Physically Based Skin rendering Using OpenGL

Spajić, Matej

Josip Juraj Strossmayer University of Osijek
School of Applied Mathematics and Informatics

Undergraduate program in Mathematics and Computer Science

Physically Based Skin Rendering using OpenGL

Bachelor's thesis

Mentor:
doc. dr. sc. Domagoj Ševerdija

Student:
Matej Spajić

Osijek, 2024

Contents

1 Abstract **1**

2 OpenGL **3**
2.1 Introduction to OpenGL 3
2.2 Specification and Implementation 3
2.3 Supporting Libraries and Extensions 4

3 Implementation **5**
3.1 Initialization of OpenGL Context and GLFW Window 5
3.2 Global OpenGL State Configuration 6
3.3 Shader Compilation and Linking 6
3.4 Loading Textures and Models 7
3.5 Framebuffer Configuration 9
3.6 Environment Mapping and Convolution 10
3.7 Main Rendering Loop 11

4 PBR Shader Implementation **15**
4.1 Lighting Calculations and BRDF 16
4.1.1 Normal Distribution Function D 18
4.1.2 Geometry Function G 18
4.1.3 Fresnel Function F 20
4.1.4 PBR Shader Main Function 21

5 Model Render **25**

References **27**

1 | Abstract

Real human skin shading based on the physically based rendering (PBR) is a challenging task that requires knowledge of mathematics, physics, and biology. The main purpose of the PBR technique is to accurately reproduce how light behaves in the real world, as opposed to earlier methods like the Phong and Blinn-Phong lighting models. Aside from its visual realism, PBR grants artists practical benefits and a convenient way of defining materials by physical properties, which reduces the use of workaround techniques and 'lighting trickery.' Furthermore, even when authored with PBR, textures, and models do not look significantly different under different lighting conditions, a factor that usually requires tremendous effort in non-PBR strategies. This paper describes and presents the basic modeling and advanced rendering techniques for generating physically accurate models, recalls the main approaches to using such techniques, and discusses new tendencies in modeling skin. The described methods in this thesis were implemented using modern GPUs and OpenGL's shading language for higher precision and improved performance.

2 | OpenGL

2.1 Introduction to OpenGL

There are several ways to approach the topic of OpenGL, but before delving into the technical details, we need to define what OpenGL is and what it serves in the context of computer graphics. OpenGL is mainly considered an Application Programming Interface (API) that provides developers with a large set of functions that they can use to manipulate graphics and images. However, it is necessary to make the important distinction here that OpenGL is not an API but a specification. This specification is created and sustained by the Khronos Group, an industry consortium that is in charge of defining open standards in graphics and parallel processing.

2.2 Specification and Implementation

Although the OpenGL specification fully describes the expected results of its functions and the sequences of behaviors of the functions, it does not describe how these functions are to be implemented. This distinction enables the development of a more generic specification, which can be implemented in several ways depending on the platform or hardware. Developers, especially graphics card developers, comply with the OpenGL specification by creating libraries that map defined functions to operations that can be performed by the underlying hardware. These implementations must be able to conform to the specification and be able to give the same output no matter which platform it is run on, even though the code may vary.

The role of the manufacturers of the graphic cards is especially important in the context of the OpenGL environment. Usually, each graphics card works with certain versions of OpenGL that are tailored to that particular piece of hardware. This versioning is important because it determines the capabilities and components that developers can implement in that card.

OpenGL like most graphical libraries can be different in terms of maintenance and implementation depending on the operating system in use. For instance, while on Apple platforms the OpenGL libraries are provided by Apple themselves, it is possible to maintain steady compatibility between the library and the hardware/software environment. However, the implementations of OpenGL in Linux may be direct implementations from the graphics card manufacturers, or

they may be implemented by third-party open-source projects that supplement these official libraries for different types of hardware and applications.

Due to these differences in implementation, developers may sometimes find that OpenGL does not behave as expected in some contexts. These problems may be due to specific support offered by the graphics card manufacturer, or the organization that maintains that library in a certain platform. It is critical to understand this layered system of OpenGL including the specification as the top tier and the hardware implementations and platform-specific adaptations as the lower tier to fully utilize OpenGL in graphic development and in case of occurring problems.

