Graphics Processing Project Documentation

Student: Meze Răzvan Gabriel, Group 30232

Lecturer: Prof. Dr. Eng. Dorian Gorgan

Practical works professor: Sl. Dr. Eng. Cosmin Nandra

Table of contents

1. Presentation of the Theme

This project explores the creation and rendering of a dynamic restaurant scene using OpenGL, focusing on achieving realistic lighting and shadow effects to enhance the visual experience. The scene is designed to simulate an ambient restaurant environment, complete with moving lights that cast omnidirectional shadows, creating a lifelike and immersive dining atmosphere.

This project's main objective is to demonstrate advanced OpenGL techniques, particularly in lighting and shadow rendering. By implementing moving lights and omnidirectional shadows, we aim to simulate the nuanced lighting found in real-world restaurant settings, where multiple light sources and their interactions with the environment play a crucial role in creating ambiance.

Key Features

Dynamic Lighting: The scene features moving lights that simulate the effect of overhead fixtures or candles being carried through the restaurant, offering a dynamic lighting environment that changes over time.

Omnidirectional Shadows: Utilizing advanced shadow mapping techniques, the project achieves realistic omnidirectional shadows, ensuring that objects in the scene cast shadows in all directions, adding depth and realism to the rendering.

Detailed Restaurant Environment: The scene is meticulously crafted to include typical restaurant elements such as tables, chairs, dinnerware, and decorative items. Each object is designed to interact naturally with the scene's lighting to produce realistic visual effects.

Interactive Camera Controls: Users can navigate through the scene using intuitive camera controls, allowing for exploration of the restaurant from multiple perspectives and enhancing the immersive experience.

2. Scenario

2.1. Description of the Scene and Objects

1. Seating Area:

Chairs and Tables: The dining area features wooden chairs with green cushioned seats and matching wooden tables, arranged neatly to accommodate multiple guests. The tables are set with blue utensil holders, which add a pop of color to the wooden tones of the furniture.

Curtains: Semi-transparent curtains in shades of red and purple hang beside the windows, diffusing the light and softening the interior.

Flooring: The flooring is composed of polished wooden planks, providing a classic look that resonates with the overall rustic theme.

2. Bar Counter:

Counter: A bar counter is present, showcasing an array of dishware on the shelves behind it. The counter itself appears to be a focal point for casual dining or drinks, with several bar stools lined up in front.

Utensils and Decor: On the counter, black utensil holders are visible, along with what appears to be a menu stand.

3. Ambient Elements:

Lighting: The lighting within the restaurant comes from both the natural daylight and the subtle artificial lights, which are not directly visible in the images. The reflections on the windows suggest the presence of ceiling lights or hanging fixtures that contribute to the shadows and highlights within the space.

Decorative Elements: The walls are adorned with wooden beams and decorative elements that give depth and character to the space. Some walls have a blue tile accent, providing a modern touch to the otherwise rustic decor.

4. Privacy Dividers:

Dividers: Wooden dividers with glass inserts offer a sense of privacy for the diners, subtly sectioning off areas of the restaurant without obstructing the open feel of the space.

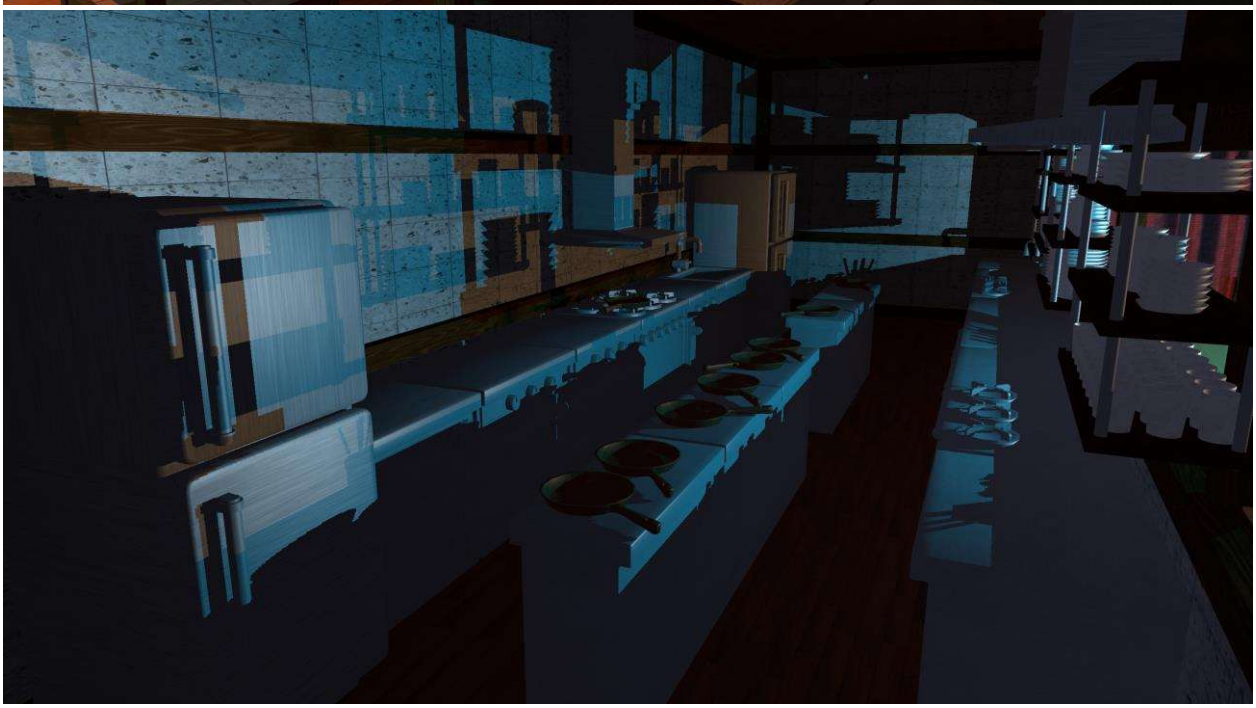
5. Kitchen Area:

Counters: The kitchen is equipped with long, dark-colored countertops which are reflective, suggesting a polished stone or metal surface. These counters are used for food preparation and display.

Cooking Stations: There are cooking stations with visible pans, possibly indicating a live cooking area or an open kitchen concept where guests can view meal preparation.

Appliances and Cabinets: Various kitchen appliances, potentially refrigerators or storage units, are present with a modern design, blending into the sleek look of the kitchen.

Shelving: Above the counters, there are shelves stocked with plates and bowls, emphasizing the readiness for service. The blue tiles in the background add a vibrant contrast to the dark counters and shelves.



2.2. Functionalities

- Navigational Controls: Users can freely navigate through the scene using keyboard controls. The 'W', 'A', 'S', 'D' keys allow for forward, leftward, backward, and rightward movement, respectively, enabling exploration of the environment from different angles and perspectives.
- Presentation Animation: An automated presentation mode can be activated with a keypress [“0”]. In this mode, the camera follows a predefined square path around the scene. This showcases the restaurant’s layout and design, smoothly transitioning from one point of interest to another to give viewers a comprehensive view of the interior.
- Blinn-Phong Lighting Model:
- The Blinn-Phong lighting model is used to render the scene, contributing to the realistic appearance of materials and surfaces. It simulates the way light interacts with objects, including specular highlights that give a sense of glossiness and depth to the textures.
- Fog Effect: A fog effect is present in the scene, adding atmospheric depth and a sense of distance. The fog density can create a mystique or morning ambiance, impacting the visibility of objects based on their distance from the camera.
- Moving Point Light Sources: There are dynamic point light sources that move within the scene, which could simulate the effect of passing people with lanterns or the movement of light from a television screen. These moving lights contribute to the lively atmosphere of the restaurant and demonstrate the real-time rendering capabilities of the lighting system.
- The scene also features omnidirectional shadows, which are essential for enhancing the realism of the lighting. These shadows are cast in all directions from the light sources, mimicking how shadows behave in the real world. This is especially noticeable with the moving point light sources, as the shadows change dynamically with the movement of the lights.

3. Implementation Details

3.1. Functions and Algorithms

The focus of the algorithm development was on realistic lighting, omnidirectional shadows and the fog effect. Also, the project was designed to be modular and thus easily scalable.

3.1.1 Possible solutions

Possible rendering solutions:

1. Rasterization:

Advantages:

- High performance: Optimized for real-time applications like video games.
- Well-supported: Graphics hardware is designed to accelerate rasterization.
- Predictable: Offers consistent frame rates crucial for interactive applications.

Disadvantages:

- Limited realism: Struggles with complex lighting and global illumination.
- Overdraw: Can be inefficient when multiple layers overlap, as hidden surfaces are processed.
- Fixed pipeline constraints: Older rasterization methods offered less flexibility for advanced effects.

2. Ray Tracing:

Advantages:

- Realistic images: Simulates many optical effects like reflections, refractions, and shadows accurately.
- Simplified asset creation: Materials and lights behave predictably, reducing the need for hacks or adjustments.
- Dynamic scenes: Handles changes in geometry and lighting without pre-computation.

Disadvantages:

- Performance-intensive: Requires significant computational power, traditionally unsuitable for real-time.
- Noise: Can produce grainy images that require complex denoising algorithms.

- Hardware dependency: Optimal performance relies on specialized hardware support.
-

3. Deferred Rendering:

Advantages:

- Efficient lighting: Calculates lighting only for visible fragments, optimizing scenes with many lights.
- Scalable: Adding more lights doesn't significantly impact performance.
- Decoupled geometry and lighting: Simplifies the rendering pipeline and shader management.

