DOCUMENTATATION

GP PROJECT: OUTER SPACE

STUDENT NAME: PELLE ANDREI
GROUP: 30434

TABLE OF CONTENTS

1. Subject Specification

The subject of this project is the photorealistic presentation of 3D objects using OpenGL library. Open Graphics Library is a specification of a standard which defines a multiplatform API, widely used for programming the 2D and 3D components of computer programs.

The scene I chose to implement is a space scene set on a planet in outer space. The user controls the scene of objects from the keyboard input. Several functionalities have been implemented, like moving objects, teleportation, transparency, fog and so on.

2. Scenario

A) Scene of object description

The scene depicts a space scene with various space related objects on a mars-like planet. The main base is formed of multiple astronauts, buildings, a spinning metal pyramid, a TIE fighter, and a military helicopter. Behind us there are 2 buildings: a prosperity dome and a big power plant.

There is also a spinning moon very far away with some astronauts and an asteroid ring. Next there is a star trek base and another planet with asteroids.

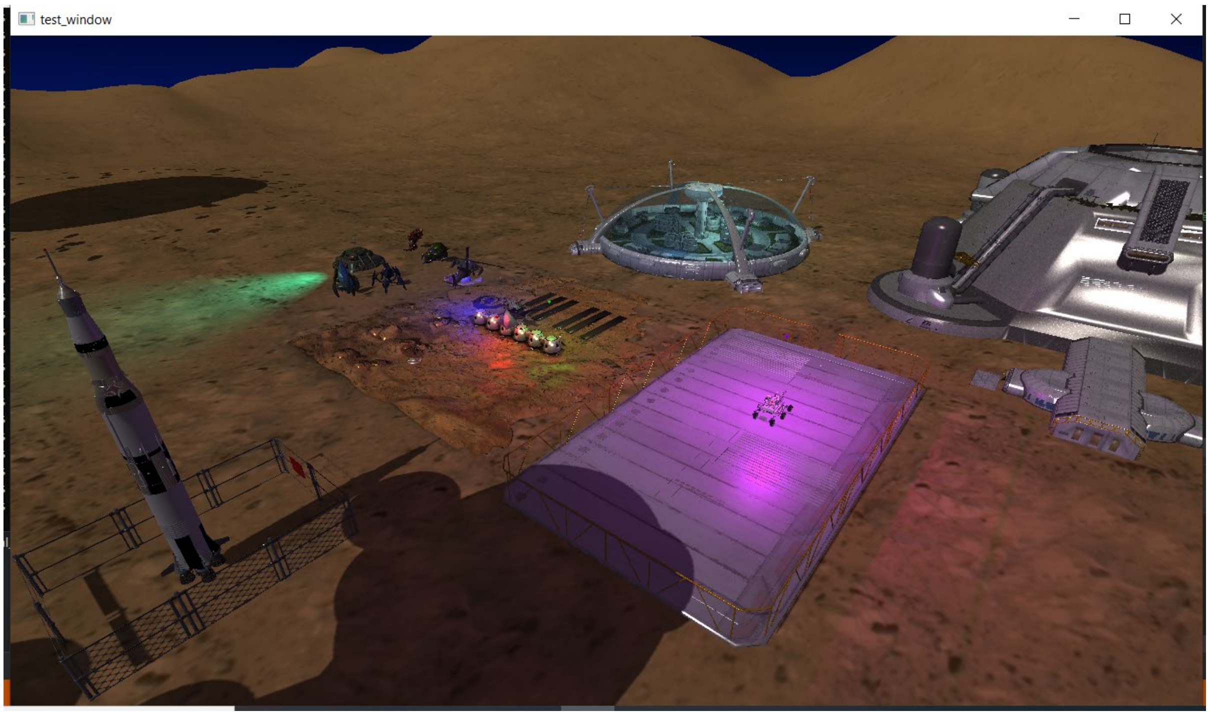


Figure 1: Main scene

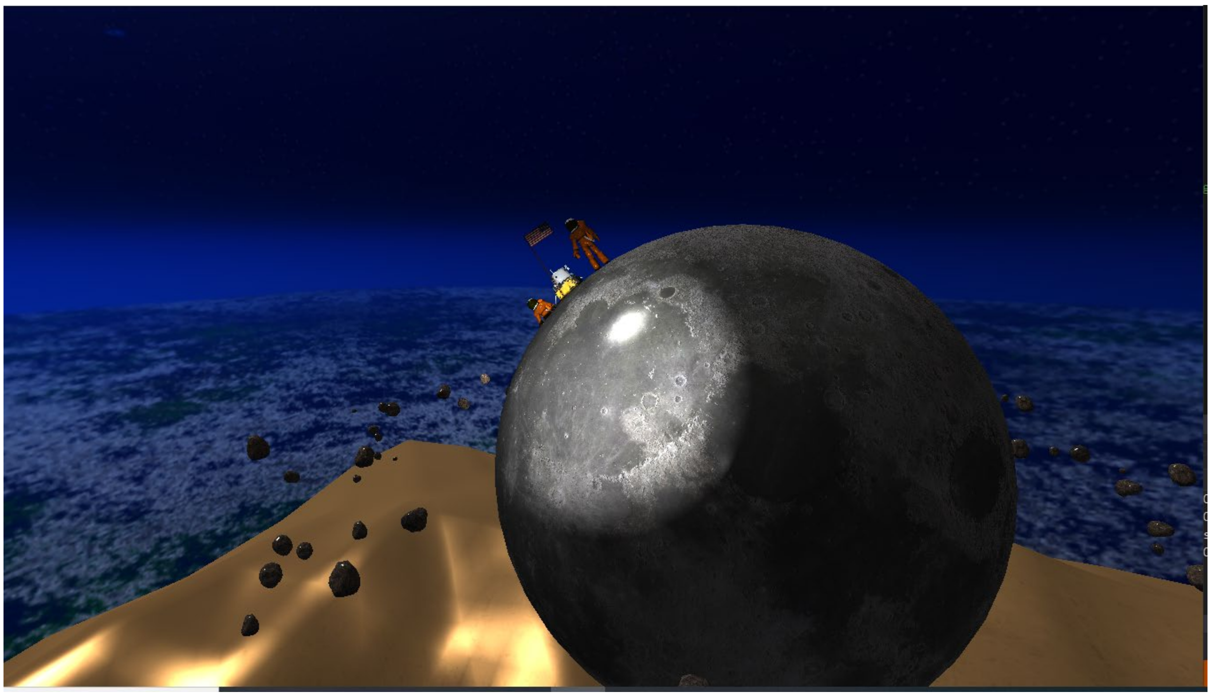


Figure 2: moon visualization

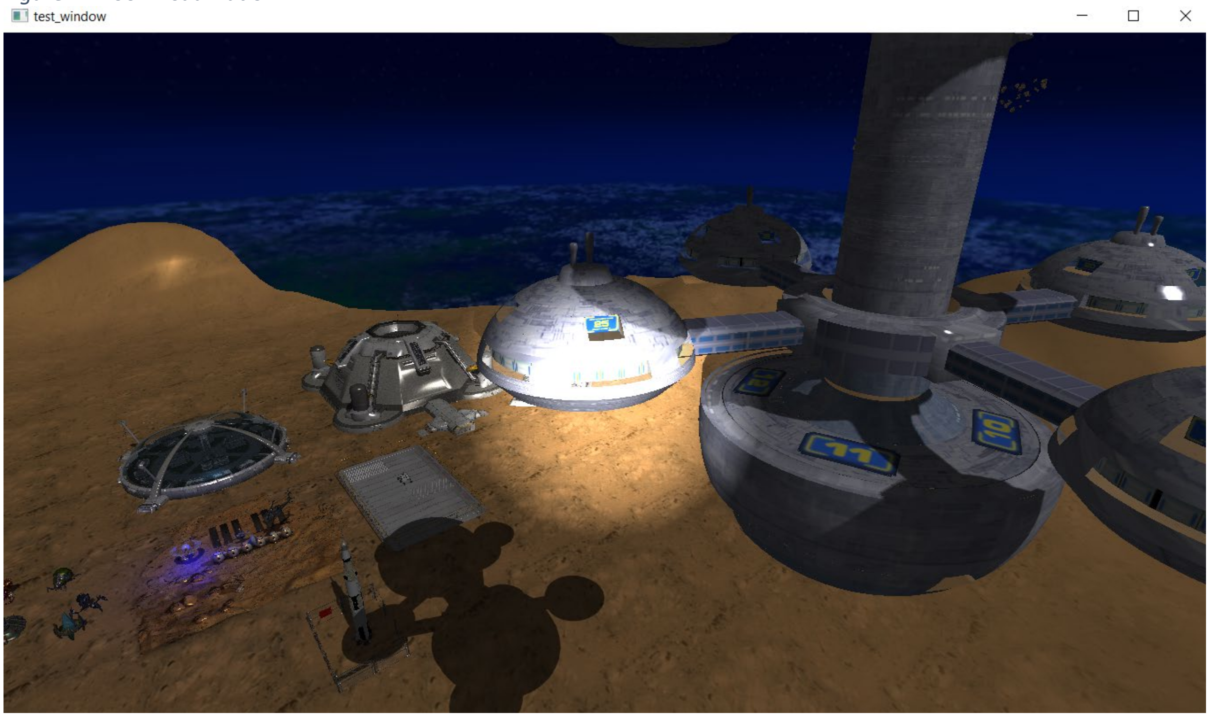


Figure 3: star trek base

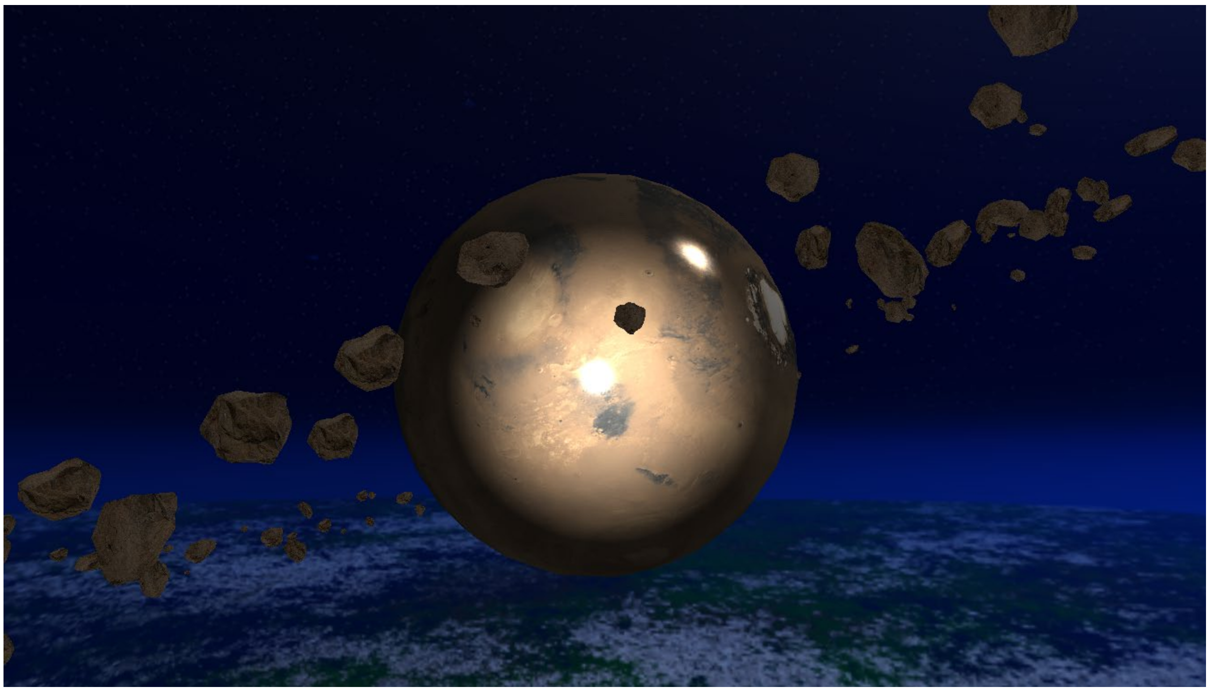


Figure 4: Saturn planet

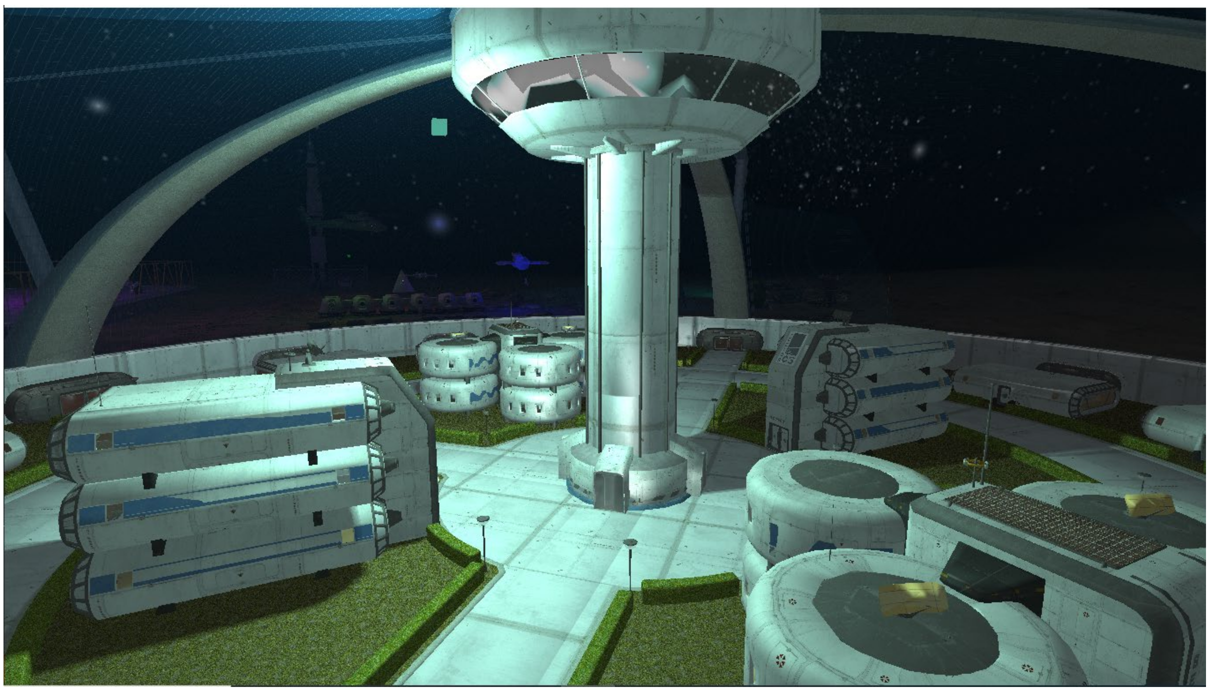


Figure 5: Inside the transparent dome

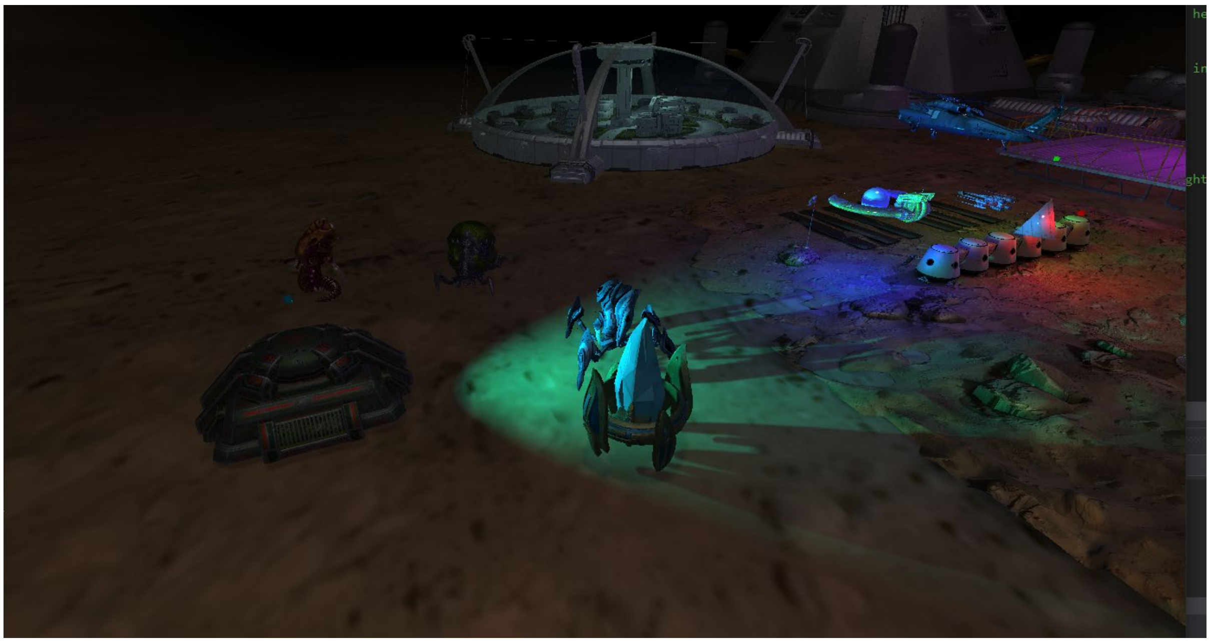


Figure 6: Bunker scene during the night

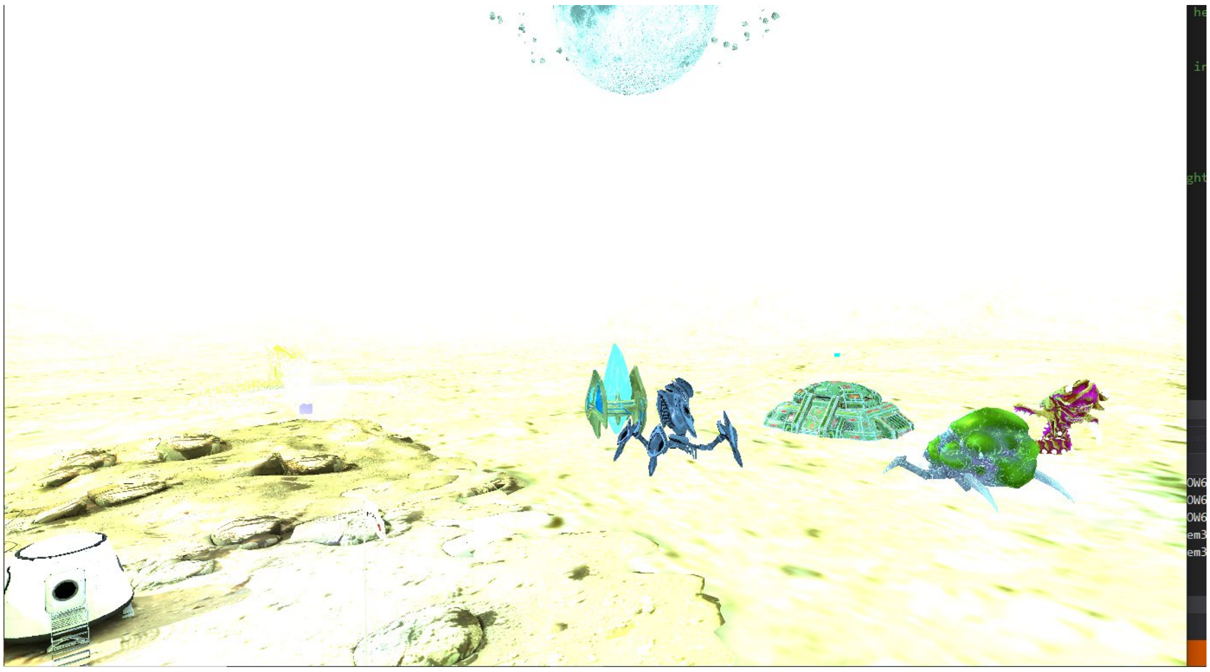


Figure 7: Screen during teleportation

B) Functionalities

As far as functionalities are concerned, I implemented the following:

- Directional, Point and Spotlights and shadows for all of them using omnidirectional shadow maps.
- Wireframe and point view for the objects

- Animation of helicopter and helicopter blades, animation of spotlight direction (bunker), controllable animation/movement of the rover, animation of the star trek base, moon, and its cloud.
- Animation independent of framerate using delta time.
- Instanced draws: instanced drawing techniques used for drawing the Saturn-like planet visualized above.
- Spotlight: a spotlight follows the camera at all times, and can supports toggling, it can be turned on or off.
- Transparency: dome and tempest have transparency attached to them
- Teleportation and sound: by pressing some buttons, one can quickly change the scene of visualization in an interactive manner. There is a 1-second timeout for this teleportation, during this time the screen gradually turns a bright white while sound is played
- Skybox: gives a sense of realism. The skybox automatically changes from the day scene to the night scene.
- Scene visualization: visualize the whole scene using animation and Bezier curves.

