Department of Computer Science
Technical University of Cluj-Napoca

Graphic Processing

Solar System Project

Potra Paul-Antonio
Group: 30431 / 2

Teaching Assistant: Boancă Vlad Cătălin

Contents

Chapter 1

Subject specification

The aim of this project was to create a representation of the Solar System using the OpenGl framework, as well as the Blender application for object modelling. One goal was the photo-realism of the final application, which is why it packs a number of features such as: camera movement, lightning, shadow computation, fog generation, skybox display and collision detection.

Chapter 2

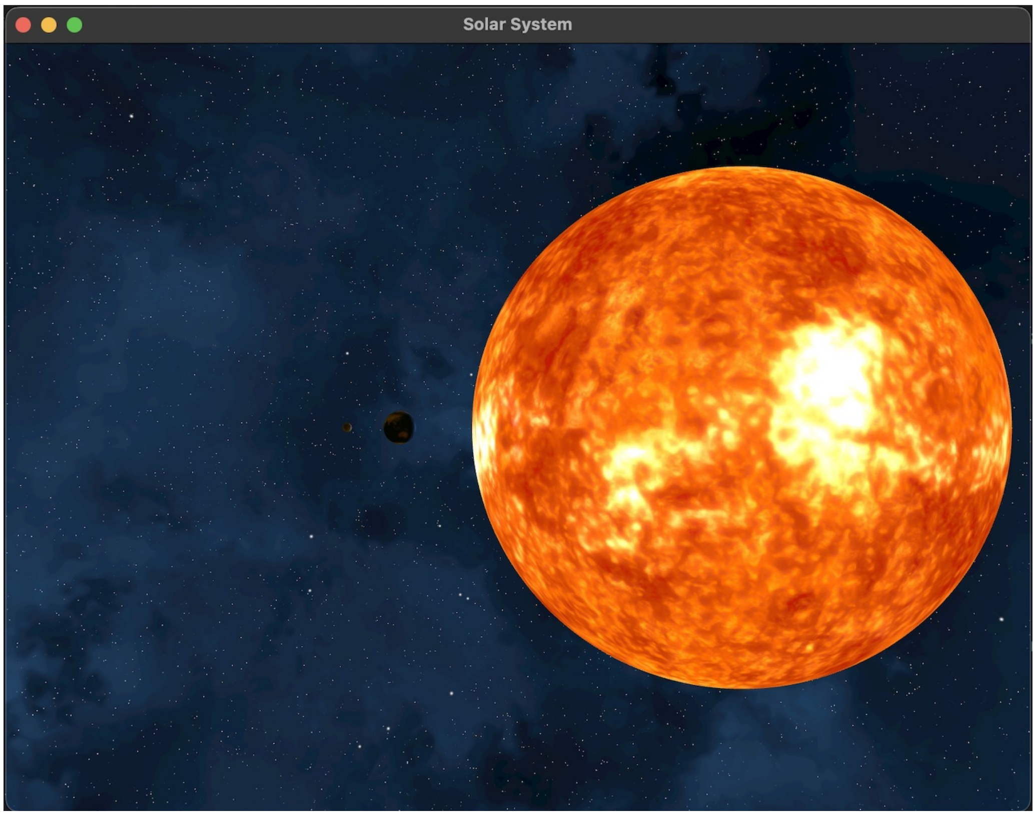
Scenario

2.1 Scene and objects description

There are 2 scenes in this project, one for the Solar System and one for the planet surface, with a total of 4 objects. The objects were all modeled in Blender, more detail on that later.