Disadvantages:

- Memory-intensive: Requires multiple render targets to store intermediate data.
- Anti-aliasing complexity: Standard MSAA is not directly compatible, requiring alternative solutions.
- Transparency handling: Requires additional passes or techniques to render transparent objects correctly.

Each technique suits different scenarios. Rasterization is the go-to for real-time applications where performance is key. Ray tracing is favored in scenarios where visual fidelity is paramount, such as cinematic content creation or architectural visualization. Deferred rendering finds its place in scenes with complex lighting, where it offers a good balance between performance and visual quality.

Possible lighting models:

1. Phong Lighting Model:

Components:

- Ambient: Simulates indirect light scattered in the environment.
- Diffuse: Models the direct light absorbed and scattered uniformly by surfaces.
- Specular: Represents the bright spots of light that appear on shiny surfaces.

Advantages:

- Simple and fast: Suitable for real-time applications.
- Specular highlights: Provides control over the shininess of surfaces.

Disadvantages:

- Unrealistic: Fails to capture complex light-material interactions.
- View-dependent: Specular highlights can change dramatically with the camera angle.

2. Blinn-Phong Lighting Model:

Components:

- Similar to Phong, with ambient, diffuse, and specular components.

Improvements over Phong:

- Uses the halfway vector between the view direction and light direction, which can be more performance-friendly for specular calculations.

Advantages:

- More realistic specular highlights compared to Phong, especially at glancing angles.
- Still relatively simple and computationally inexpensive.

Disadvantages:

- Like Phong, it does not account for real-world material properties or complex lighting scenarios.
- Highlights can still appear artificial compared to more advanced models.

3. Physically Based Rendering (PBR):

Components:

- Base Color or Albedo: Reflects the intrinsic color of the material without direct lighting.
- Metallic-Roughness/Specular-Glossiness: Two common workflows that define material properties and how they interact with light.
- Normal Mapping: Allows detailed surface definitions without high polygon counts.
- Environment Lighting: Often uses image-based lighting (IBL) for realistic reflections and ambient light.

Advantages:

- Realism: Simulates how light interacts with surfaces in the real world.
- Consistency: Materials look correct under various lighting conditions.

- Energy conservation: Ensures that the amount of light reflected doesn't exceed the light that hits a surface.

Disadvantages:

- Complexity: Requires more computational resources and detailed material definitions.
- Steeper learning curve: Artists must understand material science concepts to create realistic textures.

Realism Impact:

- Phong and Blinn-Phong: These are older models that provide reasonable approximations of how light interacts with surfaces. They are still used in many applications but lack the realism provided by more complex models. Specular highlights can appear artificial, and the materials may not respond to lighting changes realistically.
- PBR: This model offers a significant step towards photorealism, ensuring that materials behave consistently under different lighting conditions. The inclusion of real-world physical properties in the calculations results in more believable scenes. PBR has become the standard in many industries, from game development to film production.

Ultimately, the choice of lighting model affects the visual quality and performance of the scene. Phong and Blinn-Phong are suitable for applications where performance is critical, and the highest degree of realism is not necessary. PBR is the choice for state-of-the-art rendering where image quality is paramount, and the computational budget allows for it.

Possible Shadow generation solutions:

1. Standard Shadow Mapping:

Mechanism:

- Uses a depth map captured from the light's viewpoint to determine whether a fragment is in shadow or not.

Advantages:

- Straightforward to implement.
- Relatively fast and suitable for scenes where sharp shadows are acceptable.
- **Disadvantages:**

- Can produce aliasing artifacts known as "shadow acne" due to depth map resolution limits.
- Hard edges: The shadows have a uniform sharpness which may not look realistic.

2. Percentage Closer Filtering (PCF):

- Mechanism: Softens the edges of shadows by averaging the depth values around the shadow-map lookup coordinate.

Advantages:

- Provides softer, more realistic shadow edges that mimic natural light behavior.
- Reduces shadow acne by smoothing the depth comparison results.
-

Disadvantages:

- More computationally expensive than standard shadow mapping due to multiple texture samples.
- May still display artifacts at lower resolutions or with high-contrast lighting.

3. Variance Shadow Maps (VSM):

- Mechanism: Stores the depth's mean and squared mean in the shadow map to compute the variance, allowing for soft shadow edges with a probabilistic approach.

Advantages:

- Can produce very soft shadows and can handle penumbrae well.
- Less susceptible to aliasing and shadow acne compared to standard shadow mapping.

Disadvantages:

- Can suffer from light bleeding, where shadows from one object incorrectly affect other nearby objects.
- Requires more memory and GPU resources as it stores additional moments (mean and squared mean) for each pixel.

Performance Impact:

- Standard Shadow Mapping: Offers the best performance but at the cost of visual quality, especially in scenes with detailed lighting.

- PCF: Strikes a balance between performance and visual quality. It is more demanding than standard shadow mapping but provides visually pleasing results without significant performance degradation on modern hardware.
- VSM: While it can deliver high-quality soft shadows, it is the most demanding in terms of performance and resources. It is suitable for high-end systems where shadow quality is a priority.

Visual Artifacts:

- Standard Shadow Mapping: Prone to aliasing and "shadow acne" artifacts due to the discrete nature of depth map sampling.
- PCF: Can still show some aliasing, especially if the number of samples is too low, but generally reduces hard edges and artifacts.
- VSM: Can introduce light bleeding, a challenging artifact to mitigate without additional techniques like depth clamping or using higher moments in the shadow map.

In conclusion, the selection of a shadow mapping technique often depends on the specific needs of the project. If performance is crucial and the scene can tolerate some aliasing, standard shadow mapping may suffice. PCF is an excellent middle ground, offering improved visual results with acceptable performance costs. VSM provides the highest quality shadows but should be used judiciously where performance budgets allow.

3.1.2 Reasoning behind the chosen approach

1. Rendering - Rasterization:

Chosen Because:

- Efficiency: Rasterization is well-optimized for real-time applications, providing high frame rates essential for interactive scenes.
- Hardware Support: It is the standard rendering technique supported by all graphics hardware, ensuring compatibility and performance.
- Predictable Performance: Unlike ray tracing, which can have variable performance based on scene complexity, rasterization offers stable and predictable frame rates.

Not Chosen Alternatives:

- Ray Tracing: While it provides more realistic lighting and reflections, it is computationally intensive and may not perform well in real-time without high-end hardware.
- Deferred Rendering: While it efficiently handles multiple light sources, it can be overkill for scenes with few lights and introduces complexity with transparency and anti-aliasing.

2. Lighting - Blinn-Phong:

Chosen Because:

- Simplicity: Blinn-Phong is simpler to implement than physically-based models while still offering visually appealing results.
- Performance: It is computationally less expensive than PBR, making it suitable for a wide range of hardware.
- Control: Provides artistic control over the specular highlights, allowing for a fine-tuned visual aesthetic that can be adjusted to the artist's vision.

Not Chosen Alternatives:

- Phong: Blinn-Phong is often preferred over Phong due to its more realistic simulation of specular highlights.
- PBR: While PBR offers more realism, it requires a more complex setup and is computationally more demanding, which may not be necessary for the intended visual style of the project.

3. Shadows - Shadow Mapping for Moving Lights:

Chosen Because:

- Dynamic Scenes: This technique is flexible for scenes with moving lights, as it can update shadows in real-time to reflect changes in lighting conditions.
- Versatility: Works well with both point lights and directional lights, providing consistent shadowing across different light types.
- Realism: It allows for the creation of realistic shadows that enhance the three-dimensional quality of the scene.

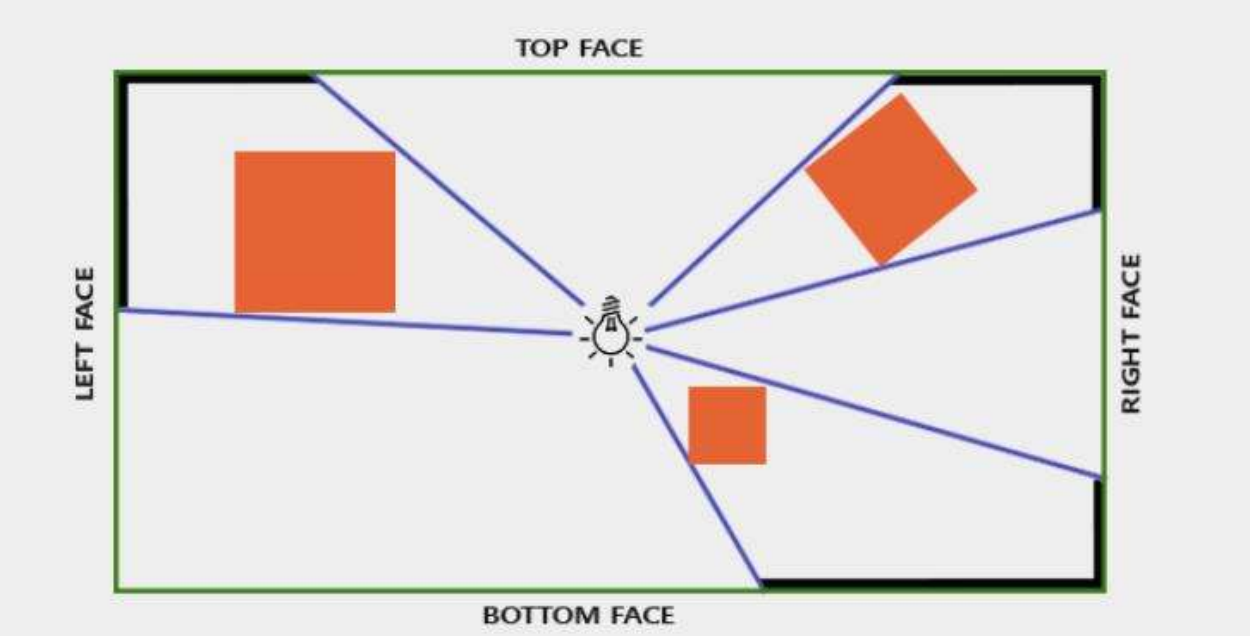
Not Chosen Alternatives:

- Static Shadow Maps: Not suitable for dynamic lights as they do not update in real-time with changes in the light's position or properties.
- VSM or PCF: While these techniques offer improvements in softness and realism, they may require additional computational resources that could be saved for other aspects of the scene.

