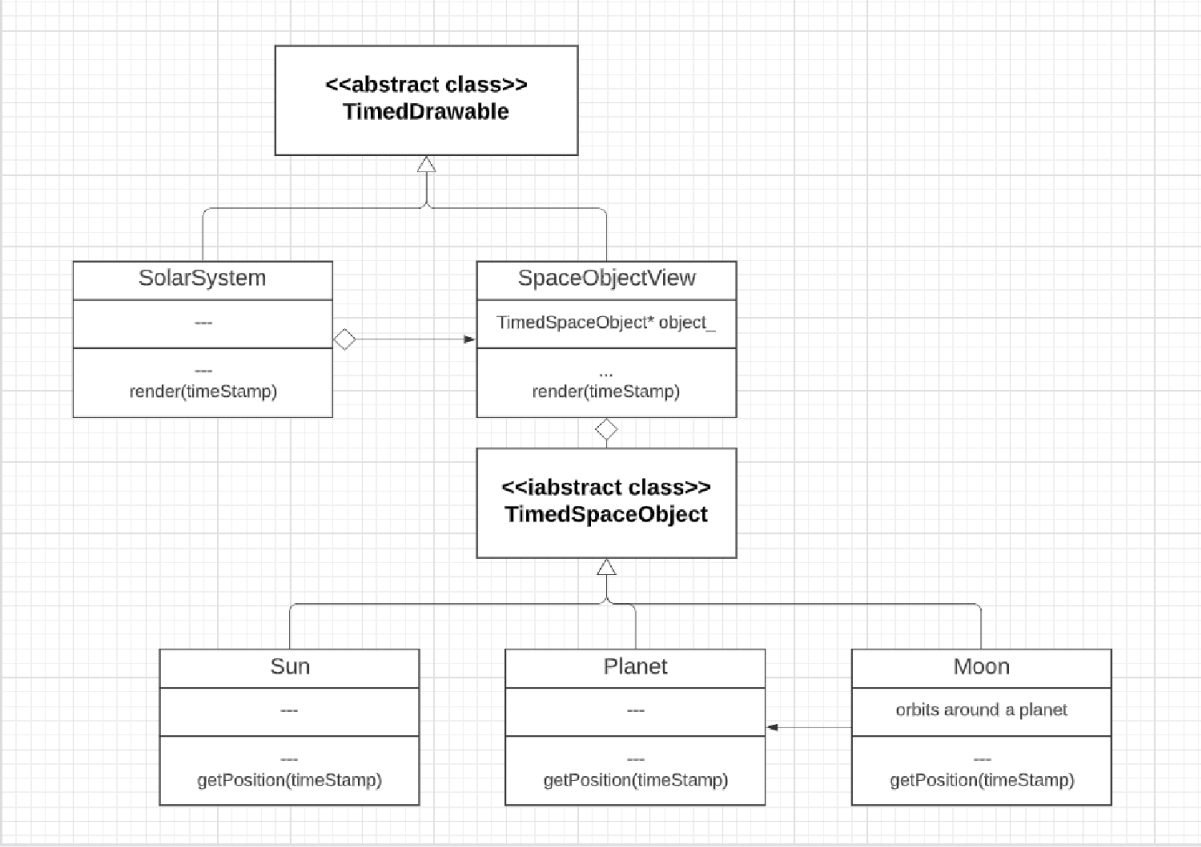


Figure 3.3: Dependencies between the View and the Model layer

3.2 Functions and Special Algorithms

3.2.1 Computing the position of the objects

As explained above, it's the Model classes' role to compute the object's positions at any time stamp. They provide a function, called

```
glm::mat4 Moon::getSpecificModel(long long current_seconds) {
    glm::mat4 my_model(1.0);

    // translate the moon to its actual position in the world space (considering the position of the orbit center): 4th transformation to be applied
    glm::vec3 current_orbit_center_pos = orbit_center_->getCurrentPosition(current_seconds);
    my_model = glm::translate(my_model, current_orbit_center_pos);

    // rotate planet (on the orbit): 3rd transformation to be applied
    float orbit_angle = computeOrbitAngle(current_seconds);
    my_model = glm::rotate(my_model, orbit_angle, orbit_axis_);

    // translate the moon to its position with respect to the orbit center: 2nd transformation to be applied
    my_model = glm::translate(my_model, initial_position_);
```

```
14
15      // rotate moon (rotation): 1st transformation to be applied
16      float rotation_angle = computeRotationAngle(current_seconds); // in
        radians
17      my_model = glm::rotate(my_model, rotation_angle, rotation_axis_);
18
19      return my_model;
20  }
21
22  float Moon::computeRotationAngle(long long current_seconds) {
23      current_seconds = current_seconds % rotation_period_seconds_;
24      float angle = ((float)current_seconds / (float)rotation_period_seconds_)
        * 2 * glm::pi<float>();;
25      return angle;
26  }
27
28  float Moon::computeOrbitAngle(long long current_seconds) {
29      current_seconds = current_seconds % orbital_period_seconds_;
30      float angle = ((float)current_seconds / (float)orbital_period_seconds_)
       * 2 * glm::pi<float>();;
31      return angle;
32  }
```