2.3 Supporting Libraries and Extensions

One of the greatest strengths of OpenGL is its support of extensions. Every time a graphics company comes up with a new trick or optimization for rendering, it is usually rolled out as an extension in the drivers. This mechanism enables the hardware vendors to bring in new functionalities that are not in the standard OpenGL. If the hardware on which an application runs supports such an extension, then the developer can use these advanced features to improve or fine-tune the graphics. This approach enables us to use new techniques as soon as they become available without having to wait for them to be incorporated into the OpenGL core versions. Before using these extensions, developers have to check if they are available, which is usually done using extension querying functions or libraries. This dynamic system allows the application to change its behavior depending on the extensions that are installed, which makes the application more versatile and efficient.

When modeling and rendering the scene's 3D head model, several libraries are used to improve the development processes and enhance the render. For example, the GLFW library is used to create and organize the window and handle the user input. GLFW is a cross-platform library that has been designed specifically for this purpose. For image loading, the stb_image library is used to help load the texture data from the image files into the OpenGL context. This library provides a comparatively limited number of operations, but it is capable of opening images of different formats. Additionally, to perform mathematical computations (like matrix and vector operations) the GLM library is used.

3 | Implementation

3.1 Initialization of OpenGL Context and GLFW Window

The pipeline begins with the creation of the GLFW window and the GLAD OpenGL loader. This function initiates the window context, verifies that the OpenGL functions have been loaded before proceeding, and sets the context version. If any of these initializations do not occur then the program is terminated.

```
GLFWwindow* initGLFWandGLAD(int width, int height) {
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
    glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* window = glfwCreateWindow(width, height, "Window",
    NULL, NULL);
    if (window == NULL) {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return nullptr;
    }

    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window,
    framebuffer_size_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return nullptr;
    }

    return window;
}
```

3.2 Global OpenGL State Configuration

The program configures and sets the OpenGL state variables upon initialization of the program. This includes enabling depth testing, setting the depth function for skybox rendering, and enabling seamless cube map sampling. These configurations are crucial to render the scene with the correct depth handling because our scene is set in a 3D environment where objects overlap each other.

```
int main(){
    GLFWwindow* window = initGLFWandGLAD(width, height);
    if (!window) return -1;

    glEnable(GL_DEPTH_TEST);

    glDepthFunc(GL_LEQUAL);

    glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);
    ...
}
```

3.3 Shader Compilation and Linking

Shader programs are important in the OpenGL architecture, as they define how vertices and fragments go through the pipeline. The implementation encompasses several shader programs like pbrShader, lensShader and backgroundShader that are compiled and linked from individual vertex and fragment shader source files. The shader implementation is a set of classes developed in their respective header files. These shaders are pre-configured as needed before the actual scene rendering begins, using the custom setupShaders() program function. The full PBR shader implementation can be found in the next chapter, and the rest of the shaders deal with the ambient light in the scene.

```
int main(){
    ...
    Shader pbrShader("pbr.vs", "pbr.fs");
    Shader lensShader("lens.vs", "lens.fs");
    Shader equirectangularToCubemapShader(
        "cubemap.vs", "equirectangular_to_cubemap.fs");
    Shader irradianceShader(
        "cubemap.vs", "irradiance_convolution.fs");
    Shader prefilterShader("cubemap.vs", "prefilter.fs");
    Shader brdfShader("brdf.vs", "brdf.fs");
    Shader backgroundShader("background.vs", "background.fs");

    setupShaders(pbrShader, backgroundShader);
    ...
}
```

3.4 Loading Textures and Models

Textures and models are located in specific directories and are bound to their respective classes through functions such as loadTexture() and through custom classes like the Model class. Without these assets, it would be impossible to develop realistic surfaces and geometry. For example, albedo, normal, specular, roughness, AO, and displacement maps are loaded for all our models in order to reach the intended result.

```
int main{
    ...
    // load all materials
    // FACE:
    unsigned int faceAlbedo =
         loadTexture((texturesPath + "Face_Albedo.jpg").c_str());
    unsigned int faceNormal =
        loadTexture((texturesPath + "Face_Normal.jpg").c_str());
    unsigned int faceSpecular =
        loadTexture((texturesPath + "Face_Specular.jpg").c_str());
    unsigned int faceRoughness =
        loadTexture((texturesPath + "Face_Roughness.jpg").c_str());
    unsigned int faceAOMap =
        loadTexture((texturesPath + "ao.png").c_str());
    unsigned int faceDisplacement =
        loadTexture((texturesPath + "Face_Displacement.jpg").c_str
    ());

    // EYES:
    ...
    // LENS:
    ...

    // construct the models
    Model mainModel((modelsPath + "model.obj").c_str());
    Model eyeModel((modelsPath + "realtime.obj").c_str());
    Model lensModel((modelsPath + "lens.obj").c_str());
    ...
}
```

