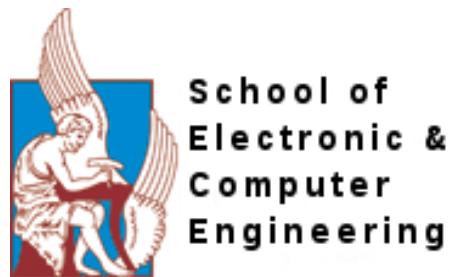


Technical University of Crete



**3D Photorealistic Reconstruction and Digital Restoration of Neoria,
Crete using Geodetic Measurement and Computer Graphics
Techniques**

Thesis By

Effrosyni Sarri

Examining Committee
Assoc. Prof. Aikaterini Mania, Supervisor
Prof. Stavros Christodoulakis
Dr Lemonia Ragia

Chania,
2015

ABSTRACT

This thesis puts forward a 3D reconstruction methodology applied to a historic building taking advantage of the combined speed, range and accuracy of a total geodetic station. The measurements representing geographical data, produced an accurate and photorealistic geometric mesh of a historic monument named ‘Neoria’. ‘Neoria’ is a Venetian building located by the old harbour at Chania, Crete, Greece. The integration of tacheometry acquisition and computer graphics puts forward an effective framework for the accurate 3D reconstruction of a historical building. The main technical challenge of this work was the production of an accurate 3D mesh based on a sufficient but small number of tacheometry measurements acquired fast and at low-cost. A combination of surface reconstruction and processing methods ensured that a detailed geometric mesh was constructed based on a few points. A fully interactive application based on game engine technologies was developed in regards to the reconstructed monument. The user can visualize and tour the monument and the area around it, as well as manipulating the model. Advanced interactive functionalities are offered to the user in relation to identifying restoration areas and visualizing the outcome of such works. Moreover, the user could visualize the co-ordinates of the points measured, calculate distances at will and navigate the complete 3D mesh of the monument. The goal of this framework is to utilize a small number of acquired data points and present a fully interactive visualization of a georeferenced 3D model.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Assoc. Prof. Katerina Mania, for giving me the chance to work on this thesis and for her guidance throughout this process.

I would like to express my gratitude to Dr. Lemonia Ragia for her help throughout this work. I would also like to thank her for providing her technical expertise and guidance during the data acquisition process, without which this thesis would not be possible.

I would like to thank Anastasia Tzigounaki, serving as the director of the 15th Ephorat of Prehistoric and Classical Antiquities in Chania, Greece, for her contribution to this work.

I would like to thank Prof. Stavros Christodoulakis for reviewing this thesis.

Finally I would like to thank my friends for their moral support and express my sincere gratitude to my family for their encouragement and for helping me realize my dreams.

TABLE OF CONTENTS

Chapter	Page
ABSTRACT	ii
ACKNOWLEDGMENTS.....	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES.....	vii
Chapter 1 : Introduction	1
1.1 Concept	1
1.2 Thesis Outline.....	2
Chapter 2 Background and Literature Review	4
2.1 Virtual Environment	4
2.2 3D Digitization.....	5
2.3 Data Acquisition Methods	5
2.3.1 Photogrammetry.....	6
2.3.2 Laser Scanning	6
2.3.3 Geodetic Total Station.....	8
2.3.4 Choice of data acquisition method	8
2.4 3D Computer Graphics	9
2.4.1 3D Modeling	9
2.4.2 Layout	11
2.4.3 3D Rendering	11
2.5 Definitions.....	12
2.5.1 Polygon Mesh	12
2.5.2 Texture Mapping	12
2.5.3 Shader	14
2.6 Surface Reconstruction	14
2.7 Subdivision Surface	15
2.8 Game Engines	16
2.9 Popular Game Engines.....	17
2.9.1 Unreal Engine / Unreal Development Kit.....	17
2.9.2 Unity 3D	17
2.9.3 CryEngine	18
2.9.4 Comparison.....	18
Chapter 3 Technological Background	19
3.1 Unity 3D	19
3.1.1 What is Unity 3D?	19
3.1.2 Project Structure in Unity 3D	19
3.1.3 Components.....	19
3.1.4 Scripting	22
3.1.5 Interface	24
3.2 Autodesk 3ds Max	24
3.3 AutoCAD Civil 3D	25