Listing 3.1: Model matrix computation example for the Moon class

The four main steps of the model class computation are:

1. rotate the object, depending on its rotation speed and the current time stamp, around its own axis
2. translate the object to its initial (time 0) position relative to the orbit axis
3. rotate the model according to the object's current position relative to the orbit center, depending on its orbital period, the current time stamp and on the current position of the center of the orbit
4. translate the model based on the position of the orbit center

3.2.2 Lighting: Point and Directional Lights

For representing the different types of light sources, the following data structures were defined, both in the fragment shaders and in the application code:

```
1  struct DirLight {
2    glm::vec3 direction;
3
4    glm::vec3 color;
5
6    float ambientStrength;
7      float diffuseStrength;
8    float specularStrength;
9  };
10
11 struct PointLight {
12     glm::vec3 position;
13
```

```
14      float constant;
15      float linear;
16      float quadratic;
17
18      glm::vec3 ambient;
19      glm::vec3 diffuse;
20      glm::vec3 specular;
21  };
```

Listing 3.2: Data Structures representing light sources

For lighting, I used **Phong's algorithm**, as it produces better quality, "smoother" results as the Gauraud algorithm, due to the fact that it works with the interpolated fragment normals in the fragment shader instead of the vertex normals available in the vertex shader.

3.2.2.1 The Solar System Scene

3.2.2.1.1 The Sun: a Point Light Source

In the context of a Solar System, the Sun becomes a point light, that I set in the origin of the World coordinate system, so the Sun's rays are falling in different angles on the different objects, depending on the object's relative location with respect to the Sun.

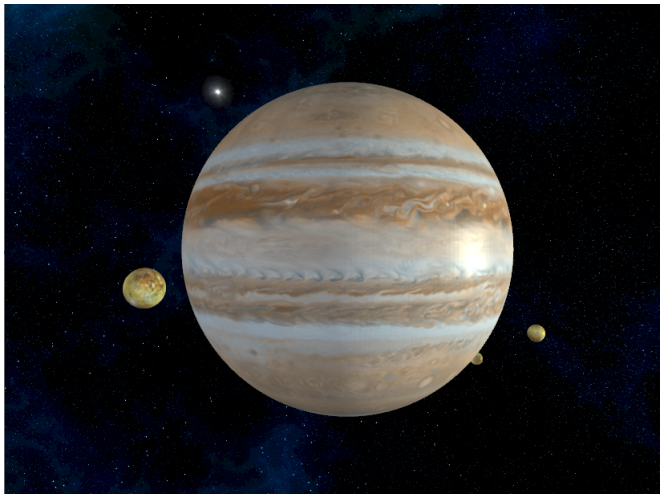


Figure 3.4: Demonstrating the lighting effect of the Sun on the Jupiter: the right side is brighter

3.2.2.1.2 Illuminating the Sun

Unfortunately, despite the Sun being expected to be the brightest object in the scene, with itself being the only light source, it will be rendered as if it were not illuminated at all by the diffuse and specular components - simply because all its vertices are facing "outwards", closing a >90 degrees angle with the light direction vector (pointing towards the light source). To overcome this issue and make the scene more realistic, I applied a second, directional light sourcem with a maximum-strength ambiental component only, only in the Sun's shader (so it has no effect on the other objects).