The Model class provides a framework for working with our 3D models. It utilizes the Open Asset Import Library (Assimp) for importing various model formats and the previously mentioned GLM library for mathematical operations. Assimp can import dozens of different model file formats by loading all the model's data into its generalized data structures. As soon as it loads the model, we can retrieve all the required data from its specific data structures. Because the data structure of Assimp stays the same, regardless of the type of file format we imported, it abstracts us from all the different file formats out there.

```
#ifndef MODEL_H
#define MODEL_H
...
class Model
{
public:
    vector<Texture> textures_loaded;
    vector<Mesh>    meshes;
    string directory;
    bool gammaCorrection;

    Model(string const& path, bool gamma = false) : gammaCorrection
    (gamma)
    {
        loadModel(path);
    }

    void Draw(Shader& shader)
    {
        for (unsigned int i = 0; i < meshes.size(); i++)
            meshes[i].Draw(shader);
    }
private:
    void loadModel(string const& path)
    {
        Assimp::Importer importer;
        const aiScene* scene = importer.ReadFile(path,
    aiProcess_Triangulate | aiProcess_GenSmoothNormals |
    aiProcess_FlipUVs | aiProcess_CalcTangentSpace);

        if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE ||
    !scene->mRootNode) // if is Not Zero
        {
            cout << "ERROR::ASSIMP:: " << importer.GetErrorString()
     << endl;
            return;
        }

        directory = path.substr(0, path.find_last_of('/'));

        processNode(scene->mRootNode, scene);
    }
    ...
}
```

3.5 Framebuffer Configuration

Following the creation of the models, the program creates the necessary frame (FBOs) and render buffers (RBOs) to effectively handle off-screen rendering which is essential in tasks such as environment mapping and post-processing. The framebuffer configuration may include, for example, the binding of an RBO for depth storage and texture attachment for color output. These configurations are important when rendering scenes, not to the screen but into textures, allowing for advanced effects like HDR environment mapping.

```
1  int main{
2      ...
3      unsigned int captureFBO;
4      unsigned int captureRBO;
5      glGenFramebuffers(1, &captureFBO);
6      glGenRenderbuffers(1, &captureRBO);
7
8      glBindFramebuffer(GL_FRAMEBUFFER, captureFBO);
9      glBindRenderbuffer(GL_RENDERBUFFER, captureRBO);
10     glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24,
       512, 512);
11     glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
       GL_RENDERBUFFER, captureRBO);
12     ...
13 }
```

3.6 Environment Mapping and Convolution

The rendering pipeline transforms an HDR equirectangular environment map into a cube map through the shader. This transformation involves the conversion of the environment map into a set of cube faces that contain imported views from the HDR image of the scene. Once the transformation is complete, the resulting cube map is used to generate an irradiance map for diffuse lighting and a pre-filtered environment map for specular reflections by using irradianceShader and prefilterShader respectively. Such maps are necessary for PBR because they allow the lighting to take into account the surrounding conditions. The following code snippet illustrates a part of the conversion process.

```
int main{
    ...
    equirectangularToCubemapShader.use();
    equirectangularToCubemapShader.setInt("equirectangularMap", 0);
    equirectangularToCubemapShader.setMat4("projection",
   captureProjection);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, hdrTexture);

    glViewport(0, 0, 512, 512);
    glBindFramebuffer(GL_FRAMEBUFFER, captureFBO);
    for (unsigned int i = 0; i < 6; ++i)
    {
        equirectangularToCubemapShader.setMat4("view", captureViews
   [i]);
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0
   , GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, envCubemap, 0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        renderCube();
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    ...
}
```

