A Multifragment Renderer for Material Aging Visualization

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Georgios Adamopoulos

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN SOFTWARE

University of Ioannina
March 2018

Examining Committee:

- **Ioannis Fudos**, Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Panos Vassiliadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Kostas Magoutis**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

Dedicated to my parents, Kostas and Voula, my brother Dimitris and my girlfriend Renia, because nothing I have achieved so far would be possible without their support and understanding.

Acknowledgements

I have to thank my supervisor Professor Ioannis Fudos for his guidance throughout my Master's Degree and all the opportunities he gave me to prove myself. Also I would like to thank all my friends that stood by me and helped me by any means. Special thanks to my colleagues Vangelis Eftaxopoulos for all the advices as a senior graduate student and Anastasia Moutafidou who also had to put up with me in everyday basis.

Table of Contents

List of Figures iii

List of Algorithms v

Glossary vi

Abstract vii

Εκτεταμένη Περίληψη viii

1 Introduction 1

1.1 Related Work 3

1.2 Thesis Structure 5

2 Theoretical Background 6

2.1 Rendering 6

2.1.1 OpenGL 7

2.1.2 Shaders 7

2.2 3D Models 8

2.2.1 Mesh 9

2.2.2 Texture Maps 10

2.3 Multipass Rendering 14

2.3.1 Order Independent Transparency 14

2.4 PBR 15

3 Our Method 18

3.1 Physical Based Rendering 18

3.2 Order Independent Transparency 20

4 Implementation **25**
4.1 Shader Loading 25
4.2 Shaders Implementation 27
4.3 Draw functions 37
4.4 Model Loading 39
4.4.1 Class model 39
4.4.2 Class mesh 45
4.5 Graphical User Interface 48

5 Experiments and Results **50**

6 Conclusion and Future Work **59**

Bibliography **60**

List of Figures

2.1 Sphere mesh with no textures 10
2.2 sphere with diffuse texture 10
2.3 Diffuse Map 11
2.4 Sphere mesh with no textures 11
2.5 Sphere with normal texture 11
2.6 Normal Map 12
2.7 Sphere mesh with no textures Maps 13
2.8 Sphere with Roughness and Metallic texture 13
2.9 Roughness/Gloss Map 13
2.10 Specular/Metallic Map 13
2.11 Sphere mesh with no texture maps 14
2.12 Sphere with every texture applied 14
2.13 Rough Surface 15
2.14 Roughness values and their result in reflection 16
2.15 How geometry affects shadowing 17
2.16 Fresnel effect 17

4.1 Main Panel of our tool and its components 48

5.1 Tool performance per Object and screen fill 51
5.2 Multiple Layer performance 52
5.3 Order Independent Transparency performance without PBR 52
5.4 PBR performance without Order Independent transparency 53
5.5 Geometry of the model 54
5.6 Geometry + Normal Texture 54
5.7 Geometry + Normal + Diffuse 55
5.8 Geometry of shield 55

5.9 Geometry + Normal Texture 55
5.10 Geometry + Normal + Diffuse 56
5.11 Outer Layer 57
5.12 Middle Layer 57
5.13 Inner Layer 58

List of Algorithms

3.1 PBR computation Algorithm for a single fragment 20
3.2 $Initialize\,Buffer$ Algorithm. 21
3.3 Render Buffer Algorithm. 22
3.4 Display Buffer Algorithm 24
3.5 Filter Fragments Algorithm 24

Glossary

abbreviation	term
PBR	Physical Based Rendering
VFX	Visual Effects
GPU	Graphics Processing Unit
OpenGL	Open Graphics Language
GLSL	OpenGL Shading Language
AO	Ambient Occlusion
RGB	Red Green Blue
BRDF	Bidirectional reflectance distribution function
NDF	Normal Distribution Function
GUI	Graphical User Interface

Abstract

Georgios Adamopoulos, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, March 2018.
A Multifragment Renderer for Material Aging Visualization.
Advisor: Ioannis Fudos, Professor.

We report on the development of a visualization tool for material aging of cultural heritage artifacts. The tool is aimed at curators, archaeologists and other users that wish to observe, visualize and prevent the process of material aging in artwork objects. The tool combines state of the art multi-fragment and physical-based rendering techniques and is built based on conclusions drawn from measurements on naturally or artificially aged objects which are comprised of multiple layers.
We introduce a method for realistically rendering outer surfaces and inner stratification with the use of transparency in real-time. The method works in three rendering passes using three buffers for storing and manipulating fragments (fragments are parts of the mesh that correspond to one pixel).
The algorithms have been tuned to provide better interaction and comprehension of the results by the user. We offer a comparative evaluation of performance and rendering quality with existing techniques.

Εκτεταμένη Περίληψη

Γεώργιος Αδαμόπουλος, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Μάρτιος 2018.
Πολυθραυσματική απόδοση για την οπτικοποίηση της σύνθεσης παλαιωμένων αντικειμένων.
Επιβλέπων: Ιωάννης Φούντος, Καθηγητής.

Σε αυτή την εργασία παρουσιάζουμε την ανάπτυξη ενός εργαλείου οπτικοποίησης με φωτορεαλιστική απόδοση. Η χρήση του προορίζεται για αρχαιολόγους, συντηρητές αλλά και απλούς χρήστες με σκοπό την απεικόνιση, παρακολούθηση και συντήρηση αντικειμένων πολιτισμικού ενδιαφέροντος. Η ανάπτυξη των αντικειμένων προς παρακολούθηση είναι αποτέλεσμα διάφορων μετρήσεων από τεχνικές ανακατασκευής επιφανειών, τόσο εξωτερικών αλλά και εσωτερικών, μέχρι χημικής ανάλυσης της σύστασης των υλικών τους ώστε να παραχθεί λεπτομερής προσομοίωση τόσο των εξωτερικών επιφανειών όσων και των εσωτερικών διαστρωμάτων.

Το σύστημα συνδυάζει τεχνικές πολυθραυσματικής απόδοσης αλλά και απόδοσης φυσικών ιδιοτήτων υλικών. Το PBR, δηλαδή η απόδοση υλικών με βάση τις ιδιότητές τους είναι μία σύγχρονη τεχνική που χρησιμοποιείται από μεγάλες κινηματογραφικές παραγωγές για την παραγωγή ρεαλιστικών εφέ ή και εξ ολοκλήρου animation ταινίες. Ακόμη πλέον θεωρείται μία από τις πλέον κοινές τεχνικές που χρησιμοποιούνται από εταιρείες παραγωγής ηλεκτρονικών παιχνιδιών για να προσδώσουν ρεαλισμό σε πραγματικό χρόνο. Βασίζεται στην υλοποίηση της BRDF συνάρτησης ανάκλασης και υλοποιείται σε fragment shaders προγράμματα γραφικών. Η χρήση της προϋποθέτει την ύπαρξη Texture Maps για την απόδοση λεπτομερειών στις επιφάνειες των υλικών και σωστής επίδρασης του φωτός σε αυτές.

Η δεύτερη τεχνική που αναπτύξαμε έχει ως σκοπό την σωστή απόδοση των εσωτερικών τους διαστρωματώσεων με την χρήση διαφάνειας σε πραγματικό χρόνο. Η

μέθοδος βασίζεται στον A-buffer και την επεξεργασία των fragments με σκοπό την διάταξη τους κατά βάθος ανά θέση pixel. Η μέθοδός μας χρησιμοποιεί τρία περάσματα κατά τη διαδικασία απόδοσης, ένα για την αρχικοποίηση ένα για την απόδοση χρώματος και την αποθήκευση καθώς και ένα για την διάταξη οπτικοποίηση των αποθηκευμένων fragments στον χρήστη. Κατά τον χειρισμό του αντικειμένου ο χρήστης δίνει μια επιλογή για το βάθος που θέλει να έχει καλύτερη οπτική. Το πρόγραμμα εκμεταλλεύεται την επιλογή αυτή παραμετροποιώντας κατάλληλα την μέθοδο. Τα fragments αποθηκεύονται σε τρείς buffer, απαραίτητων για την ευρετηριοποίηση, την αποθήκευση και την επεξεργασία τους.

Ιδιαίτερη σημασία έχει δοθεί στην παραμετροποίηση των αλγορίθμων για καλύτερη αλληλεπίδραση και κατανόηση των αποτελεσμάτων από τον χρήστη. Παρουσιάζουμε τις μεθόδους που αναπτύξαμε και συνδυάσαμε, τη σχεδίαση και ανάπτυξη του εργαλείου καθώς και αποτελέσματα σύγκρισης με υπάρχουσες τεχνικές, τόσο ως προς την αποδοτικότητα όσο και ως προς την ποιότητα της απεικόνισης. Σε μελλοντικά βήματα θα μπορούσαν να μπουν βελτιώσεις μετά από προτάσεις χρηστών καθώς και παραμετροποίηση αλγορίθμων ως προς την καλύτερη διαχείριση μνήμης καθώς και άλλες τεχνικές απεικόνισης για ακόμη πιο αληθοφανή αποτελέσματα.

Chapter 1

Introduction

1.1 Related Work
1.2 Thesis Structure

The preservation of Cultural Heritage artifacts is a very tedious but important process. Each object is crafted using selected materials that age differently and can have large impact in the appearance of it through time. Also an artifact may have different layers of materials, such as a painting with multiple coatings of colors, which decay separately and may even affect the volume of an object. Archaeologists and curators bear the responsibility to study the aging process of each material and then apply that knowledge to restore artifacts to prior condition and prevent further decay. In this context, it is imperative to have a tool that can visualize multi-layered objects in different time frames and be used by scientists while curating or studying certain artifacts.

This kind of work however require great precision and the visualization needs to be almost a simulation of real-world materials, which in traditional Computer Graphics require heavily detailed models, something that makes performance a serious issue. In this thesis we report the development of a visualization tool that gives the ability to observe physical materials both along their surface and also with their inner layers, in different points of time. Such a tool can be extended with additional functionality needed by such scientists and is able to become cross-platform if needed.

Multiple sensors are used to capture the information of an artifact needed for rendering. Some sensors are used to gather information about the color of the surface ,others are used to capture details about the texture of the surface while others calculate the volume of the artifact.

Microprofilometer, which is a laser scanner, gives us high detail information for the surface of a material. We generate Normal Maps, Roughness maps and Displacement maps based on the information we gather from it, as well as a triangulated mesh depicting the geometry of an object. However Microprofilometer can be used to scan small surfaces only, which leads us to photogrammetry techniques to obtain the entire geometry of an artifact.

Reflectance Transformation Imaging measurements, or RTI, use multiple photographs of the same object lit from different directions each time, resulting in valuable information about the reflectance abilities of its surface. We take Albedo Maps, which depict the color of a surface when is evenly lit and Metallic Maps which dictate how the light is reflected in different areas of the surface. RTI can also provide Ambient Occlusion Maps which are used for better shading in static lighting and also Normal Maps for the given surface which can be used for cross-validation with the Normal Maps from the Microprofilometer.

Ultrasound Measurements are used to calculate the inner volume of the surface and give us details about inner layers of an object which are not visible to the eye. Such details are transformed into triangulated meshes and are added to the general geometry of an object.

The data collected from these sensors need to be processed and presented in a certain format in which we will import them into our method to visualize and interact with the user. For example, the Microprofilometer data refer to small areas of an object, so analysis of them is required in order to simulate the overall surface properties before rendering.

However, the detail obtained from these sensors can be very high, resulting in very large quantity of data which can be not accepted. We encode the information in texture maps to reduce the density of the data for the rendering process.

In computer graphics visualizing 3D models using computer programs is called Rendering, and with the modern Graphical Processing Units, or GPUs, it has become efficient to render high-detailed 3D models such as artwork objects. The introduction of Shaders and their massive parallel computations made the rendering pipeline

achievable in real-time, when specific conditions are met, concerning the input data.

Our visualization tool is developed on top of a multi-fragment renderer using vertex and fragment shaders. We use modern rendering techniques, algorithms that process multiple fragments per pixel to render aging materials on artwork objects. Transparency gives user the ability to observe, both outside and inside surfaces of an artifact with low cost and in an intuitive way. Also the user has the ability to see the changes made to an object during different time periods, when provided with sufficient data. The aging data are either different versions of the entire object, or changes of characteristics, based on real-world aging measurements.

Rendered objects consist of two definite types of data, triangulated mesh which gives us a rough approximation of the geometry both in outer surface and inner layers, and texture maps which describe detailed properties each given layer. The Physically Based Rendering technique, or PBR technique which is widely used nowadays in Game and Film Industry, uses these types of data to realistically portray real world objects in virtual world with as little processing power required as possible.

In summary, in this thesis we make the following contributions:

- We develope a photorealistic visualization tool for Objects comprised of simulated aging materials generated from multiple sensors
- We give the ability to observe selected inner aspects of materials, using a multi-fragment method we developed for transperency.
- Achieved real-time rendering performance for very large objects

1.1 Related Work

While rendering objects using usual materials, aging information is of great importance because it adds realism to the result. Cracks, dents, corrosion or even cosmetic detail such as dust, smudges, fingerprints contribute in more natural looks, while an artifact rendered with spotless, ”pure” materials, seem too clean and smooth to be true. Cosmetic detail are usually added on a surface as extra textures blended with original ones, while aging details is another story. Thats way there is a lot of work done trying to simulate aged materials with success. In [1], a classification of different types of degradation is made, in an attempt to better visualize decay of different

materials in Computer Graphics. Degradation is categorized based on the phenomena causing decay, the materials affected by such phenomena, with varied effectiveness, and the different types of surface degradation as a result. In [2], the author categorizes the morphology decay types of materials instead of only outer surface degradation and the result of the general appearance of an object. These degradation types affect the volume of an aged model which is imperative to the realism of the render, for example a decaying fruit that shrinks along its decay phases.

Another subject that needs to be addressed is the appearance of a model that is not yet manufactured, or is based on an original artifact that is heavily affected by time, or other physical phenomena. Of course we do not mean the restoration of broken parts of an object, for example the broken arms of a statue, but only the material surface decay. There are many techniques used to predict the appearance of a yet to be manufactured object. Usually artists create the appearance manually and based on their perspective. In order to lower the labor of creating them manually, simulation based on sample materials is used to render a result. In [3], methods for collecting sample data needed for rendering aging materials are described, as well as ways to make use of such sample data in simulating objects of various shapes and morphology. While there are many simulating algorithms for aging materials, the most common issue is that the do not scale well in larger areas, thus making it impossible to render large scenes using these data without tedious work done by Artists. In [4] a method to well scale such algorithms is introduced that also simulates aging in large scenes smoothly.

To visualize different layers of objects separately we need to render transparency efficiently using Multi-fragment Rendering. Depth-ordered fragment determination is used in 3D game and VFX industry to achieve numerous appealing and special visual effects in graphics applications. That method is imperative for the rendering of a variety of algorithms in interactive speeds, such as photorealistic rendering, order-independent transparency for forward, deferred, volumetric shading and shadowing, visualizing hair and solid geometry.