Actual implementation

3.1.2.1 Shadows

The approach adopted for omnidirectional shadow mapping bears a strong resemblance to that of directional shadow mapping. The process involves creating a depth map from the perspective of the light source(s), then sampling this depth map using the position of the current fragment. Each fragment's depth is compared to the corresponding stored depth in the map to determine if it is occluded or in shadow. The key distinction between the two methods lies in the type of depth map utilized; while directional shadow mapping typically uses a 2D depth map, omnidirectional shadow mapping requires a depth map that encompasses all possible directions from the light source, implemented as a cube map.



First we initialize the frame buffer objects:

```
void helperConstructorInitCubeFBO(int index) {
    ///GLuint depthCubeMapFBO[NR_OF_POINT_LIGHTS] = {};
    ///GLuint solidsDepthCubeMapTexture[NR_OF_POINT_LIGHTS];
    glGenFramebuffers(1, &depthCubeMapFBO[index]);

    glGenTextures(1, &solidsDepthCubeMapTexture[index]);
    glBindTexture(GL_TEXTURE_CUBE_MAP, solidsDepthCubeMapTexture[index]);
    for (unsigned int i = 0; i < 6; ++i)
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_DEPTH_COMPONENT,
                     SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    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);

    glBindFramebuffer(GL_FRAMEBUFFER, depthCubeMapFBO[index]);
    glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, solidsDepthCubeMapTexture[index], 0);
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);///idk
}
```

glGenFramebuffers(1, &depthCubeMapFBO[index]);

- Generates one framebuffer object and stores its ID in the depthCubeMapFBO array at the given index. A framebuffer object is used as a destination for rendering operations.

glGenTextures(1, &solidsDepthCubeMapTexture[index]);

- Generates one texture object and stores its ID in the solidsDepthCubeMapTexture array at the given index. This texture will be used to store the depth information for the shadow mapping.

glBindTexture(GL_TEXTURE_CUBE_MAP, solidsDepthCubeMapTexture[index]);

- Binds the newly created texture as a cube map so that it can be configured and filled with data. A cube map texture consists of six 2D textures that represent the faces of a cube.

The for-loop with glTexImage2D calls:

- Allocates memory for each of the six faces of the cube map texture, setting each to have a depth component format. This allows the texture to store depth information from the perspective of the light source looking in six different directions.

glTexParameteri calls:

- Set texture parameters for the cube map. This includes setting the magnification and minification filters to GL_NEAREST (which means no interpolation is done

when looking up values), and clamping the texture coordinates to the edges to avoid wrapping.

glBindFramebuffer(GL_FRAMEBUFFER, depthCubeMapFBO[index]);

- Binds the framebuffer object so that it can be configured.

glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, solidsDepthCubeMapTexture[index], 0);

- Attaches the cube map texture as the depth attachment of the framebuffer object. When rendering to this framebuffer, depth information will be written to this texture.

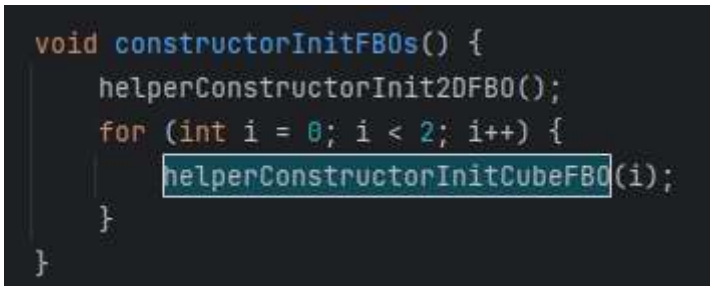
glDrawBuffer(GL_NONE); and glReadBuffer(GL_NONE);

- These calls specify that no color buffers are to be drawn into or read from, which is typical for depth-only rendering.

glBindFramebuffer(GL_FRAMEBUFFER, 0);

- Unbinds the framebuffer, reverting to the default framebuffer. This is often done at the end of the setup to avoid accidentally rendering to the wrong framebuffer.

The helper function is part of another helper function which is used inside a "TransformsAndLighting" object constructor, which is a comprehensive constructor that initializes all objects, initial parameters and sends uniforms to the shaders.



```
TransformsAndLighting::TransformsAndLighting(const ModelTransforms (&solidModelTransforms)[NR_OF_SOLID_OBJECTS],
                                             const glm::mat4 &projection, const gps::Camera &camera,
                                             const SceneLighting &sceneLighting,
                                             const gps::Shader (&solidsShader),
                                             const gps::Shader (&pointLightsShader),
                                             const gps::Shader (&depth2DMapShader),
                                             const gps::Shader (&depthCubeMapShader),
                                             const Model3D (&solidObjectMeshMods)[NR_OF_SOLID_OBJECTS],
                                             const Model3D (&lightObjectMeshMod),
                                             const int &retina_width, const int &retina_height)
        :
        projection(projection), camera(camera), sceneLighting(sceneLighting),
        solidsShader(solidsShader), pointLightsShader(pointLightsShader),
        depth2DMapShader(depth2DMapShader), depthCubeMapShader(depthCubeMapShader),
        retina_width(retina_width), retina_height(retina_height), lightObjectMeshMod(lightObjectMeshMod) {
    ///calculate view
    this->view = (this->camera).getViewMatrix();

    for (int i = 0; i < NR_OF_SOLID_OBJECTS; i++) {
        this->solidObjectMeshMods[i] = solidObjectMeshMods[i];
    }

    constructorUpdateTransformsSolidsAndLights(solidModelTransforms);
    constructorComputeSolidsLightSpaceDirTrMatrix();
    for (int i = 0; i < NR_OF_POINT_LIGHTS; i++) {
        constructorComputeDepthCubeLightSpaceCubeTrMatrices( pointLightIndex: i);
    }
    constructorInitFBOs();

    ///get the locations of all uniforms for all shaders (other than the ones that we have already got in
    /// the constructors for lights)
    constructorGetUniformLocationsForSolidsVertexShader();
    ///uniform locations for lights are obtained from light constructors
    ///uniform locations for the depth textures and far plane must be obtained from here
    constructorGetUniformLocationsForSolidsFragmentShader();
    constructorGetUniformLocationsForPointLightsVertexShader();///no uniforms in fragment shader
    constructorGetUniformLocationsForDepth2DVertexShader();///no uniforms in fragment shader
    constructorGetUniformLocationsForDepthCubeVertexShader();
    constructorGetUniformLocationsForDepthCubeGeometryShader();
    constructorGetUniformLocationsForDepthCubeFragmentShader();

    ///sendToShaders
    constructorSendUniformsToSolidsVertexShader();
    constructorSendUniformsToSolidsFragmentShader();
    constructorSendUniformsToPointLightsVertexShader();
    constructorSendUniformsToDepth2DVertexShader();
    constructorSendUniformsToDepthCubeGeometryShader();
    constructorSendUniformsToDepthCubeFragmentShader();
}
```

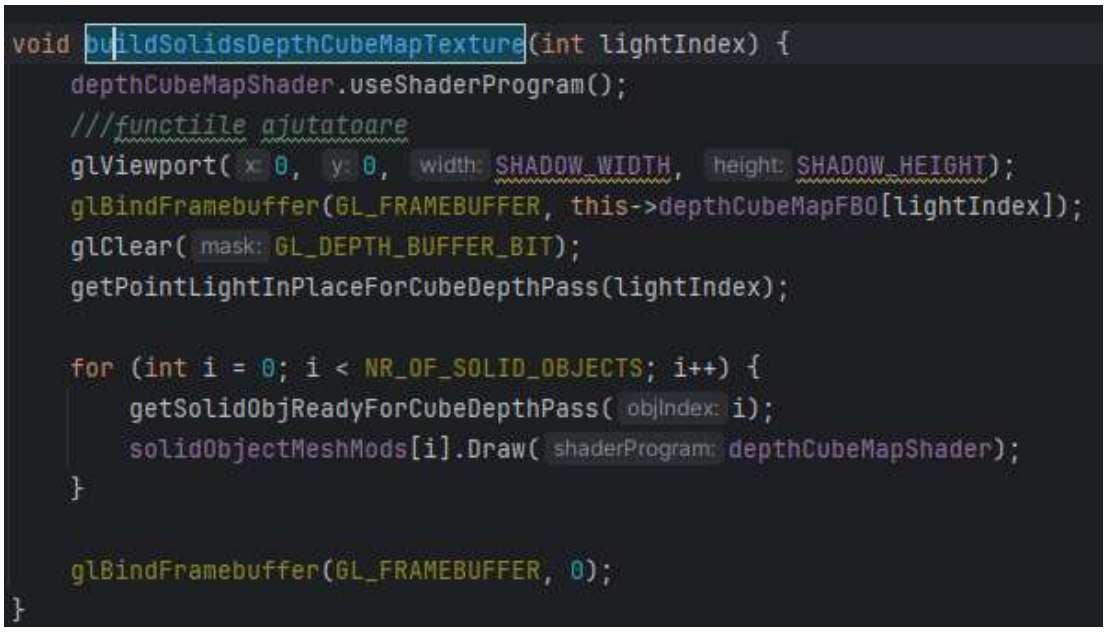
We have a **renderScene()** function in which we call buildSolidsDepthCubeMapTexture(int lightIndex) which renders all objects from the perspective of the point light with index lightIndex.

```
void renderScene(RenderMode renderMode) {

    switch (renderMode) {
        case RenderMode::SOLID:
            glPolygonMode( face: GL_FRONT_AND_BACK,  mode: GL_FILL);
            break;
        case RenderMode::WIREFRAME:
            glPolygonMode( face: GL_FRONT_AND_BACK,  mode: GL_LINE);
            break;
        case RenderMode::SMOOTH:
            glPolygonMode( face: GL_FRONT_AND_BACK,  mode: GL_POINT);
            break;
    }


    buildSolidsDepth2DMapTexture();
    for (int i = 0; i < NR_OF_POINT_LIGHTS; i++) {
        buildSolidsDepthCubeMapTexture( lightIndex: i);
    }
    drawSolidObjects();
    drawLights();
}
```