3. Implementation details

3.1) Functions and special algorithms

For generating shadows, an asteroid field and moving a rover in the scene

3.1.1) Possible solutions

For the shadow mapping we have multiple algorithms to choose from, including shadow volumes, shadow transformation, shadow Z-buffer and many more. While all of these have their advantages and disadvantages, I decided to go with the Z-buffer algorithm because it can be easily implemented using OpenGL and the programmable OpenGL graphics pipeline, and memory is of no concern to us.

For the rover movement, many things could have been done like a collision mesh and so on. Due to time constraints, I choose a simpler approach by hardcoding the limits of the rover movement, while still keeping the appearance of collision detection.

For the asteroid field, I could have done it blender using some special functions, but doing it in OpenGL does more justice to this project, so this is why I opted to create the asteroid field using object generation.

3.1.2) The motivation of the chosen approach

Multiple algorithms can be found in this project.

For starters, the rover movement: it is very similar to the one in the camera class. I chose to do something a bit different though, the tank movement seemed like a great fit for the rover since it rotates on a platform. Since the mouse is not available and decoupling it just to move the rover would have been excessive, I decided to do something like LEFT and RIGHT to rotate the rover, then take the sine and cosine in order to update the z and x of the rover front. The y direction remains constant. The rover move function just updates the position and checks the bounding box defined manually so the rover does not move from the platform.

```
void rover_rotate(bool* keys, GLfloat deltaTime, GLfloat moveSpeed)
{
    GLfloat velocity = moveSpeed * deltaTime;
    if (keys[GLFW_KEY_LEFT]) {
        roverAngle -= velocity;
    }
    if (keys[GLFW_KEY_RIGHT]) {
        roverAngle += velocity;
    }

    if (roverAngle > 360.0f) roverAngle -= 360.0f;
    if (roverAngle < 0.0f) roverAngle += 360.0f;

    roverFront.x = glm::sin(glm::radians(roverAngle));
    roverFront.z = -glm::cos(glm::radians(roverAngle));
}

void rover_move(bool* keys, GLfloat deltaTime, GLfloat moveSpeed)
{
    GLfloat velocity = moveSpeed * deltaTime;

    if (keys[GLFW_KEY_UP]) roverPosition += roverFront * velocity;
    if (keys[GLFW_KEY_DOWN]) roverPosition -= roverFront * velocity;

    // LEFT : (-36.479470, -1.100000, -0.823228) x
    // RIGHT : (-21.581999, -1.100000, 0.514411) x
    // FAR : (-30.000000, -1.100000, -19.074228) z
    // NEAR: (-27.056206, -1.100000, 10.491595)  z

    float max_x = -21.581999f;
    float min_x = -36.479470f;
    float max_z = 10.491595f;
    float min_z = -19.074228f;

    if (roverPosition.x > max_x) roverPosition.x = max_x;
    if (roverPosition.x < min_x) roverPosition.x = min_x;
    if (roverPosition.z > max_z) roverPosition.z = max_z;
    if (roverPosition.z < min_z) roverPosition.z = min_z;
}
```

Next up, we can see the algorithm that generates the asteroid field. I specify an amount of asteroids to generate and the translation vector on where the final position should be.

```
glm::mat4* generateRing(glm::vec3 translateVec, unsigned int amount)
{
    glm::mat4* modelMatrices = new glm::mat4[amount];
    srand(static_cast<unsigned int>(glfwGetTime())); // initialize random seed
    float radius = 30.0f;
    float offset = 5.0f;
    for (unsigned int i = 0; i < amount; i++)
    {
        glm::mat4 model = glm::mat4(1.0f);
        model = glm::translate(model, translateVec);
        // 1. translation: displace along circle with 'radius' in range [-offset, 
offset]
        float angle = (float)i / (float)amount * 360.0f;
        float displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
        float x = sin(angle) * radius + displacement;
        displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
        float y = displacement * 0.4f; // keep height of asteroid field smaller 
compared to width of x and z
        displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
        float z = cos(angle) * radius + displacement;
        model = glm::translate(model, glm::vec3(x, y, z));

        // 2. scale: Scale between 0.1 and 0.5f
        float scale = static_cast<float>((rand() % 50) / 100.0 + 0.1f);
        model = glm::scale(model, glm::vec3(scale));

        // 3. rotation: add random rotation around a (semi)randomly picked rotation 
axis vector
        float rotAngle = static_cast<float>((rand() % 360));
        model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

        // 4. now add to list of matrices
        modelMatrices[i] = model;
    }
    return modelMatrices;
}
```