In [5] the authors provide an overview of depth-aware methods for rendering. They also provide modifications for such algorithms in order to overcome the problem of $coplanarity$, or $Z - fighting$, which is the phenomenon in three-dimensional rendering where some rendering primitives have identical or similar values in the Z-buffer. That particularity leads to 1) either intersecting surfaces, 2) overlapping

surfaces, when primitives are coplanar and overlap or 3) non-convergent surfaces due to precision errors.

[6] introduces S-buffer, a variation of A-buffer architecture, which is efficient and memory friendly. Alternate techniques implementing A-buffer require linked-lists or fixed arrays to store fragments in the shared GPU memory, which scale well and run in linear time regarding the number of fragments, but they require increased amounts of GPU. Their approach includes an additional rendering pass in which they count the fragments per pixel, thus allocating the exact amount of memory for storing fragments. The result is an implementation which takes advantages of fragment distribution and the sparsity of pixel-space with improved memory usage and performance with the cost of one extra rendering pass. We employ this technique to achieve real-time rendering of material aging information of artwork objects.

1.2 Thesis Structure

The thesis is composed of six chapters.

In the Chapter 2, called Background, we will provide information regarding any specific meaning so that the reader can be familiarized with rendering, shader programming and any other tool which was used to our work.

In Chapter 3, the methods we developed will be presented.

In Chapter 4, the implementation of our methods will be presented with more technical details

In Chapter 5, certain experiments will be presented in order to evaluate the performance and the rendering results of our tool

In Chapter 6 we conclude our work and we propose future work based on this thesis.

Chapter 2

Theoretical Background

2.1 Rendering

2.2 3D Models

2.3 Multipass Rendering

2.4 PBR

2.1 Rendering

In Computer Graphics rendering is the automated process of creating an image based on a collection of 2D or 3D models, which are properly loaded into memory. The collection of such models is commonly named a scene and computer graphics applications take responsibility in loading all the models in memory and preparing data for rendering. After that the rasterization process begins in which the scene is separated to fragments. A fragment is the collection of information needed to visualize a single pixel in the frame buffer, which is achieved by the fragment operations. This process as a whole is called rendering pipeline and can be fully programmed in modern computer graphics with the development of shader programs. There is also the need for programmers to communicate with these shader programs and load the data into pipeline.

2.1.1 OpenGL

That role is covered by APIs responsible to interact with the *Graphics Processor Unit* (*GPU*) with OpenGL or Open Graphics Library being one of them. OpenGL is a cross-language, cross-platform library containing a set of function along with named integer constants. It is used in almost all kinds of Computer Graphics applications, ranging from standard image rendering to 3D animation and Computer Games. The latest version is 4.6 and while every version is backwards compatible, because it is designed to be entirely implemented in hardware, there might be the need to use earlier versions for maximum compatibility.

2.1.2 Shaders

As we talked about before a shader is a type of computer program that is used to program the rendering pipeline. They were originally used fro the computation of shading values, which means the combination of light, darkness and color in an image, however due to their high parallel abilities they are used in a variety of applications, not only in computer graphics, but even topics with no relativity with them at all.

With these kind of programs it has been made possible to create custom effects, against fixed-function pipeline that was the standard before their introduction, and manipulate every aspect of pixel, vertices or textures, in interactive speeds. The algorithms responsible for these effects are developed in shader programs and are externally loaded with data and modified using variables, during the calling of a shader.

Shaders are divided in types, regarding the objective they serve. The types of shaders we concern ourselves with are the Fragment Shaders and the Vertex Shaders, apart from which there are Geometry Shaders, Tesselation Shaders, Compute Shaders and Primitive Shaders, whose abilities were not necessary in our work.

Vertex Shader

The first kind of shader program created were the Vertex Shaders. Such a shader is called once for each vertex of the scene and has the ability to manipulate its position, color, texture coordinates or other properties. The purpose of it is to transform the local 3axis coordinates of each vertex to the equivalent 2D value in which it appears on screen along with the depth value passed into z-buffer. The result of the Vertex

Shader is passed to the next step of the rendering pipeline, in our case the Fragment Shader

Fragment Shader

Fragment shaders or pixel shaders, compute basically the color and also other attributes of each pixel rendered. They are mostly responsible for applying the shading algorithms that compute how the light influences each fragment of the render, and in the same context, they apply the more evolved algorithms for shading photorealistically a scene. Algorithms that take into account texture maps, such as PBR pipeline we used in our work, have most developing done in fragment shaders.

In general fragment shaders are programs that have "access" only in pixels and not in the geometry of the scene, which makes them unable to manipulate them in such way. However, because of the information they hold about pixels and their neighbors they can be used to create certain effects, such as blur, or edge detection and can also be used alone in post-processing of images or videos already rasterized. That ability gives them versatility and makes them really useful for creating certain effects.

2.2 3D Models

A strictly defined structured is followed when creating a 3D model for rendering. The data that constitute the model fall into three different categories and subcategories as seen below:

- Vertices

  Position

  Vertex Normals

  Vertex Color

  Texture Coordinates

- Faces

- Texture Maps

Diffuse map

Normal map

Metallic map

Roughness map

Displacement map

Generally Vertex and Face data comprise the mesh of a model, while textures refer to the visual details that the mesh can't reproduce, or more commonly, is too expensive to be rendered with mesh data. In certain formats such as $OBJ\ Wavefront$ file format the vertex and face data are included in a single file with extension $.obj$ while textures are present as image files in the filesystem. In that case there is usually a material file $.mtl$ with information on texture referencing and types, that are included in said mesh.

2.2.1 Mesh

Each vertex is a data structure that refers to a single point of a mesh and holds information about certain abilities of said point. The necessary information a vertex holds is the position in 3D space, usually given in local coordinates around the center of 3-axis (0.0 , 0.0, 0.0). Without position data nothing can be rendered, while other information needed can be exported from them if missing.

Vertex Normal is a directional vector that is associated with given vertex and is computed by the Face in which is a part of. They are used in many lighting models like *Gouraud* or *Phong* shading and the are need to achieve smoother shading than the flat one created by Faces.

Each vertex might have a color value in RGB format so that in rendering color appears. Certain file format do not support Vertex Color data by default, such as traditional OBJ, however there are different editions of them with proper implementation of color. In most modern graphics applications, ours included, Vertex Color is not needed due to the presence of diffuse texture maps that represent higher coloring detail than vertex color. In that case, the space not allocated to vertex color data can be used to load information not typically found in vertex structure and achieve various effects, for example use of mesh indexing in which vertex belongs to.

The combination of image (textures) and mesh data is achieved through wrapping 2d images around a 3D object. During this procedure each vertex of a 3D mesh is connected with a pixel in an image file. Because x, y, z coordinates are already used in vertex position, UV coordinates are used to point to that certain pixel.

2.2.2 Texture Maps

Image files used in the texture mapping method, in which an texture map is wrapped around a surface of a mesh to add higher resolution detail than the one provided by default. Maps can either be static data, or procedurally created images. By procedurally creating images based on algorithms it is really easy to render large areas with not repeated textures without resolution problems. Usually it procedural textures are used to render certain materials that have a randomness in their texture such as wood, marble, granite etc.

At first what started as texture mapping later was renamed to diffuse texturing, means adding color detail (Fig. 2.2), while more complex types of textures where invented to enrich a model. In PBR rendering pipeline diffuse maps are also called albedo maps, with sole difference being that the albedo map does not include shades in it so that the rendering is only affected by the light present in our scene.

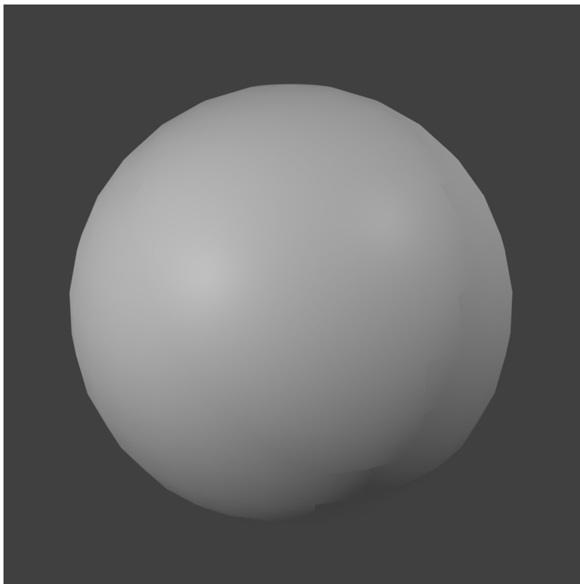

Figure 2.1: Sphere mesh with no textures

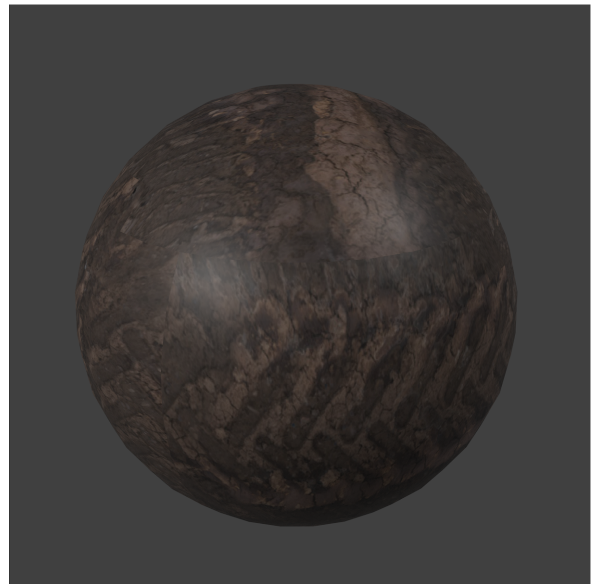

Figure 2.2: sphere with diffuse texture

Figure 2.3: Diffuse Map

Normal Texture Maps is the equivalent of vertex normals but with more information, essentially faking bumps, dents and other surface irregularities when computing shading algorithms (Fig. 2.5). They are one of the most common techniques to increase the resolution of a model dramatically without high-resolution meshes. They are images with RGB information in which 3D vector direction is encoded (Fig. 2.6). This information is used in fragment shaders when computing light radiance.

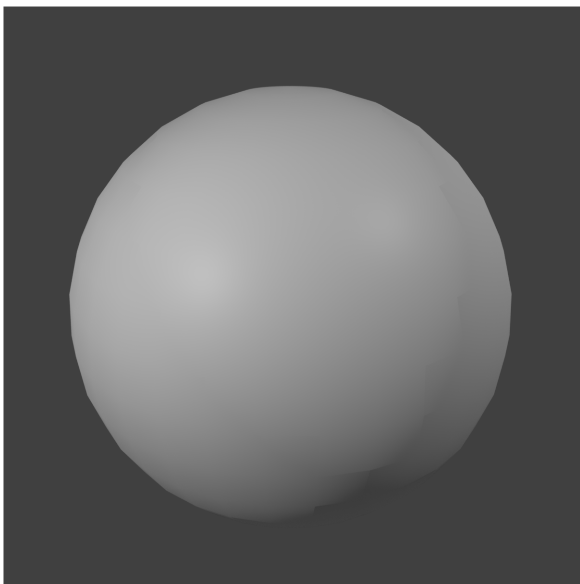

Figure 2.4: Sphere mesh with no textures

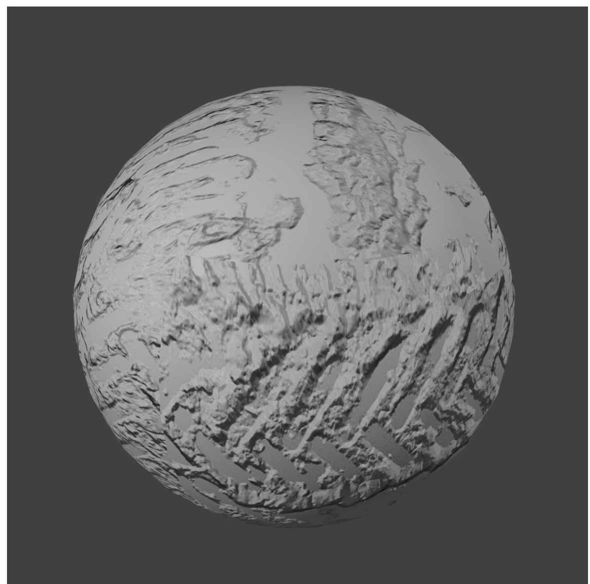

Figure 2.5: Sphere with normal texture

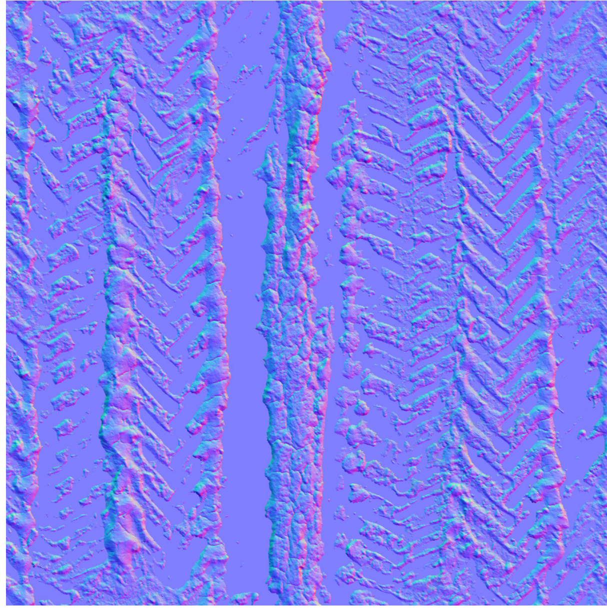

Figure 2.6: Normal Map

Metallic Textures maps are usually grayscale images (Fig. 2.10), or in some cases black and white, that determine if each fragment of the mesh is metal or not. This is needed for the renderer to know the amount of reflectance ability that a surface has.

Supplement to Metallic Texture Maps, the Roughness maps are grayscale images (Fig. 2.9) that provide information on how rough a surface is, metallic or not, thus determining how wide and blurry are the reflections captured by it (Fig. 2.8). Typically rough surfaces have wide and blurry reflections while smooth ones reflect smaller and clearer. In some cases, depending on the workflow of a renderer, instead of Roughness maps there might be a need of Smoothness/Gloss maps which are the inverted equivalent of Roughness maps.