buildSolidsDepthCubeMapTexture:



```
void buildSolidsDepthCubeMapTexture(int lightIndex) {
    depthCubeMapShader.useShaderProgram();
    ///functiile ajutatoare
    glViewport( x: 0,  y: 0,  width: SHADOW_WIDTH,  height: SHADOW_HEIGHT);
    glBindFramebuffer(GL_FRAMEBUFFER, this->depthCubeMapFBO[lightIndex]);
    glClear( mask: GL_DEPTH_BUFFER_BIT);
    getPointLightInPlaceForCubeDepthPass(lightIndex);

    for (int i = 0; i < NR_OF_SOLID_OBJECTS; i++) {
        getSolidObjReadyForCubeDepthPass( objIndex: i);
        solidObjectMeshMods[i].Draw( shaderProgram: depthCubeMapShader);
    }

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

- It activates the shader program that is used for rendering the depth map (depthCubeMapShader).
- The viewport is set to the size of the shadow map texture, which is defined by SHADOW_WIDTH and SHADOW_HEIGHT. This determines the resolution of the depth map.
- The framebuffer object (depthCubeMapFBO) corresponding to the point light with the given lightIndex is bound as the current framebuffer. This means that subsequent rendering commands will render into this framebuffer rather than the default framebuffer used for rendering to the screen.
- The depth buffer of the currently bound framebuffer is cleared to ensure that it doesn't contain any previous data.
- The **getPointLightInPlaceForCubeDepthPass** function is called with the lightIndex, which sets up the shader uniforms required to render from the point light's perspective.
- A loop iterates over all solid objects in the scene. For each object:
- The **getSolidObjReadyForCubeDepthPass** function is called with the object's index, which prepares the object for rendering by setting its model matrix and other relevant shader uniforms.
- The object is rendered with the depthCubeMapShader, which writes the depth information of each fragment into the depth map texture.

- After all objects have been rendered to the depth map, the framebuffer is unbound by binding the default framebuffer (0). This switches back the rendering target to the screen.

GetPointLightInPlaceForCubeDepthPass:

```
void TransformsAndLighting::getPointLightInPlaceForCubeDepthPass(int lightIndex) {
    depthCubeMapShader.useShaderProgram();
    constructorComputeDepthCubeLightSpaceCubeTrMatrices( pointLightIndex: lightIndex);
    glUniform3fv(depthCubeLightPosLoc, 1, glm::value_ptr( v: sceneLighting.getPointLights()[lightIndex].getPosition()));
    for (int i = 0; i < 6; i++) {
        glUniformMatrix4fv(this->depthCubeLightSpaceCubeTrMatricesLoc[i], 1, GL_FALSE,
                           glm::value_ptr( &: this->depthCubeLightSpaceCubeTrMatrices[lightIndex][i]));
    }
    solidsShader.useShaderProgram();
    glUniform3fv(sceneLighting.getPointLights()[lightIndex].getPositionLoc(), 1,
                 glm::value_ptr( v: sceneLighting.getPointLights()[lightIndex].getPosition()));
}
```

The function getPointLightInPlaceForCubeDepthPass prepares the environment for rendering the depth map from a point light's perspective in OpenGL. This process is part of omnidirectional shadow mapping, where a shadow map is created for each of the six faces of a cube surrounding the light source. This function sets up the uniforms for the shader that will render the depth map. Here's a step-by-step explanation of what this function does:

- The shader program depthCubeMapShader is activated using useShaderProgram(). This shader is specialized for rendering the depth information from the light's perspective to the cubemap textures.
- **constructorComputeDepthCubeLightSpaceCubeTrMatrices**(lightIndex) is presumably a function call (although it's unusual to have a constructor in the name as constructors don't usually return values) that computes the transformation matrices for the light's view projecting onto each face of the cube. This is used to render the scene from the light's perspective to the cubemap.
- The position of the point light with the given lightIndex is uploaded to the shader as a uniform variable using glUniform3fv. This uniform (depthCubeLightPosLoc) is used in the shader to determine the light's position relative to the geometry in the scene.
- A loop iterates over the indices 0 to 5, corresponding to the six faces of the cubemap (positive X, negative X, positive Y, negative Y, positive Z, negative Z). For each face:

- The corresponding light space transformation matrix is uploaded to the shader using glUniformMatrix4fv. These matrices transform the scene's vertices into the space of the cubemap's current face being rendered.
- The shader program for the solid objects (solidsShader) is activated, which seems to be for rendering the actual scene after the depth maps have been created.
- The light position for the solid objects shader is set using another glUniform3fv, which uses the location obtained presumably in the point light's constructor (getPositionLoc()). This is likely used for the actual lighting calculations in the scene rendering phase.
- In summary, getPointLightInPlaceForCubeDepthPass prepares the GPU to render the depth cubemap from a point light's perspective by setting the appropriate uniforms for the shader program. These uniforms include the light's position and the transformation matrices for rendering to each face of the depth cubemap.

ConstructorComputeDepthCubeLightSpaceCubeTrMatrices:

```
void constructorComputeDepthCubeLightSpaceCubeTrMatrices(int pointLightIndex) {
    float aspect = (float) this->SHADOW_WIDTH / (float) this->SHADOW_HEIGHT;
    glm::mat4 shadowProj = glm::perspective( fovy: glm::radians( degrees: 90.0f), aspect, zNear: this->pointNearPlane,
                                             zFar: this->solidsFarPlane);
    glm::vec3 lightPos = this->sceneLighting.getPointLights()[pointLightIndex].getPosition();
    depthCubeLightSpaceCubeTrMatrices[pointLightIndex][0] =
            shadowProj * glm::lookAt( eye: lightPos,  center: lightPos + glm::vec3( x: 1.0,  y: 0.0,  z: 0.0),  up: glm::vec3( x: 0.0,  y: -1.0,  z: 0.0));
    depthCubeLightSpaceCubeTrMatrices[pointLightIndex][1] =
            shadowProj * glm::lookAt( eye: lightPos,  center: lightPos + glm::vec3( x: -1.0,  y: 0.0,  z: 0.0),  up: glm::vec3( x: 0.0,  y: -1.0,  z: 0.0));
    depthCubeLightSpaceCubeTrMatrices[pointLightIndex][2] =
            shadowProj * glm::lookAt( eye: lightPos,  center: lightPos + glm::vec3( x: 0.0,  y: 1.0,  z: 0.0),  up: glm::vec3( x: 0.0,  y: 0.0,  z: 1.0));
    depthCubeLightSpaceCubeTrMatrices[pointLightIndex][3] =
            shadowProj * glm::lookAt( eye: lightPos,  center: lightPos + glm::vec3( x: 0.0,  y: -1.0,  z: 0.0),  up: glm::vec3( x: 0.0,  y: 0.0,  z: -1.0));
    depthCubeLightSpaceCubeTrMatrices[pointLightIndex][4] =
            shadowProj * glm::lookAt( eye: lightPos,  center: lightPos + glm::vec3( x: 0.0,  y: 0.0,  z: 1.0),  up: glm::vec3( x: 0.0,  y: -1.0,  z: 0.0));
    depthCubeLightSpaceCubeTrMatrices[pointLightIndex][5] =
            shadowProj * glm::lookAt( eye: lightPos,  center: lightPos + glm::vec3( x: 0.0,  y: 0.0,  z: -1.0),  up: glm::vec3( x: 0.0,  y: -1.0,  z: 0.0));
}
```

The constructorComputeDepthCubeLightSpaceCubeTrMatrices function calculates the transformation matrices needed to render the scene from the point of view of the light source to create a depth cubemap for omnidirectional shadow mapping. Each face of the cubemap corresponds to a direction the light is "looking" at, and thus, requires its own view matrix. Here's a breakdown of the code:

- aspect: The aspect ratio is calculated based on the shadow map's width and height, which will be used for the perspective projection matrix.
- shadowProj: This is a perspective projection matrix with a field of view of 90 degrees (which corresponds to the field of view for each face of the cubemap), the

previously calculated aspect ratio, and the near and far planes defined by this->pointNearPlane and this->solidsFarPlane.

- lightPos: The position of the current point light, obtained from the sceneLighting object using the pointLightIndex.
- For each face of the cubemap (six faces in total), a view matrix is calculated using glm::lookAt, which creates a view matrix for a camera looking from the light position lightPos towards a certain direction:
    - For the positive X direction: The camera looks along the positive X-axis, with down being in the negative Y direction.
    - For the negative X direction: The camera looks along the negative X-axis, with down being in the negative Y direction.
    - For the positive Y direction: The camera looks along the positive Y-axis, with the forward vector pointing along the positive Z-axis.
    - For the negative Y direction: The camera looks along the negative Y-axis, with the forward vector pointing along the negative Z-axis.
    - For the positive Z direction: The camera looks along the positive Z-axis, with down being in the negative Y direction.
    - For the negative Z direction: The camera looks along the negative Z-axis, with down being in the negative Y direction.
- Each calculated view matrix is multiplied by the perspective projection matrix shadowProj and stored in the depthCubeLightSpaceCubeTrMatrices array. This array has an entry for each face of the cubemap for each light source, allowing the scene to be rendered from the light's perspective in all six directions.
- The resulting matrices are used during the rendering process where the scene is rendered to the depth cubemap from the point light's perspective. These matrices transform the scene's vertices into the space where the point light is at the center, allowing for the depth values to be stored correctly for each direction, which will later be used to determine if a fragment is in shadow or not.

GetSolidObjReadyForCubeDepthPass:

```
void TransformsAndLighting::getSolidObjReadyForCubeDepthPass(int objIndex) {
    depthCubeMapShader.useShaderProgram();
    glUniformMatrix4fv(depthCubeModelLoc, 1, GL_FALSE, glm::value_ptr( & solidsModels[objIndex]));
}
```

Then, **drawSolidObjects()** is called to draw the objects considering the lighting model, shadows and fog defined in the shaders.

drawSolidObjects:

```
void drawSolidObjects() {
    glViewport( x: 0,  y: 0,  width: retina_width,  height: retina_height);
    glClear( mask: GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    solidsShader.useShaderProgram();

    glActiveTexture(GL_TEXTURE0);
    glBindTexture( target: GL_TEXTURE_2D,  texture: solidsDepth2DMapTexture);
    glUniform1i(solidsDepth2DMapTextureLoc, 0);

    for (int i = 0; i < 2; i++) {
        glActiveTexture(GL_TEXTURE1 + i);
        glBindTexture( target: GL_TEXTURE_CUBE_MAP,  texture: solidsDepthCubeMapTexture[i]);
        glUniform1i(solidsDepthCubeMapTextureLoc[i], i + 1);
    }

    for (int i = 0; i < NR_OF_SOLID_OBJECTS; i++) {
        getSolidObjReadyForFinalPass( objIndex: i);
        solidObjectMeshMods[i].Draw( shaderProgram: solidsShader);
    }
}
```

The drawSolidObjects function is designed to render the solid objects within a scene, applying both shadow maps and lighting effects to achieve realistic rendering. Here's a detailed explanation of the steps involved in this function:

Setting the Viewport and Clearing Buffers:

- glViewport(0, 0, retina_width, retina_height); sets the viewport size to match the dimensions of the rendering window or target. This ensures that the drawing operations affect the entire area.
- glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); clears the color and depth buffers to prepare for a new frame. This step is essential to remove the remnants of the previous frame and ensure correct depth testing.

Activating the Shader Program:

- solidsShader.useShaderProgram(); activates the shader program that will be used to render the solid objects. This shader likely includes logic for applying lighting and shadow effects based on the depth maps.

Binding the Directional Shadow Map:

- glActiveTexture(GL_TEXTURE0); activates texture unit 0.
- glBindTexture(GL_TEXTURE_2D, solidsDepth2DMapTexture); binds the 2D texture containing the depth map from the directional light's perspective.
- glUniform1i(solidsDepth2DMapTextureLoc, 0); tells the shader program that the depth map texture will be found on texture unit 0.

Binding the Point Light Shadow Maps:

The loop iterates over the point light shadow maps (assuming there are 2 based on the loop condition).

For each point light shadow map:

- glActiveTexture(GL_TEXTURE1 + i); activates the texture unit corresponding to each point light shadow map. The first point light shadow map uses texture unit 1, the second uses texture unit 2, and so on.
- glBindTexture(GL_TEXTURE_CUBE_MAP, solidsDepthCubeMapTexture[i]); binds the cube map texture that stores the depth information from the point light's perspective to the active texture unit.
- glUniform1i(solidsDepthCubeMapTextureLoc[i], i + 1); informs the shader program about the texture unit where each cube map texture can be found.

Rendering Each Solid Object:

The final loop iterates over all solid objects in the scene.

For each object:

- getSolidObjReadyForFinalPass(i); prepares the object for rendering. This likely involves setting up model transformations and any other per-object uniforms required by the shader.
- solidObjectMeshMods[i].Draw(solidsShader); draws the object using the activated shader program.

In summary, the drawSolidObjects function is responsible for rendering all solid objects in the scene with proper shadowing effects. It achieves this by setting up the viewport,

clearing previous frame data, activating the appropriate shader program, binding necessary shadow map textures, and finally drawing each object with the prepared settings.

Shader code for shadows

Typically, to render a scene onto the faces of a cubemap texture, we would bind each face individually to the framebuffer object and execute the rendering process six times, each time targeting a different face of the cubemap for the depth buffer. However, by employing a geometry shader, we have the capability to render to all cubemap faces simultaneously in just one rendering pass. This efficiency is achieved by directly attaching the entire cubemap as the depth attachment of the framebuffer using glFramebufferTexture, thereby eliminating the need to switch the depth buffer target for each face.

Shaders for building the depth textures:

Depth.vert:

```
#version 410 core

layout(location = 0) in vec3 vPos;
uniform mat4 model;

void main() {
    gl_Position = model * vec4(vPos, 1.0);
}
```

It is quite simple.

In the context of generating depth textures for shadow mapping, this shader's simplicity is key because:

- Efficiency: Only the depth information is required, so there's no need to process color, normal vectors, or texture coordinates, which are irrelevant for shadow calculations.
- Focus on Geometry: The shader ensures that the geometry of the scene is accurately transformed according to its position relative to the light source, allowing for precise depth calculations.
- Versatility: This shader can be used with different light sources (directional, point, or spotlights) by varying the view and projection matrices applied outside this shader to compute gl_Position.

Depth.glsl:

```
#version 410 core
layout(triangles) in; ///3 triangle vertices
layout (triangle_strip, max_vertices=18) out;///6S passes
uniform mat4 shadowCubeMatrices[6];

out vec4 fragPos;

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 vertex
        {
            fragPos = gl_in[i].gl_Position;
            gl_Position = shadowCubeMatrices[face] * fragPos;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Input and Output Primitives:

- layout(triangles) in; informs the shader that the input primitives are triangles, meaning the geometry shader will process input vertices in groups of three, as each triangle consists of three vertices.
- layout (triangle_strip, max_vertices=18) out; defines the output primitive type as triangle_strip and sets the maximum number of vertices that the shader can output to 18. This setup is particularly useful for rendering to all six faces of a cubemap within a single pass. The choice of max_vertices=18 is based on the need to potentially emit three vertices per face for each of the six faces of the cubemap.

Uniforms:

- uniform mat4 shadowCubeMatrices[6]; declares an array of six 4x4 matrices. These matrices are transformation matrices used to position and orient geometry relative to each of the six faces of the cubemap from the light's perspective. They effectively transform vertices into the space of the cubemap, facilitating correct depth value calculation for shadow mapping.

Output Variable:

- out vec4 fragPos; declares an output variable that will pass the transformed vertex position to the fragment shader. It's essential for calculating the depth value in the context of the fragment shader.

Main Function:

The main function iterates over the six faces of the cubemap (for(int face = 0; face < 6; ++face)). For each face:

- gl_Layer = face; sets the built-in variable gl_Layer to the current face index. This is crucial for rendering to layered framebuffers, allowing each iteration to target a different face of the cubemap texture attached to the framebuffer.
- A nested loop (for(int i = 0; i < 3; ++i)) iterates over each of the three vertices of the input triangle. For each vertex:
    - The vertex's position is assigned to fragPos.
    - gl_Position is set to the product of the transformation matrix for the current cubemap face and the vertex's position. This operation transforms the vertex position into the space defined by the current face's view, ensuring that the rendered depth values are accurate from that perspective.
    - EmitVertex(); is called to output the transformed vertex.
- EndPrimitive(); signifies the end of the current primitive (triangle), ensuring correct assembly of the triangle strip.

This geometry shader efficiently enables omnidirectional shadow mapping by transforming and emitting vertices for all six faces of a cubemap in a single pass. This approach significantly optimizes the shadow map generation process, as it avoids the need for six separate rendering passes—one for each face of the cubemap.

Depth.frag:

```
#version 410 core

in vec4 fragPos;

uniform vec3 lightPos;
uniform float far_plane;

void main()
{
    float lightDistance = length(fragPos.xyz - lightPos);

    lightDistance = lightDistance / far_plane;

    gl_FragDepth = lightDistance;
}
```

Inputs:

- in vec4 fragPos; receives the position of the fragment in world space. This input is passed from the vertex or geometry shader, which has transformed the vertex positions into the perspective of the light source.

Uniform Variables:

- uniform vec3 lightPos; specifies the position of the light source in world space. This uniform is used to calculate the distance from the light to each fragment, which is crucial for determining how far away each part of the scene is from the light source.
- uniform float far_plane; defines the farthest distance from the light that will be considered for shadow calculations. This value is used to normalize the depth values to a [0, 1] range.

Main Function:

The main function of the shader calculates the depth value for each fragment with respect to the light source.

- float lightDistance = length(fragPos.xyz - lightPos); calculates the Euclidean distance between the light source and the fragment. This distance represents how far the fragment is from the light source.
- lightDistance = lightDistance / far_plane; normalizes the calculated distance by dividing it by the far_plane value. Since the depth values in OpenGL are typically normalized to a [0, 1] range, this step ensures that the resulting lightDistance fits within this range. It represents the relative depth of the fragment from the light's perspective.