3.4 MeshLab	26
Chapter 4 Monument Survey And Data Acquisition	27
4.1 Introduction	27
4.2 Venetian Neoria	27
4.3 Monument Survey	30
4.3.1 Introduction	30
4.3.2 Geodetic Total Station.....	30
4.3.3 Field Work	31
4.4 Geodetic Data.....	35
Chapter 5 3D Reconstruction.....	37
5.1 Surface Creation.....	37
5.1.1 Data evaluation	37
5.1.2 Study of the known surface reconstruction algorithms	38
5.1.3 TIN Surfaces and Delaunay Triangulation	39
5.1.4 Reconstruction of TIN surfaces	41
5.1.5 Processing of the TIN surfaces	49
5.2 Surface Processing.....	52
5.3 Texturing.....	54
5.3.1 UV Mapping	54
5.3.2 Texture mapping	59
5.3.3 Normal mapping	61
Chapter 6 Presentation of the 3D interactive visualization and the Graphical User Interface	63
6.1 Concept	63
6.2 User Navigation and Perspective.....	64
6.3 Graphical User Interface	65
6.3.1 The Main Menu	65
6.3.2 The Submenu.....	70
6.4 Additional Features	75
Chapter 7 Implementation.....	77
7.1 3D model import	77
7.2 Creation of the 3D environment.....	78
7.3 User Navigation and Camera Control	80
7.4 Spatial Recognition	83
7.5 The Main Menu.....	84
7.5.1 Transportation functionality	86
7.5.2 Sun simulation functionality	87
7.5.3 Display measurements functionality	89
7.5.3.1 Data Storage	89
7.5.3.2 Initialization of the database	91
7.5.3.3 Display measurements	91
7.5.3.4 Clickable measurements	92
7.5.4 Display Raw Data functionality	93
7.6 Display coordinates dynamically.....	96
7.7 The Submenu	97

7.7.1 Alteration functionality	98
7.7.2 Information plane	103
7.7.3 Measure Dimensions Functionality	103
7.8 Lighting transition.....	105
7.9 Compass	108
7.10 Application utilities	108
Chapter 8 Result and System Evaluation	110
8.1 3D Reconstruction Evaluation.....	110
8.1.1 Accuracy Evaluation.....	111
8.2 Evaluation of the application.....	113
Chapter 9 Conclusions and Future Work	116
References.....	117

LIST OF FIGURES

Figure	Page
Figure 2.1 An example a of a virtual environment. ("A Coved Streetscape" by Prefurbia - Own work. Licensed under CC BY-SA 3.0 via Wikimedia Commons)	4
Figure 2.2 A Point cloud laser scan dataset (Julian Leyland -University of Southampton)	8
Figure 2.3 An example of polygonal modeling.	10
Figure 2.4 Example of a triangle mesh representing a dolphin.	12
Figure 2.5 Right cube textured with only diffuse map. Left cube textured with normal map along with diffuse map.....	14
Figure 3.1 An empty GameObject with a transform component.	20
Figure 3.2 An example of a mesh filter and a mesh renderer component.	21
Figure 3.3 Left: a material component. Right: the built-in shaders that Unity provides..	22
Figure 3.4 Unity's interface with a scene loaded.	24
Figure 3.5 3ds Max interface.	25
Figure 3.6 AutoCAD Civil 3D interface.	26
Figure 3.7 Meshlab Interface.....	26
Figure 4.1 Venetian Neoria at the old harbor of Chania.	27
Figure 4.2 The remaining Neoria. The seven continuous domes and the Grand Arsenal (image taken from Google maps).	28
Figure 4.3 Damaged areas.....	29
Figure 4.4 Inappropriately restored areas.	29
Figure 4.5 Total Station.	30
Figure 4.6 Horizontal angle θ	31
Figure 4.7 The vertical angle ZA and the slide distance SD.	31
Figure 4.8 The selected positions (stations) from which we took the measurements....	32
Figure 4.9 The measured points (in red) of a section at the south part of the monument.	33
Figure 4.10 An example of the measurements we acquired of some of the damages. We focused on measuring their outline and their depth.	34
Figure 4.11 The sketch of the south side of the seventh dome. The measured points are marked with their respective numbers.....	35
Figure 4.12 A sample set of the final GGRS87coordinates (id;easting;northing;height).	36
Figure 5.1 The points acquired displayed in 3D space.....	37
Figure 5.2 Implementation of the ball pivoting algorithm. Holes and discontinuities are shattering the mesh.....	39
Figure 5.3 A part of a TIN surface created with Delaunay triangulation.	40
Figure 5.4 Civil3D's viewport. It processes the data with a 'top' view.	41
Figure 5.5 Rotation of south group data.	42
Figure 5.6 Rotation stages of north data.	43
Figure 5.7 Rotation stages of west group data.	44
Figure 5.8 Rotation stages of east group data.	45