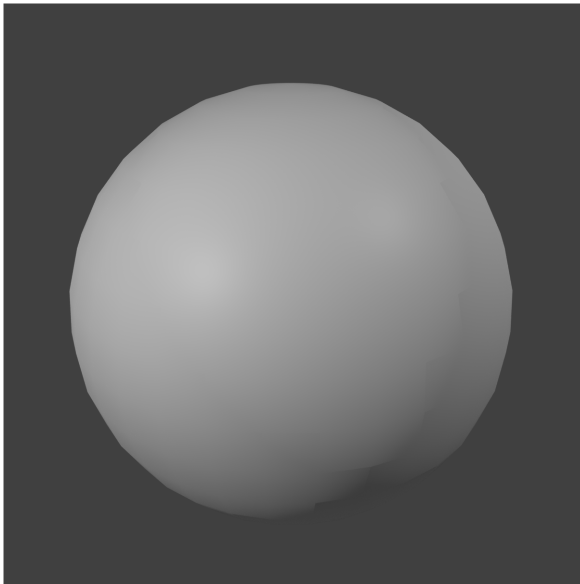

Figure 2.7: Sphere mesh with no textures Maps

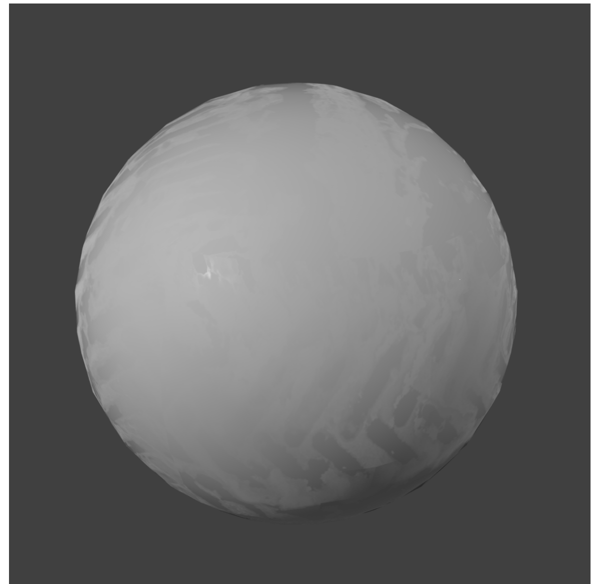

Figure 2.8: Sphere with Roughness and Metallic texture

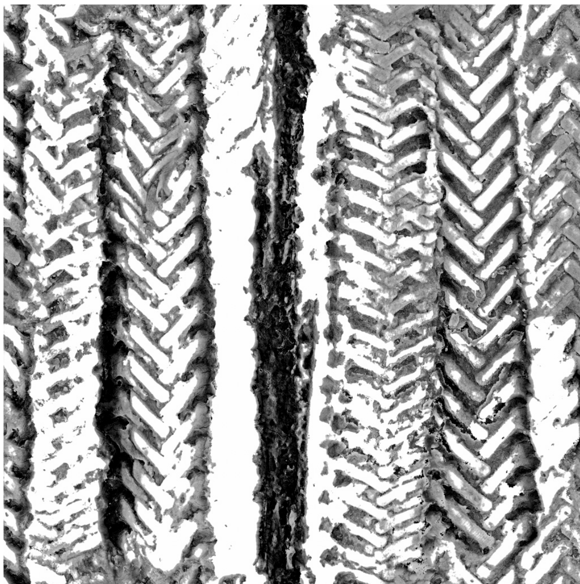

Figure 2.9: Roughness/Gloss Map

Figure 2.10: Specular/Metallic Map

Displacement maps are texture maps that are used in height mapping technique. Typically they contain information about distance of displacement of a fragment, along its' normal axis, encoded in grayscale images. The white value represent the maximum positive displacement while black the maximum negative, while middle value 0.5 represent no displacement whatsoever.

The result as we can see (Fig. 2.12) is very realistic and it only uses the number

of vertices the sphere originally had (Fig. 2.11)(482 vertices), while if we were to reproduce that result using real geometry we would might need millions of vertices making storing size many times bigger and rendering times not real-time.

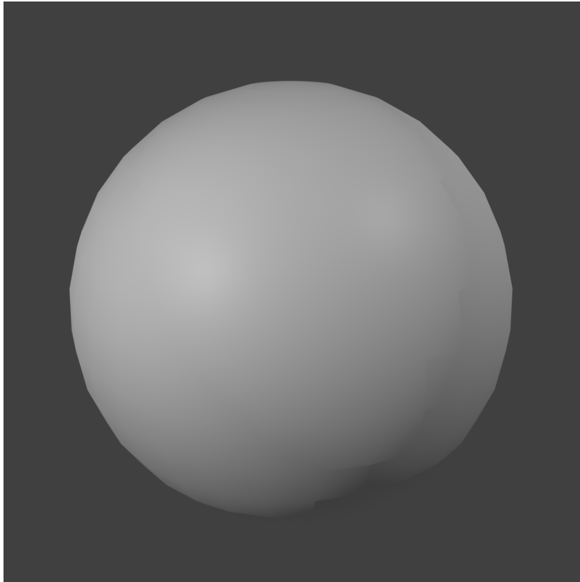

Figure 2.11: Sphere mesh with no texture maps

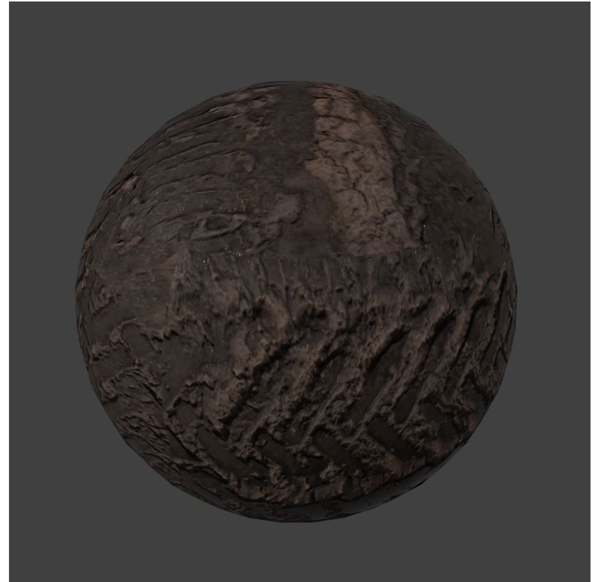

Figure 2.12: Sphere with every texture applied

2.3 Multipass Rendering

Each OpenGL shader renders output to framebuffers with screen being the most common one and that's why it is also the default one. For simple needs that pipeline is more than enough but it is limited to implementing only Vertex and Fragment shaders, which themselves are confined when in need of more complex effects. An output of a pass however, can be used as input to another one, thus expanding the options of that pass. This can be useful in cases where the output might not be in a displayable format making the second pass necessary for visualization.

2.3.1 Order Independent Transparency

In our case where transparency is needed and the performance is crucial, Order Independent Transparency was the way to go. This method is a perfect example of why the multi-pass rendering is needed, due to the preprocess the data needed before visualization. The traditional way to achieve transparency is through *Alpha*

$Blending$, where the color of each fragment in different depths is blended according to the alpha value of the fragment. The equation implemented was the following:

$$RGB_d = A_s \times RGB_s + (1.0 - A_s) \times RGB_d \tag{2.1}$$

where: RGB is referring to the color value of the fragment and A to the Alpha value while s and d mean source and destination fragment accordingly

2.4 PBR

The appearance of an object need to be realistic, or as said in computer graphics photorealistic. The amount of darkness, lightness and color values that contribute to the fragment RGB value are given by shading algorithms that are more or less based on the same underlying theory which more closely matches the behavior of light in the physical world. $Phong$, $Blinn - Phong$, or $Cook - Torrance$ are common shading algorithms present in computer graphics for many years. Our work however is based on PBR rendering, or Physically Based Rendering, which is a collection of render techniques that aim in mimicking light in a physically plausible way which gives better results than traditional lighting algorithms.

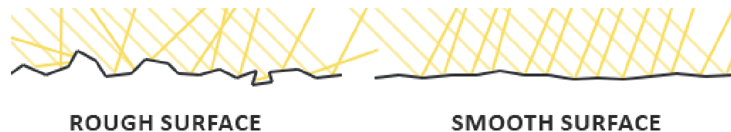

Figure 2.13: Rough Surface

It is based on the theory of microfacets, energy conservation, radiometry and $BRDF$ function. Microfacets are basically a theory that says that any surface under microscopic scale can be seen as an array of small mirrors. The roughness of the surface can drastically change the way these mirrors are aligned. That way a rough surface has wide and blurry specular reflections while a smooth one has sharper and stronger specular reflection

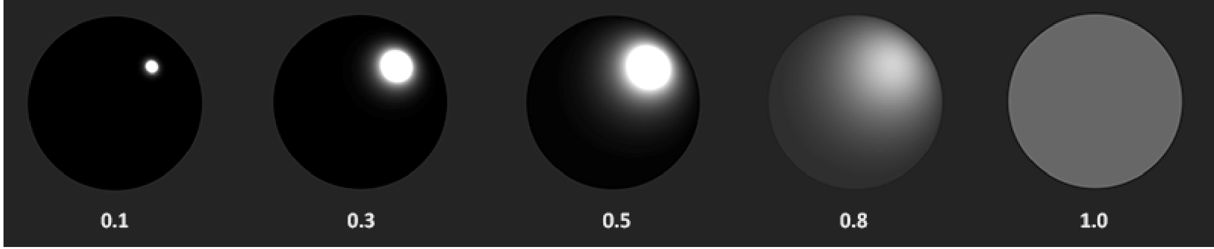

Figure 2.14: Roughness values and their result in reflection

The light that hits the surface of an object is partly reflected and partly refracted, depending on the properties of the surface. This separates the light drawn to diffuse light and reflection light. The reflection part is light that directly gets reflected and doesn't enter the surface; this is what we know as specular lighting. The refraction part is the remaining light that enters the surface and gets absorbed; this is what we know as diffuse lighting.

By observing physics law about energy conservation along with this distinction of light in two parts leads us to the observation that these two types are mutually exclusive. Whatever light energy gets reflected will no longer be absorbed by the material itself. In our implementation reflected light is computed first and diffuse light value is computed as the supplement to 1.0.

The reflectance equation is a complex equation that is used in PBR to simulate the visuals of light. It is based in theory analyzed in radiometry, the measurement of electromagnetic radiation. The equation is found below:

$$L_o(p, \omega_o) = \int_{\Omega} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i \tag{2.2}$$

The value L represents the radiance, which is the strength of light coming from a certain angle. A lot of values contribute in the computation of L such as Radiant Flux , which is the light energy represented in Watts, as well as the solid angle ω, the radiant intensity and many others. The f_r represents the $BRDF$ function which is properly analyzed below.

The $BRDF$, or Bidirectional Reflective Distribution Function takes as input data regarding the light inbound to the surface, light outbound, the microfacet roughness value a and approximates the light value of the facet of the material. Basically, combine all the above theories for microfacets and energy conservation in light to contribute to the result. Many computers shading algorithms are considered $BRDF$ functions, for example $Blinn - Phong$ takes similar inputs as $BRDF$ but it is not physically based

because they ignore the energy conservation principle. Almost all real-time render pipelines use a $BRDF$ function called $Cook - Torrance\ BRDF$.

PBR is physically based, as it has two parts for light, one diffuse and one specular that do not exceed the maximum energy of light when added together. The diffuse part is typically taking surface color into account while reflection part combines the Normal Distribution Function, Geometry function and Fresnel equation.

NDF, or Normal Distribution function, describes microfacets alignment. Geometry function describes the self-shadowing property of the microfacets. When a surface is relatively rough the surface's micro-facets can overshadow other microfacets thereby reducing the light surface reflects. And last Fresnel equation describes the ratio of surface reflection at different surface angles. There are many different implementations of these theories, other more realistic, other more performance driven. It is wise to pick one that cover all our needs.

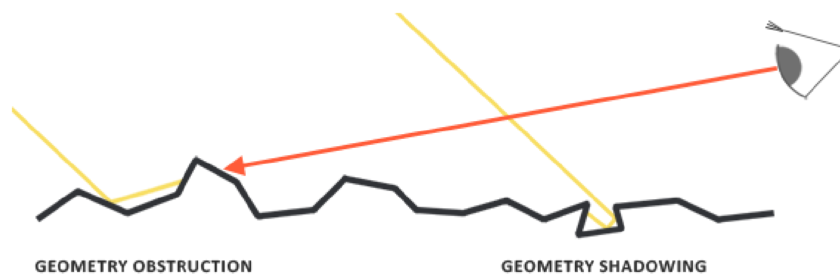

Figure 2.15: How geometry affects shadowing

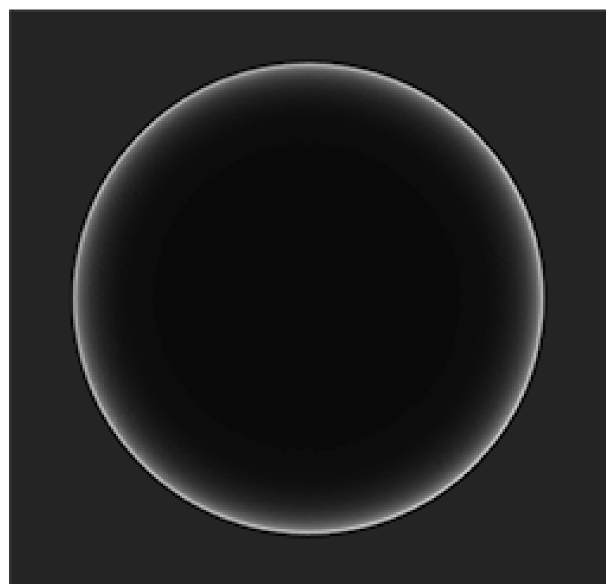

Figure 2.16: Fresnel effect

Chapter 3

Our Method

3.1 Physical Based Rendering
3.2 Order Independent Transparency

With all the data of the model there is a need for an appropriate viewer that can visualize all the details. The 3D Viewer needs to be able to render the details provided by the sensors while being fast enough. For that reason, we developed two rendering techniques. PBR (physically based rendering) and $S-buffer$. PBR is capable of visualizing all the surface details provided by the texture maps of the model, while S-buffer renders transparency and gives the user ability to view inner layers of a model.

3.1 Physical Based Rendering

Our PBR method is based in the Unreal Engine 4 functions [7] which are the $Trowbridge - Reitz\ GGX$ for the computation of normal Distribution function, the $Fresnel - Schlick$ approximation for the Fresnel equation and the $Smith's\, Schlick - GGX$ for the Geometry function. These are used in the $Cook - Torrance\ function$ to compute reflectance. We include point lights to light our scene and make the rendering more realistic. The Algorithm 3.1 presented below is describes the method we followed for PBR computation.

For each fragment in the scene this algorithm is followed to calculate it's final color in RGB value. First (line:1) we load the values from Texture Maps that correspond to the fragment's coordinates that include $albedo$ color, $metallic$, $roughness$ and AO value, as well as $normal\, vector$ of that fragment. We have to precompute 3 a value F_0 for the $Fresnel\, equation$ which is tinted from the $albedo$ color value if the material is metallic. Generally an object is metallic or dielectric but by passing a linear value from 0.0 to 1.0 in $metallic$ we can simulate variations in the surface, either aging based or dirt based, transfusing realism to our render. That value is later on (line: 13) passed in the equation along with the viewing angle of the camera, simulating the $fresnel$ effect (Fig. 2.16).