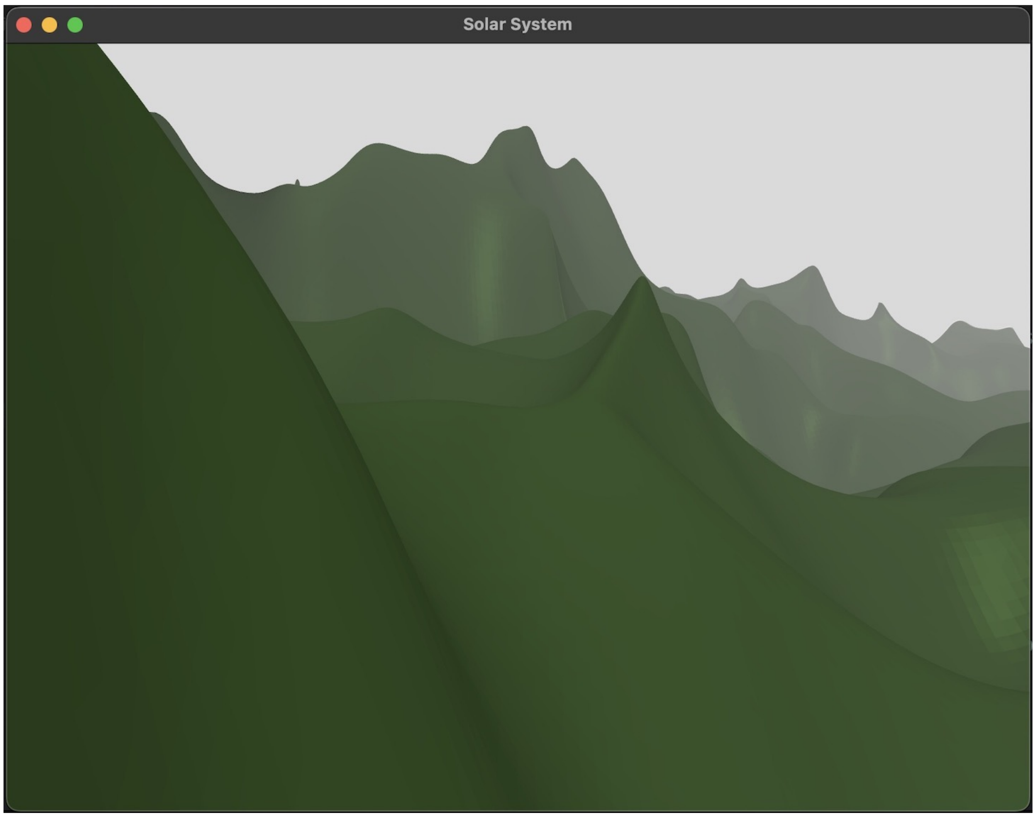
2.1.1 Solar System scene

This scene is surrounded by a space skybox, and it contains a total of 3 objects: the Sun, the Earth and the Moon. These objects were modeled in Blender as spheres with radii which coincide with the real ones, but divided by a constant factor, so that they would fit in the final scene. The distance between them is also divided, but this was done from the code, not from Blender. The objects are then imported separately and arranged from the code. Each object has a specific texture.



2.1.2 Planet surface scene

This scene consists of only one object which was modeled in Blender using the landscape add-on. The texture for this objects was created using PowerPoint.



2.2 Functionalities

2.2.1 Solar System scene

In this scene, the light source is the Sun, which is a point light. Shadow computation is also exemplified in this scene, but not by using the classical 2D depth map, but instead a 3D cube depth map.

The user can also observe animations:

- the Sun, the Earth and the Moon are rotating around their axis
- the Earth is orbiting around the Sun
- the Moon is orbiting around the Earth

During this animation, the shadows can be seen cast on the Earth by the Moon or vice-versa. Lightning can be observed at all times, since one side of the Earth will always be darker than the other side. Same thing applies for the Moon when it is not shadowed entirely by the Earth.

Last, but not least, this scene also computes the collision between the user (camera) and the Earth. When the user collided with the Earth, the planet surface scene will be displayed.

2.2.2 Planet surface scene

In this scene, the Sun is also the light Source, but this time it is a directional light source and it is not visible. The purpose of this scene is to prove that the collision computation worked correctly, as well as to exemplify fog computations. The user has the ability to move the camera around. As he moves further, the fog will become denser and denser, and the landscape will be harder to see.