Figure 5.9 Import of csv files for east and west data.....	46
Figure 5.10 Insertion of the data files in survey database.....	46
Figure 5.11 Above the measured 3D points of the south side. Below the measured 3D points of the west side.....	47
Figure 5.12 Above the measured 3D points of the north side. Below the measured 3D points of the east side.....	47
Figure 5.13 The triangulated surface of the south side.....	48
Figure 5.14 The triangulated surfaces of (clockwise from top left) the west, north and east sides.....	48
Figure 5.15 On top: the south side as it was imported from Civil3D (i.e. the TIN surface). On bottom: the north surface after some process.....	49
Figure 5.16 The stages of the west side correction.....	50
Figure 5.17 Depth details in doors and windows.....	51
Figure 5.18 Stages of attachment. Each surface was attached accordingly to form the final shape of the building.....	51
Figure 5.19 The final form of the 3D model.....	52
Figure 5.20 The 3D model after implementing midpoint subdivision.....	53
Figure 5.21 Perfectly rounded windows after further refinement.....	54
Figure 5.22 An example of UV mapping of a cube. The flattened cube can be painted to texture the cube.....	55
Figure 5.23 The Unwrap UVW modifier interface. Its menu is on the right with the UV editor open in the middle.....	56
Figure 5.24 Some examples of the face groups that were created for texturing.....	57
Figure 5.25 Left: the seams of the mesh that separate the clusters. Right: the created clusters are shown on the UV editor window.....	58
Figure 5.26 UV texture coordinates rendered in bitmap images.....	58
Figure 5.27 Photographs of the westernmost façade of the north side taken on the field.....	59
Figure 5.28 The UV template we rendered on the left, and the texture we created on the right.....	60
Figure 5.29 The textured group. Displayed with (left) and without (right) the triangle mesh.....	60
Figure 5.30 The textured model displayed from several sides.....	61
Figure 5.31 An example of a normal map.....	61
Figure 5.32 A normal map (right) generated from the texture (left).....	62
Figure 6.1 The perspective of the user. It is a first person view.....	64
Figure 6.2 The main GUI.....	65
Figure 6.4 Part of the building with the light source stopped above it (left) and in the shadows (right).....	66
Figure 6.3 The display when the user presses each transport button. Clockwise: north, south, east, west.....	66
Figure 6.5 The day/night transition. On the top left the day simulation is shown , on top right a middle time of day is shown and on the bottom the night simulation is displayed.....	67