For each light in the scene we have to compute it's outgoing reflectance value separately and add it to the overall outgoing radiance (line: 19). We calculate the incident angle of light through it's position in scene (line: 6) and the halfway vector H needed for computations. The distance of light source is used to generate the attenuation of it (line: 9), giving decreasing intensity to the light as it gets further from the fragment.

The specular term (line: 16) of the $Cook - Torrance\ BRDF$ is calculated in two parts, the nominator of the fractal and the denominatior. Nominator comprises of the Normal Distribution Function (line: 11), the Geometry Function (line: 12) and the Fresnel Approximation (line: 13) while denominator is the product of two cross-products between the $normal\, vector$ and the $incident\, angle$ and $Halfway\, vector$ (lines: 14 15).

The $Fresnel$ value corresponds to the k_S value of the material so we can use it to denote any light that reaches the material's surface (line: 17). Since we obey the energy conservation rule we can directly calculate the $refraction$ value k_D of the material as the supplement of k_S to 1.0, however if a material is metallic we need to nullify the k_D value because a metallic surface do not refract any light, thus has no diffuse reflections (line: 18). Then we have the appropriate value to contribute to the reflectance value of the light (line: 19). The overall outgoing radiance of a fragment can then be calculated using the finished calculations (line: 22) and be forwarded to the next step in the rendering pipeline.

Algorithm 3.1 PBR computation Algorithm for a single fragment

1: Load $albedo, metallic, roughness, N$ from Texture Maps for fragment's coordinates

2: $V \leftarrow normalized\,vector(camera.position - WorldPosition)$
3: $F_0 \leftarrow mix(0.04, albedo, metallic)$
4: $L_o \leftarrow 0.0$
5: **for** each light i in scene **do**
6: $L \leftarrow normalized\,vector(i.position - WorldPosition)$
7: $H \leftarrow normalized\,vector(V + L)$
8: $distance \leftarrow$ length of L
9: $attenuation \leftarrow \frac{1.0}{distance^2}$
10: $radiance \leftarrow i.color \times attenuation \times 1.5$
11: $NDF \leftarrow$amount of microfacets aligned per H vector
12: $G \leftarrow$ value Geometry Shadowing of microfacets
13: $F \leftarrow$ Fresnel equation result
14: $den_0 \leftarrow max((N \cdot V), 0.0)$
15: $den_i \leftarrow max((N \cdot L), 0.0)$
16: $specular \leftarrow \frac{NDF \times G \times F}{4 \times den_0 \times den_i}$
17: $k_S \leftarrow F$
18: $k_D \leftarrow (1.0 - k_S) \times (1.0 - metallic)$
19: $L_o \leftarrow L0 + (\frac{k_D \times albedo}{\pi + specular}) \times radiance \times (N \cdot L)$
20: **end for**
21: $ambient \leftarrow 0.03 \times albedo$
22: $Fragment\,Color \leftarrow ambient + L_o$

3.2 Order Independent Transparency

That is a general pipeline and up until now we haven't talk about our algorithms that achieved transparency and photorealism. The methodology for achieving Order independent transparency needs multi-pass rendering, because of the preprocessing needed on the data. Our method features three rendering passes before proceeding to the rasterization on the screen.

The basic idea in the Order Independent Transparency is to give shaders the ability

to sort the fragments per Z-buffer depth and manipulate each fragment's contribution to the final color of the pixel accordingly. In the common $AlphaBlending$ transparency technique for every fragment behind a pixel, the contribution to the overall color of the pixel is proportionate to the $Alpha$ value that fragment (Equation: 2.1). That technique requires either an Alpha value lower than 1.0, which is impractical because the artifacts we render are not transparent objects, or fixed Alpha values for all the fragments which does not give results that are acceptable because the characteristics of inner layers are not distinct.

We use two buffers to store the fragments as they are loaded. The first one is where fragments are stored and the second one for indexing purposes. In each pixel we can only render finite number of fragments, meaning that the size of the buffer is static, which we found out from our tests that 32 is an appropriate size for rendering multi-layered objects. Because the models we render are generated from procedures that distinct different layers we exploited that extra information for better rendering results. We introduced a third buffer to our method with the purpose of storing the index of the layer in which the fragment belongs to, and used that information in the rendering to screen pass.

In Computer Graphics the rasterization to screen results in a frame and when we want interactive programs or moving videos muliple frames are rendered per second. As a result our algorithm is called once per frame and recalculates the result discarding all previous knowledge. Our buffers are reloaded as a result many times per second and we need to be sure that there will be no leftovers from previous renders. The first pass, taken place in the $Clear\ Buffer$ Shader (Algorithm 3.2) takes the job of initializing and resetting those three buffers (lines: 1 2 3) before loading data.

Algorithm 3.2 $Initialize\ Buffer$ Algorithm.

1: reset $fragment\ store\ buffer$ for depth sorting
2: reset $fragment\ counter$ for storing fragment ids
3: reset $fragment\ layer\ buffer$ for storing layer indexes

The second pass of the method is done through the $Render\ Buffer$ Shader (Algorithm 3.3) and it is responsible for loading all fragments in the three buffers while also computing the PBR result (line: 3). Each fragment has coordinates which we use to store them in the buffers. A z coordinate is also included which gives us the

depth information of the fragment, by which we sort the fragments in later rendering pass. Along with the RGB value of the fragment (lines: 4 6) we store that z value in the fragment's alpha (line: 8) to pass it to the next rendering pass. The original alpha value of the fragment is not valuable because we give alpha values to the final render in proportion to the layers that we want to be transparent.

Because of the massive parallel ability of modern GPUs the loading times are so small that there is the immediate danger of two fragments compete for the same spot in the buffers. We used atomic operations on both incrementing values (line: 1) and storing procedures (lines: 9 10) on the buffers. That technique is introducing delay to the execution of the algorithm, however it is imperative to achieve correct results.

If the fragment belongs to the layer that user has indicated as highlighted then the full PBR computation is done giving photorealistic properties to the fragment (line: 3), while all other layers are only colored by the $albedotexture$ (line: 6) property of PBR (line: 1). This variation is proposed for improving the clarity of results while also improving performance because we selectively call the PBR computation method.

Algorithm 3.3 Render Buffer Algorithm.

1: Atomic Increment $fragment\ counter$
2: **if** fragment's layer is $Highlighted$ layer **then**
3: Calculate PBR result 3.1
4: $frag.RGB \leftarrow PBR.RGB$
5: **else**
6: $frag.RGB \leftarrow albedo.RGB$
7: **end if**
8: $frag.alpha \leftarrow$fragment $coordinates.z$
9: store $frag$ in $fragment\ store\ buffer$
10: store layer index in $fragment\ layer\ buffer$

The fragments are stored in the buffers without order in a memory that is shared between shaders. The third pass, namely $Display\ Buffer$ Shader (Algorithm: 3.4), is responsible for visualizing the fragments on the screen. First we take the fragment coordinates and load the corresponding array of fragments from the buffers, creating a local instance of them. That way the sorting can be done without affecting the original data in the buffers and eliminating the danger of memory incoherence.

Due to the high parallel abilities of GPU the sorting procedure has no great influence in the performance of the tool. Therefore many different sorting algorithms can be used. We used the common $bubble - sort$ algorithm and also a bitonic sort for testing purposes. The fragments are sorted in outer to inner order and after that they are ready for rendering.

The user has also provided of a preference regarding which layer to be highlighted in the final result. In the shader the user preference is translated as the index in the sorted array, for example if the user wanted only to see the outer layer his preference would be 0 and the shader would give priority to the fragment in 0 position of vertex. The computation of the color value of the highlighted fragment is done according to this preference using the below equation:

$$col.rgb = \sum_{i=1}^{k} col(f_i).\alpha * col(f_i).rgb), \; where \sum_{i=1}^{k} col(f_i).\alpha = 1$$

and

$$col(f_i).\alpha = \begin{cases} v_\alpha, & \text{if } f_i \text{ is highlighted} \\ \frac{1}{k}(1 - v_\alpha), & \text{if } f_i \text{ is not highlighted} \end{cases}$$

Each fragment has RGB color value along with Alpha value. The algorithm we follow in order to highlight a layer is as follows: the highlighted fragment color is multiplied by $v_\alpha = 0.5$ alpha value(lines: 8 12), which technically gives half the color in the final result. The other $1 - v_\alpha = 0.5$ is divided by the k total number of fragments 10 and then multiplied by the color value of them(line: 12). The result is then a layer highlighted and all the others present by with small contribution. Other variations of this algorithm are possible, for example showing only certain fragments around the highlighted, with a result of more clarity.

To achieve results with more clarity we chose not to render back-face triangles and we also ignore any fragments that are in inner position regarding the highlighted layer. That way we can see the highlighted layer and transparent outer layers. The user preference regarding the highlighted layer is also inputted in the $Render\,Buffer$ Shader (Algorithm 3.3), where we only compute the PBR method (Algorithm 3.1) for the highlighted layer, giving the other layers only *albedo* color, for better visualization. Another variation we implemented is ignoring two consequent fragments that belong to the same layer (line: 4) (Algorithm 3.5), resulting in transparency only where there are multiple layers in an object.

Algorithm 3.4 Display Buffer Algorithm

1: get fragment array corresponding to fragcoords from $fragment_store_buffer$
2: create local instance of $fragment\ store\ buffer$
3: use $frag.alpha$ to sort in ascending order
4: Filter fragments in buffer
5: $finalColor \leftarrow 0$
6: **for** all fragments $frag_i$ in buffer from nearest to furthest **do**
7:     **if** i is highlighted by user **then**
8:         $frag_i.alpha \leftarrow 0.5$
9:     **else**
10:         $frag_i.alpha \leftarrow \frac{0.5}{NumFrags}$
11:     **end if**
12:     $finalColor.rgb \leftarrow finalColor.rgb + frag_i.rgb \times frag_i.alpha$
13: **end for**

Algorithm 3.5 Filter Fragments Algorithm

1: **for** all fragments **do**
2:     **if** $layer_fragment_i <= layer_fragment_{i-1}$ **then**
3:         remove fragments after i
4:     **end if**
5: **end for**

Chapter 4

Implementation

4.1 Shader Loading

4.2 Shaders Implementation

4.3 Draw functions

4.4 Model Loading

4.5 Graphical User Interface

The implementation of the algorithms is made using the GLSL shading language and the tool, for loading executing them, with the Qt framework for C++ and the OpenGL 4.4 library. The C++/OpenGL/Qt part initializes the window, open files and gives the commands to visualize the shaders developed, and the GPU part with the shaders run the algorithms and produce the rendering result. It is important to clarify that the shader part of the tool is not bound with the C++ implementation and that it can easily be included in a different project even if it not developed using C++ or Qt whatsoever.

4.1 Shader Loading

The Shaders implementing our method are stored in text files with extension .frag for fragment and .vert for vertex shaders. We use functions (Listing 4.1) imple-

menting Qt wrappers for shader management to load them. Regularly the procedure needs to load the string data and compile the shaders using the OpenGL methods provided. However using Qt wrappers we only create one QOpenGLShaderProgram object (line: 1) in which we add the shaders from file and it undertakes the job of compiling and setting up the OpenGL directives(lines: 3 8). That way we can use that object whenever we want to refer one of the three shaders. That procedure is the same for the three shader pairs needed for our method.

```
QOpenGLShaderProgram *tmp = new QOpenGLShaderProgram;
// First we load and compile the vertex ...shader
bool result = tmp->addShaderFromSourceFile( QOpenGLShader::Vertex,
  vertexShaderPath );
if ( !result )
qWarning() << m_shader->log();

// ...now the fragment ...shader
result = tmp->addShaderFromSourceFile( QOpenGLShader::Fragment,
  fragmentShaderPath );
if ( !result )
qWarning() << m_shader->log();

// ...and finally we link them to resolve any references.
result = tmp->link();
if ( !result )
qWarning() << "Could not link shader program:" << tmp->log();

if (type == "Clear"){
m_shaderClear = tmp;
}
else if(type == "Disp"){
m_shaderDisp = tmp;
}
else if(type == "Rend"){
m_shader = tmp;
}
return result;

```

Listing 4.1: Load Shaders from files

4.2 Shaders Implementation

We use three buffers to store the fragments and sort them by their Z coordinate value. the first buffer called d_abufferIdx stores the number of fragments already seen in a X, Y coordinate of a pixel and has dimensions relevant to the Screen Width and Height values. That way we can access the right amount of fragments in the other two buffers when we need to. The second buffer, called d_abuffer, stores fragments which are comprised of the *RGB* value and the alpha value where we store the Z coordinate of it for sorting. The last buffer, d_abufferMeshIdx stores the layer index of the fragment stored in the same position in d_abuffer. All three are statically defined by the C++/OpenGL part of the tool according to the screen Width, Height and the max depth of the buffer, which we set to 32.

Our method as we discussed in Chapter 3 is comprised of two separate procedures combined into one and implemented into three shaders. The multi-fragment method has three steps, clearing buffers, loading fragments into them, and displaying fragments after sorting them along the Z value. The first and third shaders have the same vertex shader called passThrough.vert whose job is only to pass the fragment's position to the fragment shader (Listing: 4.2).

```
in vec4 aPos;

smooth out vec4 fragPos;

void main(){
fragPos=aPos;
gl_Position = aPos;

```

Listing 4.2: passThrough.vert

The fragment shader of the first pass is called clearBuffer.frag (Listing: 4.3) and clears the three buffers needed for the method (lines: 6 7 8) in order to ensure that there are no leftovers in memory from previous renders. It discards all the fragments because there is no need for visualization in this pass.

```

ivec2 coords=ivec2(gl_FragCoord.xy);

//Be sure we are into the framebuffer
```

```
5  if(coords.x>=0 && coords.y>=0 && coords.x<SCREEN_WIDTH && coords.y<
    SCREEN_HEIGHT ){
6  d_abufferIdx[coords.x+coords.y*SCREEN_WIDTH]=0;
7  d_abufferMeshIdx[coords.x+coords.y*SCREEN_WIDTH]=0;
8  d_abuffer[coords.x+coords.y*SCREEN_WIDTH]=vec4(0.0f);
9  }
10
11 //Discard fragment so nothing is writen to the framebuffer
12 discard;
13
```

Listing 4.3: clearBuffer.frag

The second pass of the algorithm is the one that loads the fragments of the scene in buffers needed for the $A - buffer$ and also computes the PBR result for the layer which the user has characterized as highlighted. The vertex shader of this pass, namely pbrRenderBuffer.vert (Listing: 4.4) takes as uniform variables (lines: 14 .. 16) the input form the user regarding the camera position and view as well as the position of the model in world. The in variables (lines: 1 .. 4) are loaded in the loading process of the model in the program and explained below in Section 4.4. The out Variables (lines: 6 .. 12) are the variables computed in vertex shader and pass in the fragment shader part. Those computations are mostly combination of in variables with vertex attributes in order to be translated in fragment attributes.

```
1  in vec3 aPos;
2  in vec3 aNormal;
3  in vec2 aTexCoords;
4  in int meshIndx;
5
6  out vec2 TexCoords;
7  out vec3 WorldPos;
8  out vec3 Normal;
9  flat out int meshIndex;
10 smooth out vec4 fragPos;
11 smooth out vec3 fragTexCoord;
12 smooth out vec3 fragNormal;
13
14 uniform mat4 projection;
15 uniform mat4 view;
16 uniform mat4 model;
17 uniform mat4 modelIT;
```

```
18  void main()
19  {
20    TexCoords = aTexCoords;
21    WorldPos = vec3(model * vec4(aPos, 1.0));
22    Normal = mat3(model) * aNormal;
23    fragNormal = normalize((vec4(Normal, 1.0f)*modelIT).xyz);
24    fragTexCoord.xy=aPos.xy;
25    fragTexCoord.z=abs(fragNormal.z);
26    meshIndex = meshIndx;
27
28    fragPos = projection * view * vec4(WorldPos, 1.0);
29    gl_Position =  fragPos;
30  }
31
```

Listing 4.4: pbrRenderBuffer.vert

The fragment shader of this pass is called pbrRender.frag and is divided in the part that computes the PBR value of a fragment and the part that is responsible for loading the fragment properly in the buffers for the multi-fragment rendering. In Listing 4.5 we see the code for storing the fragments in shaders. As explained in 3.2 there is a need for atomicity in storing the fragments because of the massive parallel computation of modern GPUs. Each fragment is stored in the d_abuffer array according to it's X, Y coordinate and a variable abidx incremented atomically regarding the other fragments (lines: 5 16).