One thing I need to mention is that for both scenes I used the Phong lightning model, since it produces better, smoother results than the Gouraud alternative.

Chapter 3

Implementation details

3.1 Camera

```cpp
#include "Camera.hpp"

namespace gps {

const float Camera::DFEAULT_MOVE_SPEED = 0.01;
const float Camera::DEFAULT_ROTATION_RADIANS = 0.0314;

// Camera constructor
Camera::Camera(glm::vec3 cameraPosition, glm::vec3 cameraTarget, glm::vec3
 ↪  cameraUp) {
    this->cameraPosition = cameraPosition;
    this->worldUpDirection = cameraUp;
    this->cameraFrontDirection = glm::normalize(cameraTarget -
     ↪  cameraPosition);
    updateCameraRightUpDirection();
}

// return the view matrix, using the glm::lookAt() function
glm::mat4 Camera::getViewMatrix() {
    return glm::lookAt(cameraPosition, cameraPosition + cameraFrontDirection,
     ↪  cameraUpDirection);
}

glm::vec3 Camera::getCameraPosition() {
    glm::vec3 res = cameraPosition;
    return res;
}

// update the camera internal parameters following a camera move event
void Camera::move(MOVE_DIRECTION direction, float speed) {
    // TODO: modify cameraTarget
    switch (direction) {
    case MOVE_LEFT:
        cameraPosition -= cameraRightDirection * speed;
        return;
    case MOVE_RIGHT:
        cameraPosition += cameraRightDirection * speed;
        return;
    case MOVE_FORWARD:
        cameraPosition += cameraFrontDirection * speed;
        return;
    case MOVE_BACKWARD:
        cameraPosition -= cameraFrontDirection * speed;
    }
}

// update the camera internal parameters following a camera rotate event
// yaw - camera rotation around the y axis
// pitch - camera rotation around the x axis
void Camera::rotate(float pitch, float yaw) {
    glm::vec3 previousCameraFrontDirection = cameraFrontDirection;

    // pitch
    glm::mat4 pitchRotator = glm::rotate(glm::mat4(1.0f), glm::radians(pitch),
    ↪  cameraRightDirection);
    cameraFrontDirection = pitchRotator * glm::vec4(cameraFrontDirection,
    ↪  1.0);

    // yaw
    glm::mat4 yawRotator = glm::rotate(glm::mat4(1.0f), glm::radians(yaw),
    ↪  cameraUpDirection);
    cameraFrontDirection = yawRotator * glm::vec4(cameraFrontDirection, 1.0);

    cameraFrontDirection = glm::normalize(cameraFrontDirection);
    if (cameraFrontDirection == worldUpDirection) {
        cameraFrontDirection = previousCameraFrontDirection;
    } else {
        updateCameraRightUpDirection();
    }
}

void Camera::updateCameraRightUpDirection() {
    cameraRightDirection = glm::normalize(glm::cross(cameraFrontDirection,
    ↪  worldUpDirection));
    cameraUpDirection = glm::normalize(glm::cross(cameraRightDirection,
    ↪  cameraFrontDirection));
}
} // namespace gps
```