- gl_FragDepth = lightDistance; explicitly sets the depth value of the fragment to the normalized distance calculated. gl_FragDepth is a built-in GLSL variable that allows shaders to control the depth value of each fragment directly. By setting this value, the shader determines how the depth of each fragment is recorded in the depth buffer, which is later used for shadow comparison during the lighting phase.

Overall, this fragment shader plays a critical role in the creation of depth maps for omnidirectional shadow mapping. By calculating and setting the depth value for each fragment based on its distance from the light source, it enables the generation of depth cubemaps that accurately represent the 3D geometry of the scene from the light's perspective. These depth values are essential for determining which parts of the scene are in shadow when rendering the scene from the camera's viewpoint.

Code in final pass shaders relevant to shadows:

ShaderStart.frag

```
float shadowCalculationPointLight(vec3 fragPosWorld, vec3 pointLightPosWorld, int index){
    vec3 fragToLight = fragPosWorld - pointLightPosWorld;
    float closestDepth = texture(depthCubeMapTexture[index], fragToLight).r;
    closestDepth*= far_plane;
    float currentDepth = length(fragToLight);
    if (currentDepth > far_plane){
        return 0.0f;
    }
    float bias = 0.05;
    float shadow = currentDepth -  bias > closestDepth ? 1.0 : 0.0;
    return shadow;
}
```

```
float shadow = shadowCalculationPointLight(fPosWorldP, pointLight.position, index);

return (totalAmbientThisLight + (1.0f - shadow) * (totalDiffuseThisLight + totalSpecularThisLight));
```

The shadowCalculationPointLight function in the fragment shader is designed to determine whether a given fragment (piece of a surface) is in shadow with respect to a point light source, using a shadow map stored in a cube map texture. The function takes as

input the world position of the fragment (fragPosWorld), the world position of the point light (pointLightPosWorld), and an index (index) that specifies which depth cube map texture to use. Here's how it works:

Calculate Vector from Fragment to Light:

- vec3 fragToLight = fragPosWorld - pointLightPosWorld; computes the vector from the fragment to the light source. This vector is used to determine the direction in which to look up the depth value from the cube map texture.

Fetch the Closest Depth from the Shadow Map:

- float closestDepth = texture(depthCubeMapTexture[index], fragToLight).r; samples the depth cube map texture using the direction vector fragToLight. The .r component is retrieved because depth values are stored in the red channel. This value represents the closest depth from the light source to any geometry in the scene along the direction of fragToLight.

Normalize the Closest Depth:

- closestDepth *= far_plane; normalizes the depth value by multiplying it by the far_plane distance. This step is necessary because depth values in the shadow map are normalized to [0, 1], and multiplying by far_plane converts it back to a real distance in world space.

Calculate the Current Depth:

- float currentDepth = length(fragToLight); calculates the actual distance from the light source to the fragment. This value represents how far the current fragment is from the light source along the fragToLight vector.

Early Exit for Fragments Beyond the Far Plane:

- If currentDepth > far_plane, the function returns 0.0f, indicating no shadow, as the fragment is beyond the effective range of the light source (and thus the shadow map).

Apply Bias to Prevent Shadow Acne:

- float bias = 0.05; introduces a small bias to the depth comparison to prevent shadow acne, a common artifact where surfaces incorrectly shadow themselves due to precision issues.

Determine Shadow Presence:

- float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0; compares the biased current depth to the closest depth from the shadow map. If the current depth (minus bias) is greater than the closest depth, it means an object is between the light source and the fragment, casting a shadow on the fragment. Thus, shadow is set to 1.0 (shadowed). Otherwise, it's set to 0.0 (lit).

This function effectively uses depth comparison to implement shadow mapping for point lights. By comparing the depth of the fragment to the depth stored in the shadow map, it can accurately determine whether the fragment is in shadow or directly illuminated by the light source.

3.1.2.2 Lights

Code in fragment shader relevant to lighting

ShaderStart.frag:

```
vec3 calcPointLight(PointLight pointLight, vec3 normalEye, vec3 fPosEyeP, vec3 viewDirN, int index, vec3 fPosWorldP){
    vec3 lightPosEye = vec3(view * vec4(pointLight.position, 1.0));
    vec3 lightDirEye = normalize(lightPosEye - fPosEyeP.xyz);
    float dist = distance(lightPosEye, fPosEyeP.xyz);
    float att = 1.0 / (pointLight.constant + pointLight.linear * dist+ pointLight.quadratic * dist * dist);

    vec3 halfVector = normalize(viewDirN + lightDirEye);

    float diffCoef = max(dot(normalEye, lightDirEye), 0.0f);
    float specCoeff = pow(max(dot(normalEye, halfVector), 0.0f), shininess);

    vec3 totalAmbientThisLight = pointLight.ambient * texture(diffuseTexture, fTexCoords).rgb * att;
    vec3 totalDiffuseThisLight = pointLight.diffuse * diffCoef * texture(diffuseTexture, fTexCoords).rgb * att;
    vec3 totalSpecularThisLight = pointLight.specular * specCoeff * texture(diffuseTexture, fTexCoords).rgb * att;

    float shadow = shadowCalculationPointLight(fPosWorldP, pointLight.position, index);

    return (totalAmbientThisLight + (1.0f - shadow) * (totalDiffuseThisLight + totalSpecularThisLight));
}
```

The calcPointLight function calculates the lighting contribution from a point light source to a fragment, considering ambient, diffuse, and specular components, while also incorporating shadow effects. Here's a detailed explanation:

Transform Light Position to Eye Space:

- vec3 lightPosEye = vec3(view * vec4(pointLight.position, 1.0)); transforms the light's position from world space to eye (or view) space. This is necessary because lighting

calculations are typically performed in view space for consistency and to accommodate the camera's perspective.

Calculate Light Direction and Distance:

- vec3 lightDirEye = normalize(lightPosEye - fPosEyeP.xyz); computes the direction vector from the fragment to the light source in view space.
- float dist = distance(lightPosEye, fPosEyeP.xyz); calculates the distance between the light source and the fragment, which is used to compute attenuation.

Compute Attenuation:

- float att = 1.0 / (pointLight.constant + pointLight.linear * dist + pointLight.quadratic * dist * dist); calculates the attenuation factor based on the distance to the light source. The equation incorporates constant, linear, and quadratic components to simulate realistic light falloff.

Half Vector for Specular Calculation:

- vec3 halfVector = normalize(viewDirN + lightDirEye); computes the half vector between the view direction and light direction. This vector is used in calculating the specular highlight based on the Blinn-Phong reflection model.

Diffuse and Specular Coefficients:

- float diffCoef = max(dot(normalEye, lightDirEye), 0.0f); calculates the diffuse coefficient based on the angle between the light direction and the fragment's normal. A dot product greater than zero indicates that the light hits the surface.
- float specCoeff = pow(max(dot(normalEye, halfVector), 0.0f), shininess); calculates the specular coefficient using the Blinn-Phong model, which considers the angle between the normal and the half vector, raised to the power of a shininess factor.

Calculate Lighting Components:

- Ambient: vec3 totalAmbientThisLight = pointLight.ambient * texture(diffuseTexture, fTexCoords).rgb * att; represents the ambient light multiplied by the texture color and attenuation. Ambient light is global and affects all objects equally.
- Diffuse: vec3 totalDiffuseThisLight = pointLight.diffuse * diffCoef * texture(diffuseTexture, fTexCoords).rgb * att; represents the light that scatters in many directions due to the rough surface.

- Specular: vec3 totalSpecularThisLight = pointLight.specular * specCoeff * texture(diffuseTexture, fTexCoords).rgb * att; represents the mirror-like reflection of the light source.

Shadow Determination:

- float shadow = shadowCalculationPointLight(fPosWorldP, pointLight.position, index); calculates whether the fragment is in shadow relative to the point light. The function returns 1 if the fragment is in shadow and 0 otherwise.

Final Lighting Calculation:

- The final lighting is the sum of the ambient component and the shadow-influenced combination of diffuse and specular components. If the fragment is in shadow (shadow equals 1), the diffuse and specular contributions are effectively nullified, leaving only the ambient light. If not in shadow, the full lighting effect is applied.

This function effectively calculates the light's impact on a fragment, taking into account its material properties (through texture sampling), its geometric relation to the light source (via normals and light direction), and whether it is shadowed or directly illuminated.

```
vec3 computeLightComponents(vec4 fPosEye)
{
    vec3 result = vec3(0.0f, 0.0f, 0.0f);
    //transform normal
    vec3 normalEye = normalize(fNormal);

    //compute view direction
    vec3 viewDirN = normalize(- fPosEye.xyz);

    result += calcDirLight(globalIllumination, normalEye, viewDirN);
    for (int i=0;i<2;i++){
        result += calcPointLight(pointLights[i], normalEye, fPosEye.xyz, viewDirN, i, fPosWorld.xyz);
    }
    return result;
}
```

The computeLightComponents function aggregates the lighting effects from both directional and point light sources on a fragment, resulting in the final color contribution based on the ambient, diffuse, and specular components, including shadows.

Initialization:

- vec3 result = vec3(0.0f, 0.0f, 0.0f); initializes the variable result to store the cumulative lighting effect from all light sources. Initially, it's set to black, meaning no light.

Normal Transformation:

- vec3 normalEye = normalize(fNormal); calculates the normalized normal vector in view space. The normal vector (fNormal) is assumed to be provided as an input to the fragment shader, possibly from the vertex shader, and needs to be normalized to ensure accurate lighting calculations.