3.7 Main Rendering Loop

In the main rendering loop, the program continuously updates based on the user input and the elapsed time, thus providing real-time interaction. The scene is made using the PBR shader which applies the data taken from the precomputed IBL in the form of the irradiance map, pre-filter map, and the BRDF lookup texture (LUT) to produce the desired accurate scene lighting and the objects contained within the scene. It also ensures accurate reflections of objects in the environment.

```
int main{
    ...
    while (!glfwWindowShouldClose(window))
    {
        // per-frame time logic and fps counter
        updateDeltaTime(window);

        processInput(window);

        glm::mat4 view = camera.GetViewMatrix();
        // model matrix
        glm::mat4 model = glm::mat4(1.0f);
        model = glm::scale(model, glm::vec3(0.1f));

        // Specify the color of the background
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        // Clean the back buffer and depth buffer
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // render scene, supplying the convoluted irradiance map to
    the final shader.
        pbrShader.use();
        pbrShader.setMat4("view", view);
        pbrShader.setVec3("camPos", camera.Position);

        // bind pre-computed IBL data
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_CUBE_MAP, irradianceMap);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_CUBE_MAP, prefilterMap);
        glActiveTexture(GL_TEXTURE2);
        glBindTexture(GL_TEXTURE_2D, brdfLUTTexture);
        ...
```

Transparency is achieved for the lens model by first disabling depth testing and then enabling blending while the rest of the non-transparent models have their depth testing re-enabled.

```
        ...
        glDisable(GL_DEPTH_TEST);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        // Render the lens model
        lensShader.use();
        lensShader.setMat4("view", view);
        lensShader.setMat4("projection", projection);
        lensShader.setVec3("camPos", camera.Position);
        lensShader.setMat4("model", model);

        bindTexture({ 0, lensNormal, 0, 0, lensAOMap, 0, 0 });
        lensModel.Draw(lensShader);

        glEnable(GL_DEPTH_TEST);
        glDisable(GL_BLEND);
        ...
```

The main model's shader is then activated and the model is drawn according to our PBR pipeline.

```
        ...
        pbrShader.use();
        pbrShader.setMat4("view", view);
        pbrShader.setMat4("projection", projection);
        pbrShader.setVec3("camPos", camera.Position);
        pbrShader.setMat4("model", model);
        pbrShader.setMat3("normalMatrix", glm::transpose(glm::
    inverse(glm::mat3(model))));

        bindTexture({ faceAlbedo, faceNormal, 0, faceRoughness,
    faceAOMap, faceSpecular, faceDisplacement });
        mainModel.Draw(pbrShader);

        bindTexture({ eyesAlbedo, eyesNormal, 0, eyesRoughness,
    eyesAOMap, eyesSpecular, 0 });
        eyeModel.Draw(pbrShader);
        ...
```

The property of dynamic lighting is handled by updating the position and color of the lights in the while loop, which simulates our desired light sources. We reset the texture IDs for the lights to improve object clarity. The lights in the scene are represented as spheres and their positions are managed by our PBR shader. Additionally, the skybox was rendered to cover the entire scene, preventing overdraw and enabling accurate depth information for all elements.

```
    ...
    // unbind textures for the lights
    bindTexture({ 1, 0, 0, 0, 0, 0, 0 });

    // render light source
    unsigned int nrOfLights = 1; // max 4

    for (unsigned int i = 0; i < nrOfLights; ++i) {
        glm::vec3 newPos = lightPositions[i] + glm::vec3(sin(
    glfwGetTime() * 5.0) * 5.0, 0.0, 0.0);
        newPos = lightPositions[i];
        pbrShader.setVec3("lightPositions[" + std::to_string(i) +
    "]", newPos);
        pbrShader.setVec3("lightColors[" + std::to_string(i) + "]",
     lightColors[i]);

        model = glm::mat4(1.0f);
        model = glm::translate(model, newPos);
        model = glm::scale(model, glm::vec3(0.5f));
        pbrShader.setMat4("model", model);
        pbrShader.setMat3("normalMatrix", glm::transpose(glm::
    inverse(glm::mat3(model))));
        renderSphere();
    }

    // render skybox (render last to prevent overdraw)
    backgroundShader.use();
    backgroundShader.setMat4("view", view);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, envCubemap);
    //glBindTexture(GL_TEXTURE_CUBE_MAP, irradianceMap); // display
     irradiance map
    //glBindTexture(GL_TEXTURE_CUBE_MAP, prefilterMap); // display
    prefilter map
    renderCube();

    // Swap the back buffer with the front buffer
    glfwSwapBuffers(window);
    glfwPollEvents();
    }

    // Delete window and terminate GLFW before ending the program
    glfwDestroyWindow(window);
    glfwTerminate();
    return 0;
}
```