Finally, the light sources used in the Solar System scene are the following:

```
// white directional light, for the Sun only
view_layer::DirLight dirLight = {/*direction*/ glm::vec3(-1.0f, -1.0f, 0.0f)
    ,
                                 /*.color= */ glm::vec3(1.0f, 1.0f, 1.0f),
                                 /*.ambientStrength =*/ 0.4,
                                 /*.diffuseStrength =*/ 0.0,
                                 /*.specularStrength =*/ 0.0 };

// yellow-ish sun light (positional)
view_layer::PointLight sunLight = {
    /* .position= */ glm::vec3(0, 0, 0),
    /* .constant= */ 1.0,
    /* .linear= */   0.000014,
    /* .quadratic=*/ 0.000000007,
    /* .ambient = */ glm::vec3(0.0, 0.0, 0.0),
    /* .diffuse = */ glm::vec3(1.0, 0.952, 0.741),
    /* .specular= */ glm::vec3(1.0, 0.952, 0.741),
};
```

Listing 3.3: Light Sources in the SOlar System scene

3.2.2.1.3 Light-dependent texturing for the Earth

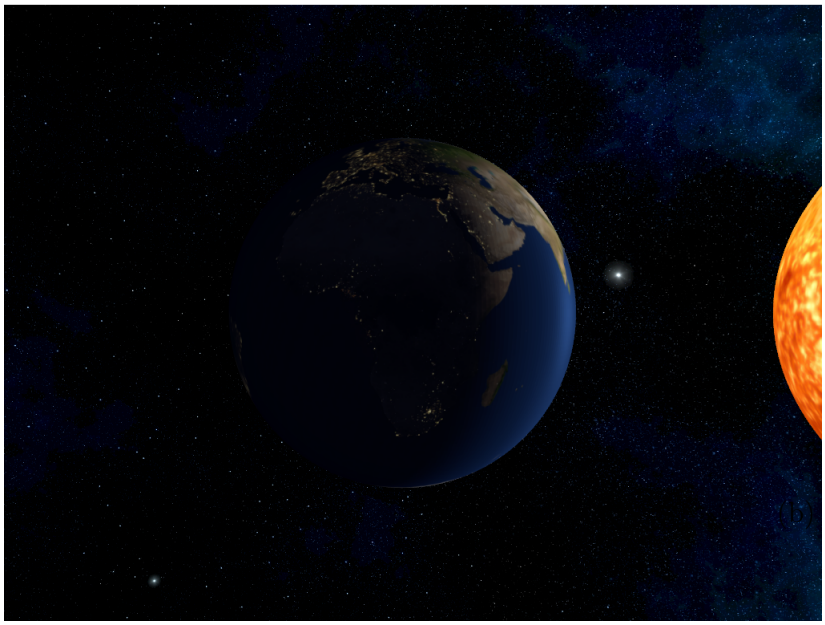
Using the Sun as a light source makes one side of all planets be darker than the other side, as expected. However, we know that for the Earth, the darkness of the colors is not the only difference between a day- and a night picture: in the nigh, street lights are turned on, land and water are more difficult to be differentiated, etc. To visualize this effect in the application, I used two different textures for the Earth, and decided in the Fragment shader which one to apply depending on the intensity of the sunlight falling on a fragment, represented by the dot product of the normalized fragment normal and the light direction vector.

- if the dot product > 0.4, then apply the day texture only
- if the dot product < 0.4, then apply the night texture
- if the dot product is in the range [-0.4, 0.4] then apply a linear combination of the two textures, the contribution of both textures being defined by the dot product, This step is necessary in order to avoid a "strict" line appearing on the border between the night and the day "side" of the Earth. To make the transition between the two sides even smoother, the smoothstep function was applied.

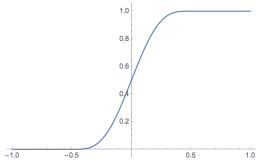
```
vec3 lightPos = (view * vec4(sun.position, 1.0)).xyz;

vec3 lightDir = normalize(lightPos - fragPos);

float sun_diff_value = dot(normal, lightDir);

sun_diff_smoothed_factor = smoothstep(-0.4, 0.4, sun_diff_value);

final_texture_color = vec3(texture(diffuseTexture, fTexCoords)) *
    sun_diff_smoothed_factor + vec3(texture(nightDiffuseTexture, fTexCoords))
     * (1 - sun_diff_smoothed_factor);
```

```
};
```

Listing 3.4: Light-dependent texturing for the Earth



(a) A demonstration of the light-dependent texturing from the Earth: the side towards the Sun has the day-texture, the other side the night-texture, and in the middle the smoothstep function was applied to smooth the transition between the two of them