Figure 6.6 The display measurements functionality. On the right picture the measurements that were conducted are displayed on the surface (The left picture is without that functionality enabled).....	68
Figure 6.7 When a green cube is clicked the corresponding geographical data from the measurements appear next to it.....	68
Figure 6.8 The display raw data toggle activated. A scroll plane appears on the screen with the geographical information acquired.....	69
Figure 6.9 When a point ID button is pressed, information about its position appears on screen and the point is displayed on the surface.	70
Figure 6.11 The material of the dome changed in order to cover the white paint. The user can toggle between surfaces.....	71
Figure 6.10 Two hotspots as they appear on the surface monument.....	71
Figure 6.12 On the left added cement can be seen on the monument's facade. The user can toggle between materials in order to visualize the original surface.....	72
Figure 6.13 The user can alter the geometry provided, by toggling between the existing one and the 'restored' one.....	72
Figure 6.14 The missing roof edge can be visualized with the toggle hotspot button....	73
Figure 6.15 The blocked door on the south side can be replaced with a regular one. ..	73
Figure 6.16 When the 'Show Info' button is clicked, a scroll plane with historical information appears on the screen.....	74
Figure 6.17 The steps for measuring the dimension between two points.....	75
Figure 6.18 The dynamic display of the coordinates. Wherever the mouse points, the corresponding coordinates of the spot appear on the bottom of the screen.....	76
Figure 7.1 Import settings for assets.	77
Figure 7.2 Bumped diffuse shader.	78
Figure 7.3 The final model imported in Unity. The normal mapping depicts a high level of surface detail.....	78
Figure 7.4 The terrain we created in 3ds Max (left) and the monument with the terrain (right).	79
Figure 7.5 Street lights as created in 3ds Max (left) and places in the scene (right).	79
Figure 7.6 The components of the parent of the first person controller.	81
Figure 7.7 The components of the main camera.	82
Figure 7.8 The east side trigger. In the inspector we have defined as trigger and we have disabled the mesh renderer.	83
Figure 7.9 Top left: sun parent object with the children objects. the components of the sun and directional light object are shown.....	88
Figure 7.10 The GUI of the database as is displayed on screen.....	94
Figure 7.11 The sphere object components.	95
Figure 7.12 The mouse points at a spot and its coordinates are displayed on the bottom.	96
Figure 7.13 The object used to alter the appearance of the dome. The white parts are gone, and are textured with similar looking material.....	99
Figure 7.14 The object representing the missing roof.	100
Figure 7.15 The part of the building as is currently (left) and with the missing roof edge (right).	100
Figure 7.16 The material list of the 3D model.....	101

Figure 7.17 The created object with the door opening. The newly created faces are rendered in red.....	102
Figure 7.18 The streetlight with the two lights attached to it and a plane with an image of a glow.....	106
Figure 8.1 The comparison of the 3D model with the original raw data.....	112
Figure 8.2 The instructions menu. When one of the buttons is clicked instructions about a particular functionality are displayed.	114

Chapter 1 : Introduction

1.1 Concept

In the last years, virtual environments have established their presence in the scientific field due to the surge of the technology advancements. They have been a huge aid in the visualization of scientific data, contributing in ways of illustrating, understanding or assessing them. The 3D digitization that virtual environments facilitate, captures scientific or cultural data and preserves them for posterity. Digital 3D reconstructions of physical objects provide not only a way of storing information but also a means for scientific research and easier data accessibility to a wide audience.

Benefited by the expansion of such technologies, the heritage community has witnessed an increase in three-dimensional photorealistic reconstructions of archaeological structures. Assisted by advances in computer graphics and digital representation, the architectural and archaeological society have been frequently developing digital content. 3D reconstruction methods promote and ensure that scientific and historical accuracy is preserved. The ability to reconstruct a historical artefact or a building's current or past structure is an invaluable tool for assessing and moving the data into the future.

The 3D digitization of current historical structures and the 3D reconstruction of their appearance should be based on accurate on-site measurements. Tacheometry acquisition methods [1] use modern total stations that result in high accuracy point measurements as well as quick and efficient spatial data acquisition in order to create a geometric mesh of existing geometry structures. The aim of virtual depiction and computer graphic techniques is the creation of accurate, high quality visualizations that faithfully represent the geometric structure in a physical environment, making it in cases, indistinguishable from the real scene. Reliable geometric measurements are necessary to create a geo-referenced 3D model which could be presented and manipulated by users.

In this thesis, the integration of tacheometry acquisition and computer graphics puts forward a framework for the scientifically accurate 3D reconstruction of 'Neoria', a historical building in Crete, Greece. Initially, a detailed 3D geometric model of the monument is created based on acquired terrestrial points on-site. Surface reconstruction and processing methods ensure that a detailed geometric mesh is constructed based on far fewer points available compared to laser scanning. The created 3D model is presented via an interactive application developed based on game engine technologies. The user can interactively manipulate as well as navigate the geometric model. Advanced interactive functionalities are developed in relation to visualizing the monument under different conditions. Moreover, the geographical data that were acquired are presented and associated with the model, putting it in context with the real world and the user can gather scientific information about the monument.

The goal of this thesis was to utilize a small number of acquired data points and present a fully interactive visualization of a georeferenced 3D model.