4 | PBR Shader Implementation

The primary fragment shader that defines how the material (primarily skin) of our 3D model looks is a PBR shader implemented with the help of the shader class developed earlier. The shader applies the formally calculated lighting value to each fragment's otherwise black color and stores the resulting value in FragColor. This is realized directly through initial lighting calculations and indirectly by accounting for material, normal displacements (Normal Mapping), and environment cube map (Image Based Lighting or IBL).

```
//  fragment shader
    #version 460 core
    out vec4 FragColor; // <- end result (output)
    in vec2 TexCoords;
    in vec3 WorldPos;
    in vec3 Normal;

    // subsurface scattering
    uniform vec3 sssColor = vec3(0.9, 0.5, 0.2);
    uniform float sssScale = 0.6;
    uniform float thickness = 0.1f;

    // material parameters
    uniform sampler2D albedoMap;
    uniform sampler2D normalMap;
    uniform sampler2D metallicMap;
    uniform sampler2D roughnessMap;
    uniform sampler2D aoMap;
    uniform sampler2D specularMap;
    ...
```

The shader employs several texture maps to establish the characteristics of the material that is being drawn. These maps include:

- **Albedo Map**: This texture provides the object with its base color before any lighting or shading is applied. The shader applies a gamma correction to the texture to ensure that the colors are correctly displayed under physically based rendering.

- **Normal Map**: This map is used to modulate the surface normals of the material, creating the illusion of a complex surface while retaining the simple geometry of the object. In other words, the normal map provides your texture depth. It's used to adjust normals from tangent space to world space, ensuring that accurate lighting calculations can occur.
- **Metallic Map**: This texture determines how metallic the surface appears. Metallic surfaces reflect more light and exhibit sharper reflections compared to non-metallic surfaces. Since the skin almost has no metallic properties, we set the metallic ID to zero when binding textures.
- **Roughness Map**: This map determines the variation of the micro-surface roughness of the material. A more irregular surface will scatter the reflected light to a greater extent which results in the blurring of the reflected image and on the other hand, a smoother surface provides more direct reflection of light.
- **Ambient Occlusion (AO) Map**: This type of texture indicates those regions where the space is small and narrow or where the light cannot penetrate. It is applied to these areas so as to produce a darker (or lighter) hue and thus make the material look as realistic as possible. At render time, this map is mixed with the albedo to describe how it reacts to light.
- **Specular Map**: This map has been added to provide finer control over the specular reflections, allowing the artist to specify regions with varying reflectivity beyond what is typically provided by the metallic map alone. Since we are not using the metallic map, a specular map is used instead.

4.1 Lighting Calculations and BRDF

The Bidirectional Reflective Distribution Function (also known as BRDF) is a model that analyzes the amount of light ω_i reflecting off a particular surface given the incoming light direction ω_i, the outgoing direction ω_o, the surface normal n, and the surface roughness parameter a. Depending on the material properties of the surface it decides the proportion of the contribution made by each of the component light rays to the resultant reflected output. For instance, if the surface is perfectly smooth and mirror-like, the BRDF will be equal to 0.0 for every ω_i except for the one at a reflective angle identical to ω_o, where it outputs 1.0.

The BRDF models the reflective and refractive properties of materials based on microfacet theory. To ensure that the models are physically realistic, the process has to conform to the law of conservation of energy, and thus the reflected intensity can never be more than the incident light. Although Blinn-Phong is a type of BRDF used for similar inputs, it does not possess true physical realism due to not maintaining energy conservation. There are several physically based BRDFs out there to approximate the surface's reaction to light. However, almost all real-time PBR render pipelines make use of a BRDF known as the Cook-Torrance BRDF.

The Cook-Torrance BRDF contains both a diffuse and specular part:

$$f_r = k_d f_{\text{lambert}} + k_s f_{\text{cook-torrance}}$$

Here, k_d represents the ratio of incoming light energy that is refracted, while k_s indicates the ratio reflected. The left side of the BRDF equation, known as $f_{lambert}$, denotes the diffuse component, similar to Lambertian diffuse used in shading. It is denoted as a constant factor:

$$f_{\text{lambert}} = \frac{c}{\pi}$$

Here, c refers to the albedo or surface color, similar to the diffuse surface texture. The division by π normalizes the diffuse light, compensating for the fact that the integral involving the BRDF is scaled by π.

