Eötvös Loránd Tudományegyetem

Faculty of Informatics

Department of Algorithms and their Applications

Efficient Simulation of Bidirectional Reflectance Distribution Functions

Supervisor:
Csaba Bálint
Assistant Professor, MSc

Author:
Asseel Ghaith Jassem Al-Mahdawi
Computer Science BSc

Budapest, 2023

Topic Declaration

Both production and real-time graphics rely on high-performance rendering methods. In computer graphics, realistic images are produced by an accurate light and material interaction calculation. This interaction is defined as the Bidirectional Reflectance Distribution Function or BRDF for short. Since it is used in various applications, for instance, game engines, movie production graphics, and many other applications, there are several BRDF model implementations with different trade-offs, ranging from accuracy to speed.

The BRDF is the keystone of the rendering equation; thus, any lighting simulation process since it describes the light reflection at a surface location. As a result, this study introduces a method to create an efficient simulation of BRDFs by simulating raytracing on a generated surface of microfacets. Then, the simulated BRDF is utilized and compared to some existing methods in a few sample scenes.

Regarding the implementation, we apply ray tracing with Monte-Carlo sampling on environment maps for numerical integration of the rendering equation. Then we visualize objects along with the material textures and world sky-map. Lastly, we present various efficiency enhancements to the BRDF simulation.

The implementation is based on the Nvidia Falcor framework in C++ with HLSL shading language. In conclusion, we implement a more efficient simulation of BRDF and create an application that may be applied for teaching and researching BRDFs.

Budapest, 2022. 11. 29

Table of Content

1. Introduction

1.1 Motivation

The Bidirectional Reflectance Distribution Function or BRDF, is the keystone of the rendering equation, and thus of any lighting simulation process since it describes the light reflection at a surface location. Consequently, building a comprehensive BRDF representation suitable to explain, describe, and/or simulate all complex phenomena involved in this process remains an active topic both in physics and computer graphics [1].

A lot of methods have been proposed and considered as a solution to many of these problems and many of them are standards now. Yet, studies are still ongoing and papers are published yearly to enhance one of the ideas or to solve a specific problem.

In this thesis, we suggest a method that simulates BRDF using a surface of microfacets and textures. Then we compare this method with a Micro-Facets Based Model called Cook Torrance. Lastly, this study is inspired by the work of Rosana Montes and Carlos Ureña [2].

1.2 Overview

Simulating the interaction between light and material in a virtual environment using a physical-based model can be expensive, time-consuming task, and dependent on various parameters such as the light wavelength, as well as, the surface properties that the light is interacting with. The surface can be reflective, rough, and light-absorbent.

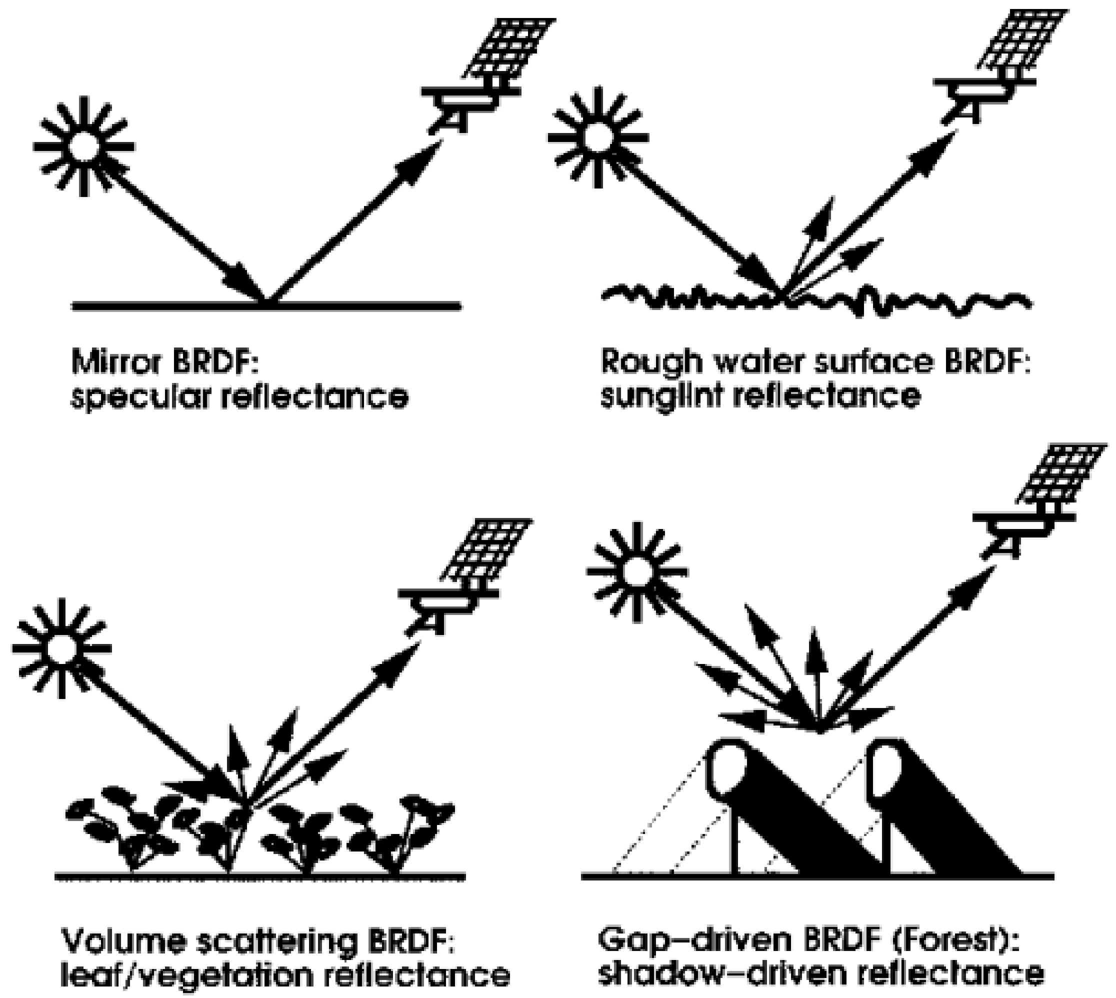

Mirror BRDF:
specular reflectance
Rough water surface BRDF:
sunglint reflectance
Volume scattering BRDF:
leaf/vegetation reflectance
Gap–driven BRDF (Forest):
shadow–driven reflectance

Figure 1.1: Types of irradiance scattering based on the properties of the material, such as roughness. Proposed by Wolfgang Luch in 1997

There are various physics-based models, which favor speed over accuracy and vice versa. Hence, some physics-based models are not recommended for real-time applications that do not require a lot of accuracy, like video games. In the previously mentioned phenomenon, the model's name is called the Bidirectional Reflectance Distribution Function, or BRDF for short. It returns the ratio of reflected radiant flux; hence, it defines the ratio of the color reflected towards a certain direction.

Hereafter, in this study we only deal with the micro-facet model, which means if the surface is rough and bumpy, the surface will scatter the light in different directions, thus, the surface is not reflective. We introduce a simulation of the BRDF, which does pre-calculation and returns a rough estimation. Lastly, we compare it with a micro-facet BRDF model called Cook Torrance.

1.3 Structure

The structure of this thesis comprises of five chapters. It starts with the introduction (Chapter 1), explaining the motivation, providing applications, and context for the reader. In Chapter 2, it provides an extensive user guide for the associated application, with both microfacets surface GUI and model GUI manuals.

. A theoretical background (Chapter 3) is provided, which offers essential definitions and algorithms required to comprehend the subsequent chapters. The implementation (Chapter 4) discusses the building and compiling process of Falcor Framework project, as well as the Bidirectional Reflectance Distribution functions implementation. The testing and validation are covered in chapter 4, as well. Lastly, the conclusion of the work is formulated (Chapter 5).

The developer documentation is divided into different chapters, facilitating improved indexing, and enhancing reader convenience. Specific aspects of the application are located by directly navigating to the relevant section of the thesis that aligns with the reader's specific interests.

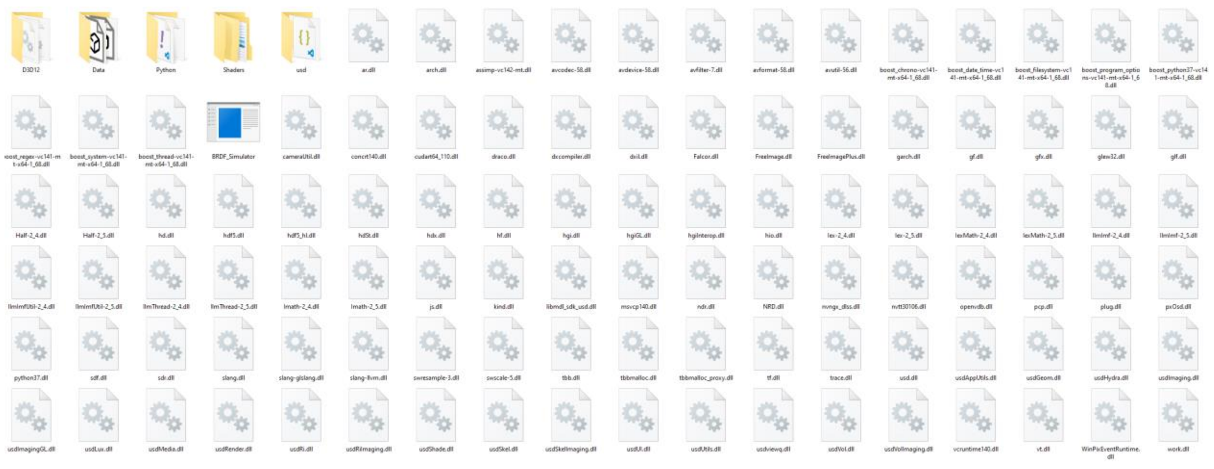
2. User Documentation

The application has two Graphical User Interfaces (GUI): A GUI for preparing the BRDF by storing it in a sky-map using a surface of microfacet, and another GUI for running the simulation and comparing it with another BRDF model on various scene models.

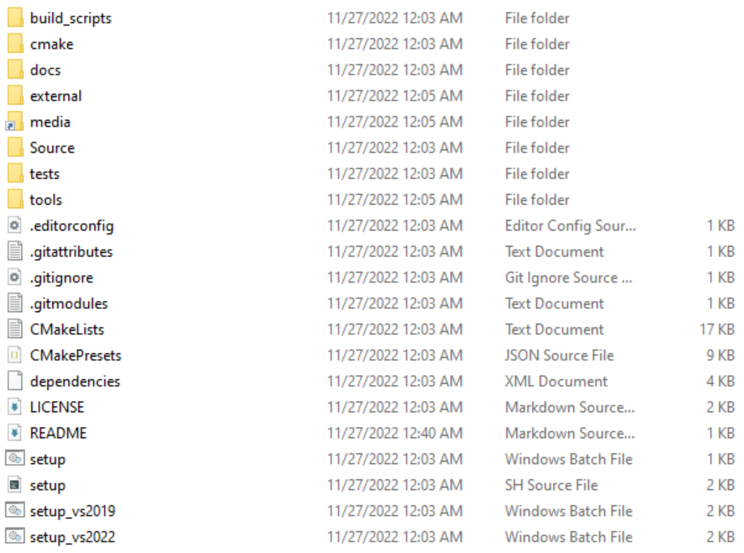
This chapter covers the installation of the application and presents a user guide to deliver an understanding of the application and its tools.

2.1 Installation

The application developed with this thesis is called Efficient Simulation of BRDF [3], and it is hosted on *GitHub*, hence, making it widely accessible. The software installation can be done in two ways. As the software is made to be an open-source project, the user can either download the built version of the software or download the source code and build the project. Moreover, if the user wants to use an already-built version of the software, it can be downloaded as well. The user can download the built in version and run *BRDF_Simulator.exe* to start the application. Moreover, the building, steps are explained in the repository *readme.md* file, and in subsection 4.1. The extracted release built of the application comprises the folders and files shown in Figure 2.1a, while the Falcor Framework source code is presented in Figure 2.1b. Finally, the BRDF_Simulator source code can be found in ***Source/Samples/BRDF_Simulator***, they are presented in Figure 2.1c.

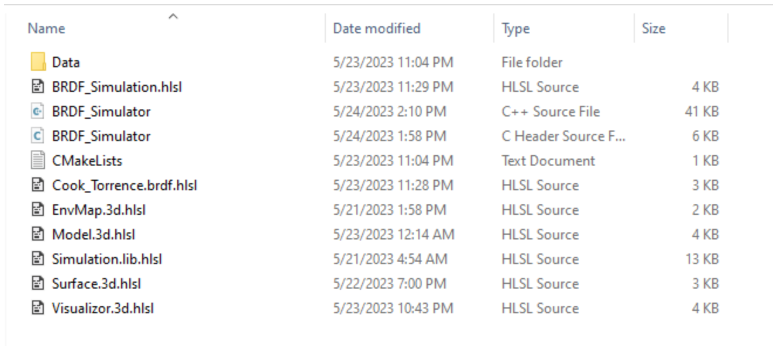


(a) The application Files and Directories after installation.



build_scripts	11/27/2022 12:03 AM	File folder	
cmake	11/27/2022 12:03 AM	File folder	
docs	11/27/2022 12:03 AM	File folder	
external	11/27/2022 12:05 AM	File folder	
media	11/27/2022 12:05 AM	File folder	
Source	11/27/2022 12:03 AM	File folder	
tests	11/27/2022 12:03 AM	File folder	
tools	11/27/2022 12:05 AM	File folder	
.editorconfig	11/27/2022 12:03 AM	Editor Config Sour...	1 KB
.gitattributes	11/27/2022 12:03 AM	Text Document	1 KB
.gitignore	11/27/2022 12:03 AM	Git Ignore Source ...	1 KB
.gitmodules	11/27/2022 12:03 AM	Text Document	1 KB
CMakeLists	11/27/2022 12:03 AM	Text Document	17 KB
CMakePresets	11/27/2022 12:03 AM	JSON Source File	9 KB
dependencies	11/27/2022 12:03 AM	XML Document	4 KB
LICENSE	11/27/2022 12:03 AM	Markdown Source...	2 KB
README	11/27/2022 12:40 AM	Markdown Source...	1 KB
setup	11/27/2022 12:03 AM	Windows Batch File	1 KB
setup	11/27/2022 12:03 AM	SH Source File	2 KB
setup_vs2019	11/27/2022 12:03 AM	Windows Batch File	2 KB
setup_vs2022	11/27/2022 12:03 AM	Windows Batch File	2 KB

(b) The application source Files and Directories found in the project repository.



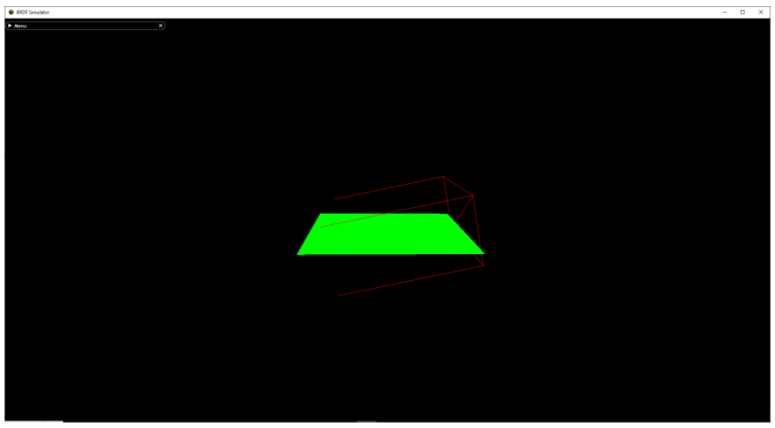
Name	Date modified	Type	Size
Data	5/23/2023 11:04 PM	File folder	
BRDF_Simulation.hlsl	5/23/2023 11:29 PM	HLSL Source	4 KB
BRDF_Simulator	5/24/2023 2:10 PM	C++ Source File	41 KB
BRDF_Simulator	5/24/2023 1:58 PM	C Header Source F...	6 KB
CMakeLists	5/23/2023 11:04 PM	Text Document	1 KB
Cook_Torrence.brdf.hlsl	5/23/2023 11:28 PM	HLSL Source	3 KB
EnvMap.3d.hlsl	5/21/2023 1:58 PM	HLSL Source	2 KB
Model.3d.hlsl	5/23/2023 12:14 AM	HLSL Source	4 KB
Simulation.lib.hlsl	5/21/2023 4:54 AM	HLSL Source	13 KB
Surface.3d.hlsl	5/22/2023 7:00 PM	HLSL Source	3 KB
Visualizor.3d.hlsl	5/23/2023 10:43 PM	HLSL Source	4 KB

(c)The source files of the BRDF_Simulator

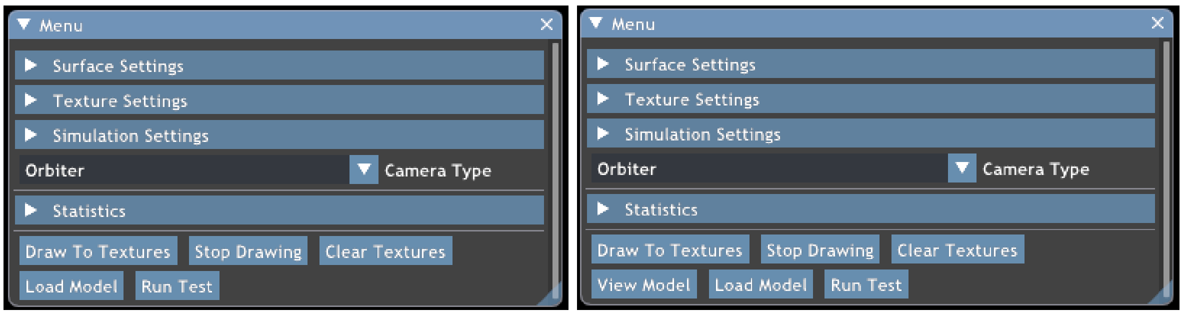
Figure 2.1: Directories and Files of **Falcor Framework**, and the **BRDF_Simulator**.

2.2 Microfacets Surface (GUI)

The user can start the application by executing the *BRDF_Simulator.exe* file. It initializes the application with the default behavior, which loads a window as seen in Figure 2.2a. The window contains a settings menu which is spawned on the left (Figure 2.2b), and a canvas, which the surface scene is viewed through. The menu holds ***Camera Type*** options, four sub-menus, which are the ***Surface Settings***, ***Texture Settings***, ***Simulation Settings***, and ***Statistics***, and ***six buttons***. The ***Camera Types*** are explained in subsection 2.2.1 and Table 2.1. The ***Surface Settings*** are explained in subsection 2.2.2, and the ***Texture Settings*** along with the ***Statistics*** are elaborated in subsection 2.2.3. Finally, the ***Simulation Settings*** with the ***Simulation Buttons*** are elaborated in subsection 2.2.4.



a)The main window of the application



b)The menu provided by this scene.

Figure 2.2: A picture from within the Surface Scene GUI. It shows the application's main window and the menu settings for this scene.

2.2.1 Camera Types

It is possible to navigate through the scene and view objects from different aspects. The movements involve the use, of both, the mouse and the keyboard. However, it differs based on the camera type. They are presented in Figure 2.3. For example, the ***orbiter type*** only involves the mouse, unlike ***First person*** and ***6-DoF***, which require key and mouse interaction. More precisely, the keys ***W***, ***A***, ***S***, ***D***, ***Q***, ***E***, and ***Shift*** on the keyboard, as well as, the ***left mouse button***, and the ***right mouse button*** in case of ***6-DoF***. Table 2.1 presents information about each camera type, their intended type of interaction, and the impact on the movement of the camera within the rendered world space.

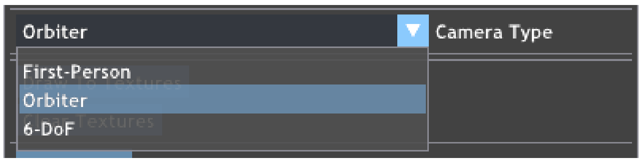


Figure 2.3: Camera Types used in the application.

Camera		
Type	**Type of Interaction**	**Description**
Orbiter	***Left** mouse button hold and drag*: To move in any direction. ***Scroll in** or **out***: To zoom in or out.	Move the camera around the model while looking at it. Furthermore, it is the default camera.
First Person	***W** press or hold:* To move forward. ***A** press or hold:* To move leftward. ***S** press or hold:* To move backward. ***D** press or hold:* To move rightward.	Move the camera in any direction without looking at the object.