Next, I want to talk about the shadow multisampling algorithm. The algorithm is pretty similar to the one we had in the laboratory, but this time a multisampling technique is used. A grid sampling disk of offsets is specified on the x, y, and z directions. The shadows are added up and averaged. The omni shadow map is a uniform that contains a sampler cube (that we access via index) and the far plane which is used for calculations.

```
// different samples for the sampler
vec3 gridSamplingDisk[20] = vec3[]
(
   vec3(1, 1,  1), vec3( 1, -1,  1), vec3(-1, -1,  1), vec3(-1, 1,  1),
   vec3(1, 1, -1), vec3( 1, -1, -1), vec3(-1, -1, -1), vec3(-1, 1, -1),
   vec3(1, 1,  0), vec3( 1, -1,  0), vec3(-1, -1,  0), vec3(-1, 1,  0),
   vec3(1, 0,  1), vec3(-1,  0,  1), vec3( 1,  0, -1), vec3(-1, 0, -1),
   vec3(0, 1,  1), vec3( 0, -1,  1), vec3( 0, -1, -1), vec3( 0, 1, -1)
);

float calcPointShadowFactor(PointLight light, int shadowIndex)
{
       // get the frag to light vector
       vec3 fragToLight = fragPos.xyz - light.position;
       float currentDepth = length(fragToLight);

       float shadow = 0.0;
       float bias   = 2.5;
       int samples  = 20;
       // modify disk radius based on the distance so we have less pixelation when we
get close
       float viewDistance = length(eyePosition - fragPos.xyz);
       float diskRadius = (1.0 + (viewDistance /
omniShadowMaps[shadowIndex].farPlane)) / 25.0;
       for(int i = 0; i < samples; ++i)
       {
              // get the closest point from the sampling disk
              float closestDepth = texture(omniShadowMaps[shadowIndex].shadowMap,
fragToLight + gridSamplingDisk[i] * diskRadius).r;
              // multiply by far plane to undo division  done by geometric shader
              closestDepth *= omniShadowMaps[shadowIndex].farPlane;
              // if the current depth is bigger than closest, we are in shadow
              if(currentDepth - bias > closestDepth)
                     shadow += 1.0;
       }
       // divide by the number of samples
       shadow /= float(samples);

       return shadow;
}
```

Many more algorithms have been used in this project, but if enough interest is shown, they shall be discussed face-to-face during the project presentation.

3.2) Graphics models

Graphics model is pretty simple, as I did not build a very complex importer. Perhaps in the future I could extend the importer to include more types of files and maybe even animations.

For all the objects in the scene, I used .obj models and .mtl pairs for materials. The models were processed using blender. A special mention about the graphics model needs to be given to the Blackhawk helicopter, which has the big propeller and the small propeller spinning. This was made possible by the use of blender split; the big mesh was split into multiple using intersection between objects. This could not have been possible without the help of my laboratory teacher, Constantin Nandra which helped me process the meshes in blender.

3.3) Data structures

Data structures used for this project have not been very complex.

The ring generation happens with an array of translation matrices that is accessed via index. The omnidirectional shadow map is made possible by a cube texture, similar to the one used for the skybox. The grid sampling disk used in the fragment shader is again, an array of vec3 values. The sampling for the directional shadow is a for loop on +- x and y. In the fragment shader, uniforms for the light and shadows are generally grouped into structs for easier processing.