3.2 Skybox

```
#include "Skybox.hpp"

namespace gps {
```

```
Skybox::Skybox() {
}

void Skybox::Load(std::vector<const GLchar *> cubeMapFaces) {
    cubemapTexture = LoadSkyBoxTextures(cubeMapFaces);
    InitSkyBox();
}

void Skybox::Draw(gps::Shader shader, glm::mat4 viewMatrix, glm::mat4
 ↪  projectionMatrix) {
    shader.useShaderProgram();

    // set the view and projection matrices
    glm::mat4 transformedView = glm::mat4(glm::mat3(viewMatrix));
    glUniformMatrix4fv(glGetUniformLocation(shader.shaderProgram,
     ↪  "viewMatrix"), 1, GL_FALSE, glm::value_ptr(transformedView));
    glUniformMatrix4fv(glGetUniformLocation(shader.shaderProgram,
     ↪  "projectionMatrix"), 1, GL_FALSE, glm::value_ptr(projectionMatrix));

    glDepthFunc(GL_LEQUAL);

    glBindVertexArray(skyboxVAO);
    glActiveTexture(GL_TEXTURE0);
    glUniform1i(glGetUniformLocation(shader.shaderProgram, "skybox"), 0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);

    glDepthFunc(GL_LESS);
}

GLuint Skybox::LoadSkyBoxTextures(std::vector<const GLchar *> skyBoxFaces) {
    GLuint textureID;
    glGenTextures(1, &textureID);
    glActiveTexture(GL_TEXTURE0);

    int width, height, n;
    unsigned char *image;
    int force_channels = 3;

    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
    for (GLuint i = 0; i < skyBoxFaces.size(); i++) {
        image = stbi_load(skyBoxFaces[i], &width, &height, &n,
         ↪  force_channels);
        if (!image) {
            fprintf(stderr, "ERROR: could not load %s\n", skyBoxFaces[i]);
            return false;
        }
        glTexImage2D(
            GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
            GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    glBindTexture(GL_TEXTURE_CUBE_MAP, 0);

    return textureID;
}

void Skybox::InitSkyBox() {
    GLfloat skyboxVertices[] = {
        -1.0f, 1.0f, -1.0f,
        -1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, 1.0f, -1.0f,
        -1.0f, 1.0f, -1.0f,

        -1.0f, -1.0f, 1.0f,
        -1.0f, -1.0f, -1.0f,
        -1.0f, 1.0f, -1.0f,
        -1.0f, 1.0f, -1.0f,
        -1.0f, 1.0f, 1.0f,
        -1.0f, -1.0f, 1.0f,

        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,

        -1.0f, -1.0f, 1.0f,
        -1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, -1.0f, 1.0f,
        -1.0f, -1.0f, 1.0f,

        -1.0f, 1.0f, -1.0f,
        1.0f, 1.0f, -1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        -1.0f, 1.0f, 1.0f,
        -1.0f, 1.0f, -1.0f,

        -1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f, 1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f, 1.0f,
        1.0f, -1.0f, 1.0f};

    glGenVertexArrays(1, &(this->skyboxVAO));
    glGenBuffers(1, &skyboxVBO);

    glBindVertexArray(skyboxVAO);
    glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices,
    ↪   GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),
    ↪   (GLvoid *)0);

    glBindVertexArray(0);
}

GLuint Skybox::GetTextureId() {
    return cubemapTexture;
}
} // namespace gps
```

3.3 Animations

This will be exemplified on the Moon, since it's animations are the most complex (rotation around its axis and orbiting around the Sun).

```
glm::mat4 Moon::getSpecificModel(long long currentSeconds) {
    glm::mat4 myModel(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 currentEarthPosition =
    ↪   earth->getCurrentPosition(currentSeconds);
    myModel = glm::translate(myModel, currentEarthPosition);

    // rotate planet (on the orbit): 3rd transformation to be applied
    float orbitAngle = computeOrbitAngle(currentSeconds);
    myModel = glm::rotate(myModel, orbitAngle, orbitAxis);

    // translate the moon to its position with respect to the orbit center:
    ↪   2nd transformation to be applied
    myModel = glm::translate(myModel, initialPosition);

    // rotate moon (rotation): 1st transformation to be applied
    float rotationAngle = computeRotationAngle(currentSeconds); // in radians
    myModel = glm::rotate(myModel, rotationAngle, rotationAxis);

    return myModel;
}

glm::vec3 Moon::getCurrentPosition(long long currentSeconds) {
    glm::mat4 myModel(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 currentEarthPosition =
    ↪  earth->getCurrentPosition(currentSeconds);
    myModel = glm::translate(myModel, currentEarthPosition);

    // rotate planet (on the orbit): 3rd transformation to be applied
    float orbitAngle = computeOrbitAngle(currentSeconds);
    myModel = glm::rotate(myModel, orbitAngle, orbitAxis);

    // translate the moon to its position with respect to the orbit center:
    ↪  2nd transformation to be applied
    myModel = glm::translate(myModel, initialPosition);

    return glm::vec3(myModel * glm::vec4(initialPosition, 1.0));
}

float Moon::computeRotationAngle(long long currentSeconds) {
    currentSeconds = currentSeconds % rotationPeriod;
    float angle = ((float)currentSeconds / (float)rotationPeriod) * 2 *
    ↪  glm::pi<float>();
    return angle;
}

float Moon::computeOrbitAngle(long long currentSeconds) {
    currentSeconds = currentSeconds % orbitPeriod;
    float angle = ((float)currentSeconds / (float)orbitPeriod) * 2 *
    ↪  glm::pi<float>();
    return angle;
}
```