	Q *press or hold:* To move downward. ***E*** *press or hold:* To move upwards. ***Shift*** *press and hold:* To move faster. ***Ctrl*** *press and hold:* To move slower.	
6–DoF	(Same types of the interaction of First Person Camera). ***Right*** *mouse button hold and drag*: Rotate the camera around itself.	Move the camera in any direction and rotate it around itself without looking at the object.

Table 2.1: Mapping camera types, their input keys, their intended type of intersection, and the impact on the camera movement.

2.2.2 Surface Settings

This subsection contains the surface manipulation settings. The ***Surface Settings*** sub-menu contains **two** input fields. See Figure 2.4.

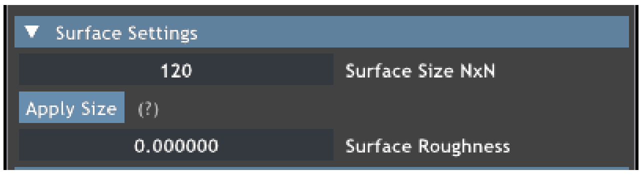
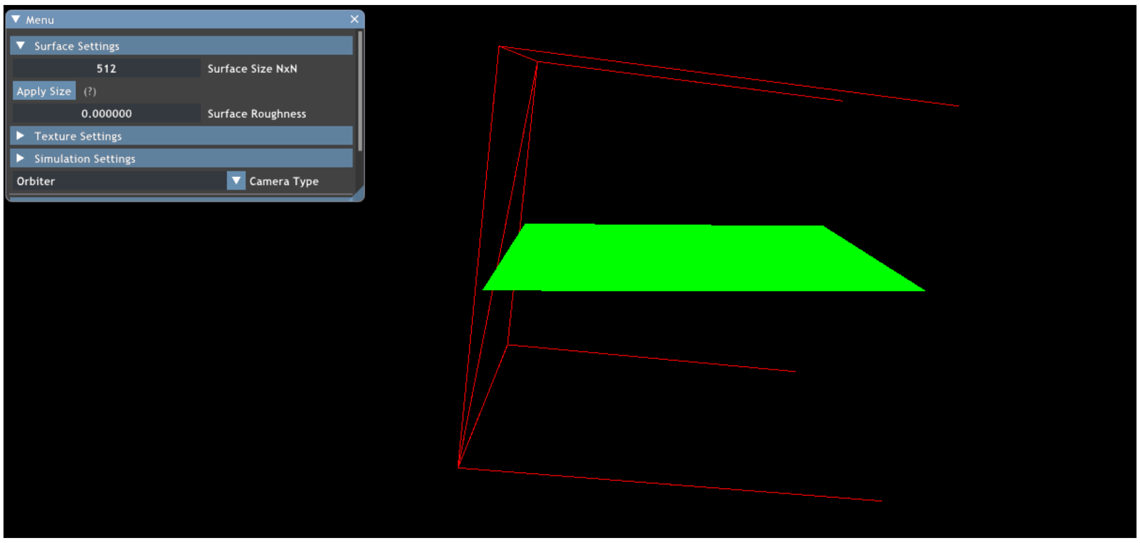


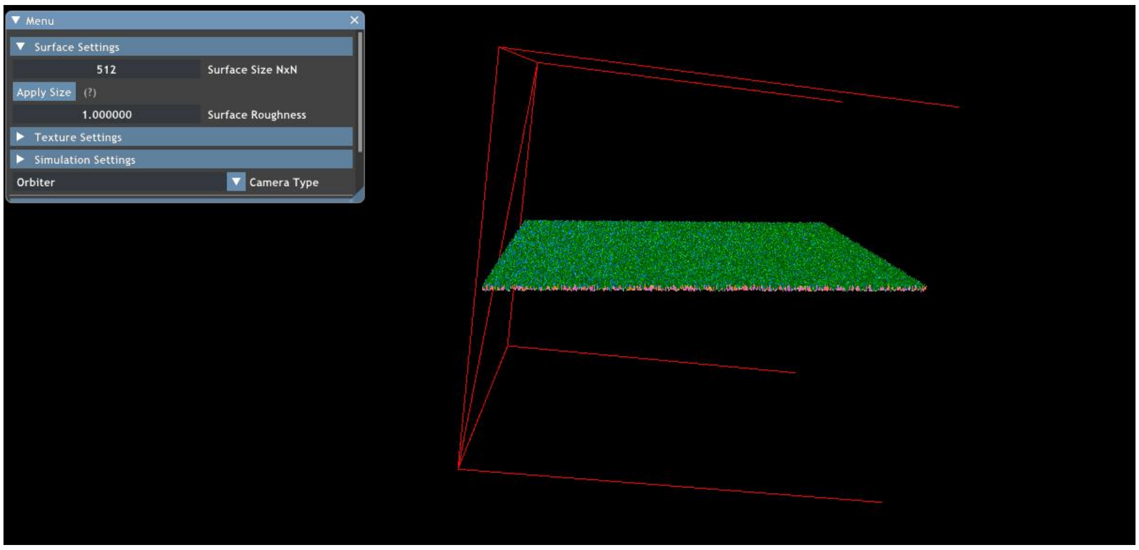
Figure 2.4: A picture that shows surface settings components.

The ***Surface Roughness*** field is responsible for determining the surface's height map. This field value ranges from 0 to 1 as represented in Figures 2.5a and 2.5b, respectively. Where 0 means the surface is smooth and it causes perfect reflection when the rays hit it, and 1 means the surface has maximum roughness and it causes the rays to be scattered randomly.



512 Surface Size NxN
0.000000 Surface Roughness
Orbiter

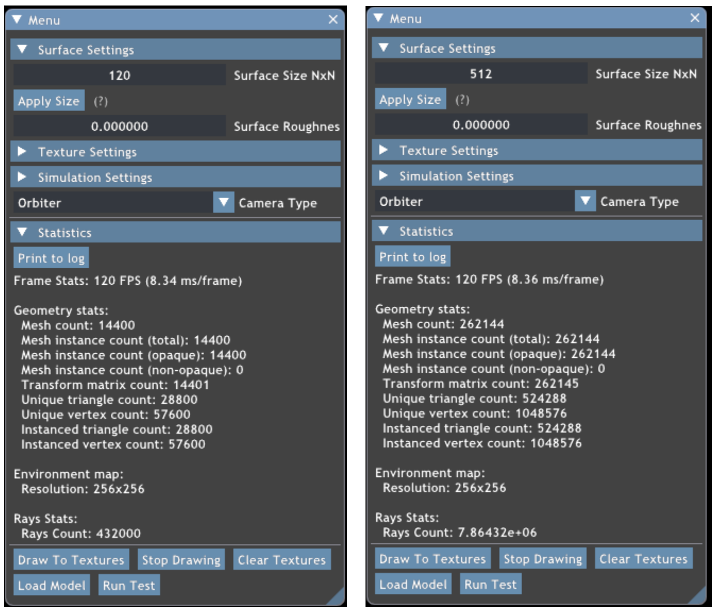
Figure 2.5a: Shows the surface when the roughness value is zero.



512 Surface Size NxN
1.000000 Surface Roughness
Orbiter

Figure 2.5b: Shows the surface when the roughness value is one.

The ***Surface Size*** input field is responsible for determining the size of the surface. Its value ranges from **120**, which is the minimum size, to **512**, which is the maximum value. After setting up the ***Surface Size*** the user must click on ***Apply Size*** to update the surface with the new size. The user can check the logs in the ***Statistics*** sub-menu to see if the surface was updated or not. See Figure 2.6. ***Statistics*** are elaborated on more in-depth in the next subsection (2.2.3).



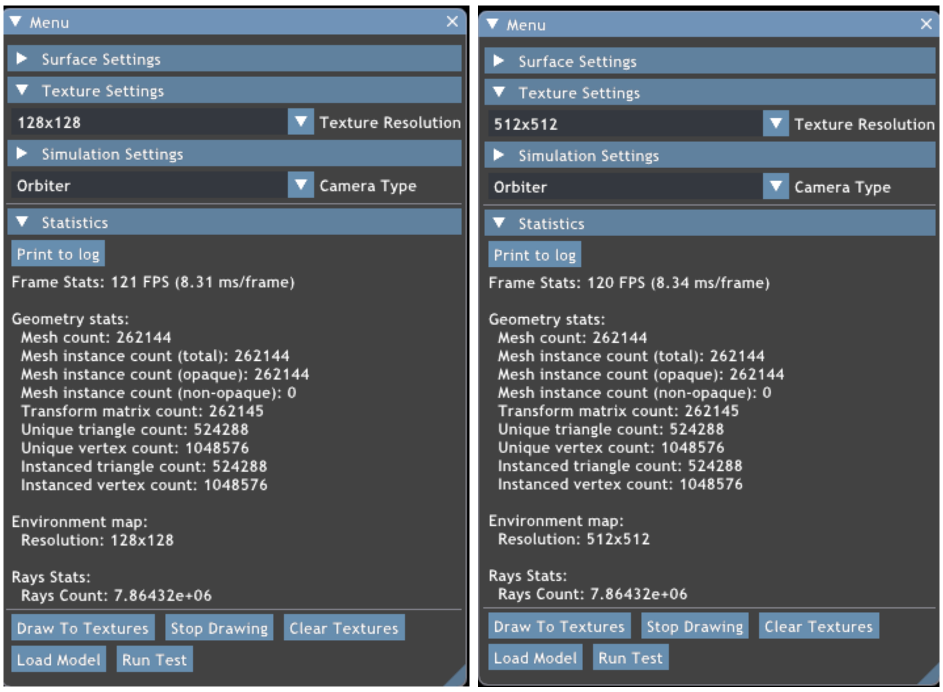
```
Menu
Surface Settings
120                  Surface Size NxN
Apply Size  (?)
0.000000             Surface Roughnes
Texture Settings
Simulation Settings
Orbiter              Camera Type
Statistics
Print to log
Frame Stats: 120 FPS (8.34 ms/frame)

Geometry stats:
 Mesh count: 14400
 Mesh instance count (total): 14400
 Mesh instance count (opaque): 14400
 Mesh instance count (non-opaque): 0
 Transform matrix count: 14401
 Unique triangle count: 28800
 Unique vertex count: 57600
 Instanced triangle count: 28800
 Instanced vertex count: 57600

Environment map:
 Resolution: 256x256

Rays Stats:
 Rays Count: 432000
Draw To Textures  Stop Drawing  Clear Textures
Load Model  Run Test
```

```
Menu
Surface Settings
512                  Surface Size NxN
Apply Size  (?)
0.000000             Surface Roughnes
Texture Settings
Simulation Settings
Orbiter              Camera Type
Statistics
Print to log
Frame Stats: 120 FPS (8.36 ms/frame)

Geometry stats:
 Mesh count: 262144
 Mesh instance count (total): 262144
 Mesh instance count (opaque): 262144
 Mesh instance count (non-opaque): 0
 Transform matrix count: 262145
 Unique triangle count: 524288
 Unique vertex count: 1048576
 Instanced triangle count: 524288
 Instanced vertex count: 1048576

Environment map:
 Resolution: 256x256

Rays Stats:
 Rays Count: 7.86432e+06
Draw To Textures  Stop Drawing  Clear Textures
Load Model  Run Test
```

Figure 2.6: We have 28800 unique triangles when the size of the surface is 120 (Left). We have 524288 unique triangles when the size of the surface is 512 (Right).

2.2.3 Texture Settings and Statistics

This subsection explains how the user can change the texture resolution. Furthermore, it discusses the components of the ***Statistics***. The ***Texture Settings*** consist of only one dropdown list, which is named the ***Texture Resolution***, it consists of five resolutions which are {***32x32***, ***64x64***, ***128x128***, ***256x256***, ***512x512***}. The textures resolution will be updated automatically after selecting the desired resolution, the user can run the simulation to see the difference or by checking the Environment Map Resolution in ***Statistics***. See Figure 2.7.

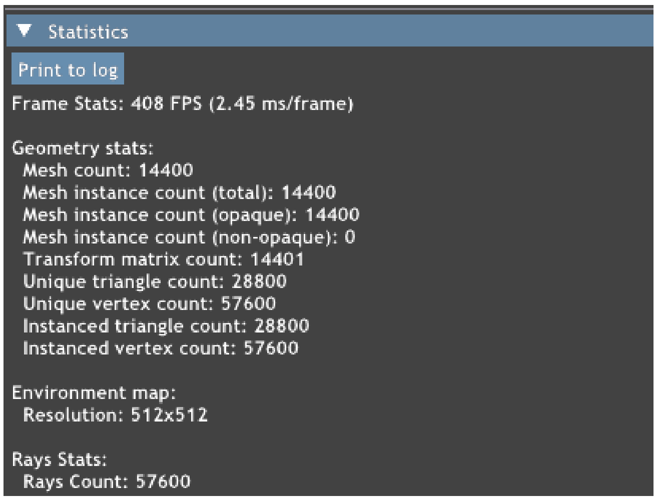


```
128x128   Texture Resolution
Orbiter   Camera Type

Frame Stats: 121 FPS (8.31 ms/frame)

Geometry stats:
  Mesh count: 262144
  Mesh instance count (total): 262144
  Mesh instance count (opaque): 262144
  Mesh instance count (non-opaque): 0
  Transform matrix count: 262145
  Unique triangle count: 524288
  Unique vertex count: 1048576
  Instanced triangle count: 524288
  Instanced vertex count: 1048576

Environment map:
  Resolution: 128x128

Rays Stats:
  Rays Count: 7.86432e+06
```

```
512x512   Texture Resolution
Orbiter   Camera Type

Frame Stats: 120 FPS (8.34 ms/frame)

Geometry stats:
  Mesh count: 262144
  Mesh instance count (total): 262144
  Mesh instance count (opaque): 262144
  Mesh instance count (non-opaque): 0
  Transform matrix count: 262145
  Unique triangle count: 524288
  Unique vertex count: 1048576
  Instanced triangle count: 524288
  Instanced vertex count: 1048576

Environment map:
  Resolution: 512x512

Rays Stats:
  Rays Count: 7.86432e+06
```

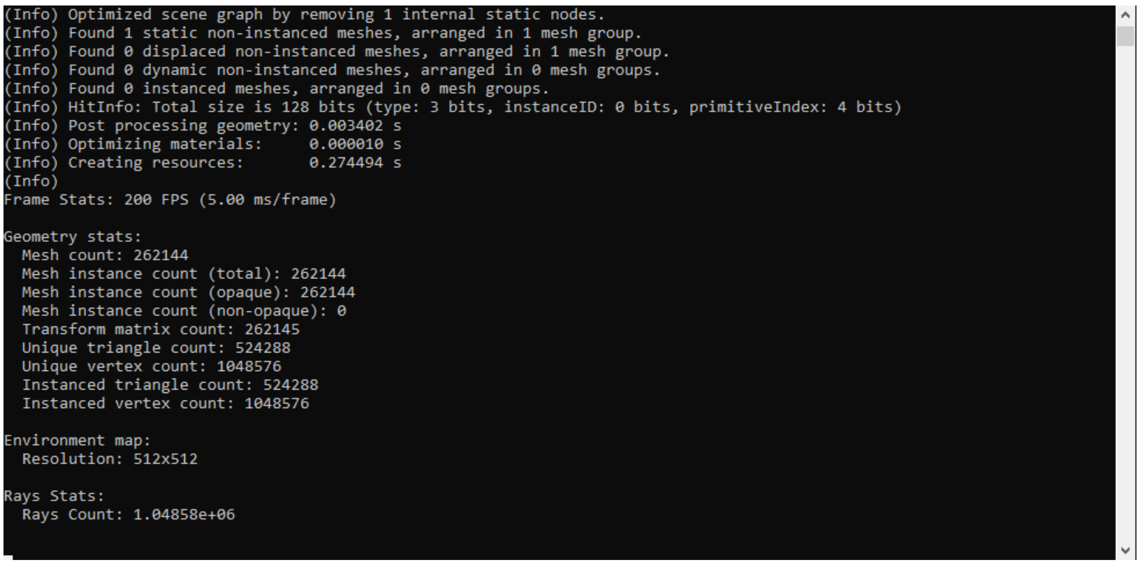
Figure 2.7: A picture that shows how the statistics got updated accordingly. When resolution is 128x128(Left), and when resolution is 512x512 (Right).

The ***Statistics*** subsection displays details regarding the scene content. It is used to measure the rendering speed in terms of *Frames Per Second* and *Milliseconds Per Frame*, *Geometry stats*, such as *Mesh count*, *Unique triangle count*, *Unique vertex count…etc.*, and *Environment map resolution*. Finally, the ***print to log*** button displays the statistics in the console. See Figure 2.8a and Figure 2.8b.



```
Frame Stats: 408 FPS (2.45 ms/frame)

Geometry stats:
  Mesh count: 14400
  Mesh instance count (total): 14400
  Mesh instance count (opaque): 14400
  Mesh instance count (non-opaque): 0
  Transform matrix count: 14401
  Unique triangle count: 28800
  Unique vertex count: 57600
  Instanced triangle count: 28800
  Instanced vertex count: 57600

Environment map:
  Resolution: 512x512

Rays Stats:
  Rays Count: 57600
```

Figure 2.8a: A picture that shows the statistics in the GUI.

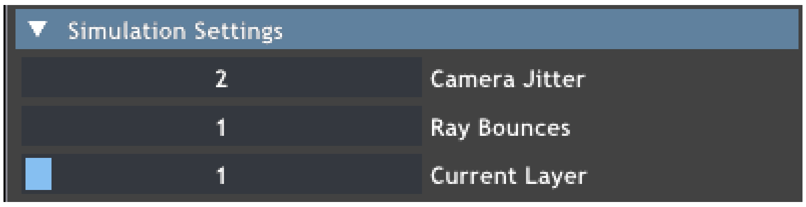


```
(Info) Optimized scene graph by removing 1 internal static nodes.
(Info) Found 1 static non-instanced meshes, arranged in 1 mesh group.
(Info) Found 0 displaced non-instanced meshes, arranged in 1 mesh group.
(Info) Found 0 dynamic non-instanced meshes, arranged in 0 mesh groups.
(Info) Found 0 instanced meshes, arranged in 0 mesh groups.
(Info) HitInfo: Total size is 128 bits (type: 3 bits, instanceID: 0 bits, primitiveIndex: 4 bits)
(Info) Post processing geometry: 0.003402 s
(Info) Optimizing materials:     0.000010 s
(Info) Creating resources:       0.274494 s
(Info)
Frame Stats: 200 FPS (5.00 ms/frame)

Geometry stats:
  Mesh count: 262144
  Mesh instance count (total): 262144
  Mesh instance count (opaque): 262144
  Mesh instance count (non-opaque): 0
  Transform matrix count: 262145
  Unique triangle count: 524288
  Unique vertex count: 1048576
  Instanced triangle count: 524288
  Instanced vertex count: 1048576

Environment map:
  Resolution: 512x512

Rays Stats:
  Rays Count: 1.04858e+06
```

Figure 2.8b: A picture that shows the statistics in the console after clicking ***print to log***

2.2.4 Simulation Settings

This subsection covers the drawing to texture settings. The **Simulation Settings** sub-menu contains **Three** fields as seen in Figure 2.9.