1.2 Thesis Outline

In the following chapters the framework that was followed is explained thoroughly. Beginning with the data acquisition process, the 3D reconstruction and continuing with the development of the application, the stages of the implementation of the project are analytically presented, as well as the tools and features that were used. More specifically:

In the second chapter the research that was conducted is presented. The use and significance of 3D technologies is explained. The chapter researches and evaluates the available methods of acquiring 3D data as well as providing an insightful review of surface reconstruction algorithms. Moreover a small review of game engines is provided.

In the third chapter the software tools that were used in the development of the project are described. Unity 3D's main components are presented and explained, as well as how the tools and features that it provides are combined to develop interactive applications. The rest of the modeling software that was used in the duration of the thesis development are also presented.

The fourth chapter describes the data acquisition process. After a brief historical background of the examined monument, it presents how the field work was conducted, what equipment was used and the methodology of the measurements. The characteristics of the data that were acquired are presented, as well as their initial processing.

The fifth chapter thoroughly presents the 3D model creation. It begins with the examination of the data and the established methods of surface reconstruction, and describes the applied method and the reasoning behind its use. It continues with the surface processing and the methodologies that were employed in order to result in a photorealistic 3D representation of the model.

In the sixth chapter the 3D interactive application that was developed is introduced. The Graphical User Interface is presented, along with the functionalities that were implemented.

The seventh chapter contains the analytical description of the implementation of the application. It presents how the functionalities were created, the methods that were applied and the process of the development in order to create the final application.

In the eighth chapter the evaluation of the project is presented. The assessment of the quality of the 3D model is conducted, through an accuracy evaluation. The evaluation of the final application by users is described, along with the changes that were implemented based on their suggestions and comments.

In the ninth chapter the conclusions of the whole development process are presented, along with suggestions about future extensions.

Chapter 2 Background and Literature Review

2.1 Virtual Environment

Virtual environment is considered a simulated real or imaginary environment, created with computer graphics and programming. It is a three-dimensional space, which users can interact with and manipulate. These environments often referred to as 3D interactive environments are categorized in two groups, the desktop virtual reality and the immersive virtual reality. In the desktop virtual reality the user can observe or interact with a 3D environment that is displayed on a screen, whereas in immersive virtual reality the users can feel that are present in the 3d world, creating the illusion of reality. This can be achieved through the use of head-mounted displays (stereoscopic goggles) or cave automatic virtual environments (CAVE). [2]

As the technology improved, the use of such environments became more frequent. Along with the advances in computer graphics, 3D interactive environments have found their place in fields such as science, education, entertainment and medicine. The use of immersive virtual reality applications has rapidly increased not only in entertainment but also in training (pilots and astronauts are trained through simulations). Desktop virtual reality doesn't require any expensive hardware or software and it is more accessible to the public. Having started in the entertainment industry, such technologies have rapidly spread and have been a huge aid in teaching, learning, the industrial field and the visualization of scientific data (architectural, physical, medical, etc) [3]. The latter has played an important role in illustrating, understanding and assessing the data.



Figure 2.1 An example a of a virtual environment. ("A Coved Streetscape" by Prefurbia - Own work. Licensed under CC BY-SA 3.0 via Wikimedia Commons)

Our interest is focused in the use of such 3D interactive environments to display scientific data. In our case geographical data are used to form a three dimensional model displaying an archaeological monument which is presented through a virtual environment. The research that was conducted, in order to create the framework we followed, is presented in this chapter.

2.2 3D Digitization

Various scientific fields are interested towards digitally representing physical objects. The depiction of objects as digital models is becoming a significant part of fields such as engineering, scientific research, entertainment, architecture and archaeology. 3D technology is utilized so that we could capture the world around us in an accurate way.

Cultural heritage has embraced such technologies. Three-dimensional digitization is a way of moving the data into the future, preserving invaluable artifacts and monuments for posterity. Objects of historic value face the danger of deterioration from erosion, weathering or damage. 3D digitization not only can replicate geometry and texture with precision but also can be used for restoration, access or research.[\[4\]](#)

The process of digitization relies heavily on the size of the object, as well as its complexity. In some cases it is implemented by modeling the object using computer aided design programs based on empirical measurements. This method can be applied in objects with simple shape and not many differentiations on their surface. The drawbacks are the possible lack of accuracy and the human error that this method may entail. Proper methods for 3d digitization acquire data with the use of electronic equipment. In this way the digital model is created with a high level of accuracy, containing all the surface details. [\[5\]](#)

It has become clear, that to obtain an accurate record of the current condition of an object, a ranging device must be employed. That approach generates data with which the geometric and thematic information can be reproduced. The variety of those ranging devices that are available is presented in the next section. [\[6\]](#)

2.3 Data Acquisition Methods

Three dimensional digitization should be based on accurate on site measurements [\[8\]](#). Technological advances have given the ability of fast and high quality data obtainment through a variety of electronic devices. The electronic equipment has given a reliable and automatic nature to the data acquisition process that has led to accurate and high quality results.