After computing the PBR value (line: 9) the fragment color is replace by it (line: 11) and in the $alpha$ value of it (line: 12), the Z coordinate of the fragment is stored, to be used in the sorting procedure. That final fragment value is stored in the d_abuffer (line: 16) and at the d_abufferMeshIdx (line: 17), the meshIndex, referring to the mesh the fragment belongs to, is stored in order to be distinct in later stages. As in the first pass all fragments are discarded because there is no need for visualization in this stage.

```
1  ivec2 coords=ivec2(gl_FragCoord.xy);
2  //Check we are in the framebuffer
3  if(coords.x>=0 && coords.y>=0 && coords.x<SCREEN_WIDTH && coords.y<
    SCREEN_HEIGHT){
4    int abidx;
5      abidx=(int)atomicIncWrap(d_abufferIdx+coords.x+coords.y*SCREEN_WIDTH,
     ABUFFER_SIZE);
```

```
6
7       vec4 abuffval;
8       vec3 col;
9       col = computePBR();
10
11      abuffval.rgb=col;
12      abuffval.w=fragPos.z; //Will be used for sorting
13
14      //Put fragment into A-Buffer
15      int temp = coords.x+coords.y*SCREEN_WIDTH + (abidx*SCREEN_WIDTH*
    SCREEN_HEIGHT);
16      d_abuffer[temp]=abuffval;
17      d_abufferMeshIdx[temp] = meshIndex;
18  }
19    discard;
20
```

Listing 4.5: pbrRenderBuffer.frag

For the $Cook-Torrance\ BRDF$ function in PBR computation we use the functions based on the Epic Game's Unreal Engine 4 (**REF HERE**) technical report and specifically $Trowbridge-ReitzGGX$ (Listing 4.6) function for computing the Normal Distribution Function, the $Fresnel-Schlick$ approximation (Listing 4.7) for Fresnel equation and the $Smith'sSchlick-GGX$ (Listing 4.8) method for the Geometry function.

```
1  float a = roughness*roughness;
2  float a2 = a*a;
3  float NdotH = max(dot(N, H), 0.0);
4  float NdotH2 = NdotH*NdotH;
5
6  float nom    = a2;
7  float denom = (NdotH2 * (a2 - 1.0) + 1.0);
8  denom = PI * denom * denom;
9
10 return nom / denom;
11
```

Listing 4.6: NDF Trowbridge-Reitz GGX

```
1  return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
```

```
2
```

Listing 4.7: Fresnel-Schlick approximation.frag

```
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;

```

Listing 4.8: Smith's Schlick-GGX

The main part of the computation (Listing 4.9) takes values from texture maps (lines: 1 .. 5) and then uses them to calculate the $BRDF$ function (line: 32), while taking into account the contribution of each of the four lights present in the scene and their color. The energy conservation of the method is implemented when computing as a supplement to 1 the k_D (line: 39) and and k_S (line: 35)variables, that indicate the portion of light refracted as diffuse color and the portion reflected (line: 49). The result is returned to main method of the shader (Listing 4.5, line: 9) to be saved to the buffer.

```
vec3 albedo   = pow(texture(texture_diffuse1, TexCoords).rgb, vec3(2.2));
float metallic  = texture(texture_normal1, TexCoords).r;
float roughness = texture(roughnessMap, TexCoords).r;
float ao        = texture(aoMap, TexCoords).r;
vec3 N = getNormalFromMap();

vec3 V = normalize(camPos - WorldPos);

// calculate reflectance at normal incidence; if dia-electric (like plastic) use F0
// of 0.04 and if it's a metal, use the albedo color as F0 (metallic workflow)
vec3 F0 = vec3(0.04);
F0 = mix(F0, albedo, metallic);

// reflectance equation
vec3 Lo = vec3(0.0);
for(int i = 0; i < 4; i++)
{
```

```
18        // calculate per-light radiance
19        vec3 L = normalize(lightPositions[i] - WorldPos);
20        vec3 H = normalize(V + L);
21        float distance = length(lightPositions[i] - WorldPos);
22        float attenuation = 1.0 / (distance * distance) * 1000.0;
23        vec3 radiance = lightColors[i] * attenuation ;
24
25        // Cook-Torrance BRDF
26        float NDF = DistributionGGX(N, H, roughness);
27        float G   = GeometrySmith(N, V, L, roughness);
28        vec3 F    = fresnelSchlick(max(dot(H, V), 0.0), F0);
29
30        vec3 nominator    = NDF * G * F;
31        float denominator = 4 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) +
         0.001; // 0.001 to prevent divide by zero.
32        vec3 specular = nominator / denominator;
33
34        // kS is equal to Fresnel
35        vec3 kS = F;
36        // for energy conservation, the diffuse and specular light can't
37        // be above 1.0 (unless the surface emits light); to preserve this
38        // relationship the diffuse component (kD) should equal 1.0 - kS.
39        vec3 kD = vec3(1.0) - kS;
40        // multiply kD by the inverse metalness such that only non-metals
41        // have diffuse lighting, or a linear blend if partly metal (pure
         metals
42        // have no diffuse light).
43        kD *= 1.0 - metallic ;
44
45        // scale light by NdotL
46        float NdotL = max(dot(N, L), 0.0);
47
48        // add to outgoing radiance Lo
49        Lo += (kD * albedo / PI + specular) * radiance * NdotL;  // note that
         we already multiplied the BRDF by the Fresnel (kS) so we won't multiply
         by kS again
50    }
51
52    // ambient lighting
53    vec3 ambient = vec3(0.03) * albedo * ao;
54    vec3 color = ambient + Lo;
```

```
55
56  // HDR tonemapping
57  color = color / (color + vec3(1.0));
58  // gamma correct
59  color = pow(color, vec3(1.0/2.2));
60
61  return color;
62
```

Listing 4.9: PBR computation method

The third pass renders the result to the screen of the user, taking as import the buffers where we loaded the fragments from the previous pass. The vertex shader is the same as the first pass (Listing 4.2) with responsibility to pass the coordinates to the fragment shader, which is called DisplayBuffer.frag (Listing 4.10).

From the d_abufferIdx buffer we get the number of fragments stored for the pixel position X, Y (line: 7). We use a method which takes that number and the coordinates of the pixel to store all the fragments in local arrays (Listing 4.11) to ensure that the data in the three buffers are kept and not meshed with. For sorting algorithm we use a common bubble sorting algorithm(Listing 4.12) but have also used a bitonic sorting algorithm(Listing 4.13) to use for our experiments on the performance of our method.

The visualization of the fragments using transparency is done i

```
1   ivec2 coords=ivec2(gl_FragCoord.xy);
2   int abNumFrag;
3   if(coords.x>=0 && coords.y>=0 && coords.x<SCREEN_WIDTH && coords.y<
      SCREEN_HEIGHT ){
4
5   //Load the number of fragments in the current pixel.
6
7   abNumFrag=(int)d_abufferIdx[coords.x+coords.y*SCREEN_WIDTH];
8
9   if(abNumFrag<0 )
10    abNumFrag=0;
11  if(abNumFrag>ABUFFER_SIZE ){
12    abNumFrag=ABUFFER_SIZE;
13  }
14  if(abNumFrag > 0){
15    //Copy fragments in local array
```

```
    fillFragmentArray(coords, abNumFrag);
    //Sort fragments in local memory array
    bubbleSort(abNumFrag);
    outFragColor = resolveKBlend(SHOW_INDEX, 4, abNumFrag);
  }
  else{
    discard;
  }

```

Listing 4.10: DisplayBuffer.frag

```
  for(int i=0; i<abNumFrag; i++){
    if (ABUFFER_USE_TEXTURES == 1){
      fragmentList[i]=imageLoad(abufferImg, ivec3(coords, i));
    }
    else{
      fragmentList[i]=d_abuffer[coords.x+coords.y*SCREEN_WIDTH + (i*
 SCREEN_WIDTH*SCREEN_HEIGHT)];
      meshIndexList[i]=d_abufferMeshIdx[coords.x+coords.y*SCREEN_WIDTH + (i
 *SCREEN_WIDTH*SCREEN_HEIGHT)];
    }
  }

```

Listing 4.11: fill fragments in local arrays

```
  for (int i = (array_size - 2); i >= 0; --i) {
    for (int j = 0; j <= i; ++j) {
      if (fragmentList[j].w > fragmentList[j+1].w) {
        vec4 temp = fragmentList[j+1];
        fragmentList[j+1] = fragmentList[j];
        fragmentList[j] = temp;

        float temp2= meshIndexList[j+1];
                meshIndexList[j+1] = meshIndexList[j];
              meshIndexList[j] = temp2;
      }
    }
  }

```

Listing 4.12: Bubble Sort

```
void bitonicSort( int n ) {
  int i,j,k;
  for (k=2;k<=n;k=2*k) {
    for (j=k>>1;j>0;j=j>>1) {
      for (i=0;i<n;i++) {
        int ixj=i^j;
        if ((ixj)>i) {
          if ((i&k)==0 && fragmentList[i].w>fragmentList[ixj].w){
            swapFragArray(i, ixj);
          }
          if ((i&k)!=0 && fragmentList[i].w<fragmentList[ixj].w) {
            swapFragArray(i, ixj);
          }
        }
      }
    }
  }
}

void swapFragArray(int n0, int n1){
  vec4 temp = fragmentList[n1];
  fragmentList[n1] = fragmentList[n0];
  fragmentList[n0] = temp;
}

```

Listing 4.13: bitonic Sort

The visualization of fragments using transparency is done in a different method (line: 19)(Listing 4.14) using the highlighted layer preference, passed as a uniform to the shader, after we process them first.

The cleaning of the fragments loaded in the local shaders is imperative for better visualizing the layers of an object (see REF HERE TO METHOD). In our implementation we use layer indexes for identifying the separate layers but we assume that these indexes are given in an order from the outer to the inner layer, for example layer 0 is the outmost layer and layer N is the inner one. We need that classification to achieve that cleaning on the array of fragments as follows. We keep all the fragments until we find one from a layer that should be in an earlier position or is repeated for a second time(lines: 4). That way we only see transparency in the front part of an object and not something that we do not need to see.

The resulted cleaned array is for rendering and start by taking each fragment in it one by one. When we find the highlighted layer we pass an alpha value of 0.5 to the fragment belonging to it (lines: 19) and we put a value of $\frac{0.5}{nuumber\ of\ fragments\ in\ pixel}$ in all the other layers (lines: 24) for clarity. The final RGB value is computed (lines: 26) according to that alpha value of each fragment. All the fragments after the highlighted one are not rendered in screen for visualization purposes (lines: 12).

```
vec4 finalColor=vec4(0.0,0.0,0.0,1.0);
finalColor=vec4(0.0f);

\* INSERT LAYER RESOLVING CODE HERE
.
.
.
*/
for(int i=0; i<abNumFrag; i++){

  if (i > showIndex-1 )
    continue;

  vec4 frag=fragmentList[i];
  vec4 col;

  if (i == showIndex-1){
    col.rgb = frag.rgb;
    col.w = 0.5f;
  }
  else{

    col.rgb=frag.rgb;
      col.w = 0.5/float(abNumFrag);
  }
  finalColor.rgb = finalColor.rgb + col.rgb*col.a;
}

finalColor.a = 1.0f;

return finalColor;

```

Listing 4.14: Render Transparency

4.3 Draw functions

The main C++ part also has a paint function which is called once per frame and is responsible for setting the dynamic parameters in our shader program and call the appropriate shaders. In Listing 4.15 we give a part of paint method where we set some uniform variables in shaders and we call the display methods for drawing each one of the three.

Each of the four lights present in the scene has it's own position so we have to pass each one separately (line: 2). The positions can be changed from the GUI part of the tool. The camera and the projection of view is set accordingly(line: 9) and the same is done for the position of the model(line: 16).

Then the drawing functions of the three shaders is called in the right order after some implementation uniform variables are set for each one(lines: 20 22 24).

```
for (int i = 0; i< 4 ; i++){
m_shader->setUniformValue(("lightPositions[" + std::to_string(i) + "]").
  c_str(), lightPositions[i]);
}

m_shader->setUniformValue(projection, m_proj);
m_shader->setUniformValue(view, m_view * m_world);
m_shader->enableAttributeArray(0);
m_shader->enableAttributeArray(1);
m_shader->enableAttributeArray(2);

m_model.setToIdentity();
m_model.scale(scaleVal);
m_model.rotate(180.0f - (m_xRot / 16.0f), 1, 0, 0);
m_model.rotate(m_yRot / 16.0f, 0, 1, 0);
m_model.rotate(m_zRot / 16.0f, 0, 0, 1);
m_shader->setUniformValue(model, m_model);

DisplayRender = 1;

displayClearABuffer_Basic();
shaderBooleanUpdates();
displayRenderABuffer_Basic();
shaderBooleanUpdates();
displayResolveABuffer_Basic();
DisplayRender = 1;
```

```
update();

```

Listing 4.15: Paint method

Each of the functions has the same logic (Listing 4.16) of binding the appropriate variables needed for rendering and then calling a glDraw OpenGL function(Listing 4.17).

```
GLuint prog = m_shader->programId();
//Assign uniform parameters

functions2->glProgramUniformui64NV(prog, glGetUniformLocation(prog, "
  d_abuffer"), abufferGPUAddress);

functions2->glProgramUniformui64NV(prog, glGetUniformLocation(prog, "
  d_abufferIdx"), abufferCounterGPUAddress);

functions2->glProgramUniformui64NV(prog, glGetUniformLocation(prog, "
  d_abufferMeshIdx"), abufferMeshIndex);



//Pass matrices to the shader

functions->glProgramUniformMatrix4fv(prog, glGetUniformLocation(prog, "
  projection"), 1, GL_FALSE, m_proj.constData());

functions->glProgramUniformMatrix4fv(prog, glGetUniformLocation(prog, "
  model"), 1, GL_FALSE, m_model.constData());

QMatrix4x4  modelViewMatrixIT=m_model.inverted().transposed();

functions->glProgramUniformMatrix4fv(prog, glGetUniformLocation(prog, "
  modelIT"), 1, GL_FALSE, modelViewMatrixIT.constData());

//Render the model

drawModel(prog);

```

Listing 4.16: Draw shader function

```
glUseProgram (prog);
```

```
2   ourModel.Draw(m_shader);
3
```

Listing 4.17: Draw model function

4.4 Model Loading

We used the ASSIMP library to open 3D model files, which makes our program compatible with every common 3D model format available in the industry. It is quite possible that a model consists of many parts, either for organization purposes or for best memory management. Each part is a separate mesh with its own list of vertices, faces, normals, Textures, etc and for that reason we need an procedure that can load multi-meshed models and keep track of all their properties. So, we developed two C++ classes to load each model to our scene, the model.h and the mesh.h needed for importing.