3.4) Class hierarchy

As far as the class hierarchy is considered, the code is structured in the following classes:

- **Camera** : contains all the information regarding the camera and its functions for moving the camera in the scene
- **Shader:** contains all the information regarding the initialization and use of the shader.
- **Material:** models a material, has 2 fields: shininess (how concentrated the ray is) and specular intensity (how bright the specular is)
- **Mesh:** models a mesh, a basic representation with VAO/VBO/IBO
- **Model:** a 3D model, the ASSIMP library is used for loading objects.
- **Skybox:** models a skybox and the associated 3D texture
- **Window:** models the window and all the GLFW interactions with the window.
- **Texture:** models a simple texture, used by the model class
- **Light:** models a simple light, with attributes common to all of them (color, intensity, etc.)
- **DirectionalLight:** models a directional light with a view direction, inherits from light
- **PointLight:** models a point light that has a position, inherits from light
- **SpotLight:** models a spotlight that has both a position and a direction, inherits from the point light class
- **ShadowMap:** models a shadow map, has a width and a height.

- **OmniShadowMap:** models an omnidirectional shadow map, with 6 sides.
- **UtilityFunctions:** contains many utility functions used in the project

4. User manual

By default, the mouse cursor disappears when the application is started. The mouse is used to pan around in the world, and the **WASD keys** are used to navigate the world.

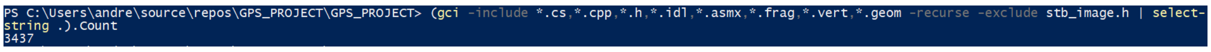
The user can interact with the scene of objects by using the following keys on the keyboard:

- **UP, DOWN, LEFT, RIGHT ARROWS** to move the rover on the platform. UP will cause the rover to go forward in the direction it is facing, while down will cause it to go backwards. LEFT and RIGHT are used to rotate the rover to the left and to the right respectively. The movement of the rover resembles that of tank, where it rotates the "tracks" with left/right and moves with forward and backwards.
- **F** – toggle the spotlight that follows the camera on or off. By default, the flashlight is turned off.
- **Q** – successively change between full, wireframe and point visualization of the scene.
- **R** – disable the shadows for the point lights and spotlights: visualize the shadows easier or increase performance.
- **T** – disable/enable all but the most necessary lights in the scene. Improves framerate significantly having fewer lights.
- **E** – switch to night scene. This enables/disables fog and changes the intensity of the directional light and changes the skybox to a different one.
- **1** – teleport to the main base.
- **2** – teleport to the moon.
- **3** – teleport to rotating star trek base.
- **4** – teleport to Saturn planet
- **5** – start scene animation

5. Conclusions

In conclusion, I would like to say that this is, by far, the hardest project I have ever done. A total of 50 hours has been clocked helping this project become a reality, and I am proud to say that the finished project closely resembles many modern aspects of OpenGL.

Running a command that generates LoC, we get the following result, the project has a total of 3437 LoC if we do not include the STBI image.



```
PS C:\Users\andre\source\repos\GPS_PROJECT\GPS_PROJECT> (gci -include *.cs,*.cpp,*.h,*.idl,*.asmx,*.frag,*.vert,*.geom -recurse -exclude stb_image.h | select-
string .).Count
3437
```

Still, much more work can be done. The quality can be improved and also many performance improvements can be made:

- Proper collision detection for the rover
- Rain
- MSAA only when the camera is close to the shadow. In general, using a dynamic shadow calculation algorithm would result in large performance improvements
- Depth sorting algorithm for transparency
- Tessellation
- Ray tracing/ray casting and PBR, global illumination model
- Skeletal animation
- Area lights

6. Bibliography

- https://moodle.cs.utcluj.ro/course/view.php?id=526 -> Moodle course site and all of the laboratories
- https://learnopengl.com/ -> website with loads of tutorials and resources for building a modern OpenGL application. A book format is also available for free
- https://www.youtube.com/playlist?list=PLysLvOneEETPlOI_PI4mJnocqIpr2cSHS -> Michael Greico OpenGL tutorials, very informative tutorials with lots of in-depth knowledge
- https://www.youtube.com/@OGLDEV/playlists -> OGLDEV has many interesting and useful tutorials for learning OpenGL, for both beginners and advanced users.

Websites for model loading such as:

- Turbosquid: https://www.turbosquid.com/Search/3D-Models/free
- Sketchfab: https://sketchfab.com/search?features=downloadable&type=models
- Free3D: https://free3d.com/
- cgTrader: https://www.cgtrader.com/
- CgPersia: https://cgpersia.com/category/download/cg/3d/models (The website hosts links to many free 3D models from various companies that are usually only available with payment. No copyright infringement intended, and the website only hosts links, actual content is hosted on different websites)