The main methods to acquire data for the documentation of objects in three dimensional environments are photogrammetry, laser scanning and the topographic method which employs a geodetic total station [7]. They are explained in the following sections.

2.3.1 Photogrammetry

Photogrammetry is the science of making measurements from photographs. It is used in fields, such as topographic mapping, architecture, engineering, and geology, as well as by archaeologists and by meteorologists in cases where objective weather data cannot be obtained [9]. Photogrammetry refers to acquiring reliable information about physical objects and the environment through processes of recording, measuring and interpreting photographic images.

It is widely adopted for 3d digitization, by mainly using a series of photo images to computationally map a 3d model or space. This is accomplished without any physical contact with the object which is the main difference to other methods of surveying. It is categorized in two basic fields : aerial photogrammetry and close-range photogrammetry. In aerial photogrammetry the images are taken from an aircraft vertically pointed towards the ground. In close-range photogrammetry the camera is typically on a tripod or hand-held.[10]

In order to be employed for architectural and archeological 3d reconstruction, it requires specialized hardware and software, such as a non standard digital camera and a geodetic total station. The acquisition of the digital images is a meticulous process, as it requires controlled shooting in order to capture every visible surface of the object. The content and the angle of each image should be determined beforehand in order to have continuous images which overlap with each other, something that is required for the software process afterwards. It is also necessary to measure coordinates in order to orient the acquired images. The geodetic total station measurements provide the necessary geographical information to result in a geo-referenced 3d model.

Photogrammetry's outcome is a high quality 3d model which includes geometric information. However, it has drawbacks as it can lead to unspecified geometry. This can be caused by inadequate images, referring to not only the number and the consecutiveness of the photos but also to the content. Dark shadows and low luminance areas can prevent efficient data acquisition, leading to inconclusive results. In addition, the amount of 3d data can be quite big, requiring computational power and special software to process it. [11],[12],[13]

2.3.2 Laser Scanning

Laser scanning is considered the most advanced step in the technological progress in the field of digital documentation in architecture and archaeology. It is referred to the

controlled steering of laser beams followed by a distance measurement at every pointing direction and is used to rapidly capture shapes of objects , buildings and landscapes [14]. The devices used in this method, the laser scanners analyze objects and collect data on their appearance and shape that are used to construct digital three dimensional models.[15]

There is a variety of laser scanners available. Different technologies are used in their functionality. They are mainly categorized as contact and non-contact. Contact scanners probe the subject through physical touch to acquire information, while non-contact scanners acquire information from distance. Non-contact scanners are further categorized in active, where they emit some kind of radiation or light and detect its reflection or radiation passing through object, and passive , which they don't emit radiation themselves but detect reflected ambient radiation from objects. Each of these scanners is suitable for different uses. In the field of digital documentation of objects non-contact active scanners are used.

Laser scanners acquire a large number of points on an object's surface, which are used to form a point cloud. The point cloud, which is the direct outcome of the laser scanner, represents the dense set of points that the device has measured. Each point is represented by a set of three dimensional coordinates (X,Y,Z) and a variable i which refers to the intensity of the reflected signal. A single scan usually is not adequate to fully capture the morphology of a structure or an object. Several scans are required from different locations. These scans must be fused together correctly, into a common coordinate system in order to form a complete 3d model. This process is called registration.