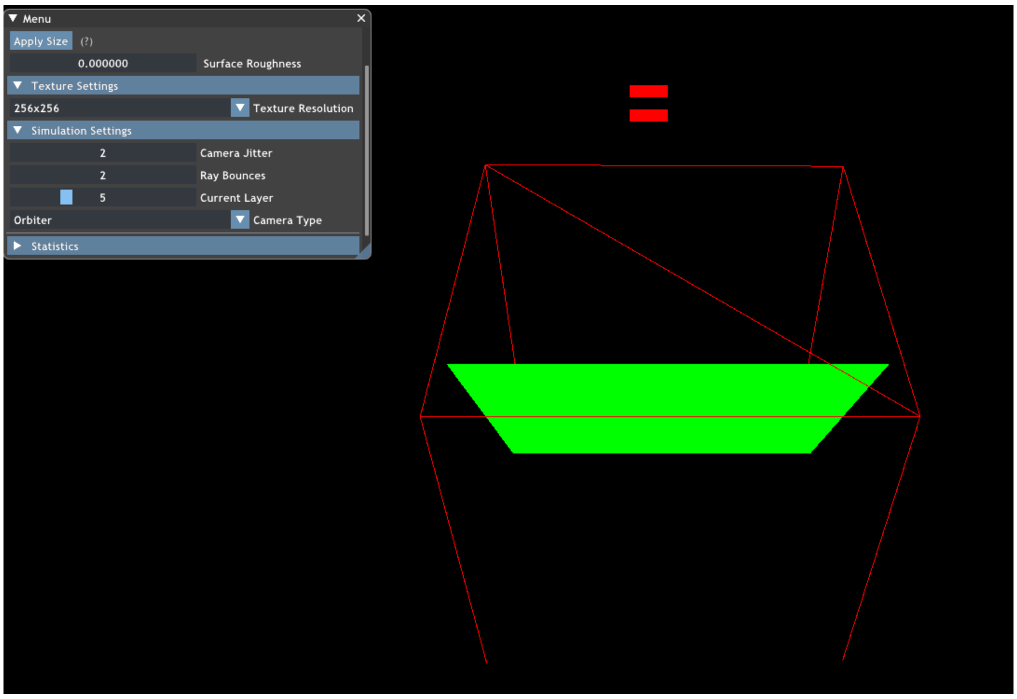


▼ Simulation Settings

2	Camera Jitter
1	Ray Bounces
1	Current Layer

Figure 2.9: Shows the input values of the Simulation Settings

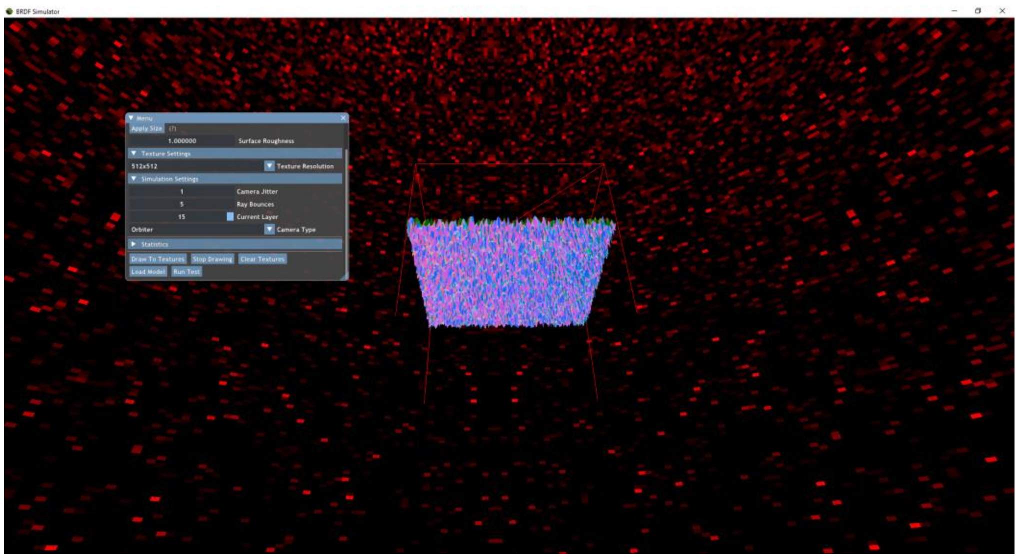
The ***Camera Jitter*** field is responsible for setting up how many times the camera needs to be jittered (Shifted) during the simulation. The minimum jittering value that can be passed is one. Moreover, the camera jittering position is set to be random with less than a pixel size. See Figure 2.10 (Note: in the figure, the jittering position was set to be more than the size of one pixel for demonstration purposes only).



0.000000 Surface Roughness
256x256 Texture Resolution
2 Camera Jitter
2 Ray Bounces
5 Current Layer
Orbiter Camera Type

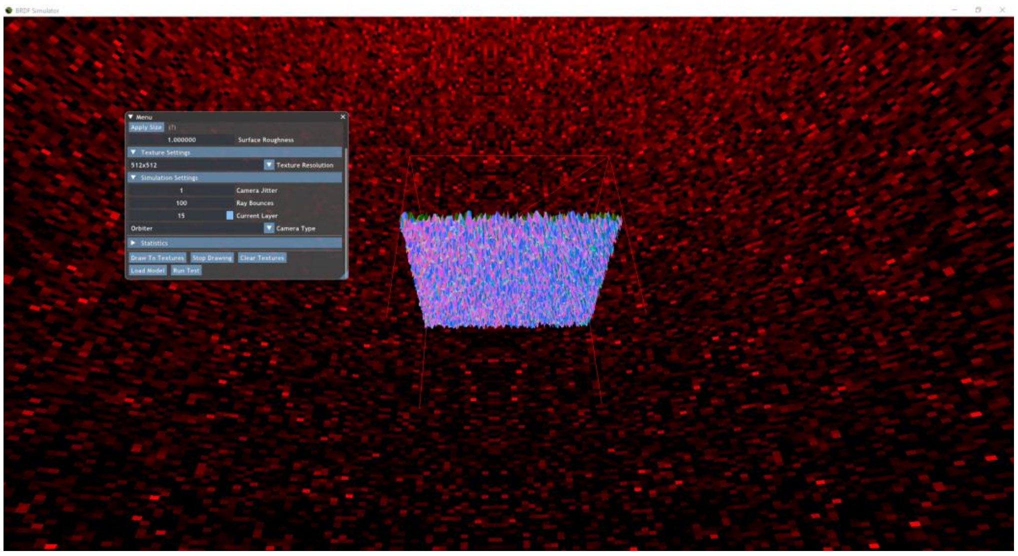
Figure 2.10: A picture shows how the camera was jittered during the simulation (The jittering size is set to be more than one pixel).

The ***Ray Bounces*** field is responsible for setting up the energy of the ray. Therefore, for every bounce the ray loses energy until it is reflected away from the surface, or all of its energy is converted into heat due to multiple collisions. See Figures 2.11a and 2.11b.



1.000000 Surface Roughness
512x512 Texture Resolution
1 Camera Jitter
5 Ray Bounces
15 Current Layer
Orbiter Camera Type

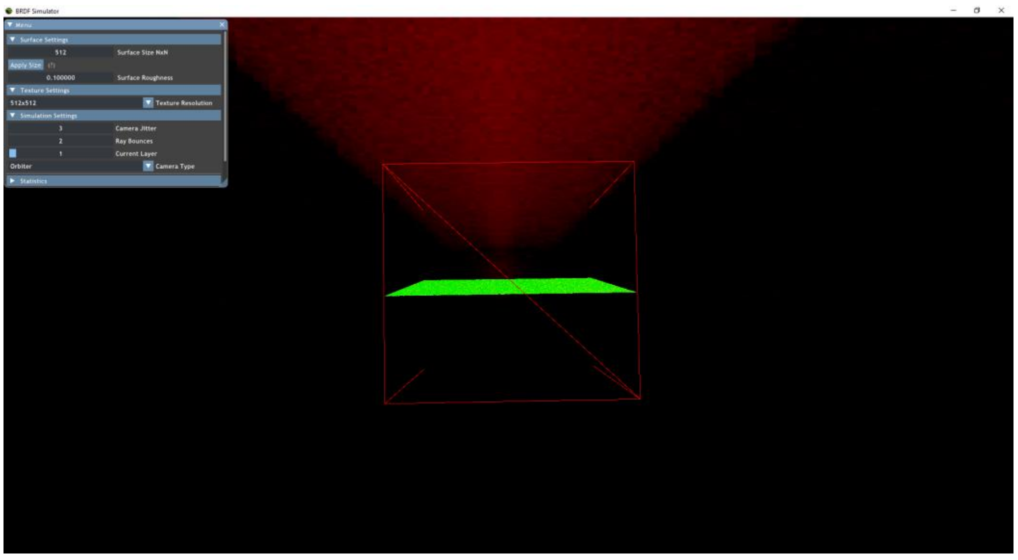
Figure 2.11a: A picture shows how the ray behaves when it has only few bounces.



1.000000 Surface Roughness
512x512 Texture Resolution
1 Camera Jitter
100 Ray Bounces
15 Current Layer
Orbiter Camera Type

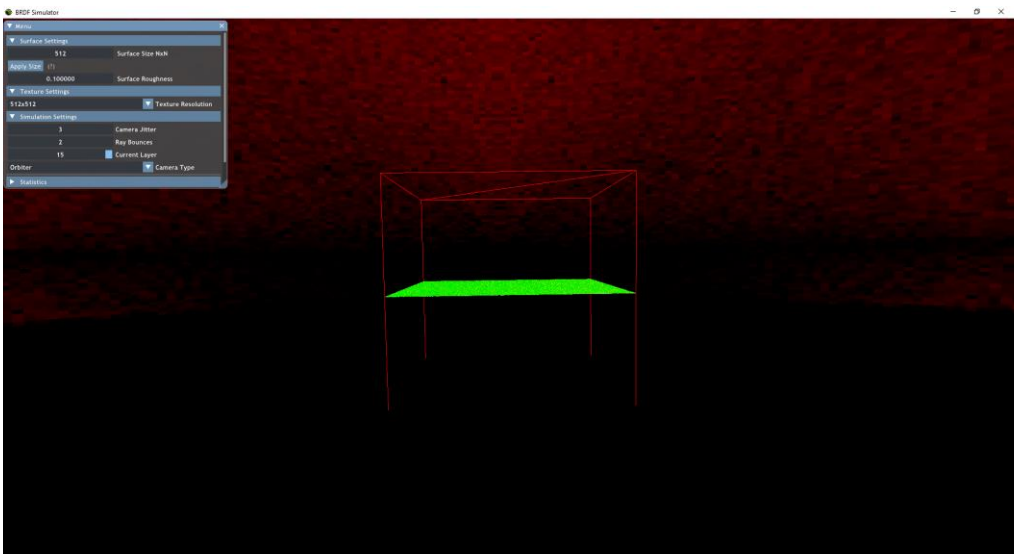
Figure 2.11b: A picture shows how the ray behaves when it has ten bounces.

The ***Current Layer*** field is responsible for choosing the texture of the sky-map, those textures are referred to by **layers** in this thesis. Its value ranges from 1 to 15. Each value is mapped to a different texture as represented in Figures 2.12a, and 2.12b Furthermore, changing the layer will change the incoming rays angle, where 1 is ~0 degrees and 15 is 90 degrees. During the simulation, the layers will be filled in a descending manner starting from layer 15 to layer 1.



512 Surface Size NxN
0.100000 Surface Roughness
512x512 Texture Resolution
3 Camera Jitter
2 Ray Bounces
1 Current Layer
Orbiter Camera Type

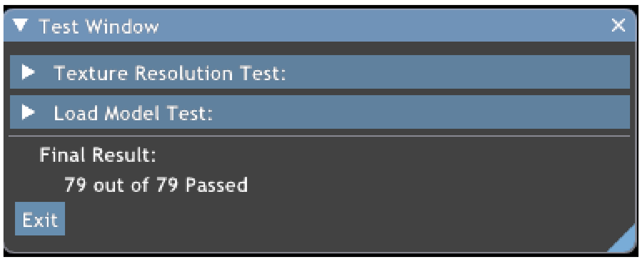
Figure 2.12a: Represents the texture of layer 1 after the simulation.



512 Surface Size NxN
0.100000 Surface Roughness
512x512 Texture Resolution
3 Camera Jitter
2 Ray Bounces
15 Current Layer
Orbiter Camera Type

Figure 2.12b: Represents the texture of layer 15 after the simulation.

The Buttons in this GUI are ***Draw to Texture***, ***Stop Drawing***, ***Clear Texture***, ***View Model***, ***Load Model***, and ***Run Test***. The ***Draw to Textures*** will fill up the layers based on the specified settings that were aforementioned. ***Stop Drawing*** will terminate the process of drawing to texture, and it will clear all the layers. ***Clear Textures*** will clear all the layers. *The* ***Run Test*** button checks the internal functionalities of Texture Resolution, and Loading a Model, see Figure 2.13a, those functionalities are discussed more in-depth in the ***Testing and Validation*** Chapter (Subsection 4.3.2). The ***Load Model*** will allow the user to upload a model as shown in Figures 2.13b and 2.13c, furthermore, it switches from the surface scene to the model scene. Finally, ***View Model*** this button switches to the model view, however, it will not appear unless there is a model already uploaded.



Final Result:
79 out of 79 Passed

Figure 2.13a: Test window after clicking ***Run Test*** button.

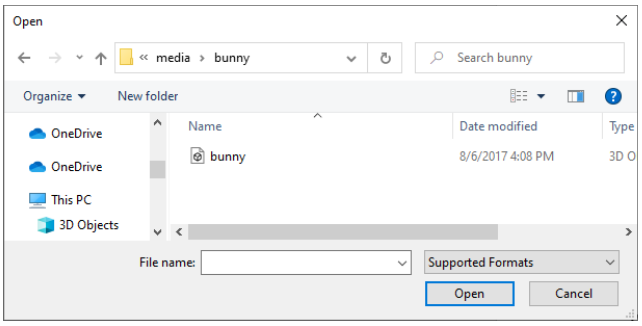


Figure 2.13b: Shows the model uploading window.

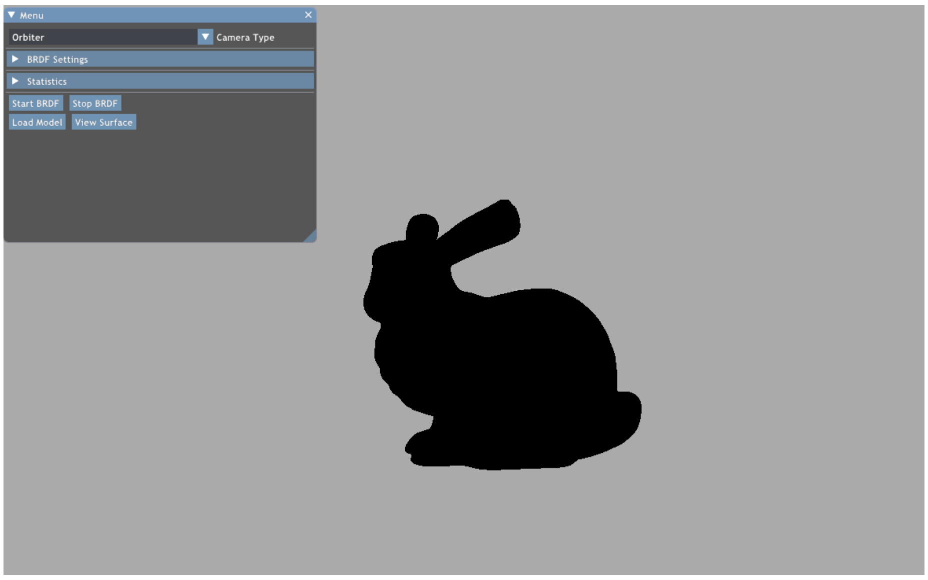


Figure 2.13c: A picture of the uploaded model in the model scene.

2.3 Model Scene (GUI)

In the Model Scene GUI, there are the ***Camera Type***, ***BRDF Settings***, ***Statistics,*** and ***Four buttons***, see Figure 2.14. The ***Camera Type*** was explained in subsection 2.2.1, and ***Statistics*** were elaborated within subsection 2.2.3. Finally, the four buttons are ***Start BRDF***, which will run BRDF based on the chosen model either ***Cook-Torrance*** or the ***Simulated*** BRDF, ***Stop BRDF***, stops running BRDF, ***Load Model***, it loads a new model based on the user's choice, and ***View Surface*** changes the view to the surface. The ***BRDF Settings*** are elaborated in subsection 2.3.1.

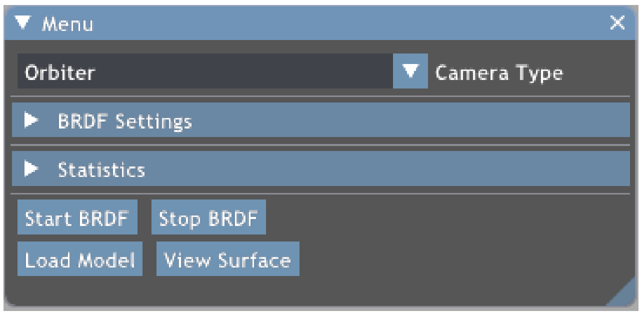


Figure 2.14: A picture of the model scene menu.

2.3.1 BRDF Settings

In this sub-menu, there are two types of ***BRDF*** models, which are the ***Cook Torrance*** model and the ***Simulated*** model as shown in Figure 2.15. ***Cook Torrance*** Model settings are elaborated in Subsection 2.3.1.1. and The ***Simulated*** model settings are explained in Subsection 2.3.1.2.

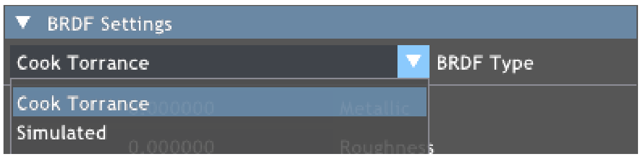
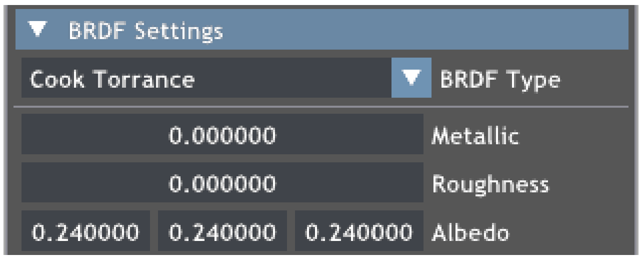


Figure 2.15: Shows the BRDF Types in the application.

2.3.1.1 Cook Torrance Model

When choosing ***Cook Torrance Model*** the menu will be expanded with three fields ***Metallic***, ***Roughness***, and ***Albedo***. See Figure 2.16.


BRDF Settings

Cook Torrance BRDF Type

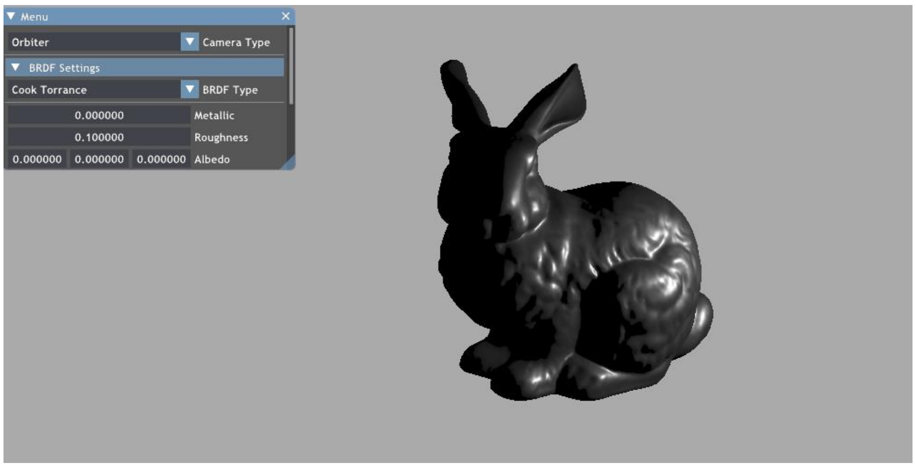
0.000000 Metallic

0.000000 Roughness

0.240000 0.240000 0.240000 Albedo

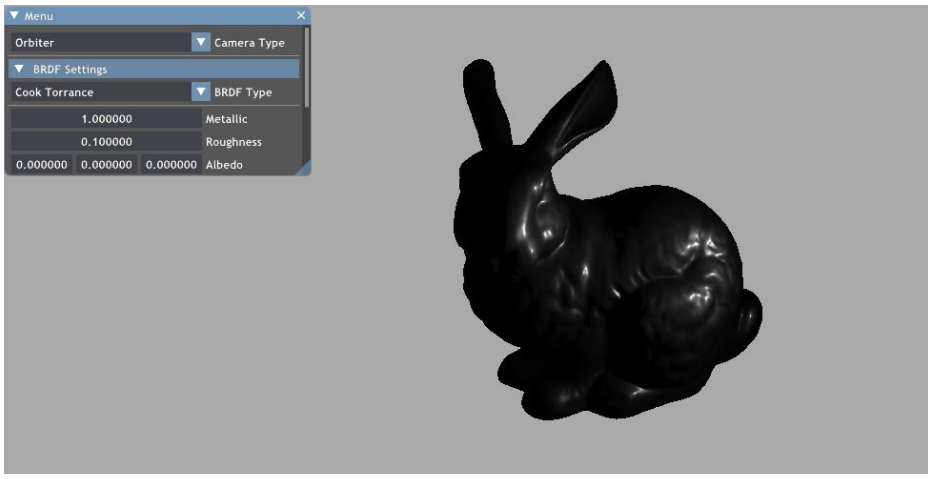
Figure 2.16: A picture of Cook Torrance fields.