View Direction Calculation:

- vec3 viewDirN = normalize(-fPosEye.xyz); computes the view direction vector in view space. It's obtained by normalizing the negative of the fragment's position in view space (fPosEye), effectively pointing from the fragment back to the camera.

Directional Light Calculation:

- result += calcDirLight(globalIllumination, normalEye, viewDirN); calculates the contribution of the directional light (simulating sunlight or other faraway light sources) to the fragment's color. This involves calculating the ambient, diffuse, and specular components based on the light's properties, the fragment's normal, and the view direction.

Point Lights Calculation:

- The loop iterates over the point lights (for (int i=0; i<2; i++)) and adds their contributions to the result. For each point light:
    - calcPointLight(pointLights[i], normalEye, fPosEye.xyz, viewDirN, i, fPosWorld.xyz) calculates the light contribution from the i-th point light. It factors in the light's position, attenuation, the fragment's position in both view and world space, normal, and view direction. It also determines whether the fragment is in shadow relative to each point light.
    - The result of calcPointLight is added to result, accumulating the lighting effects from each point light source.

Return Final Lighting Contribution:

- The function returns result, which now contains the combined lighting effects from the directional light and all point lights considered. This includes ambient lighting that affects the fragment uniformly, diffuse lighting that depends on the angle between the light direction and the fragment's normal, and specular highlights that depend on the view direction.

This function effectively synthesizes the contributions of different types of light sources to determine the final visual appearance of a fragment based on its material properties and its interaction with light and shadow within the scene.

3.1.2.3 Fog

```
float computeFog(vec4 fragmentPosEyeSpace)
{
    float fogDensity = 0.05f;
    float fragmentDistance = length(fragmentPosEyeSpace.xyz);
    float fogFactor = exp(-pow(fragmentDistance * fogDensity, 2.0));

    return clamp(fogFactor, 0.0, 1.0);
}
```

The computeFog function calculates the intensity of fog at a given fragment's position, ultimately affecting the color of that fragment based on its distance from the camera (or eye space position). This function implements a simple exponential squared fog model, where the fog's density decreases exponentially with the square of the distance from the viewer. Here's a step-by-step explanation:

Fog Density:

- float fogDensity = 0.05f; sets the density of the fog. A higher value results in a denser fog that becomes opaque at shorter distances.

Calculate Fragment's Distance from Camera:

- float fragmentDistance = length(fragmentPosEyeSpace.xyz); computes the distance between the camera and the fragment. This distance is calculated in eye (or view) space, where fragmentPosEyeSpace is the position of the fragment transformed into view space coordinates. The .xyz component extracts the vector part of the position, ignoring any homogeneous coordinate.

Compute Fog Factor:

- float fogFactor = exp(-pow(fragmentDistance * fogDensity, 2.0)); calculates the fog factor using an exponential squared decay based on the fragment's distance from the camera. The computation involves squaring the product of the fragment distance and fog density, then applying a negative exponent to it. This results in a value that decreases rapidly as the distance increases, simulating the effect of fog becoming denser (or more opaque) with distance.

Clamp Fog Factor:

- return clamp(fogFactor, 0.0, 1.0); ensures the fog factor is within the range [0, 1]. A value of 0 means the fragment is completely obscured by fog, whereas a value of 1 means the fragment is unaffected by fog. This clamping is essential to ensure the fog factor can be directly used to mix the fragment's color with the fog color in a linear interpolation, preventing unrealistic values that could result from the exponential calculation.

Overall, the computeFog function provides a way to simulate atmospheric fog effects in a 3D scene. By applying this fog factor to the final color computation of fragments, scenes can achieve a more realistic appearance, with distant objects gradually fading into the fog, enhancing the depth perception and atmospheric quality of the rendered image.

```
void main()
{
    vec3 fogColor = vec3(0.7, 0.7, 0.7); // Light gray fog

    vec4 fPosEye = view * fPosWorld;
    vec3 result = computeLightComponents(fPosEye);


    vec3 color = min(result, 1.0f);

    float fogFactor = computeFog(fPosEye);
    vec3 finalColor = mix(fogColor, result, fogFactor);

    fColor = vec4(finalColor, 1.0f);
}
```

The main function in the shader is where the final color of a fragment is calculated, taking into account lighting and fog effects. Here's a breakdown of the process:

Define Fog Color:

- vec3 fogColor = vec3(0.7, 0.7, 0.7); sets the color of the fog to a light gray. This color will be blended with the fragment's color based on the distance from the camera, simulating the appearance of fog in the scene.

Transform Fragment Position to Eye Space:

- vec4 fPosEye = view * fPosWorld; calculates the position of the fragment in view (or eye) space by multiplying the world space position (fPosWorld) by the view matrix (view). This transformation is essential for both lighting calculations and determining the fragment's distance from the camera for fog computation.

Compute Lighting Components:

- vec3 result = computeLightComponents(fPosEye); calculates the lighting contribution to the fragment based on its position in eye space. This includes contributions from ambient, diffuse, and specular light components, as well as shadows cast by other objects in the scene.

Clamp Resulting Color:

- vec3 color = min(result, 1.0f); ensures that each component of the resulting color does not exceed 1.0, preventing overexposure. This step is necessary because lighting calculations can sometimes produce values greater than 1.0, leading to unrealistic colors.

Compute Fog Factor:

- float fogFactor = computeFog(fPosEye); calculates the intensity of the fog at the fragment's position. The closer the fragment is to the camera, the less it is affected by fog, and vice versa. The fog factor ranges between 0 (completely obscured by fog) and 1 (unaffected by fog).

Blend Final Color with Fog:

- vec3 finalColor = mix(fogColor, result, fogFactor); blends the fragment's color (result) with the fog color based on the fog factor. If the fog factor is close to 1, the fragment's color dominates, and if the fog factor is close to 0, the fog color dominates. This blending simulates the effect of objects becoming less visible and taking on the color of the fog as they recede into the distance.

Set Fragment Color:

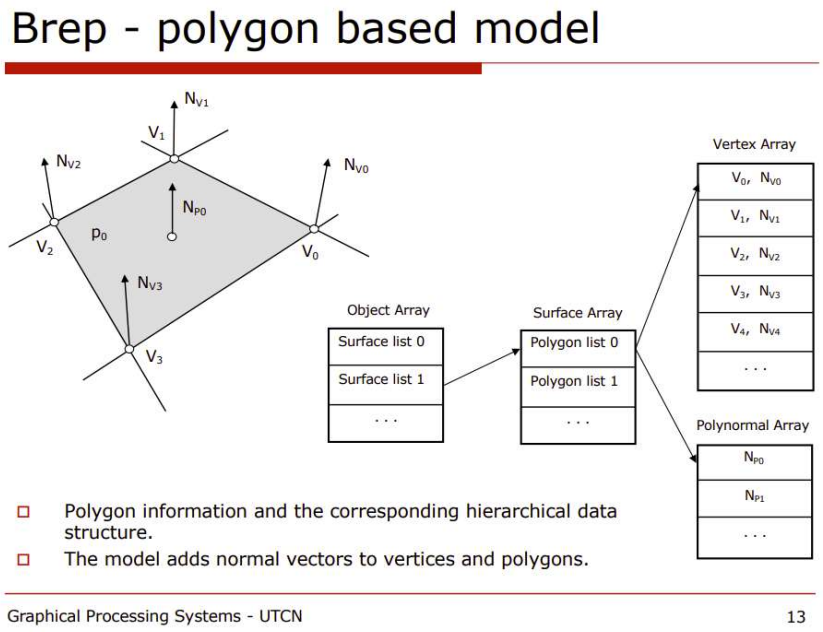
- fColor = vec4(finalColor, 1.0f); assigns the final color to the fragment, including an alpha value of 1.0 to indicate full opacity. This color is the result of the lighting calculations adjusted for fog effects, and it determines how the fragment will appear in the rendered scene.

The main function effectively combines the effects of lighting and fog to produce a realistic rendering of each fragment in the scene. By adjusting for fog based on the distance from the camera, it enhances the depth and atmospheric quality of the rendering.

3.2. Graphic Model

The model used is brep (boundary representation) and it is a polygon based model. The model stores an array of objects, each object contains an array of polygons (in this case triangles) and each polygon contains a list of vertices (with vertex normal) and a polynormal array.

The boundary representation (often abbreviated as B-rep or BREP) is a method for representing the geometry of 3D objects in computer graphics and solid modeling. It describes a 3D object in terms of its topological and geometric information. Here is a detailed description:



Topological Elements:

- Vertices: The points where edges meet and the most basic element of the B-rep model.
- Edges: Curved or straight line segments that connect two vertices and define the object's shape.

- Faces: The surfaces enclosed by a set of edges, typically in the form of polygons or more complex surface patches.
- Loops: The set of edges bounding a face. There can be an outer loop and several inner loops if there are holes or internal boundaries within the face.

Geometric Information:

- Surface Data: Defines the 3D geometry of the faces, which can be flat (planar) or curved (non-planar), like NURBS surfaces, Bezier patches, etc.
- Curve Data: Defines the geometry of the edges, which can be straight lines, circles, splines, etc.
- Vertex Coordinates: The (x, y, z) spatial coordinates of each vertex in a 3D space.

In a B-rep model, the geometric information provides the shape of the surfaces and curves, while the topological information provides the connectivity between these geometric elements. This system can represent complex objects with intricate details and is particularly powerful in CAD (Computer-Aided Design) applications, where precise control over surfaces and edges is required.