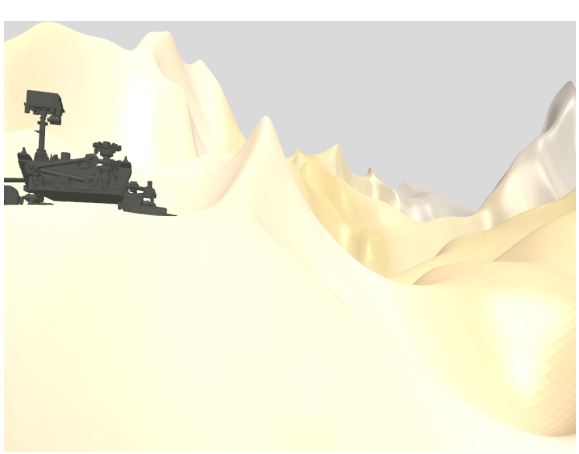
(b) The smoothstep function

Figure 3.5: Light-dependent texturing for the Earth

3.2.2.2 The Planet Surface Scene

3.2.2.2.1 The Sun: a Directional Light Source

When landing on a planet, due to the smaller size of the scene the Sun becomes a directional light source.



(a) Lighting with all light components

(b) The same scene, if the diffuse and specular components were removed

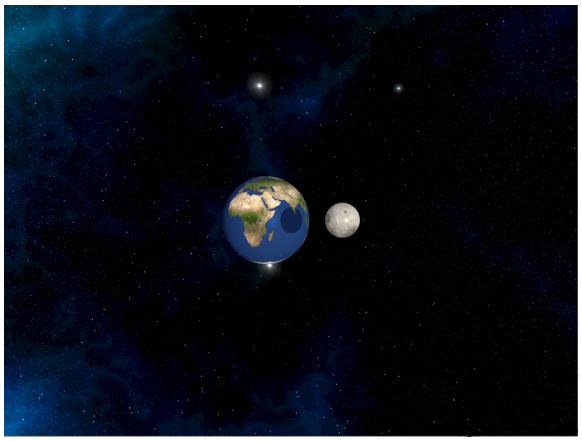
Figure 3.6: Lighting in the Planet Surface Scene

3.2.3 Shadowing: for Point Lights

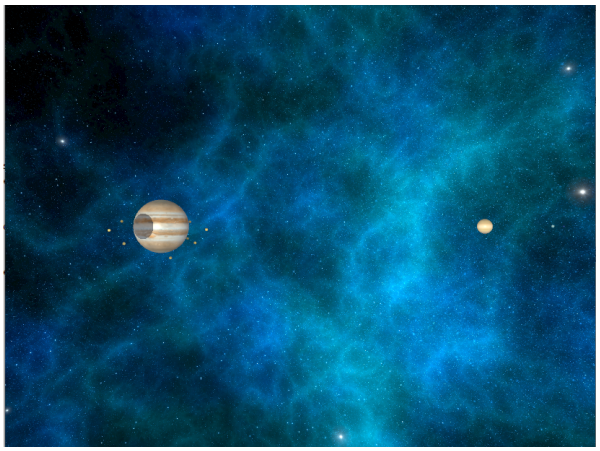
One of the major challenges for me in this project was implementing point shadows. For this, I integrated with slight adjustments the **Shadow Mapping** algorithm for point lights explained on the website learnopengl.com.

The challenge: the light rays go into all surrounding directions →a 2D shadow map cannot be applied anymore. The Solution: instead of a 2D shadow map apply a **depth cubemap**.

Unfortunately, even with the highest resolution that I could set on my GPU, the shadows are still not entirely accurate, but they allow me to demonstrate how the 3D shadow mapping with cubemaps work.