4.4.1 Class model

The model.h class uses the ASSIMP structs to import a model to the scene. We use two vectors for each model, one that keeps a list of the meshes loaded in that model and one for the textures. The constructor of the model.h class call the loadModel method which creates an ASSIMP::aiScene object in which it imports all the data of the file given (line: 3). After error checking it starts the processnode method with the first node at scene->mrootNode position.

```
1   Assimp::Importer *importer = new Assimp::Importer;
2
3   const aiScene* scene = new const aiScene;
4
5   scene = importer->ReadFile(path.c_str(), aiProcess_Triangulate |
     aiProcess_FlipUVs | aiProcess_CalcTangentSpace | aiProcess_GenNormals);
6
7   // check for errors
8   if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->
     mRootNode) // if is Not Zero
9   {
10  cout << "ERROR::ASSIMP:: " << importer->GetErrorString() << endl;
```

```
return;
}

// retrieve the directory path of the filepath
directory = path.substr(0, path.find_last_of('/'));

// process ASSIMP's root node recursively
processNode(scene->mRootNode, scene, m_vertexBuffer);

```

Listing 4.18: Load File with ASSIMP

The structure of the ASSIMP::aiScene object follows a tree structure scheme with multiple children nodes and meshes. That way if a node has only mesh children it is considered an object and when a node has only node children it refers to a family of objects. That structure is not useful in our pipeline because we only import objects that visualize cultural heritage artifacts, however we have included that utility to our code in order to be scheme independent and be used in different cases.

The processNode method recursively processes nodes (line: 12) and when there are meshes it creates an ASSIMP::aiMesh object (line: 6) which calls the processMesh method for each mesh in the node, while pushing the result in the list of meshes (line: 7). This way we keep a list with all the meshes in the scene which we can address later on, eg. while rendering.

```
// process each mesh located at the current node
for (unsigned int i = 0; i < node->mNumMeshes; i++)
{
// the node object only contains indices to index the actual objects in
  the scene.
// the scene contains all the data, node is just to keep stuff organized
  (like relations between nodes).
aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
meshes.push_back(processMesh(mesh, scene, m_vertexBuffer, i));
}
// after we've processed all of the meshes (if any) we then recursively
  process each of the children nodes
for (unsigned int i = 0; i < node->mNumChildren; i++)
{
processNode(node->mChildren[i], scene, m_vertexBuffer);
}
```

```
14
```

Listing 4.19: processNode method

For each mesh we keep three vectors for vertices, indices and textures. A vertex is a struct containing the vertex Position, Normal, texture Coordinates, Tangent, Bitangent and the index of the mesh it belongs. When the processMesh method is called we first do a pass of all the vertices comprising the mesh and we find the minimum and maximum values for each of teh X, Y, Z coordinates. This way we can compute the Bounding Box (Listing: 4.20) of the mesh and therefore move the mesh in the center of the scene.

```
float bboxMinX=1000000.0f;float bboxMinY=1000000.0f;float bboxMinZ
  =1000000.0f;
float bboxMaxX=-1000000.0f;float bboxMaxY=-1000000.0f;float bboxMaxZ
  =-1000000.0f;
for(uint v=0; v < mesh->mNumVertices; ++v){
   float valX=mesh->mVertices[v].x;
   float valY=mesh->mVertices[v].y;
   float valZ=mesh->mVertices[v].z;
   if(valX<bboxMinX)
      bboxMinX=valX;
   if(valX>bboxMaxX)
      bboxMaxX=valX;
   if(valY<bboxMinY)
      bboxMinY=valY;
   if(valY>bboxMaxY)
      bboxMaxY=valY;
   if(valZ<bboxMinZ)
      bboxMinZ=valZ;
   if(valZ>bboxMaxZ)
      bboxMaxZ=valZ;
}

float sizeX=(bboxMaxX-bboxMinX);
float sizeY=(bboxMaxY-bboxMinY);
float sizeZ=(bboxMaxZ-bboxMinZ);
float maxSize=max(sizeX, max(sizeY, sizeZ));

```

Listing 4.20: Find Bounding Box

We load each vertex and each properties from the ASSIMP::aiMesh in the vertices vector using the Vertex struct fields (Listing: 4.21).

```
for (unsigned int i = 0; i < mesh->mNumVertices; i++)
{
  Vertex vertex;
  glm::vec3 vector; // we declare a placeholder vector since assimp uses its own vector class that doesn't directly convert to glm's vec3 class so we transfer the data to this placeholder glm::vec3 first.
  // positions

  float tmpx = (mesh->mVertices[i].x-bboxMinX)/(maxSize);
  float tmpy = (mesh->mVertices[i].y-bboxMinY)/(maxSize);
  float tmpz = (mesh->mVertices[i].z-bboxMinZ)/(maxSize);
  vector.x = mesh->mVertices[i].x;
  vector.y = mesh->mVertices[i].y;
  vector.z = mesh->mVertices[i].z;
  vertex.Position = vector;
  // normals
  vector.x = mesh->mNormals[i].x;
  vector.y = mesh->mNormals[i].y;
  vector.z = mesh->mNormals[i].z;
  vertex.Normal = vector;
  vertex.meshIndx = MeshIndex;
  // texture coordinates
  if (mesh->mTextureCoords[0]) // does the mesh contain texture coordinates?
  {
    glm::vec2 vec;
    // a vertex can contain up to 8 different texture coordinates. We thus make the assumption that we won't
    // use models where a vertex can have multiple texture coordinates so we always take the first set (0).
    vec.x = mesh->mTextureCoords[0][i].x;
    vec.y = mesh->mTextureCoords[0][i].y;
    vertex.TexCoords = vec;
  }
  else
    vertex.TexCoords = glm::vec2(0.0f, 0.0f);
  // tangent
  vector.x = mesh->mTangents[i].x;
```

```
34      vector.y = mesh->mTangents[i].y;
35      vector.z = mesh->mTangents[i].z;
36      vertex.Tangent = vector;
37      // bitangent
38      vector.x = mesh->mBitangents[i].x;
39      vector.y = mesh->mBitangents[i].y;
40      vector.z = mesh->mBitangents[i].z;
41      vertex.Bitangent = vector;
42      vertices.push_back(vertex);
43    }
44
```

Listing 4.21: Load vertex data

We then proceed to load the indices of the mesh to indices vector (Listing: 4.22) with the same logic.

```
1   // now wak through each of the mesh's faces (a face is a mesh its
      triangle) and retrieve the corresponding vertex indices.
2   for (unsigned int i = 0; i < mesh->mNumFaces; i++)
3   {
4     aiFace face = mesh->mFaces[i];
5     // retrieve all indices of the face and store them in the indices
      vector
6     for (unsigned int j = 0; j < face.mNumIndices; j++)
7       indices.push_back(face.mIndices[j]);
8   }
9
```

Listing 4.22: Load indices data

The last thing that comprises a mesh is the material with the Textures. The ASSIMP::aiScene keeps a struct of materials in relation to the meshes they refer to and which we take and load each Texture separately by its type (Listing: 4.23). It is easier to load the textures right now in the GPU memory and we use newTextureFromFile method (Listing: 4.24) to do that.

```
1   aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
2
3   // 1. diffuse maps
4   vector<Texture> diffuseMaps = loadMaterialTextures(material,
      aiTextureType_DIFFUSE, "texture_diffuse");
5   textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
```

```
6   // 2. specular maps
7   vector<Texture> specularMaps = loadMaterialTextures(material,
     aiTextureType_SPECULAR, "texture_specular");
8   textures.insert(textures.end(), specularMaps.begin(), specularMaps.end())
     ;
9   // 3. normal maps
10  std::vector<Texture> normalMaps = loadMaterialTextures(material,
     aiTextureType_HEIGHT, "texture_normal");
11  textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());
12  // 4. height maps
13  std::vector<Texture> heightMaps = loadMaterialTextures(material,
     aiTextureType_AMBIENT, "texture_height");
14  textures.insert(textures.end(), heightMaps.begin(), heightMaps.end());
15
```

Listing 4.23: read Material Data

In this method we have to load the data of the Texture using the stbimage library for image manipulation. When we load texture data to a shader use the glGenTextures method (line: 2) of OpenGL to create a Texture Buffer which we then bind (line: 17) when we want to pass data (line: 18) to it along with some parameters (line: 24). While reading the Texture data from file (line: 5) we also get some information regarding the image encoding and size (line: 4) where the Texture is saved and we also pass it to the buffer (line: 18).

```
1   unsigned int textureID;
2   glGenTextures(1, &textureID);
3
4   int width, height, nrComponents;
5   unsigned char *data = stbi_load(filename.c_str(), &width, &height,
                                     &nrComponents, 0);
6   if (data)
7   {
8     GLenum format;
9     if (nrComponents == 1)
10    format = GL_RED;
11    else if (nrComponents == 3)
12    format = GL_RGB;
13    else if (nrComponents == 4)
14    format = GL_RGBA;
15
```

```
16      QOpenGLFunctions *f = QOpenGLContext::currentContext()->functions();
17      glBindTexture(GL_TEXTURE_2D, textureID);
18      glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0,
                     format, GL_UNSIGNED_BYTE, data);
19      f->glGenerateMipmap(GL_TEXTURE_2D);
20
21      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
22      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
23      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR);
24      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
25
26      stbi_image_free(data);
27    }
28    else
29    {
30      std::cout << "Texture failed to load at path: " << path << std::endl;
31      stbi_image_free(data);
32    }
33
34    return textureID;
35
```

Listing 4.24: newTextureFromFile method

Now all the information about the mesh are loaded in the Mesh vector and then we pass them in the mesh.h class constructor (Listing: 4.25) which is responsible for loading them in GPU memory, except for the texture information which are already loaded.

```
1   Mesh *m = new Mesh(vertices, indices, textures, m_vertexBuffer);
2
```

Listing 4.25: call mesh.h constructor after data parsed

4.4.2 Class mesh

the mesh.h class job is to pass the mesh data to the GPU memory appropriately without conflicts with other buffers and by binding different buffers for each mesh we ensure that the memory integrity is ensured. For each mesh we create three buffers, one vertex array object (line: 2), one array buffer (line: 3) where we store the

vertex data and one element buffer object (line: 4) where we store the indices data. The vertex array object is basically a descriptor of how the vertex attributes are stored in memory and also introduces an easier way of rendering in later stages.

After binding the vertex array object and the vertex buffer object we pass vertex data by calling the glBufferData method (line: 9).The same procedure is followed for the indices and the element buffer object (line: 12). Finally we need to update the shader with the size and the memory position where each vertex attributes can be found during rendering stage (line: 19).

```
// create buffers/arrays
f2->glGenVertexArrays(1, &VAO);
f->glGenBuffers(1, &VBO);
f->glGenBuffers(1, &EBO);

f2->glBindVertexArray(VAO);
// load data into vertex buffers
f->glBindBuffer(GL_ARRAY_BUFFER, VBO);
f->glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex),
                &vertices[0], GL_STATIC_DRAW);

f->glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
f->glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() *
        sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);


// set the vertex attribute pointers
// vertex Positions

f->glEnableVertexAttribArray(0);
f->glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                        (void*)0);
// vertex normals
f->glEnableVertexAttribArray(1);
f->glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                        (void*)offsetof(Vertex, Normal));
// vertex texture coords
f->glEnableVertexAttribArray(2);
f->glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                        (void*)offsetof(Vertex, TexCoords));
f->glEnableVertexAttribArray(3);
```

```
f->glVertexAttribPointer(5, 1, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (void*)offsetof(Vertex, meshIndx));
// vertex tangent
f->glEnableVertexAttribArray(4);
f->glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (void*)offsetof(Vertex, Tangent));
// vertex bitangent
f->glEnableVertexAttribArray(5);
f->glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (void*)offsetof(Vertex, Bitangent));

```

Listing 4.26: mesh.h constructor

DRAW FUNCTIONS

After we open the 3D model files we need to interpret the data of them accordingly, in order to be passed in the GPU memory for rendering. As we analyzed in the basics chapter, a 3D model is com-prised of mesh and textures.

A mesh is made of triangles that each has, of course three vertices. The data in files are generally given as vertices, properties of these vertices and also the indices of the vertices that take part in each triangle/face are given. We create four lists that we fill with mesh data, one for vertex position, one for vertex normal, one for the UV coordinates of that vertex and one for the face index-es. Then we use OpenGL functions to create the appropriate buffers in GPU memory to pass the data and we transfer them. A similar process is used for the textures too.

We have three fragment shaders, the *clearShader.frag* the *renderShader.frag* and the *displayShader.frag*. We also have two vertex shaders that are only used as medium for the data in the fragment shader because we do not do any vertex manipulation in our work.

The interface of the renderer is divided in three parts: Model selection and different aging times for said model. An object manipulation part that is used to move, rotate, scale the model, the camera or the lights in the scene.

And finally, a shader algorithm selection part that is used to render both PBR shader and S-buffer. User is given the ability to select the number of layers to render and the specified one to be high-lighted.

The User Interface successfully passes variables to the shader program which then are properly manipulated by the shader language to be rendered.