3.4 Collision

```
bool SolarSystem::hasLandedOnEarth(long long currentSeconds, glm::vec3
↪  currentPosition) {
    if (length(earth->getCurrentPosition(currentSeconds) - currentPosition) <
    ↪  earth->getRadius() * 1.1) {
        return true;
    }
    return false;
}
```

3.5 Point shadows

I used an algorithm which implies a cubemap and the use of a geometry shader.

```
// vertex
#version 330 core

layout (location = 0) in vec3 vPosition;

uniform mat4 modelMatrix;

void main() {
    gl_Position = modelMatrix * vec4(vPosition, 1.0);
}

// geometry
#version 330 core

in vec4 FragPos;

uniform vec3 lightPosition;
uniform float farPlane;

void main() {
    // get distance between fragment and light source
    float lightDistance = length(FragPos.xyz - lightPosition);

    // map to [0;1] range by dividing by farPlane
    lightDistance = lightDistance / farPlane;

    // write this as modified depth
    gl_FragDepth = lightDistance;
}

// geometry
#version 330 core

layout (triangles) in;
layout (triangle_strip, max_vertices=18) out;

uniform mat4 shadowMatrices[6];

out vec4 FragPos; // FragPos from GS (output per emitvertex)

void main() {
    for(int face = 0; face < 6; ++face) {
        gl_Layer = face; // built-in variable that specifies to which face we
         ↪  render.
        for(int i = 0; i < 3; ++i) { // for each triangle's vertices
            FragPos = gl_in[i].gl_Position;
            gl_Position = shadowMatrices[face] * FragPos;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Chapter 4

User manual

- W: move the camera forward
- S: move the camera backward
- A: move the camera to the left
- D: move the camera to the right
- Q: rotate the entire scene to the left
- E: rotate the entire scene to the right
- T: rotate the camera to the left
- Y: rotate the camera to the right
- P: rotate the camera upwards
- L: rotate the camera downwards (pitch control)
- ENTER: enable/disable the animations (enabled by default)
- M: enable/disable mouse control (disabled by default)
- Z: switch between wire-frame/solid display mode (wire-frame by default)
- UP: take off if landed on Earth

Chapter 5

Conclusions and further developments

Even at a first glance, it is fairly easy to see some possible future developments for the project:

- adding the rest of the planets to the Solar System
- adding more details to make the scene more photo-realistic (e.g. Saturn's rings, asteroids, satellites)
- making a more complex planet surface

Some more advanced improvements could be:

- allowing the camera to follow a single planet through the course of its animation
- handling interactions between multiple moons (e.g. for a planet like Jupiter, make sure its moons don't collide)
- changing the animation speed while the application is running

Still, the application allows us to view a simple representation of the Solar System, and it allowed me to get some hands-on experience with the OpenGl framework.

Chapter 6

References

- https://ogldev.org/www/tutorial43/tutorial43.html
- https://tools.wwwtyro.net/space-3d/index.html#animationSpeed=1&fov=80&nebulae=true&pointStars=true&resolution=1024&seed=30zhg6cqahmo&stars=true&sun=true
- https://learnopengl.com/Lighting/Multiple-lights
- https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows
- https://ogldev.org/www/tutorial20/tutorial20.html
- https://www.solarsystemscope.com/textures/