(a) The Moon casts a shadow on the Earth



(b) The Mercury casts a shadow on Jupiter

Figure 3.7: Shadowing

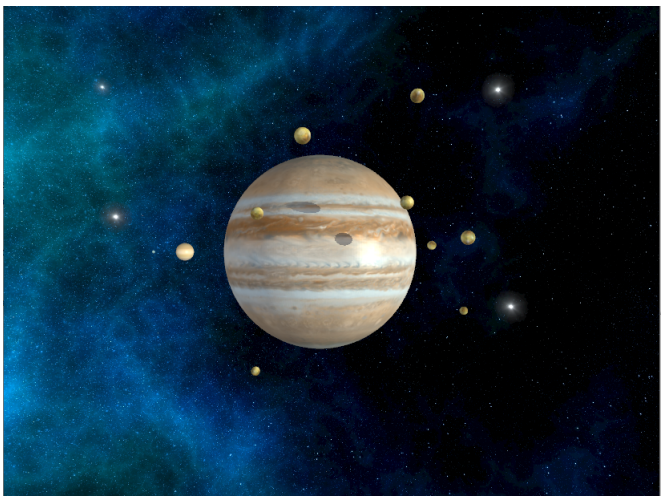


Figure 3.8: Two of Jupiter's moons casting a shadow on the planet

3.2.4 Collision Detection

A critical part of the interaction with the system is the collision detection between the camera and the planets/moons. Fortunately, since my objects where all spheres/close to spheres, I didn't need to apply any complicated approximations for the object surfaces. What I did was simply to compute the distance between the camera position and a planet/moon's center in the world coordinate system. If the distance was lower than the sphere's radius, then an object was signalled.

```
bool SolarSystem::hasLandedOnPlanet(long long current_seconds, glm::vec3
    current_pos) {
  for (auto& planet : space_objects_) {
    if (length(planet.getCurrentPosition(current_seconds) - current_pos) <
    planet.getRadius() * 1.1) {
      return true;
    }
  }
  return false;
}
```

Listing 3.5: Collision detection

Notice that instead of comparing the distance with the actual radius of the object, I compared it with 110% of the radius. This change frees users from having to perfectly position the camera on the surface of an object (since once they get past the surface, the object is not visible anymore), and makes the application more user-friendly.

Chapter 4

User Manual

4.0.1 Solar System Scene

When the user is in the Solar System scene, the buttons have the following effects:

1. **W**: move the camera forward
2. **S**: move the camera backward
3. **A**: move the camera to the left
4. **D**: move the camera to the right
5. **Q**: rotate the entire scene to the left
6. **E**: rotate the entire scene to the right
7. **T**: rotate the camera to the left (yaw control)
8. **Y**: rotate the camera to the right (yaw control)
9. **P**: rotate the camera upwards (pitch control)
10. **L**: rotate the camera downwards (pitch control)
11. **ENTER**: pause/resume the simulation (the animation will be frozen in moment of pressing the button. Allows users to better observe a certain configuration of the objects.)
12. **M**: enable/disable mouse control. Disabled by default.
13. **Z**: switch between the wireframe/solid display mode. Solid is the default.

Additionally, if the mouse control is enabled, then the camera direction can also be controlled by moving around the mouse.

4.0.2 Planet Surface Scene

When the user is in the Planet Surface scene, the buttons have the following effects:

1. **T**: rotate the camera to the left (yaw control)
2. **Y**: rotate the camera to the right (yaw control)
3. **P**: rotate the camera upwards (pitch control)
4. **L**: rotate the camera downwards (pitch control)
5. **M**: enable/disable mouse control. Disabled by default.
6. **Z**: switch between the wireframe/solid display mode. Solid is the default.
7. **UP**: take off from the planet. Redirects the viewer to the Solar System scene.

Additionally, similarly to the other scene, if the mouse control is enabled, then the camera direction can also be controlled by moving around the mouse.

Chapter 5

Conclusions and further developments

Overall, the algorithms and graphic models explained above allow us to get a realistic picture of the Solar System and enjoy it's breathtaking views.

There are however several further extensions that could make the application even nicer/more user friendly. For example:

- the application could allow users to attach the camera to a certain planet and view the scene continuously from the perspective of that planet.
- UI controls could be added to modify the simulation speed while running the application (currently, a parameter in the code must be modified before compiling it)
- UI controls could be added to modify the camera movement and rotation speed while running the application (currently, a parameter in the code must be modified before compiling it)
- in the Planet Surface scene, the texture of the surface could depend on the planet on which the vuser landed
- ... there are endless possibilities :)

Chapter 6

References

- planet sizes: solarsystem.nasa.gov
- planets' distances from the Sun: www.jpl.nasa.gov
- textures: www.solarsystemscope.com and www.space.com
- skybox images: opengameart.org
- planet rotation speeds: https://spaceplace.nasa.gov/days/en/
- planet orbital periods: https://nssdc.gsfc.nasa.gov/planetary/factsheet/
- point shadows: learnopengl.com