The ***Metallic*** field is responsible for determining the material type of the model, the input value ranges from 0 to 1. Where 0 means the model has no metallic characteristics, and 1 means the model is metallic. See Figures 2.17a and 2.17b.



Menu
Orbiter — Camera Type
BRDF Settings
Cook Torrance — BRDF Type
0.000000 — Metallic
0.100000 — Roughness
0.000000 0.000000 0.000000 — Albedo

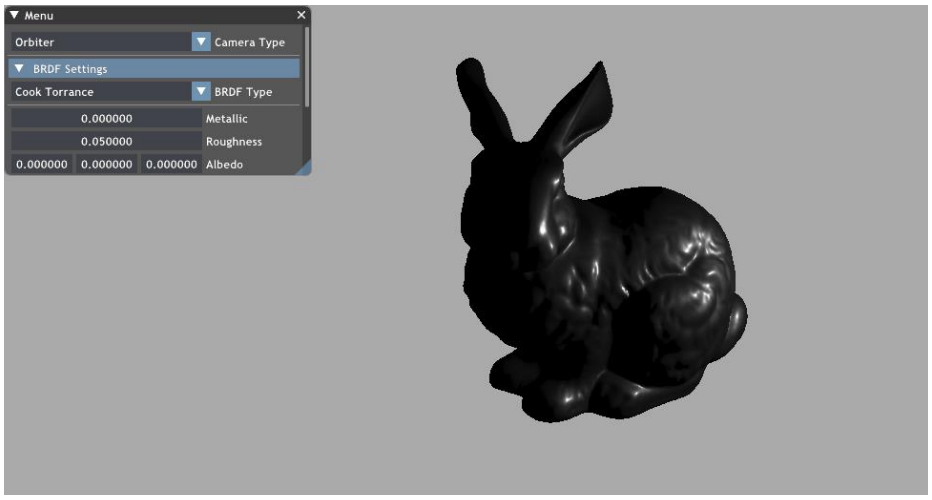
Figure 2.17a: A picture of the bunny model when it is 0% metallic.



Menu
Orbiter — Camera Type
BRDF Settings
Cook Torrance — BRDF Type
1.000000 — Metallic
0.100000 — Roughness
0.000000 0.000000 0.000000 — Albedo

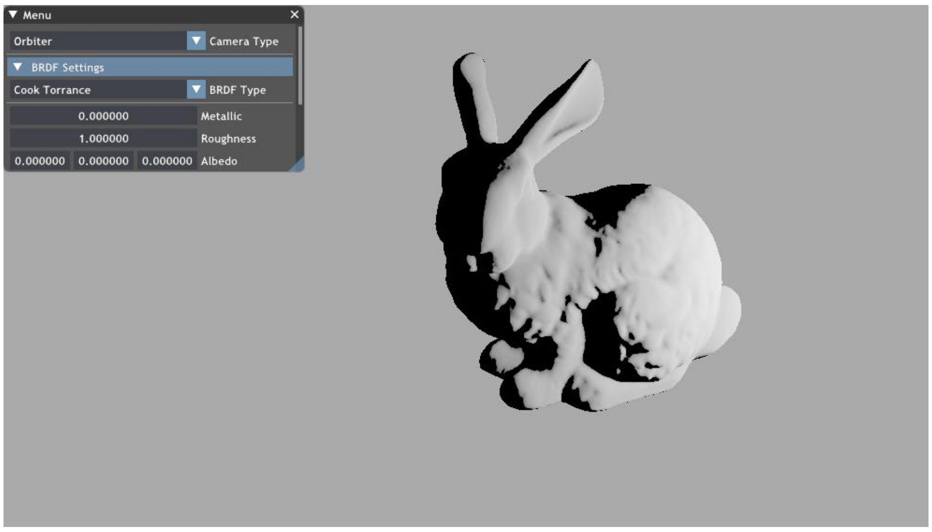
Figure 2.17b: A picture of a bunny model when it is 100% metallic.

The ***Roughness*** field is responsible for determining how rough the model surface is. The input value ranges from 0 to 1. A low roughness creates a defined, sharp highlight, while a high roughness diffuses the specular highlight across the surface. Specular Reflectance adjusts the intensity of the specular highlight. See Figures 2.18a and 2.18b.



Orbiter Camera Type
Cook Torrance BRDF Type
0.000000 Metallic
0.050000 Roughness
0.000000 0.000000 0.000000 Albedo

Figure 2.18a: Shows the model when its roughness is low.



Orbiter Camera Type
Cook Torrance BRDF Type
0.000000 Metallic
1.000000 Roughness
0.000000 0.000000 0.000000 Albedo

Figure 2.18b: Shows the model when its roughness is high.

The ***Albedo*** consists of three input values. Each field corresponds to a coloring channel red, green, and blue, respectively. Each field can be filled with a value from 0 to 1. Zero means the color is not applied and one means the model is 100% colored with the corresponding color. See Figure 2.19.

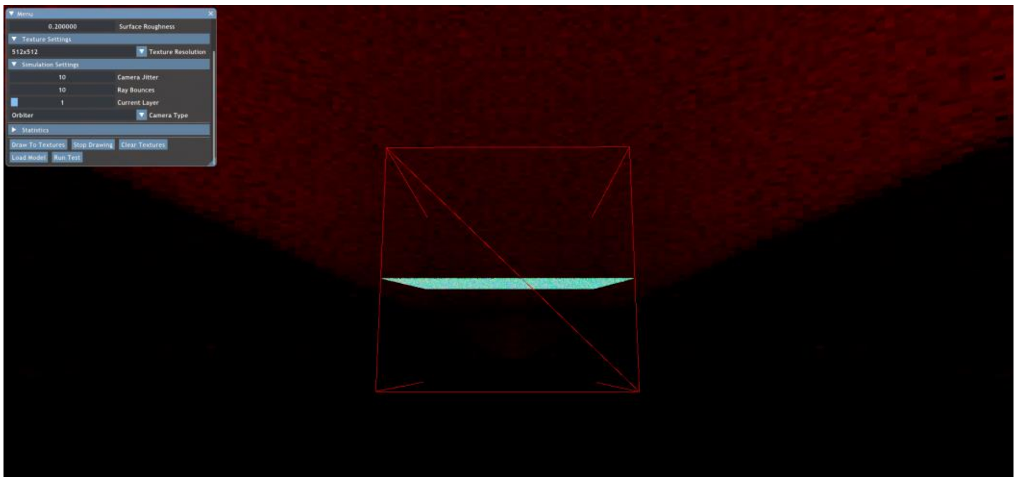
Figure 2.19: Shows the model coloring.

2.3.1.2 Simulated BRDF Model

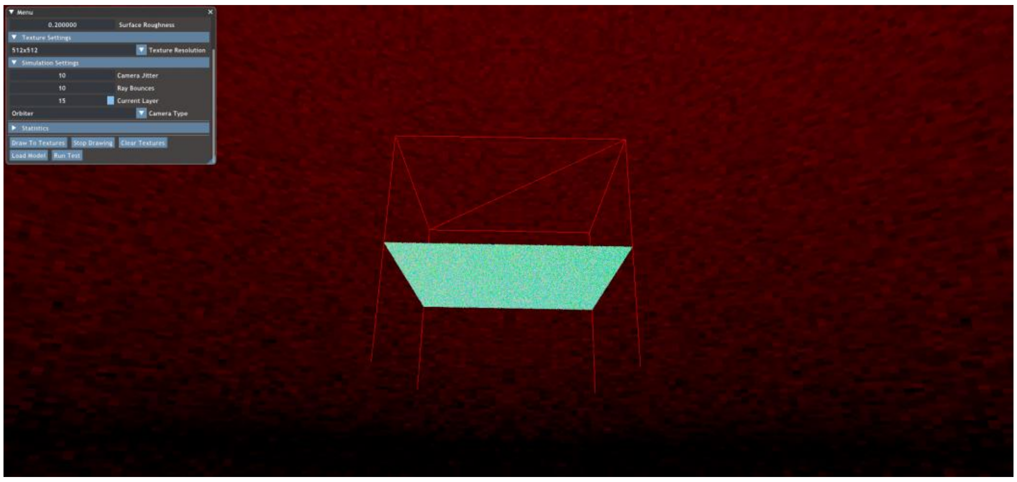
This subsection explains how to run the simulated BRDF model. For running the ***Simulated Model*** the user must prepare the sky-map textures. As they are considered to be our BRDF. For doing that the following steps are needed:-

In the Surface Scene:-

1- The user must set up the ***Surface Settings***. Which are the ***Surface Roughness*** and the ***Surface Size***.
2- Setting up the ***Texture Settings***. Maximizing the resolution creates better BRDF Simulation.
3- Fill up the ***Simulation Settings***. The users can set the ***Camera Jitter*** and ***Ray Bounces*** with the value they want.
4- After setting up all the previous data. The user can click ***Draw To Textures*** to write into all the textures according to the settings. See Figure 2.20.



```
0.200000    Surface Roughness
512x512     Texture Resolution
10          Camera Jitter
10          Ray Bounces
1           Current Layer
Orbiter     Camera Type
```

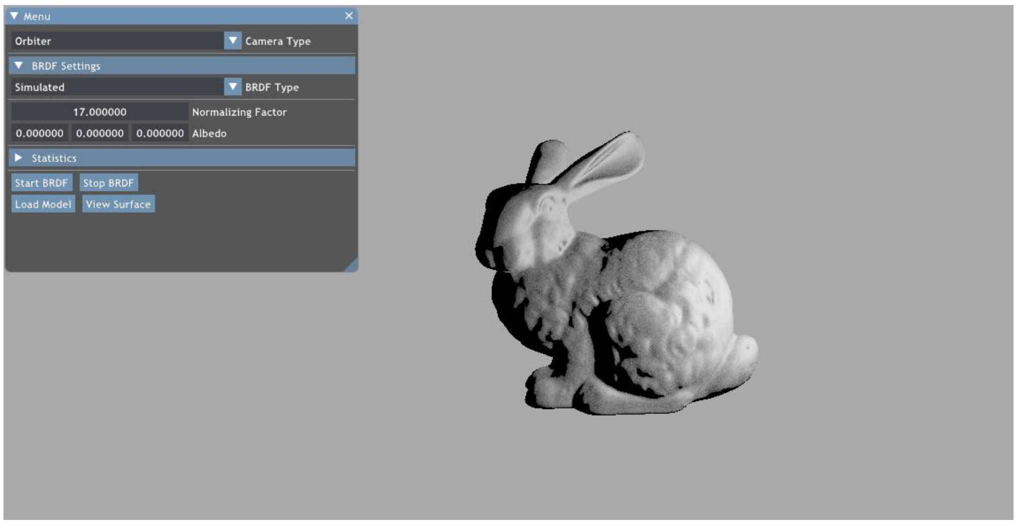


```
0.200000    Surface Roughness
512x512     Texture Resolution
10          Camera Jitter
10          Ray Bounces
15          Current Layer
Orbiter     Camera Type
```

Figures 2.20: Show the Simulation result when Surface Size and Texture Resolution are maximized. (Top) shows Layer 1. (Bottom) shows Layer 15.

In the ***Model Scene***:-

1- The user must set the ***BRDF Type*** to ***Simulated***.
2- There are two input fields. Firstly, the ***Normalize Factor*** which normalizes the specular light of the model. Secondly, the ***Albedo*** which is the color of the model.
3- The user can click ***Start BRDF*** to see the result. Note, the user can change the ***Albedo*** and ***Normalize Factor*** inputs during run-time as they update the model in real-time. See the result in Figure 2.21.


Orbiter Camera Type
BRDF Settings
Simulated BRDF Type
17.000000 Normalizing Factor
0.000000 0.000000 0.000000 Albedo

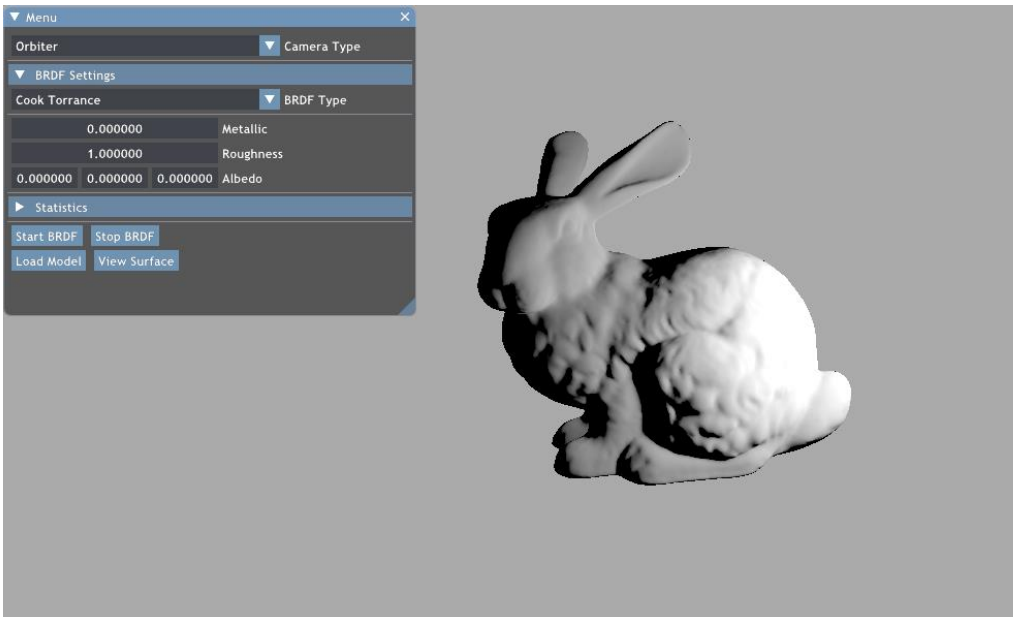

Orbiter Camera Type
BRDF Settings
Cook Torrance BRDF Type
0.000000 Metallic
1.000000 Roughness
0.000000 0.000000 0.000000 Albedo

Figure 2.21: (Top)The Simulated BRDF Result. (Bottom) Cook-Torrance Result

3. Theoretical Background

Centuries ago, the concept of drawing an image of the world from a specific point of view originated. Moreover, many theoretical and practical methods were suggested for this purpose. For example, the ray tracing technique came as early as the 16th century when it was described by ***Albrecht Dürer***, who is credited for its invention [4]. Moreover, the first application of using a computer for ray tracing to obtain shaded pictures was achieved by ***Arthur Appel*** in 1968 [5]. Ray tracing is only one of many techniques for projecting 3D scenes into an image plane. There is also another well-known technique called rasterization. Nowadays, these two methods are widely used solutions to solve visualization problems. This chapter presents the concept of the rendering equation, Monte-Carlo Sampling, the theory of microfacets, and Algorithms used in this thesis.

3.1 Rendering Equation

The Rendering Equation is one of the most important topics in computer graphics. It was presented by ***James Kajiya*** in 1986 [6]. The various realistic rendering techniques, that aim to simulate real-world images, in computer graphics attempt to solve this equation. The rendering equation is an integral equation that describes the light propagation in a scene according to a given illumination model. In other words, it describes the total amount of light emitted from a point along a particular viewing direction, given a function for incoming light and BRDF, see Figure 3.1.

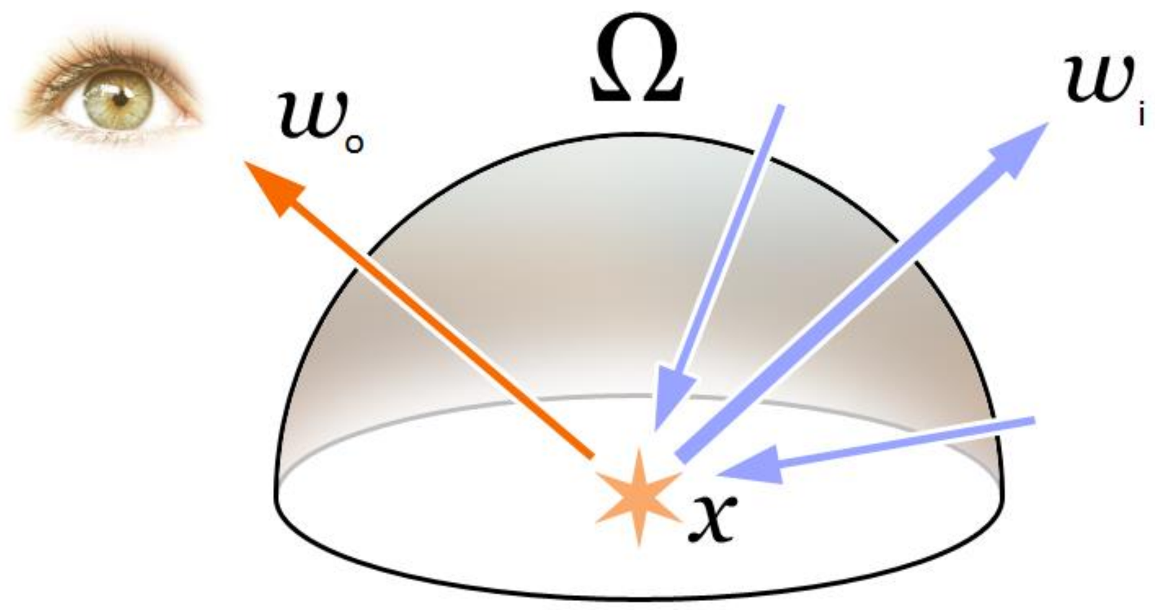


Figure 3.1: Shows the cumulative quantity of light generated from a specific point within the scene. Figure source: wikimedia.org.

The formal representation of the rendering equation is given in Equation 3.1. Some parameters are left out of the equation for ease of implementation. Equation 3.2 shows the final form in most circumstances. The wavelength can be essential in physics simulations; hence, the equation can be modified based on its usage.

$$L_0(\vec{x},\overrightarrow{w_0},\lambda,t) = L_e(\vec{x},\overrightarrow{w_0},\lambda,t) + \int_\Omega \mathrm{brdf}(\vec{x},\overrightarrow{w_0},\overrightarrow{w_i},\lambda,t) * \mathrm{L_i}(\vec{x},\overrightarrow{w_i},\lambda,t) * (\overrightarrow{w_i}\cdot\vec{\mathrm{n}})\,\mathrm{d}\overrightarrow{w_i} \quad (3.1)$$

$$L_0(\vec{x},\overrightarrow{w_0}) = L_e(\vec{x},\overrightarrow{w_0}) + \int_\Omega \mathrm{brdf}(\vec{x},\overrightarrow{w_0},\overrightarrow{w_i}) * \mathrm{L_i}(\vec{x},\overrightarrow{w_i}) * (\overrightarrow{w_i}\cdot\vec{\mathrm{n}})\,\mathrm{d}\overrightarrow{w_i} \quad (3.2)$$

- $\boldsymbol{L_0}$ is the total light directed outward along direction $\overrightarrow{\boldsymbol{w_o}}$, from a particular position $\vec{\boldsymbol{x}}$,
- $\vec{\boldsymbol{x}}$ is the location in space
- $\overrightarrow{\boldsymbol{w_0}}$ is the view direction in which the point is viewed from
- $\boldsymbol{\lambda}$ is a particular wavelength of light
- $\boldsymbol{t}$ is the time
- $\boldsymbol{L_e}$ is the emitted light from the point $\overrightarrow{\boldsymbol{w_o}}$
- $\int_\Omega$ is the continuous sum of all incoming light or the irradiance

- Ω is the unit hemisphere centered around $\vec{n}$ brdf is the Bidirectional Reflectance Distribution Function
- $\vec{n}$ is the surface normal

It is challenging to solve the rendering equation for any given scene in a realistic rendering. Therefore, one of the methods that are used to solve the equation is called ***Monte-Carlo***, which is used in this application. Monte-Carlo is a sampling method used in computer graphics to approximate the rendering equation. It is discussed more in-depth in Subsection 3.2.