Different, more realistic, (but also more computationally intensive) equations exist for the BRDF's diffuse component. For our purposes, though, the Lambertian diffuse meets the needs of all our real-time rendering tasks. The specular part of the BRDF is a bit more advanced and is described as:

$$f_{\text{CookTorrance}} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

The Cook-Torrance specular BRDF is composed of three functions and a normalization factor in the denominator. Each of the D, F, and G symbols represents a type of function that approximates a specific part of the surface's reflective properties. These are defined as the normal Distribution function, the Fresnel equation, and the Geometry function:

- **Normal distribution function**: approximates the amount the surface's microfacets are aligned to the halfway vector, influenced by the roughness of the surface; this is the primary function approximating the microfacets.
- **Geometry function**: describes the self-shadowing property of the microfacets. When a surface is relatively rough, the surface's microfacets can overshadow other microfacets reducing the light the surface reflects.
- **Fresnel equation**: The Fresnel equation describes the ratio of surface reflection at different surface angles.

Each function approximates its physical counterpart with multiple versions aiming to capture the underlying physics more realistically or efficiently. It's acceptable to choose any approximated version for use. This implementation employs the Trowbridge-Reitz GGX for D, the Fresnel-Schlick approximation for F, and Smith's Schlick-GGX for G.

4.1.1 Normal Distribution Function D

The normal distribution function D statistically approximates the relative surface area of microfacets exactly aligned to the (halfway) vector h. There are a multitude of NDFs that statistically approximate the general alignment of the microfacets given some roughness parameter and the one we'll be using is known as the Trowbridge-Reitz GGX:

$$\text{NDF}_{\text{GGX TR}}(n, h, a) = \frac{a^2}{\pi((n \cdot h)^2(a^2 - 1) + 1)^2}$$

Here h is the halfway vector to measure against the surface's microfacets, with a being a measure of the surface's roughness. The Trowbridge-Reitz GGX normal distribution function translates to the following code:

```
float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a = roughness*roughness;
    float a2 = a*a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH*NdotH;

    float nom   = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;

    return nom / denom;
}
```

4.1.2 Geometry Function G

The geometry function statistically estimates how much the microscopic surface details block each other and cause light rays to be occluded, taking the roughness parameter as input where rougher surfaces are more likely to overshadow microfacets. The chosen method combines GGX with Schlick-Beckmann approximations, known as Schlick-GGX:

$$G_{\text{SchlickGGX}}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

Here k is a remapping of a based on whether we're using the geometry function for either direct lighting or IBL lighting:

$$k_{\text{direct}} = \frac{(\alpha + 1)^2}{8}$$
$$k_{\text{IBL}} = \frac{a^2}{2}$$

To effectively approximate the geometry we need to take account of both the view direction (geometry obstruction) and the light direction vector (geometry shadowing). We can take both into account using Smith's method:

$$G(n, v, l, k) = G_{\text{sub}}(n, v, k) G_{\text{sub}}(n, l, k)$$

The geometry function scales between [0.0, 1.0], where 1.0 (white) indicates no microfacet shadowing and 0.0 (black) signifies complete microfacet shadowing. The geometry function translates to the following code:

```
1   float GeometrySchlickGGX(float NdotV, float roughness)
2   {
3       float r = (roughness + 1.0);
4       float k = (r*r) / 8.0;
5
6       float nom   = NdotV;
7       float denom = NdotV * (1.0 - k) + k;
8
9       return nom / denom;
10  }
11  float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
12  {
13      float NdotV = max(dot(N, V), 0.0);
14      float NdotL = max(dot(N, L), 0.0);
15      float ggx2 = GeometrySchlickGGX(NdotV, roughness);
16      float ggx1 = GeometrySchlickGGX(NdotL, roughness);
17
18      return ggx1 * ggx2;
19  }
```

4.1.3 Fresnel Function F

Using the Fresnel equation, we learned how the reflection-to-refraction ratio varies with the angle of incidence which is the angle between the normal and the ray of light. It determines the percentage of light reflected when it impacts a surface, allowing us to calculate the refracted light based on energy conservation. Materials exhibit a base reflectivity at the normal viewing angle but show increased reflectivity at glancing angles. This can be seen by looking at surfaces such as desks, where reflections are more pronounced as the viewing angle approaches 90 degrees. This effect, known as Fresnel, occurs with all surfaces fully reflecting light at perfect right angles. While the Fresnel equation itself is complex, it's commonly approximated by the Fresnel-Schlick approximation:

$$F_{\text{Schlick}}(h, v, F_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5$$

In code, the Fresnel Schlick approximation translates to:

```
vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
return F0 + (1.0 - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0), 5.0);
}
```

The irradiance map captures the diffuse component of the reflectance integral from all indirect environmental light, treating both diffuse and specular indirect lighting as ambient light. To appropriately account for the diffuse component, the Fresnel equation is used to assess the surface's indirect reflectance ratio, from which the diffuse ratio is derived.