Advantages:

- Precision: Allows for the exact representation of complex surfaces and solids.
- Efficiency: Supports efficient algorithms for Boolean operations, feature recognition, and more.
- Flexibility: Easy to implement changes to the geometry without losing the structure of the model.

Challenges:

- Complexity: Requires careful management of the relationship between topological and geometric elements.
- Computationally Intensive: Certain operations, like Boolean operations, can be computationally demanding.

B-rep is one of the primary methods used for solid modeling and is supported by most modern CAD systems. It contrasts with other representations like Constructive Solid Geometry (CSG), which uses primitives and Boolean operations to define objects.

3.3. Data Structures

◆ TransformsAndLighting()

```
gps::TransformsAndLighting::TransformsAndLighting ( const ModelTransforms(&) solidModelTransforms[NR_OF_SOLID_OBJECTS],
                                                    const glm::mat4 &          projection,
                                                    const gps::Camera &        camera,
                                                    const SceneLighting &      sceneLighting,
                                                    const gps::Shader &        solidsShader,
                                                    const gps::Shader &        pointLightsShader,
                                                    const gps::Shader &        depth2DMapShader,
                                                    const gps::Shader &        depthCubeMapShader,
                                                    const Model3D(&)           solidObjectMeshMods[NR_OF_SOLID_OBJECTS],
                                                    const Model3D &            lightObjectMeshMod,
                                                    const int &                retina_width,
                                                    const int &                retina_height )
```

◆ DirLight()

```
gps::DirLight::DirLight ( const glm::vec3 &      direction,
                          const glm::vec3 &      ambient,
                          const glm::vec3 &      diffuse,
                          const glm::vec3 &      specular,
                          const gps::Shader &    solidObjectsShader )
```

```
DirLight::DirLight(const glm::vec3 &direction, const glm::vec3 &ambient, const glm::vec3 &diffuse, const glm::vec3 &specular,
        const gps::Shader &solidObjectsShader)
       : direction(direction), ambient(ambient), diffuse(diffuse), specular(specular),
         solidObjectsShader(solidObjectsShader) {
    this->solidObjectsShader.useShaderProgram();
    directionLoc = glGetUniformLocation(this->solidObjectsShader.shaderProgram, "globalIllumination.directionEye");
    ambientLoc = glGetUniformLocation(this->solidObjectsShader.shaderProgram, "globalIllumination.ambient");
    diffuseLoc = glGetUniformLocation(this->solidObjectsShader.shaderProgram, "globalIllumination.diffuse");
    specularLoc = glGetUniformLocation(this->solidObjectsShader.shaderProgram, "globalIllumination.specular");
}
```

◆ ModelTransforms() [1/2]

```
gps::ModelTransforms::ModelTransforms ( const glm::vec3 & translation,
                                        const glm::vec3 & rotation,
                                        const glm::vec3 & scaling )
```

◆ PointLight() [1/2]

```
gps::PointLight::PointLight ( const glm::vec3 &      position,
                              float                  constant,
                              float                  linear,
                              float                  quadratic,
                              const glm::vec3 &      ambient,
                              const glm::vec3 &      diffuse,
                              const glm::vec3 &      specular,
                              const gps::Shader & solidObjectsShader )
```

```
int PointLight::nrOfPointLights = 0;

PointLight::PointLight(const glm::vec3 &position, float constant, float linear, float quadratic, const glm::vec3 &ambient,
          const glm::vec3 &diffuse, const glm::vec3 &specular, const gps::Shader &solidObjectsShader) : position(
       position), constant(constant),
                                                                                                        linear(linear),
                                                                                                        quadratic(
                                                                                                               quadratic),
                                                                                                        ambient(ambient),
                                                                                                        diffuse(diffuse),
                                                                                                        specular(
                                                                                                               specular),
                                                                                                        solidObjectsShader(
                                                                                                               solidObjectsShader) {
    this->solidObjectsShader.useShaderProgram();
    ambientLoc = glGetUniformLocation(solidObjectsShader.shaderProgram,
                                      ("pointLights[" + std::to_string( val: nrOfPointLights) + "].ambient").c_str());
    diffuseLoc = glGetUniformLocation(solidObjectsShader.shaderProgram,
                                      ("pointLights[" + std::to_string( val: nrOfPointLights) + "].diffuse").c_str());
    specularLoc = glGetUniformLocation(solidObjectsShader.shaderProgram,
                                       ("pointLights[" + std::to_string( val: nrOfPointLights) + "].specular").c_str());
    constantLoc = glGetUniformLocation(solidObjectsShader.shaderProgram,
                                       ("pointLights[" + std::to_string( val: nrOfPointLights) + "].constant").c_str());
    linearLoc = glGetUniformLocation(solidObjectsShader.shaderProgram,
                                     ("pointLights[" + std::to_string( val: nrOfPointLights) + "].linear").c_str());
    quadraticLoc = glGetUniformLocation(solidObjectsShader.shaderProgram,
                                        ("pointLights[" + std::to_string( val: nrOfPointLights) + "].quadratic").c_str());
    positionLoc = glGetUniformLocation(solidObjectsShader.shaderProgram,
                                       ("pointLights[" + std::to_string( val: nrOfPointLights) + "].position").c_str());
    ++nrOfPointLights;
}
```

◆ SceneLighting()

```
gps::SceneLighting::SceneLighting ( const DirLight &      globalIllumination,
                                    const PointLight(&) pointLights[NR_OF_POINT_LIGHTS] )
```

3.4. Class Hierarchy

4. User Manual

Navigating the Scene

- W: Move forward

- S: Move backward
- A: Strafe left
- D: Strafe right

Camera Control

- The camera can be controlled to look around in the scene. Use your mouse to change the view direction of the camera, providing a full 360-degree exploration capability.

Showcase Animation

To start the showcase animation, which automatically navigates and displays key features of the scene, press:

- 0: Starts the showcase animation

Changing View Modes

The application supports multiple rendering modes for viewing the scene, allowing users to toggle between solid, wireframe, and smooth views. These modes can be accessed using the following keys:

- 1: Solid View Mode - This mode renders the scene with solid surfaces, providing a realistic view of the objects in the scene.
- 2: Wireframe View Mode - This mode displays the scene as a wireframe, allowing you to see the underlying geometric structure of the objects.
- 3: Smooth View Mode - This mode allows you to see individual vertices

5.Conclusions and Further Developments

The development and implementation of this application have successfully leveraged advanced graphical techniques, including omnidirectional shadow mapping, the Blinn-Phong lighting model, and fog effects, to enhance the realism and depth of the 3D scenes. These technologies have been instrumental in achieving high-quality visualizations, providing users with an immersive and interactive experience.

- Omnidirectional Shadow Mapping: This technique has enabled the creation of dynamic shadows that accurately represent the interaction of light with objects in the environment, regardless of the light source's position. It has significantly improved the scene's realism, making the lighting and shadows more consistent with real-world observations.
- Blinn-Phong Lighting Model: By adopting this lighting model, we have enhanced the visual quality of surfaces within the scene, offering a more nuanced representation of how light interacts with different materials. This model has been particularly effective in simulating the specular highlights and diffuse reflections characteristic of various textures and finishes.
- Fog Effects: The integration of fog effects has added depth and atmosphere to the scenes, simulating the attenuation of light over distance and creating a sense of scale and space. This feature has not only contributed to the aesthetic appeal of the environment but also helped in conveying more realistic environmental conditions.

Further Developments

While the current implementation has achieved a high level of visual fidelity and user engagement, there are several areas identified for further development to enhance the application's capabilities and user experience:

- Advanced Shadow Techniques: Exploring more sophisticated shadow mapping techniques, such as Cascaded Shadow Maps (CSM) or Variance Shadow Maps (VSM), could provide even more accurate and performance-efficient shadow rendering, especially for large-scale scenes.
- Physically Based Rendering (PBR): Transitioning to a PBR framework could further improve the realism of materials and lighting in the scene. PBR offers a more comprehensive approach to rendering light interaction with surfaces, based on physical principles, which could replace or complement the Blinn-Phong model.

- Dynamic Weather Systems: Incorporating dynamic weather systems, including variable fog density, precipitation effects, and cloud shadows, could enhance the environmental realism and offer users interactive scenarios that reflect changing weather conditions.
- Performance Optimization: As we add more complex graphical features, performance optimization will become increasingly important. Techniques such as Level of Detail (LOD) models, frustum culling, and efficient resource management will be crucial in maintaining smooth performance across a wide range of hardware.
- User-Defined Lighting and Fog Parameters: Allowing users to adjust lighting and fog parameters in real-time could provide a more interactive and personalized experience, enabling experimentation with different environmental setups and lighting conditions.

Conclusion

The application's current state represents a significant achievement in 3D scene rendering, effectively utilizing advanced graphical techniques to create immersive and visually compelling environments. As we look to the future, the planned developments aim to push the boundaries of realism, performance, and interactivity, ensuring that the application continues to provide a cutting-edge platform for exploring and interacting with 3D scenes.

6.References

Graphical Processing Systems lectures and practical works, Technical University of Cluj-Napoca 2024

https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

https://www.youtube.com/watch?v=BZw2olDmZo4&list=PLPaoO-vpZnumdcb4tZc4x5Q-v7CkrQ6M-&index=21&ab_channel=VictorGordan

https://www.youtube.com/watch?v=9g-4aJhCnyY&list=PLPaoO-vpZnumdcb4tZc4x5Q-v7CkrQ6M-&index=26&ab_channel=VictorGordan

https://www.youtube.com/watch?v=Q8w_z2Ye-Go&list=PLPaoO-vpZnumdcb4tZc4x5Q-v7CkrQ6M-&index=27&ab_channel=VictorGordan