3.2 Monte-Carlo Sampling (Stratified Sampling)

Stratified Sampling or Jittered sampling is a method that is in between random and regular distributed samples. It was introduced by ***Robert L. Cook*** in his paper "Stochastic Sampling in Computer Graphics" in 1986 [7]. In this paper, different sampling methods were studied. Stratified sampling is one of the main sampling methods that were discussed in that paper.

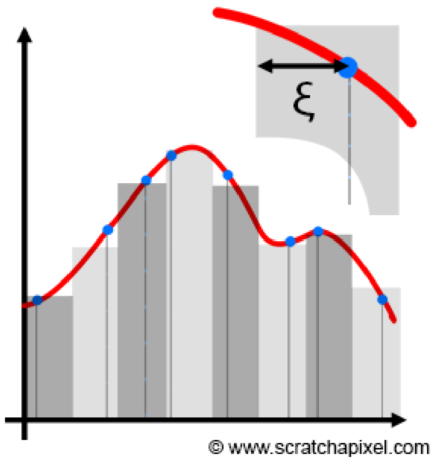


Figure 3.2: The integration interval is divided into regular intervals and a random sample within each subinterval is generated. Figure source: https://www.scratchapixel.com

Stratified sampling combines two types of sampling methods regular and random sampling, Figure 3.2. The integration interval is divided into N cells, known as ***strata***. Samples are placed within the center of these cells; however, they are slightly adjusted with a random offset that can be either positive or negative. Furthermore, the offset value must be less than or equal to half the width of the cell. For example, if $\boldsymbol{h}$ is the width of the cell: $\frac{-\boldsymbol{h}}{\boldsymbol{2}} \leq \boldsymbol{\xi} \leq \frac{\boldsymbol{h}}{\boldsymbol{2}}$. Hence, the result can be determined as the sum of $\boldsymbol{N}$ Monte Carlo integrals over sub-domains [8]. Equation 3.3.

$$\langle F^N \rangle = \frac{(b-a)}{N} \sum_{i=0}^{N-1} f\left(a + \left(\frac{i+\xi}{N}\right)(b-a)\right) \tag{3.3}$$

It can be extended to 2D by dividing the domain of the integration into $\boldsymbol{N} * \boldsymbol{M}$ cells, and a random sample is placed within each cell. It can be extended even to higher dimensions. In stratified sampling, variance reduces linearly with the number of samples ($\boldsymbol{\sigma} \propto \mathbf{1} / \boldsymbol{N}$). Hence, stratified sampling is more reliable than random sampling and it is used more often [8].

3.3 Microfacets Theory

This section presents an overview of the microfacets theory. Moreover, it provides the components of the microfacets-based BRDFs.

3.3.1 Overview

Microfacets theory is the keystone of all Physically-Based-Rendering techniques. Microfacets are very tiny perfectly reflective mirrors which exist on any surface at a microscopic level. Depending on the roughness of the surface, the alignment of the microfacets differs.

The rougher the surface is, the more randomly aligned each microfacet will be along the surface. As a result, these microfacets affect the specular lighting, as the incoming light rays will be scattered in random directions on rougher surfaces. On the other hand, when the surface is smooth the light rays will be reflected in the same direction. Therefore, it causes perfect reflection. See Figure 3.3. Hence, we can see that higher roughness values display a much larger specular reflection shape, in contrast with the smaller and sharper specular reflection shape of smooth surfaces [9]. See Figure 3.4.

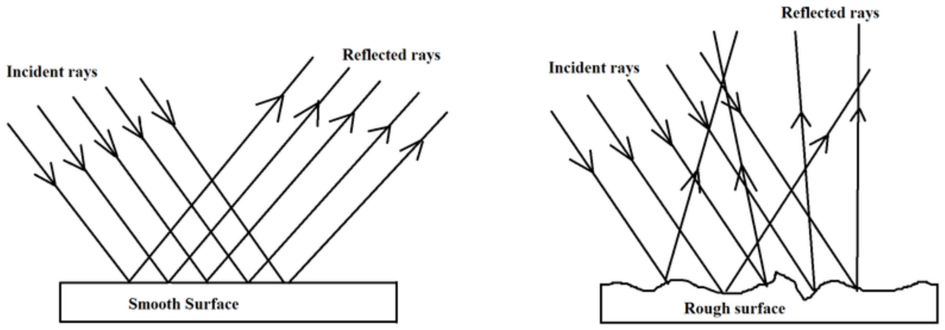


Figure 3.3: (Left) when surface is smooth the specular will be reflected in the same direction. (Right) When the surface is rough the specular will be scattered. Figure credits to Design and development of soft x-ray multi-layer optics [24] Figure No. 2.8

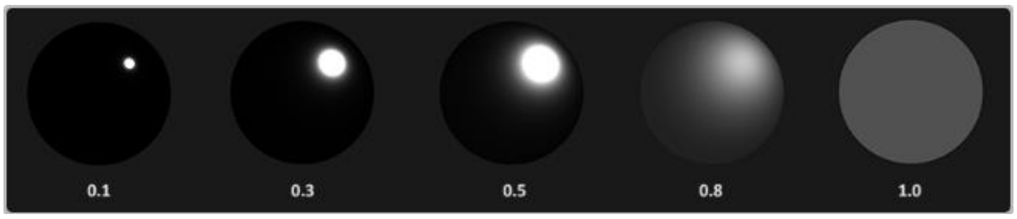


Figure 3.4: The change of the specular reflection based on the roughness. Figure source: https://learnopengl.com/PBR/Theory

The surface approximation employs a form of ***energy conservation***, which means the outgoing light energy should never exceed the incoming light energy, except if it is an emissive surface. From Figure 3.4, we notice that when the roughness increases, it causes the brightness to decrease with more specular reflection area. However, if the specular reflection kept its energy, then that means the surface itself is emitting more energy, hence, it violates the principle of energy conservation. Therefore, the specular reflection is dimmer on rough surface and more intense on smooth ones [9].

In physics, light is considered as a beam that has energy. Therefore, it loses some of its energy by collision, as surfaces have tiny particles that absorb the light's energy and convert it into heat, see Figure 3.5. However, generally, not all energy is absorbed, and the light beam will be reflected in a random direction until it is reflected away from the surface or loses all of its energy [9]. The distance between the entrance and exit of the ray is denoted by the ***scattering distance***.

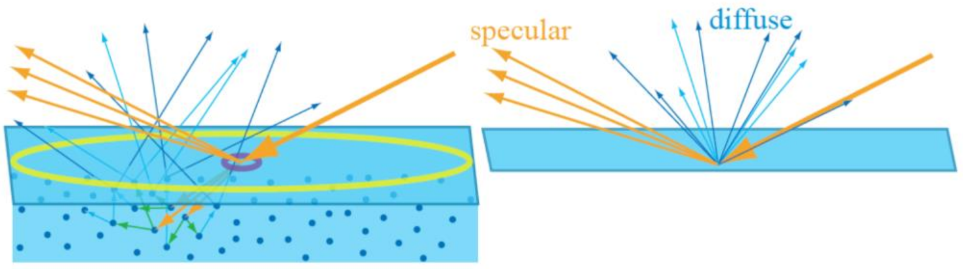


Figure 3.5: Demonstrates how the surface interacts with the light beam. The left one shows a close view of the reflection and refraction phenomena, where the blue circles represent the particles of the surface. The blue arrows are the diffused rays, and the yellow ones are the specular rays. Figure credits to Real-Time rendering [13].

Metallicity has an important role when it comes to reflection and refraction. As metallic surfaces behave differently to light compared to non-metallic ones. Metallic surfaces follow the same laws of reflection and refraction. However, all the refracted light gets absorbed without scattering. Hence, metallic surfaces only leave specular light; metallic surfaces show no diffuse colors [9].

To approximate the material's reflective and refractive properties a BRDF is used, which estimates how much each light ray contributes to the final reflected light of an opaque surface given its material properties [9]. However, the laws of physics impose two constraints on any BRDF, which are *Helmholtz reciprocity* and *conservation of energy*. However, in interactive applications, it is possible to ignore these laws when an approximation is sufficient to create a sense of realism. Conversely, offline renderers to be physically plausible they must respect these laws, as their main purpose is to create a realistic scene.

Between 1977 and 1982, ***Blinn***, ***Cook***, ***Sparrow***, and ***Torrance*** introduced the concepts of micro-geometry and nano-geometry representations, which are employed to create physically accurate BRDFs [10, 11, 12]. In the micro-geometry model, each deformity is considered as a tilted mirror, enabling the interaction between light and material using geometric optics principles. Conversely, the nano-geometry models utilize a wave optics description, allowing light waves to interfere constructively or destructively [13, 14].

The *micro-facet* models were derived from the *micro-geometric* representation of surfaces. Hence, they are considered *physically based*, yet they might be physically inaccurate [17]. There are three important functions compose these micro-facets based BRDFs. The *Fresnel Function* (Subsection 3.3.2) describes the ratio of the surface reflection, the *Geometry Function* (Subsection 3.3.3), describes the self-shadowing property of the microfacets, and the *Normal Distribution Function* (Subsection 3.3.4), deals with the ratio of normal to the given direction.

3.3.2 Fresnel-Schlick Approximation

The Fresnel equations describe the intensity of the light that gets reflected over the light that gets refracted based on the angle the surface is being looked at. Those equations were introduced by ***Augustin-Jean Fresnel***. From the ratio of reflection and the energy conservation principle, we can obtain the refracted portion of light.

Every surface or material has a level of reflectivity when looking straight at its surface, however, when looking at the surface from a specific angle all reflections become more intense compared to the surface's base reflectivity. Hence, all surfaces fully reflect light if seen from perfect 90-degree angles. This phenomenon is called Fresnel and it is described by the Fresnel equation [9].

However, the Fresnel equation is complicated. As a result, it can be approximated using ***Fresnel-Schlick*** approximation, Equation 3.3, and Code 3.1 for implementation [9], Schlick approximation [15] is used in interactive applications as it works with RGB Triplets, making it easier to implement. The result of the Schlick model is almost realistic. However, in some cases, like Zinc [13, 15], the inaccuracy is high, although in most cases, this error is ignored.

$$F_{Schlick}(\mathrm{n}, v) = F_0 + (1 - F_0)\big(1 - (\mathrm{n} \cdot v)\big)^5 \tag{3.3}$$

- ***n*** is the surface normal
- ***v*** is the incident direction
- ***F₀*** is the base color of the specular reflection that is depicted when the incoming light hits the surface at normal incidence ($\boldsymbol{\theta_i = 0}$°). As the incident angle increases, the Fresnel function output for all light frequencies rises to 1. As shown in Equation 3.4, if **θ_i = 90**°, then ***F*(θᵢ) = *1***, where ***F*(θᵢ)** is the Fresnel function.

$$\lim_{\theta \to 90^\circ} f\,(\theta) \;=\; 1 \;\&\; \lim_{\theta \to 0^\circ} f\,(\theta) = F_0 \tag{3.4}$$

For unknown F_0 values, equation 3.5 can be used.

$$F_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{3.5}$$

```
float3 fresnelSchlick(float cosTheta, float3 F0)
{
    return F0 + (1.0 - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0), 5.0);
}
```

Code 3.1: ***Fresnel-Schlick*** implementation in ***hlsl***.

3.3.3 Geometry Function

The Geometry Function approximates the relative surface area where sub-surfaces overlap with each other. It was first addressed by Torrance and Sparrow [11] in their investigation of the off-specular peak phenomenon.

The Geometry Function manages the self-shadowing property that occurs between the incoming light and the direction being viewed. Figure 3.6 represents the surface self-shadowing resulting in a reduction in the light intensity when viewed from a specific angle. In other words, self-shadowing (self-masking) occurs when the incoming light hits a sub-surface and then being reflected in a direction that may collide with another sub-surface before reaching the viewer, causing the light to be scattered or reflected in another direction. Based on the energy conservation principle, the light will keep bouncing on the surface of microfacets until it loses all of its energy or exits the surface.

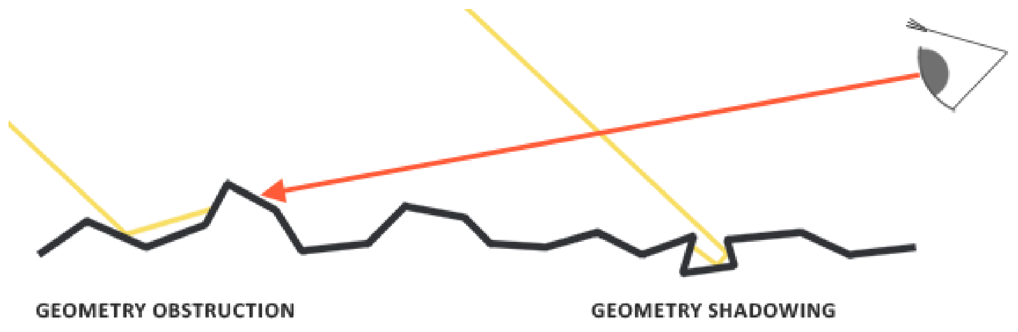


Figure 3.6: Explains how sub-surfaces overshadow each other. The image explains self-shadowing as the Incoming rays cannot reach the viewer because of the over lapped sub-surface, and self-masking as the viewer cannot see the sub-surfaces behind some sub-surfaces. Figure source: https://learnopengl.com/PBR/Theory.

Many Geometric functions are used and utilized nowadays. However, ***Brain Karis*** from epic games did a range of studies on many types of approximations [16]. Moreover, he stated that the one used by Epic Game's Unreal Engine 4 is ***Smith's Schlick-GGX***, given in Equation 3.6, and it is represented in Code 3.2 [9]. According to ***Heitz*** [17], Smith's ***Schlick-GGX*** model proved to be the most accurate among all approximations.

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k} \tag{3.6}$$

```
float GeometrySchlickGGX(float NdotV, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r * r) / 8.0;

    float num = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return num / denom;
}
```

Code 3.2: An implementation of ***Schlick*** method in HLSL.

K is a remapping of ***a***, which can be $\frac{(\alpha+1)^2}{8}$ or $\frac{\alpha^2}{2}$, where ***a*** is the roughness of the surface.

To approximate the geometric function, we need to consider both the view direction and the light direction. The final representation is seen in Equation 3.7, and Code 3.3 for implementation [9].

$$G(n,v,l,k) = G_{SchlickGGX}(n,l,k) * G_{ShlickGGX}(n,v,k) \tag{3.7}$$

```
float GeometrySmith(float3 N, float3 V, float3 L, float roughness)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx2 = GeometrySchlickGGX(NdotV, roughness);
    float ggx1 = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}
```

Code 3.3: An implementation of ***Smith*** method in HLSL

3.3.4 Normal Distribution Function

As the Geometric Function considers the overlap of the sub-surfaces, it does not consider the orientation of these sub-surfaces toward a given direction. Hence, another factor is added to microfacets-based BRDFs to address the problem and is denoted by the *Normal Distribution Function*.

The ratio of microfacets facing a specific direction is described by Normal Distribution Function. Similar to the Geometric Functions, Many Normal Distribution Functions are used in production, however, the one utilized in Epic Games Unreal Engine 4 and in this thesis is ***Trowbridge-Reitz GGX*** [16], as seen in Equation 3.8, and Code 3.4 for implementation.

$$NDF_{GGXTR}(n,h,\alpha) = \frac{\alpha^2}{\pi((n\cdot h)^2(\alpha^2-1)+1)^2} \tag{3.8}$$

- ***n*** is the normal direction

- ***h*** is the intended direction, which is the halfway vector between the view and incidence direction
- ***a*** is the surface roughness

Lastly, it must be energy conservative. Hence, the sum of the distribution function for all given directions must equal 1, see equation 3.9.

$$\int_h D(n, h, \alpha) \cdot (n \cdot h)\, dh = 1 \tag{3.9}$$

```
float normalDistributionGGX(float3 N, float3 H, float roughness)
{
    float a = roughness * roughness;
    float a2 = a * a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH * NdotH;
    float num = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;
    return num / denom;
}
```

Code 3.4: An implementation of ***Trowbridge-Reitz GGX*** method in HLSL.

3.4 Algorithms

This subsection presents and explains algorithms used on the surface of microfacets simulation, that is used in this study. It starts by explaining raymarching in section (3.4.1), then the ***Möller-Trumbore*** algorithm in section (3.4.2) for ray and triangle intersection. Finally, the ray and line intersection in section (3.4.3).

3.4.1 Ray Marching

Ray marching is a method employed to create entirely procedural environments from a fragment shader. When ray marching is used dealing with polygons becomes unneeded. Instead, *signed distance functions* (*SDFs*) are utilized. SDFs are mathematical functions that assess the distance between a given point in space and the surface.

For instance, assume there is a point, denoted as P in a three-dimensional space, along with a sphere positioned at the center of the world and having a radius of one unit. To determine the distance between point P and the nearest point on the sphere, we compute the distance that corresponds to the length of the vector that points from P to the center of the sphere, minus the radius of the sphere, hence, this will return the distance from ***p*** to the closest point on the sphere. This calculation is categorized as a signed distance function(SDF) because it yields a negative value, zero, or a positive value, depending on whether P is located inside, on the surface of, outside the sphere, respectively [18]. As seen in Equation 3.10, and Figure 3.7.

$$d = ||p - c|| - r \tag{3.10}$$

- ***p*** is the point,
- ***c*** is the center of the sphere
- ***r*** is the radius of the sphere

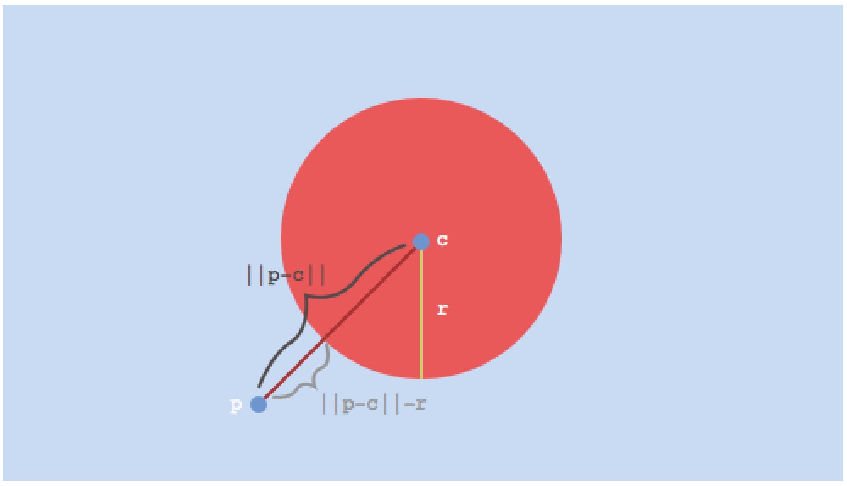


Figure 3.7: Shows an example of SDF to calculate the distance between the point and the closest point on the sphere. Figure credits to Michael Walczyk [18].

At a high level, numerous rays will be emitted from a virtual camera that observes our world. Each of these rays will undergo ray marching. In simple terms, we will progressively move along the direction of the ray, evaluating the SDF at each step. This approach allows us to gather various information, such as the position of the point, and how far it is from the closest point on the surface. However, when the distance is extremely small, as determined by a predefined threshold, we can classify it as a "hit". Subsequently, we can compute the reflected light and repeat the entire process again.

In this thesis, ray marching was utilized on the surface of microfacets. Hence, two SDFs were used to calculate the ray steps during ray marching. The ray and line intersection was used to get the distance to move the ray from one block to another, and the ray and triangle intersection was used to determine if the ray collided with one of the sub-surfaces, if so the ray will be reflected based on the normal of the sub-surface, then the process will be repeated, see Figure 3.8. However, the ray marching will stop if the ray exits the surface, or the ray has lost all of its energy, as can be seen in code 3.5.