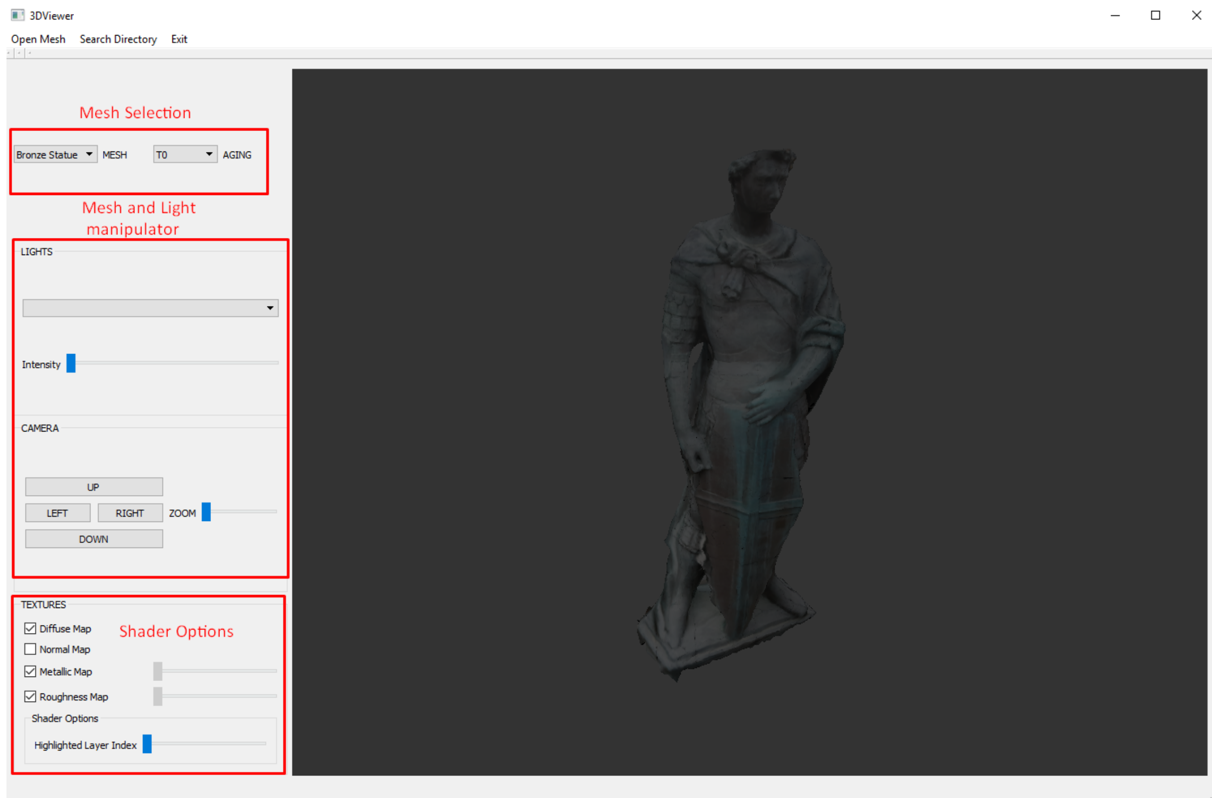

Figure 4.1: Main Panel of our tool and its components

4.5 Graphical User Interface

As seen in Fig. 4.1, our tool has two distinct components, the general options panel and the OpenGL window where the rendering result is shown. We have a mesh selector part that we can use to navigate through different models and time periods for the selected object which have previously imported in the tool. If we decide to ignore this button the time t_0 will be automatically chosen.

In the Mesh and Light Manipulation section are some basic functions such as rotation translation and scale, so that we can have visual contact with our object so we can very easily focus on a desired area of interest. Of course, we are given the opportunity to come as close or as far as we want (zoom) to our mesh, so that we can observe very small details that may concern us. The same manipulation can be done in the lights present to the scene. We have included four light that can be moved around the object and which also can vary in the intensity of their radiance.

The last part of our interface belongs to the shader options of our tool. The most important step of our approach is to define textures for our meshes. For each mesh we have a number of possible textures to add on. Each texture provides a different

optical result. In the process of importing the mesh in the tool, we look the textures included with the mesh and we enable the option to the user to include them in the render or not. In the background we have stored default values in case where there is no texture present and we also give the option to give uniform values in roughness and metallic values of the mesh.

Chapter 5

Experiments and Results

The results of our work can be measured regarding the performance of the method in different cases. In Computer Graphics the performance is usually calculated using frames per second (FPS), where each frame is a rasterization of the rendering pipeline in the screen or a picture. Real-time rendering value is considered when each frames takes less than 1/30th of a second to render, or when we get more than 30FPS.

Our method is affected by many parameters such as the triangle count of the mesh, the screen dimensions and the percentage of them filled with fragments. Because our method is fragment based the amount of fragments, eg. pixels, affects the performance more than the triangle count of a mesh. Below we can see (Fig. 5.1) the FPS count of three different objects in the same screen resolution (1024x1024) and on different screen fill percentages(50% 75% 100%).

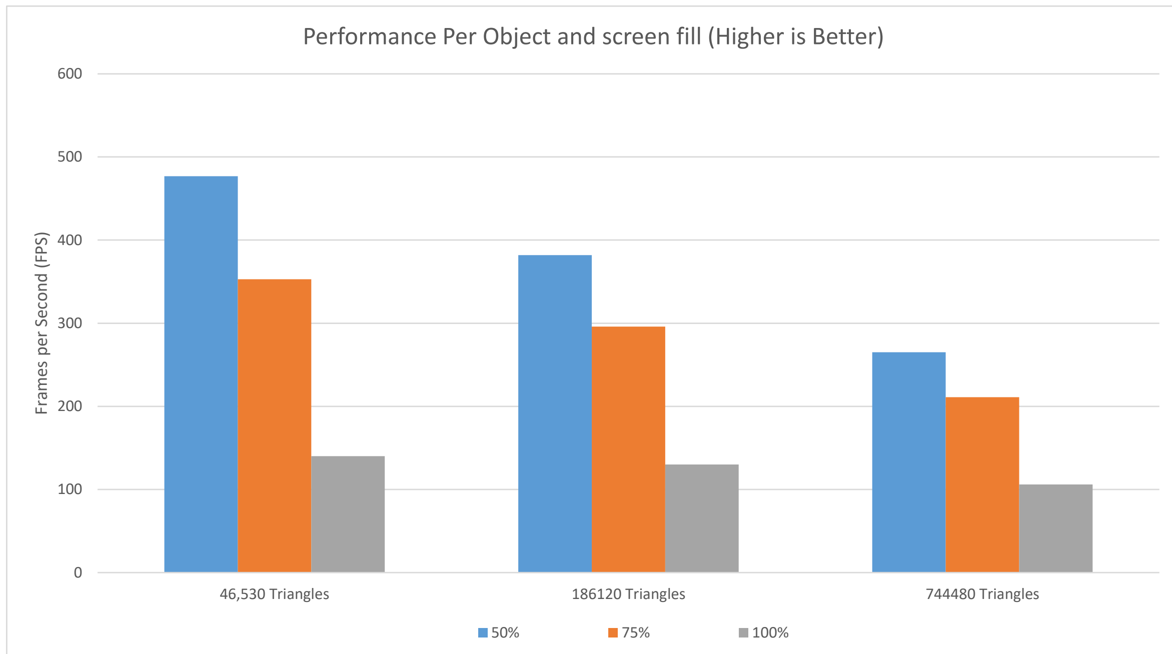

Figure 5.1: Tool performance per Object and screen fill

The objects we used are comprised of three layers and they are exactly the same but with different number of triangles. The low polycount one has 46.530 triangles, the mid polycount has 186.120 triangles and the High poly one has 744.480 triangles. We can see than the number of triangles, despite been largely varied does not have equivalent drop in FPS. Even if we multiply the number of layers there is not considerable difference (Fig. 5.2). What has a big influence is the screen percentage covered by the model. That can be explained because the method implements buffers for all the fragments, therefore when a larger percentage of the screen is filled with a mesh there are more buffers filled with fragments and there are way more computations to be done.

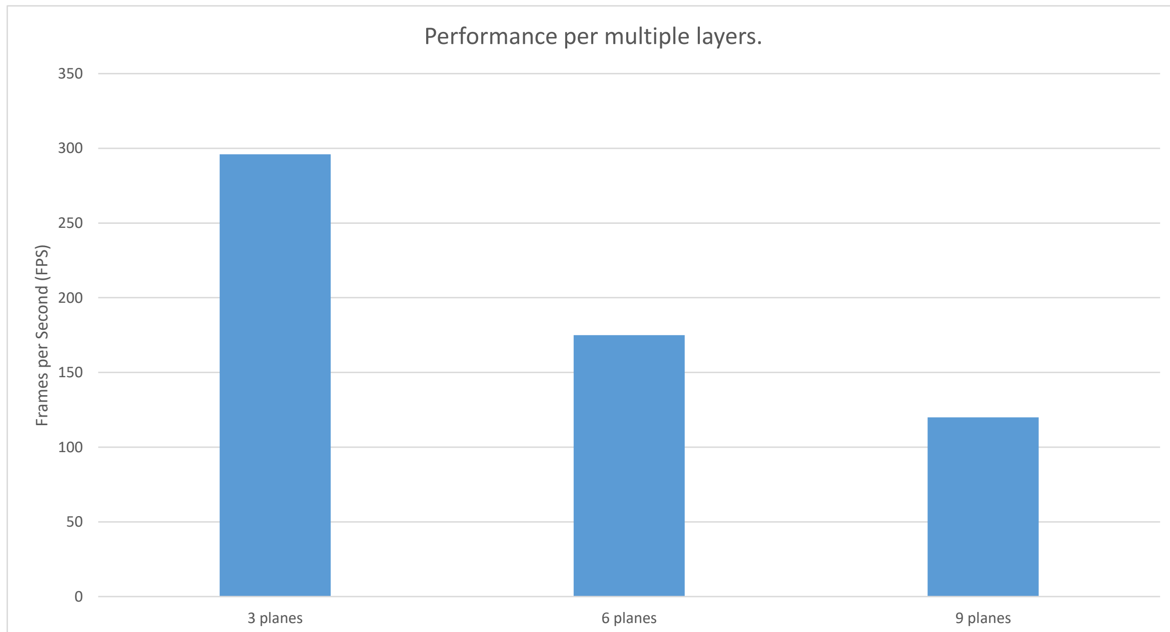

Figure 5.2: Multiple Layer performance

We need to evaluate the performance of each method separately to understand the influence it has in the overall performance of the tool. It is easily understood that the Multi-fragment rendering algorithm has a great overhead because of the buffers used and below we can see that the influence of PBR is almost negligible comparing to multi-fragment (Fig. 5.3).

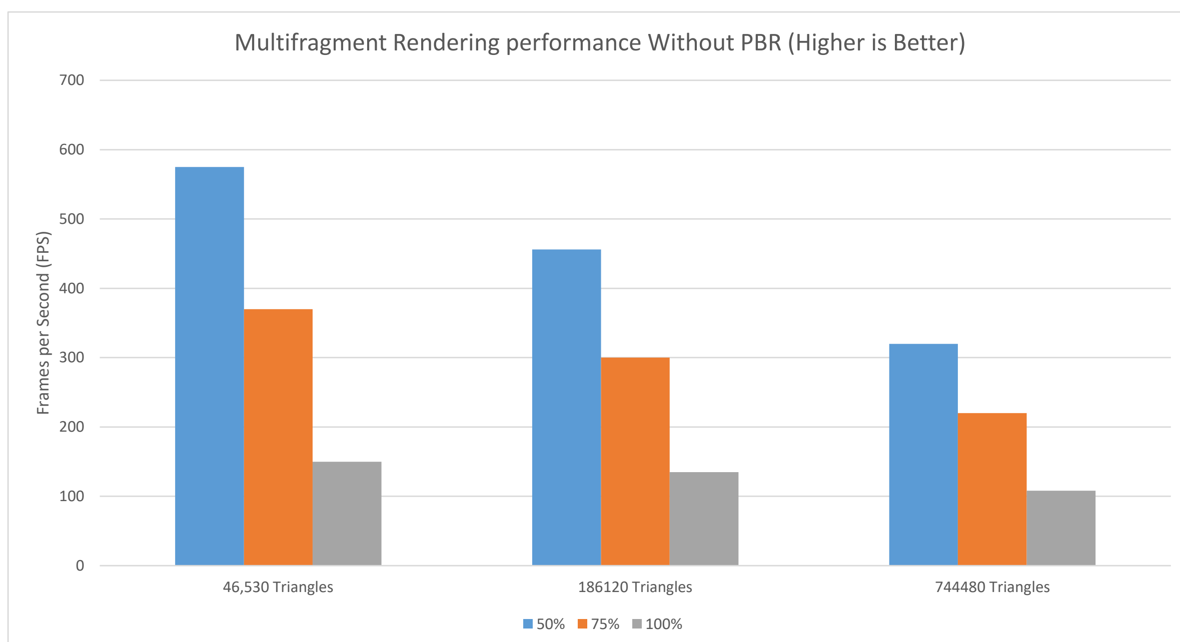

Figure 5.3: Order Independent Transparency performance without PBR

In Fig. 5.4 we see that for high number of triangles we have pretty low performance rate, while for small number of triangles we see tremendous performance values. Taking into respect that the Low polycount can portray almost the same level of detail as the high polycount one we can understand why the PBR pipeline is so common in the Game and Movie industry.

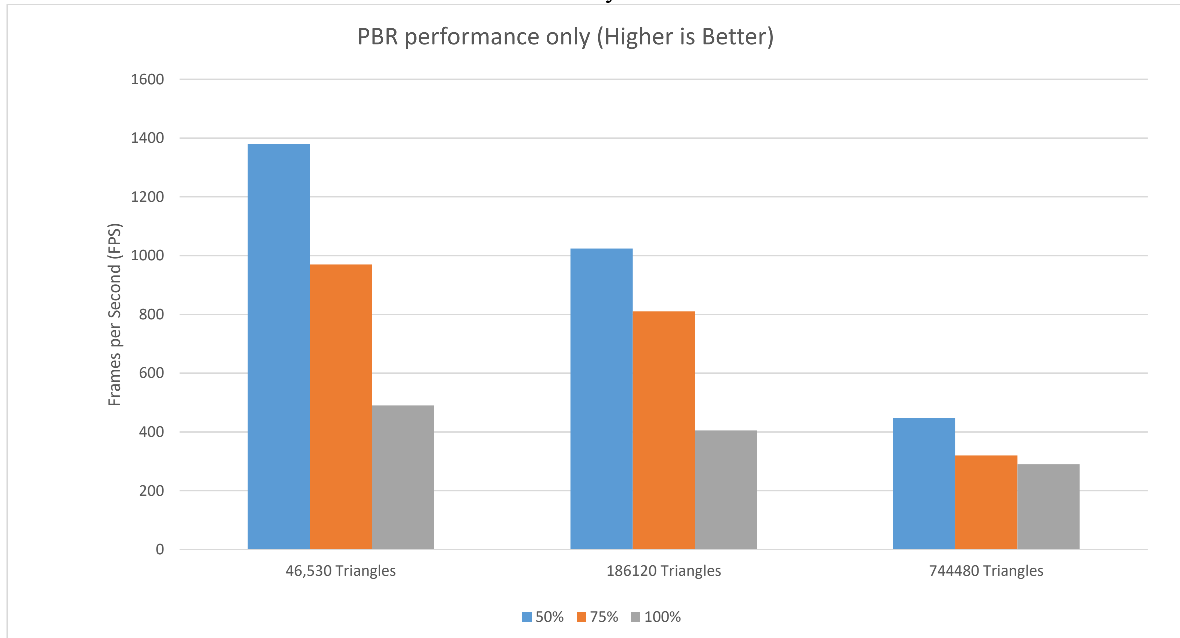

Figure 5.4: PBR performance without Order Independent transparency

The Results of the PBR pipeline can be seen below where we have the geometry of the model in Fig. 5.5 without any color information. In Fig. 5.6 we can observe the little details in the surface of the model such as the corrosion in the right arm, which was captured by photogrammetry , however the file size of the original mesh is many times larger. In Fig. 5.7 the final result of the PBR method is shown.