Unlike direct lighting, where the Fresnel response can be determined by a halfway vector, ambient light's omnidirectional nature precludes a singular vector for Fresnel calculations. Instead, Fresnel is calculated using the angle between the normal and view vector, disregarding surface roughness, which typically results in a high reflective ratio. However, acknowledging that rough surfaces reflect less at the edges, a roughness term can be incorporated into the Fresnel-Schlick equation, a method proposed by Sébastien Lagarde:

```
vec3 fresnelSchlickRoughness(float cosTheta,
                             vec3 F0,
                             float roughness)
{
    return F0 + (max(vec3(1.0 - roughness), F0) - F0) * pow(clamp
    (1.0 - cosTheta, 0.0, 1.0), 5.0);
}
```

4.1.4 PBR Shader Main Function

The first part of the main function retrieves material properties, which are found in different texture maps. This process is done by sampling from pre-loaded 2D textures at the fragment's texture coordinates. Gamma correction is performed on the albedo map to linearize the color values for accurate lighting calculations. Then the shader dynamically calculates the normals from the normalMap by using a method of converting tangent space normals to world space. This allows for enhanced surface detailing by accounting for lighting correctly across the 3D geometry's orientation. In the following snippet, N represents the surface normal in world space, and V is the view direction from the camera position to the fragment's world position. It also prepares the reflection vector R for calculating specular reflections later on.

```
void main() {
    vec3 albedo = pow(texture(albedoMap, TexCoords).rgb,vec3(2.2));
    float metallic = texture(metallicMap, TexCoords).r;
    float roughness = texture(roughnessMap, TexCoords).r;
    float ao = texture(aoMap, TexCoords).r;
    float specularValue = texture(specularMap, TexCoords).r;

    vec3 N = getNormalFromMap();
    vec3 V = normalize(camPos - WorldPos);
    vec3 R = reflect(-V, N);
    ...
```

The shader computes the Fresnel reflectance at normal incidence (F_0) based on the surface albedo and its metallic property, which is later adjusted by the specular map value. This step is crucial in simulating how different materials reflect light, where metals have a higher reflectance and a colored tint to their reflections compared to non-metals. In our case, the skin's metallic value is set to 0.

```
void main() {
    ...
    vec3 F0 = vec3(0.04);
    F0 = mix(F0, albedo, metallic);
    vec3 F0WithSpecular = mix(F0, vec3(specularValue), metallic);
    ...
```

The direct lighting loop iterates over all the light sources and computes the contribution of each given light according to the Cook-Torrance BRDF equation. The loop accumulates the diffuse and specular components for each light source which are reduced by distance. It also accounts for subsurface scattering, making the final render slightly more authentic. The loop is customizable and depends on the number of rendered lights in the scene.

```
void main(){
    ...
    vec3 Lo = vec3(0.0);
    for(int i = 0; i < 4; ++i)
    {
        // calculate per-light radiance
        vec3 L = normalize(lightPositions[i] - WorldPos);
        vec3 H = normalize(V + L);
        float distance = length(lightPositions[i] - WorldPos);
        float attenuation = 1.0 / (distance * distance);
        vec3 radiance = lightColors[i] * attenuation;

        // Cook-Torrance BRDF
        float NDF = DistributionGGX(N, H, roughness);
        float G   = GeometrySmith(N, V, L, roughness);
        vec3 F    = fresnelSchlick(max(dot(H, V), 0.0),
    F0WithSpecular);

        vec3 numerator    = NDF * G * F;
        float denominator = 4.0 * max(dot(N, V), 0.0) * max(dot(N,
    L), 0.0) + 0.0001;
        vec3 specular = numerator / denominator;

        // kS is equal to Fresnel
        vec3 kS = F;
        vec3 kD = vec3(1.0) - kS;
        kD *= 1.0 - metallic;

        // scale light by NdotL
        float NdotL = max(dot(N, L), 0.0);

        // Subsurface scattering term (approximated)
        vec3 sssDiffuse = sssColor * (1.0 - exp(-thickness *
    sssScale * NdotL));

        // add to outgoing radiance Lo
        Lo += (kD * albedo / PI + specular + sssDiffuse) * radiance
     * NdotL;
    }
    ...
```