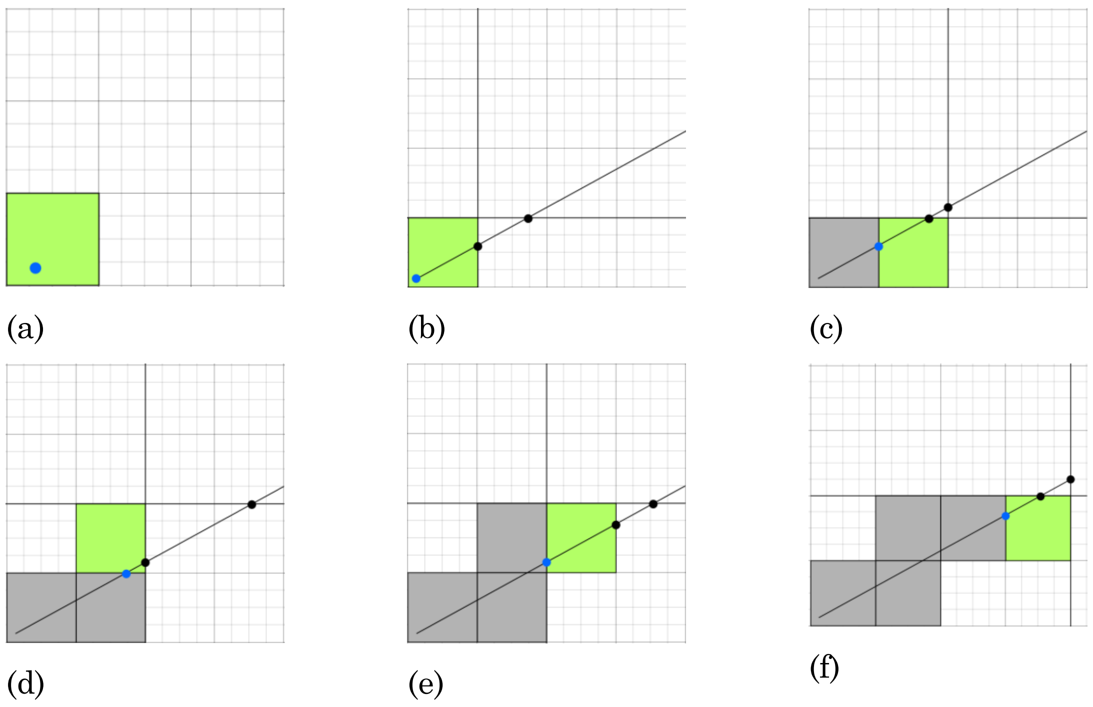


Figure 3.8: A top-down view that illustrates the ray marching process. (a) shows the initial block of the ray. (b) the point will be marched in its direction and will be shifted to the closest border. (c) checking if there is an intersection with any triangle in the current block, if there is an intersection the point will be reflected based on the normal of that sub-surface then the process will be repeated, else the marching will continue till the ray exits the surface.

```
bool ray_march(inout float3 rayOrigin, inout float3 rayDir, inout int hitCount)
{
    float3 current_block = floor(rayOrigin);
    float3 v1 = float3(0.f);
    float3 v2 = float3(0.f);
    float3 v3 = float3(0.f);
    float3 v4 = float3(0.f);
    bool isTriaIntersect = false;
    while (hitCount >= 0 && isPointInCube(rayOrigin, surfaceSize) &&
           intersectionCnt >= 0)
    {
            //Finding the Current Block
            current_block = floor(rayOrigin);
            fillVertices(current_block.x, current_block.z, v1, v2, v3, v4);

            //Checking if there is a ray-triangle intersection
            isTriaIntersect = isTriangleIntersection(rayOrigin, rayDir, v1, v2,
              v3, v4, hitCount);

            if (!isTriaIntersect) {
                 //Updating the block/cell
                 updateCell(rayOrigin, rayDir, v1, v2, v3, v4);
            }
    }
    return !isPointInCube(rayOrigin, surfaceSize) && hitCount >= 0;
}
```

Code 3.5: Demonstrates the ray-marching method used to iterate through the surface in this thesis.

3.4.2 Ray-Triangle Intersection

In this study, the Ray-Triangle intersection algorithm that was applied is called the ***Möller-Trumbore*** ray-triangle intersection algorithm. The Möller-Trumbore algorithm is a fast ray-triangle intersection algorithm that was introduced in 1997 by ***Tomas Möller*** and **Ben Trumbore** in a paper titled "Fast, Minimum Storage Ray/Triangle Intersection" [19]. Möller-Trumbore is a fast method that is utilized for calculating the intersection of a ray and a triangle in a three-dimensional space. One of the advantages of this method is that there is no need to precompute the plane equation of the plane containing the triangle.

The algorithm translates the origin of the ray and then changes the base of that vector which returns a vector ***(t u v)**^{T}*. Therefore, this method highly relies on the *barycentric coordinate* also known as *areal coordinates*. Barycentric coordinates is used to express the position of any point located on the triangle with three scalars, usually referred to as ***(u, v, w)***. The location of this point includes any position inside the triangle, any of its edges, or even any of the triangle's vertices themselves. To compute the position of the point using barycentric coordinates, see Equation 3.11. The barycentric coordinates are normalized ***(w + u + v = 1)***, hence, from any two coordinates the third one can be obtained easily, for example ***(w = 1 - u - v)***, from which we can establish that ***u + v <= 1*** [20].

$$P = uA + vB + wC \tag{3.11}$$

The algorithm takes advantage of the parameterization of ***P***, the intersection in barycentric coordinates. See Equation 3.11, it can be represented as 3.12 as well and simplified as 3.13. Since the ray is described as an origin point ***O*** and a direction vector ***D,*** the intersection ***P*** can also be written using the ray's parametric equation, see Equation 3.14 . Lastly, we get equation 3.15 after rearranging the terms. Therefore, the three unknown values ***(t, u, v)*** can be found by solving a linear system of equations [19, 21]. Hence, it can be thought of as translating the triangle to the origin and transforming it to a unit triangle in ***y*** and ***z*** with the ray direction aligned with ***x***, see Figure 3.9. The algorithm implementation can be seen in Code 3.6.

$$P = (1 - u - v)A + uB + vC \tag{3.12}$$

$$P = A + u(B - A) + v(C - A) \tag{3.13}$$

$$O + tD = A + u(B - A) + v(C - A) \tag{3.14}$$

$$O - A = -tD + u(B - A) + v(C - A) \tag{3.14}$$

$$\begin{bmatrix} -D & (B - A) & (C - A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - A \tag{3.15}$$

- **t** is the distance from the ray origin
- ***(u, v)*** are barycentric coordinates.
- ***A***, ***B***, and ***C*** are the triangle vertices
- ***(B - A)*** is edge ***AB***
- ***(C - A)*** is edge ***AC***

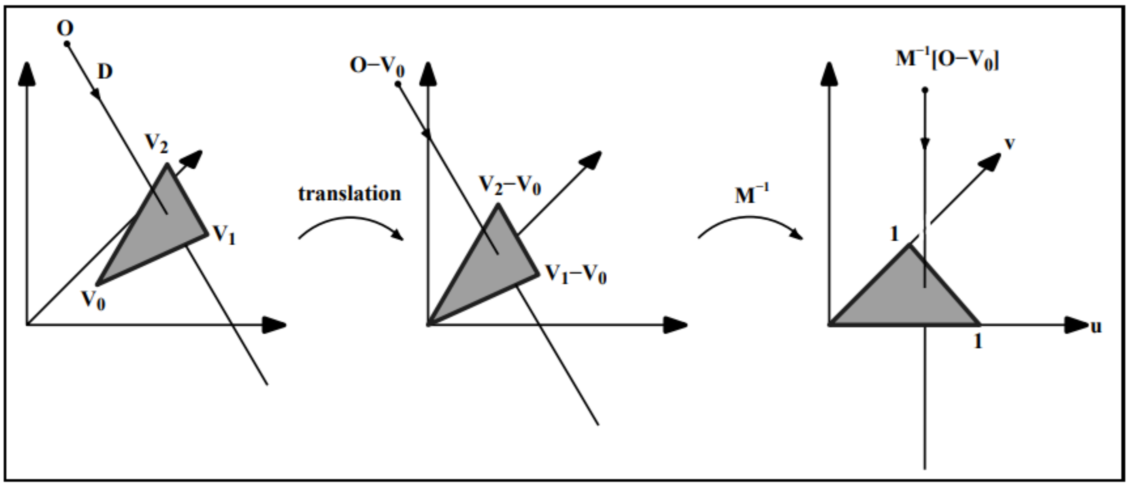


Figure 3.9: Translation and change of base of the ray origin. Where ***M = [-D, (B - A), (C - A)]***. Figure credits to "Fast, minimum storage ray-triangle intersection" paper [19].

The solution of equation 3.15 can be obtained by using ***Cramer's rule***, See Equations and 3.17.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D,\ E_1,\ E_2|} \begin{bmatrix} |\ T,\ E_1,\ E_2| \\ |-D,\ T,\ E_2| \\ |-D,\ E_1,\ T| \end{bmatrix} \tag{3.16}$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{3.17}$$

Where,

- **E$_1$**, and **E$_2$** are edges (B - A) and (C - A), respectively.
- ***T*** is (O - A)
- ***P*** is (D x E$_2$)
- ***Q*** is (T x E$_1$)

```
/*Moller-Trumbore Method*/
#define kEpsilon 0.000001
bool rayTriangleIntersect(
    const inout float3 orig, const inout float3 dir,
    const inout float3 v0, const inout float3 v1, const inout float3 v2,
    inout float t, inout float u, inout float v)
{
    float3 v0v1 = v1 - v0;
    float3 v0v2 = v2 - v0;
    float3 pvec = cross(dir, v0v2);
    float det = dot(v0v1, pvec);
    // ray and triangle are parallel if det is close to 0
    if (( det < kEpsilon)) return false;
    float invDet = 1.0 / det;
    float3 tvec = orig - v0;
    u = dot(tvec, pvec) *invDet;
    if (u < 0.f || u > 1.f) return false;
    float3 qvec = cross(tvec, v0v1);
    v = dot(dir, qvec) *invDet;
    if (v < 0.f || v + u> 1.f) return false;
    t = dot(v0v2, qvec) * invDet;
    if (t <= 0) return false;
    return true;
}
```

Code 3.6: Implementation of ***Möller–Trumbore*** ray-triangle intersection algorithm in HLSL. Code source: Möller–Trumbore, Fast, minimum storage ray-triangle intersection paper [19].

3.4.3 Ray-Line Intersection

To determine the position of the intersection in relation to line segments, we can express lines ***L₁*** and ***L₂*** in terms of first-degree Bézier parameters [22], see equation 3.18.

$$L_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + t \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix}, \quad L_2 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} + u \begin{bmatrix} x_4 - x_3 \\ y_4 - y_3 \end{bmatrix} \tag{3.18}$$

The intersection point of the lines is found with one of the following values of ***t*** or ***u***. Equation 3.19 represents their calculation.

$$t = \frac{(x_1-x_3)(y_3-y_4)-(y_1-y_3)(x_3-x_4)}{(x_1-x_2)(y_3-y_4)-(y_1-y_2)(x_3-x_4)} \quad (3.19)$$

$$u = \frac{(x_1-x_3)(y_1-y_2)-(y_1-y_3)(x_1-x_2)}{(x_1-x_2)(y_3-y_4)-(y_1-y_2)(x_3-x_4)} \quad (3.19)$$

To calculate the intersection point, see equation 3.20.

$$(P_x, P_y) = (x_1 + t(x_2 - x_1), y_1 + t(y_2 - y_1)) \quad (3.20)$$

$$(P_x, P_y) = (x_3 + u(x_4 - x_3), y_3 + u(y_4 - y_3)) \quad (3.20)$$

An intersection exists when ***0*** $\leq$ ***u*** $\leq$ ***1*** and ***0*** $\leq$ ***t*** $\leq$ ***1***. If ***t*** falls beween 0 and 1, it indicates that the intersection point lies within the first line segment. Similarly, if ***u*** is between 0 and 1, it means that the point lies within the second line segment. For implementation see code 3.7.

```
/*Ray Line intersection*/
bool rayLineIntersect(const float2 rayOrigin, const float2 rayDirection, const
float2 lineStart, const float2 lineEnd, inout float t) {
    float2 lineDir = lineEnd - lineStart;
    float d = rayDirection.x * lineDir.y - rayDirection.y * lineDir.x;
    if (d == 0.f) return false;
    float u = ((lineStart.x - rayOrigin.x) * rayDirection.y - (lineStart.y –
                rayOrigin.y) * rayDirection.x) / d;
    t = ((lineStart.x - rayOrigin.x) * lineDir.y - (lineStart.y - rayOrigin.y) *
          lineDir.x) / d;
    if ((u < 0.f || u > 1.f) || (t < 0.f || t > 1.f)) return false;
    return true;
}
```

Code 3.7: An implementation of the ray line intersection method using HLSL.

4. Development & Implementation

This chapter explains the technical parts of the application. Firstly, the building steps of the application is shown (Section 4.1). Then, the implementation of the two BRDFs used in this thesis (Section 4.2). Lastly, it discusses testing and validation (Section 4.3).

4.1 Building & Compiling the Application

This software solution was developed and maintained using a Nvidia Framework called ***Falcor***. Falcor is real-time rendering framework that supports DirectX 12 and Vulkan APIs. The purpose of it is to enhance the productivity of research and prototype projects. It has many features such as Render Graph system, DirectX Raytracing abstraction, Abstracting common graphics operations, and many more.

4.1.1 Prerequisites

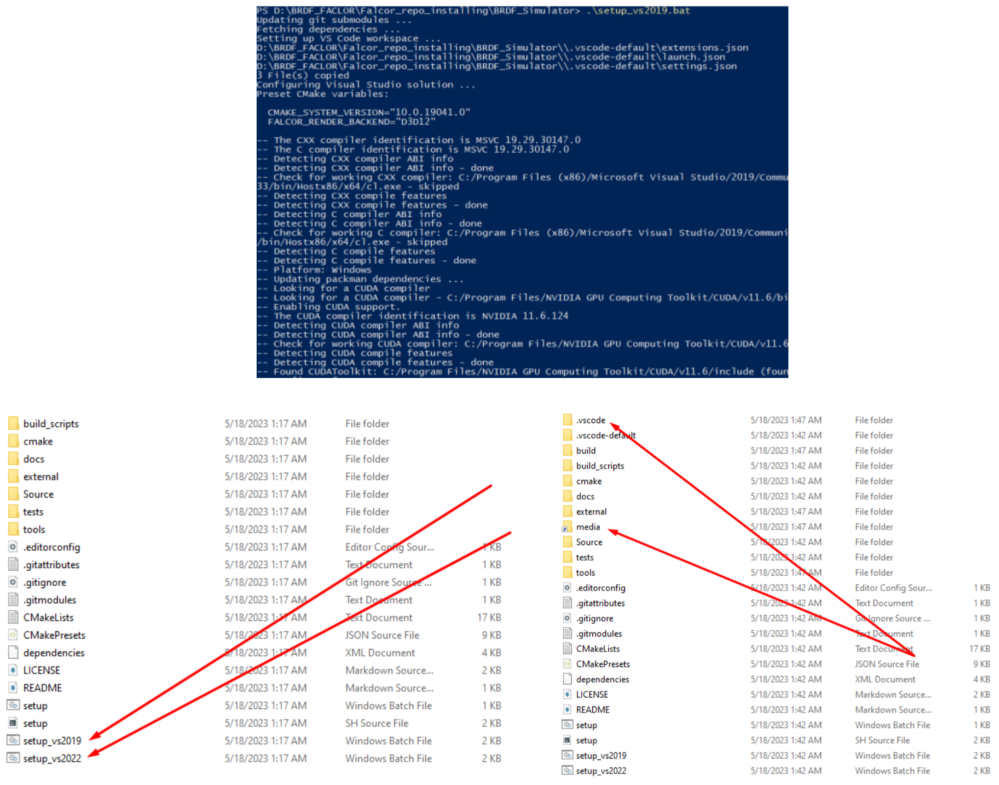
Such a big framework needs prerequisites to run projects on it. The user's machine must be compatible with the following prerequisites [23]:

- Windows 10 version 20H2 or newer, OS build revision .789 or newer.
- Visual Studio 2019 or 2022
- Windows 10 SDK
- A GPU which supports DirectX Raytracing, such as the NVIDIA Titan V or GeForce RTX
- NVIDIA driver 466.11 or newer
- The user might need to download optional prerequisites. Refer to [23].

4.1.2 Building & Compiling Process

There are some steps needed to build the framework. The user must have ***Visual Studio*** installed in their machine (It should either be ***2019*** or ***2022***). Firstly, the repository must be cloned[3]. After that run ***./setup_vs2019.bat*** (or ***./setup_vs2022.bat***), this bash script will prepare the ***./vscode*** folder and will install the needed ***dependencies*** along with the ***media*** folder, where the models are stored, see Figure 4.1. Then, to make sure that the building was done, open ***./build/windows-vs2019*** directory and check if the **solution files** are there, see Figure 4.2. The binary output is located in ***./build/windows-vs2019/bin***, compiling the solutions is needed first.

Remark, make sure that the path does not contain white space before running the bash script. Otherwise, the script will throw an error that it could not find the path. Moreover, ***Git*** must be installed on the user's machine.

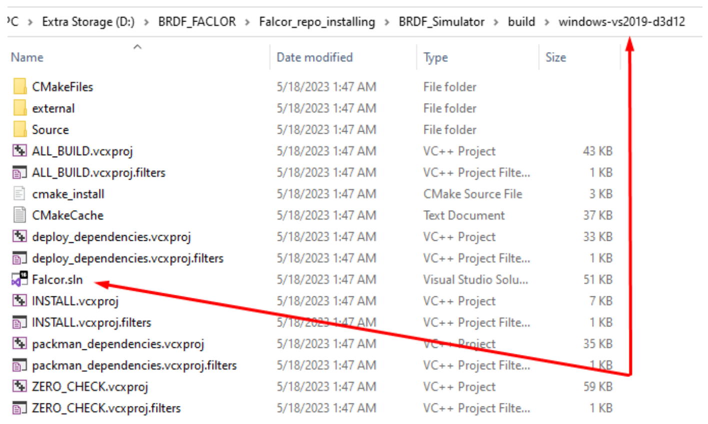


```
PS D:\BRDF_FACLOR\Falcor_repo_installing\BRDF_Simulator> .\setup_vs2019.bat
Updating git submodules ...
Fetching dependencies ...
Setting up VS Code workspace ...
D:\BRDF_FACLOR\Falcor_repo_installing\BRDF_Simulator\\.vscode-default\extensions.json
D:\BRDF_FACLOR\Falcor_repo_installing\BRDF_Simulator\\.vscode-default\launch.json
D:\BRDF_FACLOR\Falcor_repo_installing\BRDF_Simulator\\.vscode-default\settings.json
3 File(s) copied
Configuring Visual Studio solution ...
Preset CMake variables:

  CMAKE_SYSTEM_VERSION="10.0.19041.0"
  FALCOR_RENDER_BACKEND="D3D12"

-- The CXX compiler identification is MSVC 19.29.30147.0
-- The C compiler identification is MSVC 19.29.30147.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/2019/Commu
33/bin/Hostx86/x64/cl.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/2019/Communi
/bin/Hostx86/x64/cl.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Platform: Windows
-- Updating packman dependencies ...
-- Looking for a CUDA compiler
-- Looking for a CUDA compiler - C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.6/bi
-- Enabling CUDA support.
-- The CUDA compiler identification is NVIDIA 11.6.124
-- Detecting CUDA compiler ABI info
-- Detecting CUDA compiler ABI info - done
-- Check for working CUDA compiler: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.6
-- Detecting CUDA compile features
-- Detecting CUDA compile features - done
-- Found CUDAToolkit: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.6/include (foun
```

build_scripts	5/18/2023 1:17 AM	File folder	
cmake	5/18/2023 1:17 AM	File folder	
docs	5/18/2023 1:17 AM	File folder	
external	5/18/2023 1:17 AM	File folder	
Source	5/18/2023 1:17 AM	File folder	
tests	5/18/2023 1:17 AM	File folder	
tools	5/18/2023 1:17 AM	File folder	
.editorconfig	5/18/2023 1:17 AM	Editor Config Sour...	1 KB
.gitattributes	5/18/2023 1:17 AM	Text Document	1 KB
.gitignore	5/18/2023 1:17 AM	Git Ignore Source ...	1 KB
.gitmodules	5/18/2023 1:17 AM	Text Document	1 KB
CMakeLists	5/18/2023 1:17 AM	Text Document	17 KB
CMakePresets	5/18/2023 1:17 AM	JSON Source File	9 KB
dependencies	5/18/2023 1:17 AM	XML Document	4 KB
LICENSE	5/18/2023 1:17 AM	Markdown Source...	2 KB
README	5/18/2023 1:17 AM	Markdown Source...	1 KB
setup	5/18/2023 1:17 AM	Windows Batch File	1 KB
setup	5/18/2023 1:17 AM	SH Source File	2 KB
setup_vs2019	5/18/2023 1:17 AM	Windows Batch File	2 KB
setup_vs2022	5/18/2023 1:17 AM	Windows Batch File	2 KB

.vscode	5/18/2023 1:47 AM	File folder	
.vscode-default	5/18/2023 1:42 AM	File folder	
build	5/18/2023 1:47 AM	File folder	
build_scripts	5/18/2023 1:42 AM	File folder	
cmake	5/18/2023 1:42 AM	File folder	
docs	5/18/2023 1:42 AM	File folder	
external	5/18/2023 1:47 AM	File folder	
media	5/18/2023 1:47 AM	File folder	
Source	5/18/2023 1:42 AM	File folder	
tests	5/18/2023 1:42 AM	File folder	
tools	5/18/2023 1:47 AM	File folder	
.editorconfig	5/18/2023 1:42 AM	Editor Config Sour...	1 KB
.gitattributes	5/18/2023 1:42 AM	Text Document	1 KB
.gitignore	5/18/2023 1:42 AM	Git Ignore Source ...	1 KB
.gitmodules	5/18/2023 1:42 AM	Text Document	1 KB
CMakeLists	5/18/2023 1:42 AM	Text Document	17 KB
CMakePresets	5/18/2023 1:42 AM	JSON Source File	9 KB
dependencies	5/18/2023 1:42 AM	XML Document	4 KB
LICENSE	5/18/2023 1:42 AM	Markdown Source...	2 KB
README	5/18/2023 1:42 AM	Markdown Source...	1 KB
setup	5/18/2023 1:42 AM	Windows Batch File	1 KB
setup	5/18/2023 1:42 AM	SH Source File	2 KB
setup_vs2019	5/18/2023 1:42 AM	Windows Batch File	2 KB
setup_vs2022	5/18/2023 1:42 AM	Windows Batch File	2 KB

Figure 4.1: (Top) Shows the ***setup_vs2019.bat*** installing dependencies. (Left) Shows how the folder structure looks before running the script. (Right) Shows the folder structure after running the script.