Figure 5.5: Geometry of the model

Figure 5.6: Geometry + Normal Texture

Figure 5.7: Geometry + Normal + Diffuse

We can also see a different and more interesting angle were we can see high detail in the shield of the statue and the influence of PBR in the final result (Figures 5.8 5.9 5.10.

Figure 5.8: Geometry of shield

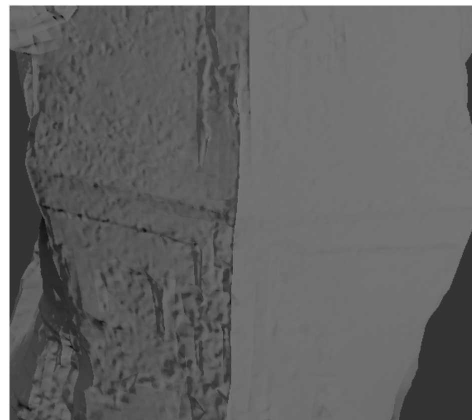

Figure 5.9: Geometry + Normal Texture

Figure 5.10: Geometry + Normal + Diffuse

Here in Fig. 5.11 we can see the outer layer of the object zoomed in the lower part of the right arm and in the next Fig. 5.12 we see the inner layer, while in the last Fig. 5.13 we can fully observe the three layers of the model, all in real-time. The inner layers of the model are colored differently for clarity purposes and the layers were simulated and not based on ultrasound measurements. However we can see that the results are rendered without issues, they are straightforward and the rendering is completed with real-time performance.

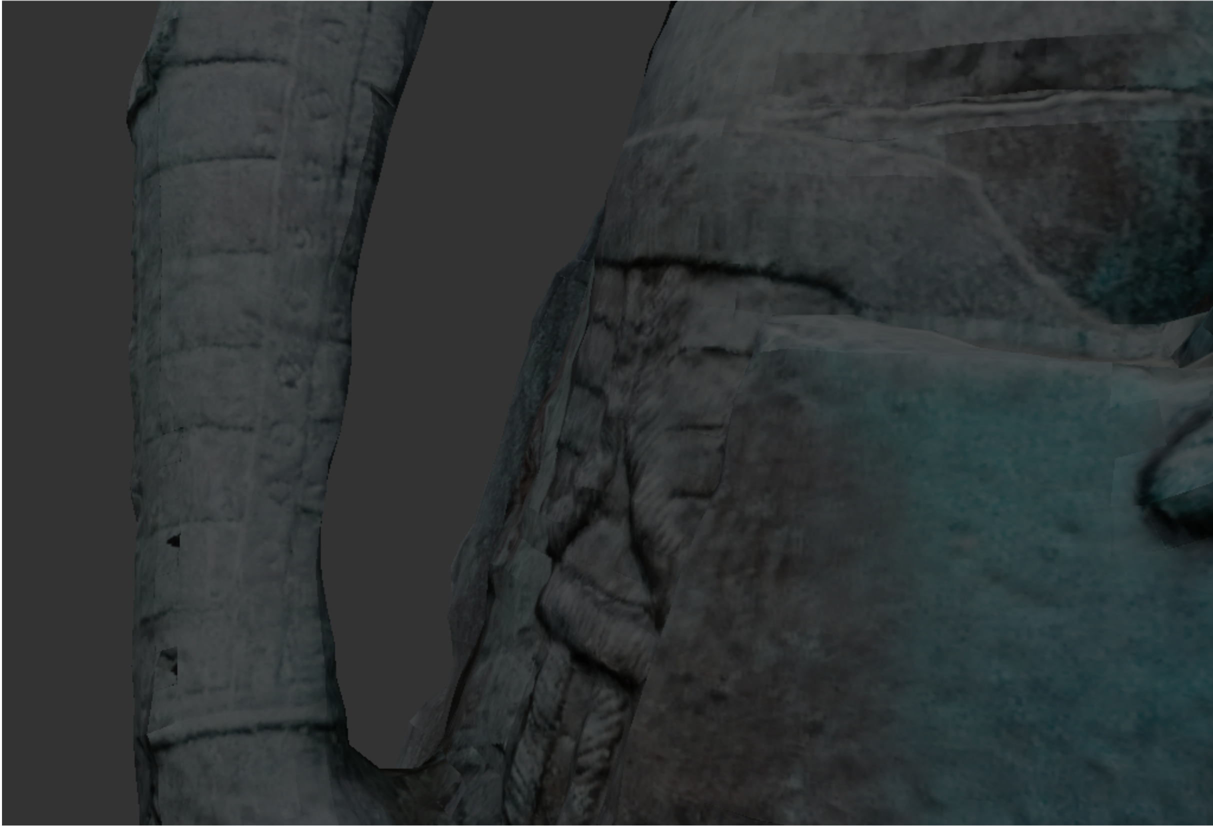

Figure 5.11: Outer Layer

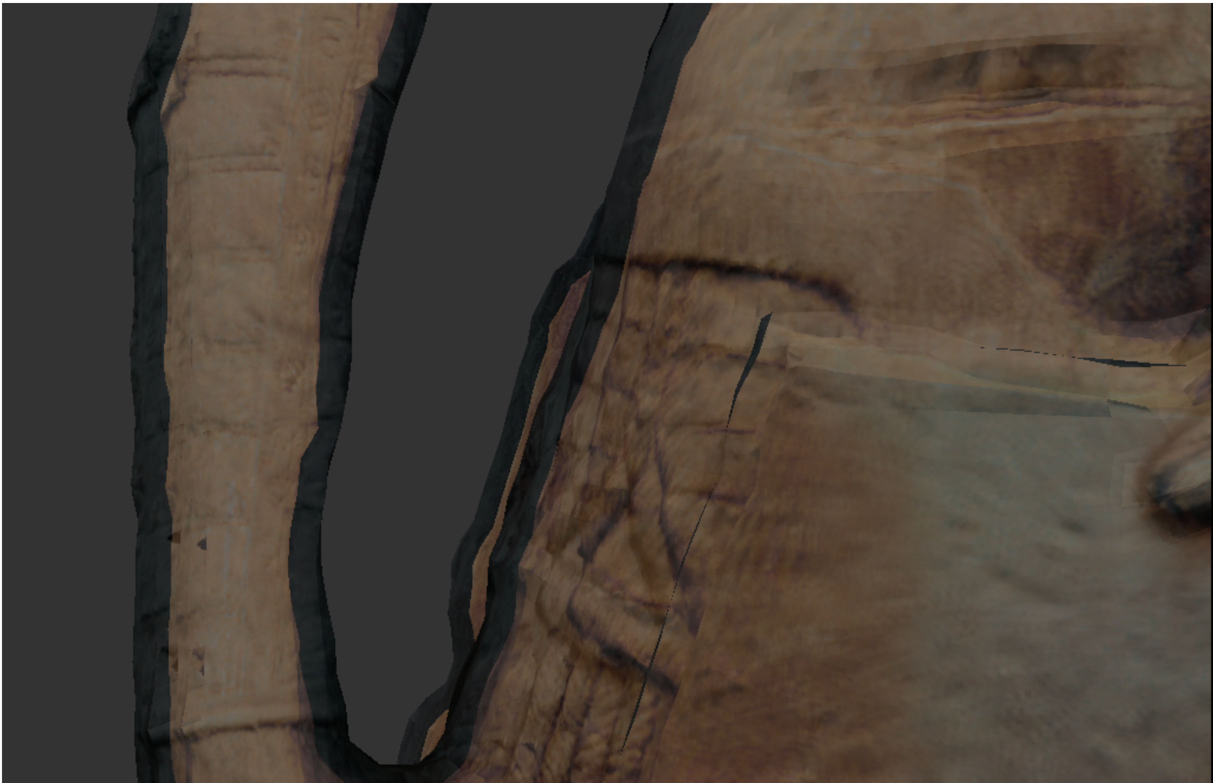

Figure 5.12: Middle Layer

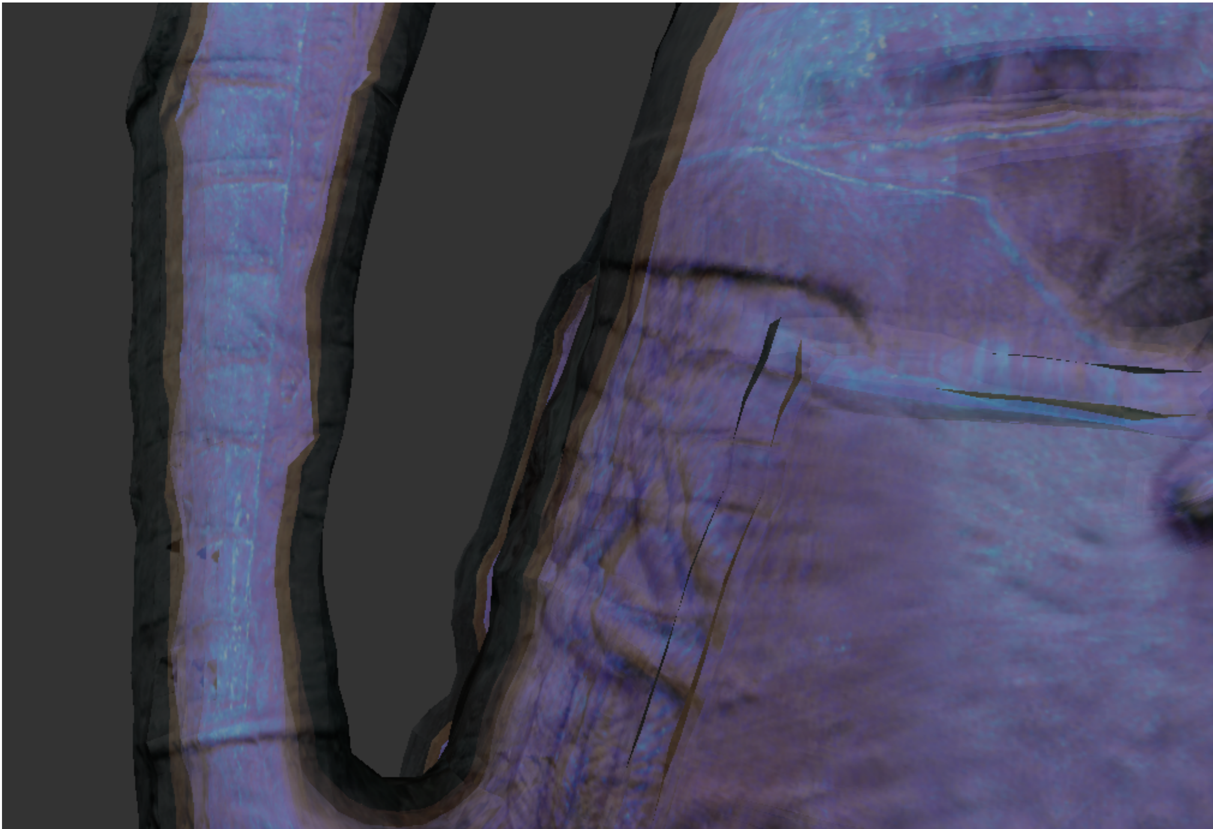

Figure 5.13: Inner Layer

Chapter 6

Conclusion and Future Work

The complexity of data and their amount led us to state of the art solutions for visualization. In our work we presented the methods we developed and combined for a tool that will be of great help for archaeologists, curators and even everyday users that have the curiosity to observe the aging process of a cultural heritage object. Our tool can load almost any common 3D file format along with its properties and render the result in real-time. Overall we report on a finished tool that can be used as a standalone viewer or be integrated in a bigger project if needed.

From that point on propositions regarding the improvement of the tool can be made through the feedback taken from the end users in a form of a case study. Also interface changes can be made to improve the interaction user has with the tool and overall usability of it. Some features like user annotation and observation notes can be implemented in later versions for more detailed analysis on an artifact.

Regarding the algorithmic part of our work there are certain improvements that could be made. In the multi-fragment part, the memory allocation for the 3 buffers is currently static which leads to excessive memory needs. An extra pass could be added to analyze the number of fragments in each pixel, thus enabling us to allocate proper amount of memory for our model. In that case we could investigate whether an extra pass, and its affect on the performance, is tradeable for smaller memory demands. Performance improvements can also being researched Also in the PBR part of our method we could add Global Lightning for better results or implement methods that are currently being introduced in the industry.

Bibliography

[1] S. Mérillou and D. Ghazanfarpour, "Technical section: A survey of aging and weathering phenomena in computer graphics," *Comput. Graph.*, vol. 32, pp. 159–174, Apr. 2008.

[2] D. Frerichs, A. Vidler, and C. Gatzidis, "A survey on object deformation and decomposition in computer graphics," *Comput. Graph.*, vol. 52, pp. 18–32, Nov. 2015.

[3] H. Rushmeier, *Computer Graphics Techniques for Capturing and Rendering the Appearance of Aging Materials*, pp. 283–292. Springer US, 2009.

[4] J. T. Kider, Jr., *Simulation of Three-dimensional Model, Shape, and Appearance Aging by Physical, Chemical, Biological, Environmental, and Weathering Effects*. PhD thesis, Philadelphia, PA, USA, 2012. AAI3542821.

[5] A.-A. Vasilakis and I. Fudos, "Depth-fighting aware methods for multifragment rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 6, pp. 967–977, 2013.

[6] A. Vasilakis and I. Fudos, "S-buffer: Sparsity-aware multi-fragment rendering," in *Eurographics (Short Papers)*, pp. 101–104, 2012.

[7] B. Karis and E. Games, "Real shading in unreal engine 4."

Short Biography

I was born and raised in Kalamata city in southern Greece. Computers and modern technologies were always my passion. That passion led me in 2009 as an undergraduate student in the Department of Computer Science and Engineering in the University of Ioannina from which I graduated in 2015. Since February 2016 I am a graduate student in the same Department and also part of the Computer Graphics Reasearch Group under the supervision of Professor Ioannis Fudos. I have also worked as an research associate in CERTH (CEntre for Research and Technology Hellas) in the team of Dr Dimitrios Tsovaras since July 2017.