Direct lighting is then followed by shader processing of the ambient lighting using IBL techniques. It samples the irradianceMap to simulate diffuse lighting from the environment and uses the prefilterMap and brdfLUT for environment reflections. This part 'places' the object and its environment into the scene with its surroundings according to the roughness and view direction.

```
1 void main{
2     ...
3     vec3 irradiance = texture(irradianceMap, N).rgb;
4     vec3 diffuse = irradiance * albedo;
5     vec3 prefilteredColor = textureLod(prefilterMap, R, roughness *
      MAX_REFLECTION_LOD).rgb;
6     vec2 brdf = texture(brdfLUT, vec2(max(dot(N, V), 0.0),
     roughness)).rg;
7     vec3 specularEnvironment = prefilteredColor * (F * brdf.x +
     brdf.y);
8     ...
```

Last but not least, the shader combines the calculated ambient, diffuse, and specular components alongside the direct light. The final output is then passed through HDR tonemapping to address issues with high dynamic range lighting scenarios and gamma correction to convert linear color space values back to sRGB for display purposes. This last step makes sure what the viewer sees in the end is as close as possible to the artist's intention, with realistic lighting and material appearances.

```
1 void main{
2     ...
3     vec3 color = ambient + Lo;
4     color = color / (color + vec3(1.0));
5     color = pow(color, vec3(1.0/2.2));
6     FragColor = vec4(color, 1.0);
7 }
```

5 | Model Render

We can see the finished product by building the project and rendering the scene in Visual Studio which is available on the *GitHub repository*.

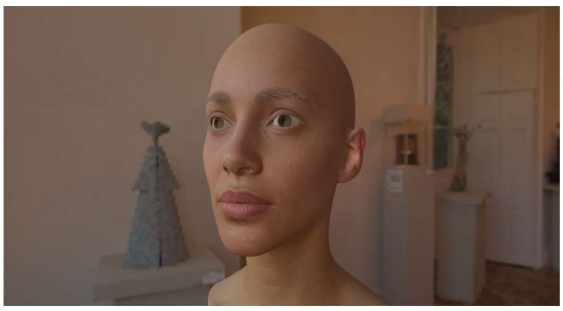

Figure 5.1: Top-right view

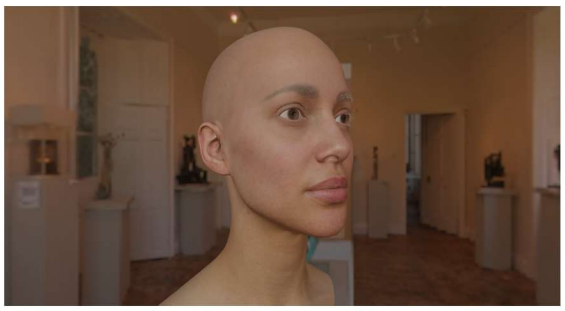

Figure 5.2: Top-left view

Figure 5.3: Bottom-right view

Figure 5.4: Bottom-left view

References

[1] Heidrich, Wolfgang, and Eugene d'Eon. "Advanced Techniques for Realistic Real-Time Skin Rendering." In GPU Gems 3, edited by Hubert Nguyen, 319-342. Springer, 2008. Available online at https://link.springer.com/chapter/10.1007/978-3-642-03452-7_1

[2] d'Eon, Eugene. 2007. "NVIDIA Demo Team Secrets–Advanced Skin Rendering." Presentation at Game Developer Conference 2007. Available online at http://developer.download.nvidia.com/presentations/2007/gdc/Advanced_-Skin.pdf

[3] "Different Maps in PBR Textures", A23D, available online at https://www.a23d.co/blog/different-maps-in-pbr-textures

[4] "GPU Gems 3 - Chapter 14. Advanced Techniques for Realistic Real-Time Skin Rendering", NVIDIA Developer, available online at https://developer.nvidia.com/gpugems/gpugems3/part-iii-rendering/chapter-14-advanced-techniques-realistic-real-time-skin

[5] "LearnOpenGL," available online at https://learnopengl.com/

[6] Sébastien Lagarde "Adopting a physically based shading model", available online at https://seblagarde.wordpress.com/2011/08/17/hello-world/