›C › Extra Storage (D:) › BRDF_FACLOR › Falcor_repo_installing › BRDF_Simulator › build › windows-vs2019-d3d12

Name	Date modified	Type	Size
CMakeFiles	5/18/2023 1:47 AM	File folder	
external	5/18/2023 1:47 AM	File folder	
Source	5/18/2023 1:47 AM	File folder	
ALL_BUILD.vcxproj	5/18/2023 1:47 AM	VC++ Project	43 KB
ALL_BUILD.vcxproj.filters	5/18/2023 1:47 AM	VC++ Project Filte...	1 KB
cmake_install	5/18/2023 1:47 AM	CMake Source File	3 KB
CMakeCache	5/18/2023 1:47 AM	Text Document	37 KB
deploy_dependencies.vcxproj	5/18/2023 1:47 AM	VC++ Project	33 KB
deploy_dependencies.vcxproj.filters	5/18/2023 1:47 AM	VC++ Project Filte...	1 KB
Falcor.sln	5/18/2023 1:47 AM	Visual Studio Solu...	51 KB
INSTALL.vcxproj	5/18/2023 1:47 AM	VC++ Project	7 KB
INSTALL.vcxproj.filters	5/18/2023 1:47 AM	VC++ Project Filte...	1 KB
packman_dependencies.vcxproj	5/18/2023 1:47 AM	VC++ Project	35 KB
packman_dependencies.vcxproj.filters	5/18/2023 1:47 AM	VC++ Project Filte...	1 KB
ZERO_CHECK.vcxproj	5/18/2023 1:47 AM	VC++ Project	59 KB
ZERO_CHECK.vcxproj.filters	5/18/2023 1:47 AM	VC++ Project Filte...	1 KB

Figure 4.2: Shows how should the ***./build/windows-vs2019*** directory look, and checks if ***Solution*** files exist.

After running the script, which will install the required files. Now, we can open the solution and compile it using Visual Studio. First, open Visual Studio. Then, click on ***Open a project or Solution***. After that, a window will open, in that window navigate to ***./build/windows-vs2019***(or ***./build/windows-vs2022***), and choose the file named ***Falcor.sln***. This will open the ***Solution*** of ***Falcor***, see Figures 4.2 and 4.3. After successfully opening the solution. The user needs to navigate to ***Samples*** directory from the ***Solution Explorer***, then right-click on the project desired to be opened, this thesis application is the one named ***BRDF_Simulator*** project, then choose ***Set as Startup Project***. Finally, make sure that the ***Solution Configurations*** is set to ***Release***, then click on the ***build and run button***, and it will compile and start the application, as seen in Figure 4.4. Now, to see the binary files open **./build/windows-vs2019/bin**, see Figure 4.5.

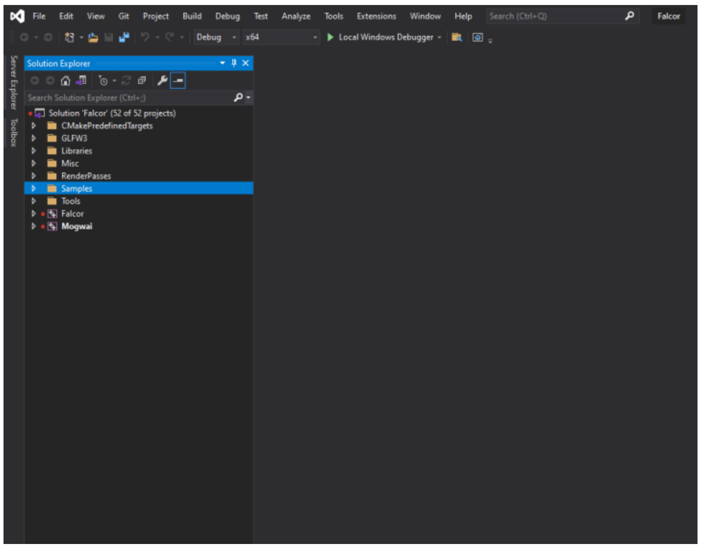


Figure 4.3: Shows the ***Faclor*** Solution

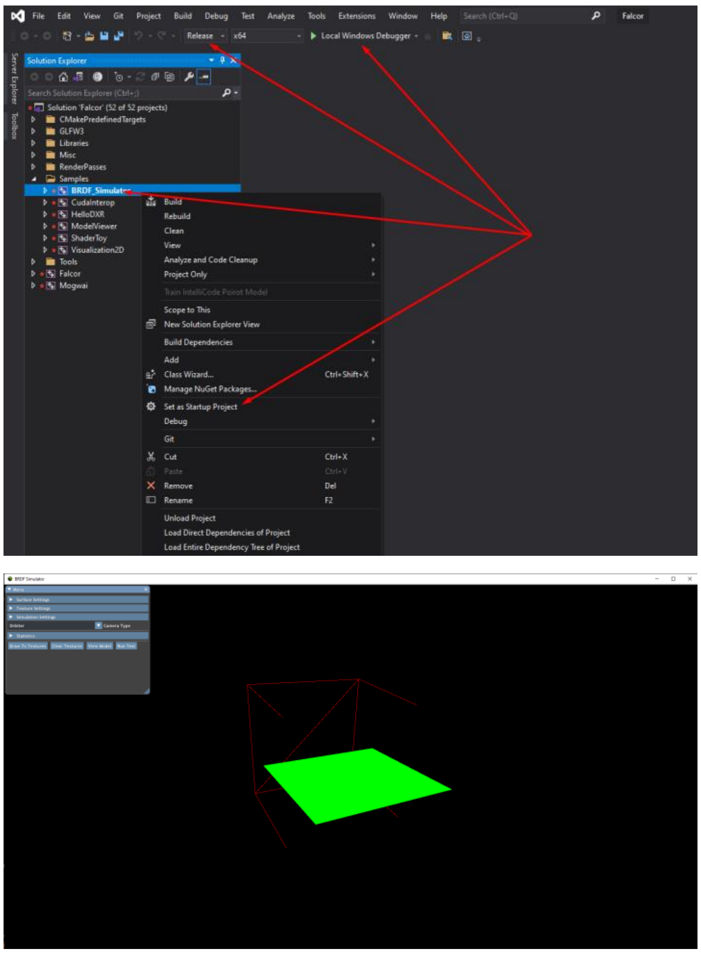


Figure 4.4: (Left) Shows how to compile the application. (Right) Shows the application running

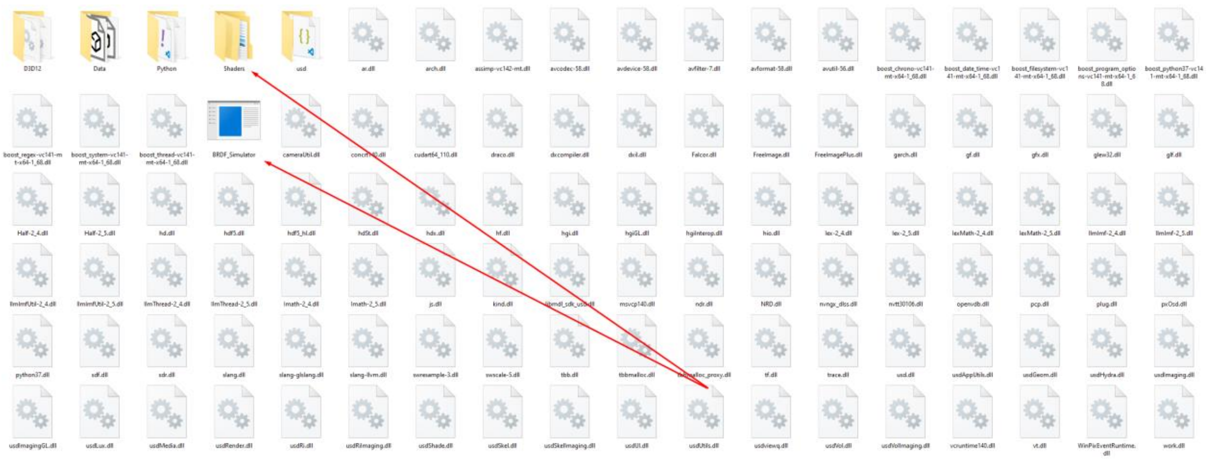


Figure 4.5: Shows the binary file of ***BRDF_Simulator*** project and its shaders. However, the rest of the files are ***Falcor framework*** related ***libraries***, hence, they must not be deleted.

4.2 Bidirectional Reflectance Distribution Functions Implementation

A number of different BRDFs models exist to predict the distribution of microfacets. This section demonstrates the two BRDFs models that are used in this study, which are ***Cook-Torrance*** model (Subsection 4.2.1) and the ***Simulated*** model (Subsection 4.2.2).

4.2.1 Cook-Torrance Model

Cook-Torrance is a BRDF model that is used widely in physically based rendering render pipelines. Moreover, it contains both a diffuse and specular part:

$$f_r = k_d f_{lambert} + k_s f_{cook-torrance} \tag{4.1}$$

Where,

- ***K_d*** is the ratio of incoming light energy that gets refracted
- ***K_s*** is the ratio of incoming light energy that gets reflected

- $\boldsymbol{f_{lambert}}$ is the Lambertian diffuse which is a constant factor denoted as: $\frac{\boldsymbol{c}}{\boldsymbol{\pi}}$ (Where $\boldsymbol{c}$ is the ***albedo*** or ***surface color*** divided by $\boldsymbol{\pi}$ to normalize the diffuse light).

Remark, there are various equations for the diffuse part to look more realistic. However, they are also computationally expensive. Moreover, Lambertian diffuse is sufficient enough for most real-time rendering purposes.

However, the specular part of the BRDF is a bit more advanced, See equation 4.2.

$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \tag{4.2}$$

The cook-Torrance specular BRDF is composed of three functions and a normalization factor in the denominator. Each of these functions ***D***, ***F*** and ***G*** represents an approximation of a specific part of the surface's reflective properties. These are defined as the ***Normal Distribution Function***, the ***Fresnel Equation*** and the ***Geometry Function***. Those functions were discussed in Subsections 3.3.4, 3.3.2, and 3.3.3, respectively.

Including the physically based BRDF into the final reflectance equation, see Equation 4.3. However, the ***Fresnel*** term ***F*** represents the ratio of light that gets reflected on a surface. This is the ratio $\boldsymbol{k_s}$, hence, the specular part contains the reflectance ratio implicitly. Therefore, the final equation can be represented as Equation 4.4, see Code 4.1 for implementation.

$$L_o(p, \omega_o) = \int_\Omega \left(k_d \frac{c}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}\right) L_i(p, \omega_i) n \cdot \omega_i d\omega_i \tag{4.3}$$

$$L_o(p, \omega_o) = \int_\Omega \left(k_d \frac{c}{\pi} + \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}\right) L_i(p, \omega_i) n \cdot \omega_i d\omega_i \tag{4.4}$$

```
float3 cook_torrence_BRDF(float3 N, float3 V, float3 L) {
   float3 H = normalize(V + L);
   float3 F0 = lerp(float3(0.04), albedo, metallic);
   float3 F = fresnelSchlick(max(dot(H, V), 0.0), F0);
   float NDF = DistributionGGX(N, H, roughness);
   float G = GeometrySmith(N, V, L, roughness);
    float3 numerator = NDF * G * F;
    float denominator = 4.0 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) +0.0001;
    float3 specular = numerator / denominator;
    return specular;
}
```

Code 4.1: Implementation of ***Cook-Torrance*** BRDF in HLSL

4.2.2 Simulated Model

The concept behind the *simulated model* is to pre-compute the BRDF (Bidirectional reflectance Distribution Function) and save it in textures, which are represented as the sky-map of the scene. These textures contain information about the number of rays that are reflected away from the microfacets surface.'

Its implementation involved utilizing ray marching for each ray during the simulation of the microfacet surface, as explained in Section 3.4. Following this, if the ray beam retains energy and is reflected away from the surface, the spherical coordinate mapping equation (4.5) is employed to convert the ray's direction into a texture coordinate. Lastly, these texture coordinates are used as indices to increment the amount of rays that hit that specific point in the texture, as demonstrated in Code 4.2.

Remark, each texture can be considered as a 2d array.

$$u = \frac{atan2(x,-z)}{2\pi} + \frac{1}{2} \tag{4.5}$$

$$v = \frac{acos(y)}{\pi} \tag{4.5}$$

```
// Spherical Coordinate Mapping Equation
float2 world_to_latlong_map(float3 dir)
{
    float3 p = normalize(dir);
    float2 uv;
    uv.x = atan2(p.x, -p.z) * M_1_2PI + 0.5f;
    uv.y = acos(p.y) * M_1_PI;
    return uv;
}


void updateTexture(float3 posIn, float3 dirIn)
{
    uint twidth;
    uint theight;
    tex2D_uav.GetDimensions(twidth, theight);
    int hitCount = bounces;
    float3 pos = posIn;
    float3 dir = dirIn;
    //Applying Ray marching
    bool render = ray_march(pos, dir, hitCount);
    if (render) {
        float2 res = world_to_latlong_map(normalize(dir));

       //Increasing the texture value on threads
        InterlockedAdd(tex2D_uav[uint2((res.x * twidth), (res.y * theight))],1);
    }
}
```

Code 4.2: Shows the implementation of how the BRDF is stored in the texture.

In the end, the stored BRDF is utilized on a model. To extract the BRDF information from the texture, the direction from the surface to the light source must be transformed into the local space. Subsequently, the spherical coordinates mapping equation is employed to obtain the corresponding $\boldsymbol{u}$ and $\boldsymbol{v}$ coordinates, which are then used as indices to retrieve data from the 2D texture, as demonstrated in Code 4.3. The outcome of the simulated BRDF can be observed in Figure 5.1.

```
float3 render_simulated(float3 N, float3 V, float3 L, Texture2D currTexture,
SamplerState cuurSampler) {

    float3 fst_axis = normalize(cross(V, N));        // (1,0,0)
    float3 sec_axis = normalize(cross(fst_axis, N)); // (0,0,1)
    float3x3 axis = float3x3(fst_axis, N, sec_axis); // [(1,0,0), (0,1,0),
(0,0,1)] Using right hand style. |/_
    float3x3 inv_axis = transpose(axis);
    float3 dir = mul(L, inv_axis);
    float2 uv = world_to_latlong_map(dir);
    uint twidth;
    uint theight;
    currTexture.GetDimensions(twidth, theight);
    float4 specular = currTexture.Sample(cuurSampler, uv).xxxx;
    uint3 res = uint3(asuint(specular.r), asuint(specular.g),
    asuint(specular.b));
    //normalizing parameter is passed through the GUI
    float3 normSpecular = float3(res / (gSamples * normalizing));
    return normSpecular;
}
```

Code 4.3: Shows the Implementation of how to read the BRDF in HLSL.

4.3 Testing

This section discusses the testing methods that were applied to the application. The testing of the software is divided into two subsections: Black box testing (4.3.1), which tests how the application is supposed to behave, and white box testing (4.3.2), which tests internal functionalities.

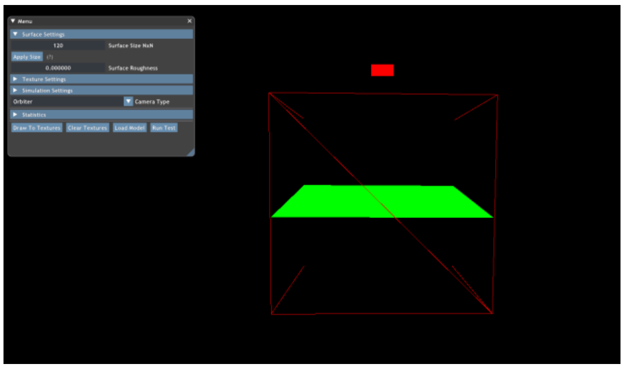
4.3.1 Black Box

Black box testing is done to ensure that the software is usable without looking at the source code. Those test cases are:

Test Case	Explanation
Change surface size	Test whether the size of the surface will be updated or not, see Figures 2.6.

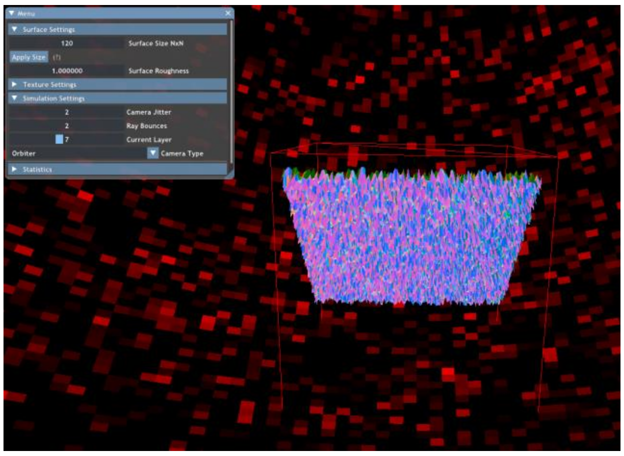
Change surface roughness	Test whether the surface roughness will be changed or not, see Figures 2.5a and 2.5b.
Change texture resolution	Test whether the texture resolution will be updated or not, see Figure 2.7.
Update camera jittering	Test whether the texture will be updated based on the camera jittering times or not, see Figure 2.10.
Increase ray bounces	Test whether more rays will hit the sky-map or not, see Figures 2.11a, and 2.11b.
Switch current layers	Test whether the layers (Sky-Map textures) will be change based on the inputted value or not, see Figures 2.12a, and 2.12b.
Camera types	Test whether the camera view will be changed based on the chosen type or not.
Drawing to textures button	Test whether the color of the texture is changed in descending order or not.
Clear textures button	Test whether the textures color is cleared or not.
View model	Test whether the scene will be changed to the model scene or not, see Figures 2.13b, 2.13c.
BRDF type	Testing whether the BRDF Settings subsection will be changed based on the chosen BRDF Type or not, see Figures 2.16, 2.21.
Cook-Torrance BRDF	Testing Cook-Torrance BRDF on a model, see Figures 2.17a, and 2.17b. And checking inner parameters such as albedo, roughness and metallic, see Figures 2.17a, 2.17b, 2.18a, and 2.18b.

Simulated BRDF	Testing the Simulated BRDF on a model, Figures 2.21.
Start simulation button	Testing if Start Simulation button will trigger the simulation.
Stop Simulation button	Testing if Stop Simulation button will turn off the simulation.
View Surface	Testing whether the scene will be changed to the surface scene or not.
Perfect Reflection	Testing the perfect reflection when the surface has 0 roughness, see Figure 4.6.
Scattered Reflection	Testing the Scattered Reflection when the surface has 1 roughness, see Figure 4.7.
Statistics	Testing the correctness of the returned statistics, see Figures 2.6, and 2.7
Total of 17 test cases passed!	



120 Surface Size NxN
0.000000 Surface Roughness
Orbiter Camera Type

Figure 4.6: Perfect reflection when roughness is low.



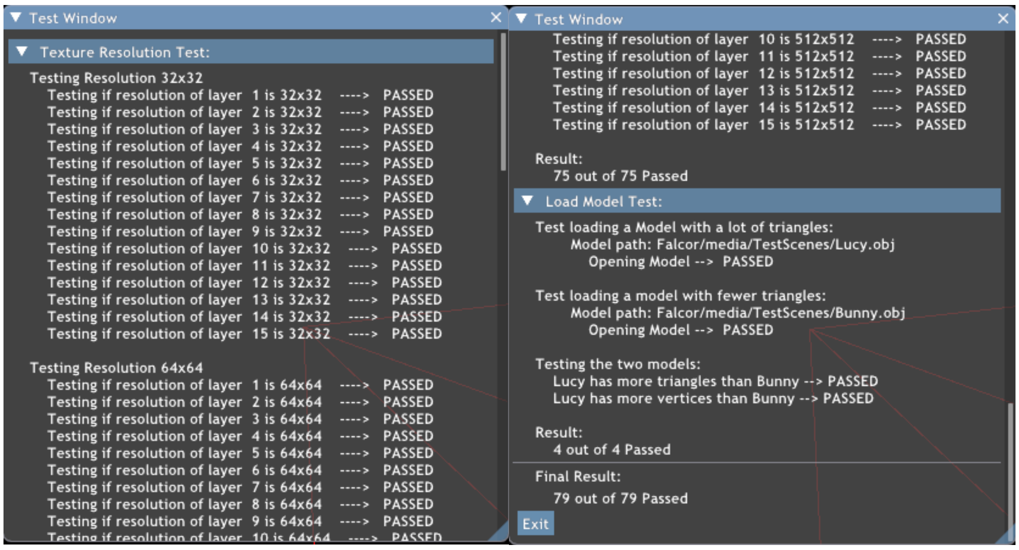
120 Surface Size NxN
1.000000 Surface Roughness
2 Camera Jitter
2 Ray Bounces
7 Current Layer
Orbiter Camera Type

Figure 4.7: Scattered reflection when roughness is high.

4.3.2 White Box

White box testing is done to ensure that code is done correctly, see Figure 4.8. The covered tests are the following:

Test Case	**Explanation**
Update textures	Tests whether the function *updateTextures()* will update all the 15 textures with all available resolutions {32x32, 64x64, 128x128, 256x256, 512x512}.
Load model from file	Tests the *loadModelFromFile()* when uploading two models, one with big size, and the other one has small size. Then comparing their triangle and vertices for validity.
Total of 79 test cases passed!	



```
Texture Resolution Test:

Testing Resolution 32x32
    Testing if resolution of layer  1 is 32x32   ---->  PASSED
    Testing if resolution of layer  2 is 32x32   ---->  PASSED
    Testing if resolution of layer  3 is 32x32   ---->  PASSED
    Testing if resolution of layer  4 is 32x32   ---->  PASSED
    Testing if resolution of layer  5 is 32x32   ---->  PASSED
    Testing if resolution of layer  6 is 32x32   ---->  PASSED
    Testing if resolution of layer  7 is 32x32   ---->  PASSED
    Testing if resolution of layer  8 is 32x32   ---->  PASSED
    Testing if resolution of layer  9 is 32x32   ---->  PASSED
    Testing if resolution of layer  10 is 32x32   ---->  PASSED
    Testing if resolution of layer  11 is 32x32   ---->  PASSED
    Testing if resolution of layer  12 is 32x32   ---->  PASSED
    Testing if resolution of layer  13 is 32x32   ---->  PASSED
    Testing if resolution of layer  14 is 32x32   ---->  PASSED
    Testing if resolution of layer  15 is 32x32   ---->  PASSED

Testing Resolution 64x64
    Testing if resolution of layer  1 is 64x64   ---->  PASSED
    Testing if resolution of layer  2 is 64x64   ---->  PASSED
    Testing if resolution of layer  3 is 64x64   ---->  PASSED
    Testing if resolution of layer  4 is 64x64   ---->  PASSED
    Testing if resolution of layer  5 is 64x64   ---->  PASSED
    Testing if resolution of layer  6 is 64x64   ---->  PASSED
    Testing if resolution of layer  7 is 64x64   ---->  PASSED
    Testing if resolution of layer  8 is 64x64   ---->  PASSED
    Testing if resolution of layer  9 is 64x64   ---->  PASSED
    Testing if resolution of layer  10 is 64x64   ---->  PASSED
```

```
    Testing if resolution of layer  10 is 512x512   ---->  PASSED
    Testing if resolution of layer  11 is 512x512   ---->  PASSED
    Testing if resolution of layer  12 is 512x512   ---->  PASSED
    Testing if resolution of layer  13 is 512x512   ---->  PASSED
    Testing if resolution of layer  14 is 512x512   ---->  PASSED
    Testing if resolution of layer  15 is 512x512   ---->  PASSED

Result:
    75 out of 75 Passed

Load Model Test:

Test loading a Model with a lot of triangles:
        Model path: Falcor/media/TestScenes/Lucy.obj
            Opening Model -->  PASSED

Test loading a model with fewer triangles:
        Model path: Falcor/media/TestScenes/Bunny.obj
            Opening Model -->  PASSED

Testing the two models:
    Lucy has more triangles than Bunny --> PASSED
    Lucy has more vertices than Bunny --> PASSED

Result:
    4 out of 4 Passed

Final Result:
    79 out of 79 Passed
```

Figure 4.8: Shows the methods test results.

5. Conclusion

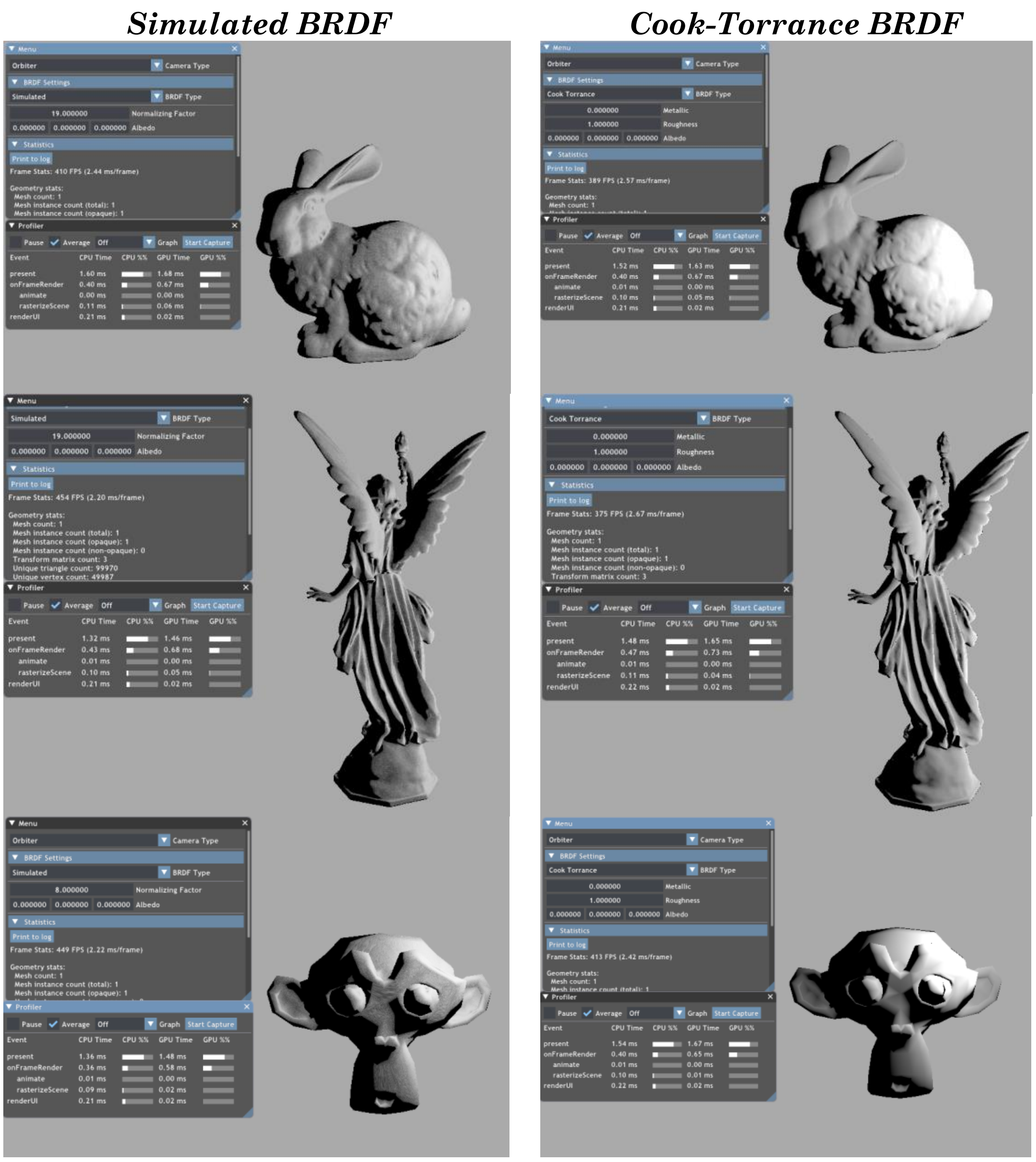
In this thesis, we present a simulation of Bidirectional Reflectance Distribution Functions. This study was inspired by the work of Rosana Montes and Carlos Ureña [2] that introduce a method to simulate the BRDFs models. An application using one of Nvidia researching tools, called Falcor, is shipped with the thesis.

By leveraging Nvidia's Falcor framework and exploring the theory behind physically based BRDFs, this thesis offers valuable insights and a practical implementation for the computer graphics community. Furthermore, a concise explanation of Bidirectional Reflectance Distribution Functions (BRDFs) and delves into the underlying theory of physically based BRDFs, specifically focusing on the microfacet model.

The developed application offers the capability to calculate and visualize the reflected incoming light rays from a rough surface. By simulating the interaction between light and the surface, the application generates a map that effectively captures the distribution of these reflected rays. This map serves as a visual representation of how the surface roughness influences the scattering and directionality of the reflected light. By providing a clear and intuitive depiction of the impact of surface roughness on the distribution of reflected rays, the application enhances our understanding of light behavior and facilitates analysis and comparison of different rough surface configurations.

One of the strengths of the software is its user-friendly interface, which enables users to view various models and experiment with both the *simulated BRDF* and the *Cook-Torrance BRDF*. This interactive aspect allows for real-time exploration and provides a valuable tool for researchers and practitioners in the field of computer graphics.

When comparing the simulated BRDF and Cook-Torrance BRDF in terms of performance, both approaches demonstrate similar capabilities. While the simulated BRDF utilizes pre-calculated values stored in textures, allowing for efficient retrieval and real-time rendering, the Cook-Torrance BRDF offers a physically-based approximation considering various surface properties. Despite their different implementation methods, both approaches yield comparable results in terms of rendering performance. Ultimately, whether opting for the simulated BRDF or the Cook-Torrance BRDF, developers can achieve realistic and visually appealing material rendering without significant differences in performance. Figure 5.1 illustrates the studied simulation of the BRDFs, along with Cook-Torrance model.



```
Orbiter                          Camera Type
BRDF Settings
Simulated                        BRDF Type
19.000000                        Normalizing Factor
0.000000  0.000000  0.000000     Albedo
Statistics
Frame Stats: 410 FPS (2.44 ms/frame)

Geometry stats:
  Mesh count: 1
  Mesh instance count (total): 1
  Mesh instance count (opaque): 1

Pause  ✓ Average  Off  Graph
Event            CPU Time   CPU %%   GPU Time   GPU %%
present          1.60 ms             1.68 ms
onFrameRender    0.40 ms             0.67 ms
  animate        0.00 ms             0.00 ms
  rasterizeScene 0.11 ms             0.06 ms
renderUI         0.21 ms             0.02 ms
```

```
Orbiter                          Camera Type
BRDF Settings
Cook Torrance                    BRDF Type
0.000000                         Metallic
1.000000                         Roughness
0.000000  0.000000  0.000000     Albedo
Statistics
Frame Stats: 389 FPS (2.57 ms/frame)

Geometry stats:
  Mesh count: 1

Pause  ✓ Average  Off  Graph
Event            CPU Time   CPU %%   GPU Time   GPU %%
present          1.52 ms             1.63 ms
onFrameRender    0.40 ms             0.67 ms
  animate        0.01 ms             0.00 ms
  rasterizeScene 0.10 ms             0.05 ms
renderUI         0.21 ms             0.02 ms
```

```
Simulated                        BRDF Type
19.000000                        Normalizing Factor
0.000000  0.000000  0.000000     Albedo
Statistics
Frame Stats: 454 FPS (2.20 ms/frame)

Geometry stats:
  Mesh count: 1
  Mesh instance count (total): 1
  Mesh instance count (opaque): 1
  Mesh instance count (non-opaque): 0
  Transform matrix count: 3
  Unique triangle count: 99970
  Unique vertex count: 49987

Pause  ✓ Average  Off  Graph
Event            CPU Time   CPU %%   GPU Time   GPU %%
present          1.32 ms             1.46 ms
onFrameRender    0.43 ms             0.68 ms
  animate        0.01 ms             0.00 ms
  rasterizeScene 0.10 ms             0.05 ms
renderUI         0.21 ms             0.02 ms
```

```
Cook Torrance                    BRDF Type
0.000000                         Metallic
1.000000                         Roughness
0.000000  0.000000  0.000000     Albedo
Statistics
Frame Stats: 375 FPS (2.67 ms/frame)

Geometry stats:
  Mesh count: 1
  Mesh instance count (total): 1
  Mesh instance count (opaque): 1
  Mesh instance count (non-opaque): 0
  Transform matrix count: 3

Pause  ✓ Average  Off  Graph
Event            CPU Time   CPU %%   GPU Time   GPU %%
present          1.48 ms             1.65 ms
onFrameRender    0.47 ms             0.73 ms
  animate        0.01 ms             0.00 ms
  rasterizeScene 0.11 ms             0.04 ms
renderUI         0.22 ms             0.02 ms
```

```
Orbiter                          Camera Type
BRDF Settings
Simulated                        BRDF Type
8.000000                         Normalizing Factor
0.000000  0.000000  0.000000     Albedo
Statistics
Frame Stats: 449 FPS (2.22 ms/frame)

Geometry stats:
  Mesh count: 1
  Mesh instance count (total): 1
  Mesh instance count (opaque): 1

Pause  ✓ Average  Off  Graph
Event            CPU Time   CPU %%   GPU Time   GPU %%
present          1.36 ms             1.48 ms
onFrameRender    0.36 ms             0.58 ms
  animate        0.01 ms             0.00 ms
  rasterizeScene 0.09 ms             0.02 ms
renderUI         0.21 ms             0.02 ms
```

```
Orbiter                          Camera Type
BRDF Settings
Cook Torrance                    BRDF Type
0.000000                         Metallic
1.000000                         Roughness
0.000000  0.000000  0.000000     Albedo
Statistics
Frame Stats: 413 FPS (2.42 ms/frame)

Geometry stats:
  Mesh count: 1

Pause  ✓ Average  Off  Graph
Event            CPU Time   CPU %%   GPU Time   GPU %%
present          1.54 ms             1.67 ms
onFrameRender    0.40 ms             0.65 ms
  animate        0.01 ms             0.00 ms
  rasterizeScene 0.10 ms             0.01 ms
renderUI         0.22 ms             0.02 ms
```

Figure 5.1: Shows both models capabilities on different objects (Bunny, Lucy, and suzanne). The Simulated Results to the left, and Cook-Torrance Results to the right.

Finally, a space for improvements can be done, and more methods and algorithms may be added to the application. Also, enhancing the performance of the simulation through the use of Parallax Mapping is a valid consideration.

6. Bibliography

[1] Pacanowski, Romain & Salazar Celis, Oliver & Schlick, Christophe Granier, Xavier & Poulin, Pierre & Cuyt, Annie. (2012). Rational BRDF. IEEE Transactions on Visualization and Computer Graphics. 18. 1824-1835. 10.1109/TVCG.2012.73.

[2] Rosana Montes and Carlos Ureña. "An Overview of BRDF Models". In: (2012).url:https://digibug.ugr.es/bitstream/handle/10481/19751/rmontes_LSI-2012-001TR.pdf.

[3] Asseel Ghaith Jassem Al-Mahdawi. BRDF Simulator. 2023. url: https://github.com/RealGorandos/BRDF_Simulator/tree/development

[4] Hofmann, G.R. (1990) 'Who invented ray tracing?', *The Visual Computer*, 6(3), pp. 120–124. doi:10.1007/bf01911003.

[5] Appel, A. (1968) 'Some techniques for shading machine renderings of Solids', *Proceedings of the April 30--May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)* [Preprint]. doi:10.1145/1468075.1468082.

[6] Kajiya, J.T. (1986) 'The rendering equation', *ACM SIGGRAPH Computer Graphics*, 20(4), pp. 143–150. doi:10.1145/15886.15902.

[7] Cook, R.L. (1986) 'Stochastic sampling in Computer Graphics', *ACM Transactions on Graphics*, 5(1), pp. 51–72. doi:10.1145/7529.8927.

[8] Scratch a pixel. url: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/introduction-quasi-monte-carlo.html

[9] Learnopengl. url: https://learnopengl.com/PBR/Theory

[10] Blinn, J.F. (1977) 'Models of Light Reflection for computer synthesized pictures', *ACM SIGGRAPH Computer Graphics*, 11(2), pp. 192–198. doi:10.1145/965141.563893.

[11] Torrance, K.E. and Sparrow, E.M. (1967) 'Theory for off-specular reflection from roughened surfaces*', *Journal of the Optical Society of America*, 57(9), p. 1105. doi:10.1364/josa.57.001105.

[12] Cook, R.L. and Torrance, K.E. (1981) 'A reflectance model for computer graphics', *ACM SIGGRAPH Computer Graphics*, 15(3), pp. 307–316. doi:10.1145/965161.806819.

[13] Akenine-Mo¨ller, T., Haines, E., & Hoffman, N. (2018). Real-Time Rendering, Fourth Edition (4th ed.). A K Peters/CRC Press. https://doi.org/10.1201/b22086

[14] Huygens-Fresnel principle. url:
https://en.wikipedia.org/wiki/Huygens%E2%80%93Fresnel_principle

[15] Schlick, C. (1994) 'An inexpensive BRDF model for physically-based rendering', *Computer Graphics Forum*, 13(3), pp. 233–246. doi:10.1111/1467-8659.1330233.

[16] Brian Karis from Epic Games. url:
http://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html

[17] Eric Heitz. "Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs". In: Journal of Computer Graphics Techniques (JCGT) 3.2 (2014), pp. 48–107. issn: 2331-7418. url: http://jcgt.org/published/ 0003/02/03/.

[18] Michael Walczyk. url: https://michaelwalczyk.com/blog-ray-marching.html

[19] Möller, T. and Trumbore, B. (1997b) 'Fast, minimum storage ray-triangle intersection', *Journal of Graphics Tools*, 2(1), pp. 21–28. doi:10.1080/10867651.1997.10487468.

[20] Scratch a pixel, url: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates.html

[21] Scratch a pixel, url: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.html

[22] Line-Line intersection. url:
https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection

[23] Falcor Framework Repository. url: https://github.com/NVIDIAGameWorks/Falcor

[24] Singam, Panini & Narendranath, Shyama & Sreekumar, Parameswaran. (2015). Design and development of sof tx-ray multi-layer optics. 10.13140/RG.2.1.1828.8082.