The quantity of data acquired with laser scanners can be up to millions of points, which can be a drawback for its post processing. When a high quality result is aimed at, those numbers are common making the raw data treatment very time consuming and quite complicated. The computational cost can be high and even with the latest hardware and software products the quantity of the data should be reduced, so that the data processing can run smoothly. Additionally, it should be noted that the laser scanning equipment is highly expensive.[16] [17] [18]



Figure 2.2 A Point cloud laser scan dataset (Julian Leyland - University of Southampton)

2.3.3 Geodetic Total Station

The topographic method of data acquisition employs a geodetic total station, which is an electronic, optical instrument used in modern surveying. It takes measurements by emitting laser beams to a target and detecting light reflected off it and calculating the deviation of the wavelength of the reflected light. It carries out goniometric measurements at the same time with measurements of lengths. Specifically, it measures slope distances, straight and perpendicular angles, and heights. The raw data of the total station result in three dimensional coordinates x,y,z or easting, northing and elevation which are computed from the measurements using trigonometry and triangulation. (The functionality of the total station is further explained in chapter 4.)

The geodetic total stations are portable, inexpensive in comparison with the laser scanners and widely available. They don't require special training and can acquire fast a collection of spatial coordinates on site. They are considered instruments of high accuracy and can produce precise results even in cases of documentation of huge objects with big dimensions. Due to the fact that the total stations measure selected , individual points the post processing is low-cost , flexible and simpler in relation to the previous methods. [19] [20]

2.3.4 Choice of data acquisition method

Both the photogrammetry and the laser scanning methods result in high quality 3d models. They can capture great detail and depict accurately the exact morphology of the surface. However, they include huge amounts of data and in most cases require special and expensive hardware and software. In the case of laser scanning the

material cost as well as the computational cost is extremely high and the model reconstruction process is very time consuming.

On the other hand the total station is fast, flexible, low-cost and highly accurate. What can be considered as a disadvantage, is that it measures one point at a time, making difficult to acquire a large quantity of data at once. However, if the points acquired are focused on interest areas and the amount is sufficient enough, the result can be highly accurate with great quality.

What it should not be forgotten is that there is always a trade-off between economic resources, computational power and technology available. Having considered the resources and the outcome, our choice for the data acquisition process is a geodetic total station. The framework that was implemented with the employment of the total station is presented in later chapters.

2.4 3D Computer Graphics

The field of computer graphics is concerned with generating and displaying three-dimensional geometric data in a two-dimensional space (e.g., computer monitor, screen). Whereas a single point in a two dimensional graphic has the properties of position, color, and brightness, a 3D point adds a depth property that indicates where the point lies on an imaginary Z-axis.

The creation of 3D graphics begins with the modeling process, followed by the layout and interactivity of the modeled objects in a scene. The process ends with the 3D rendering, which refers to the computer calculations required to display the graphic on a screen. [21]

2.4.1 3D Modeling

3D modeling refers to the process of creating the three dimensional surface of an object. The product of this process, the 3D model, is formed by a collection of points in 3D space connected by geometric entities such as lines, triangles, and curved surfaces. The 3D model can originate manually, created by a user of a 3D modeling tool, algorithmically or it can be scanned into a computer from real world objects (via methods that are mentioned in the previous sections).

The 3D models are separated in two categories. One are the **solid** models, which define the volume of the object they represent. These are more realistic, but more difficult to build. Solid models are mostly used for non visual simulations such as medical and engineering simulations. The second are the **shell or boundary** models. These models represent the surface, the boundary of the object, not its volume. These

are easier to work with than solid models. Because the appearance of an object depends largely on the exterior of the object, boundary models are the ones mainly used in computer graphics. Almost all visual models used in the entertainment industry are shell models.

A 3D model consists of points called **vertices** that define the shape and form **polygons**. A polygon is an area formed from at least three vertices, a *triangle*, or four vertices, which is called a *quad*. The line that joins one vertex with another is called **edge**.

There are several modeling methods. The most commonly used ones are:

Polygonal modeling: is a modeling process that represents the surface of an object using polygons. The vertices are connected by line segments to form a polygonal mesh. The vast majority of 3D models today are built as textured polygonal models, because they are flexible and rendered quickly. However, polygons are planar and can only approximate curved surfaces using many polygons.

Curve modeling: Surfaces are defined by curves, which are determined by weighted control points. The curve follows but does not necessarily interpolate the points. Increasing the weight for a point will pull the curve closer to that point.

Digital sculpting: Still a fairly new method of modeling, 3D sculpting has become very popular in the last years. It refers to the use of software that offers tools to push, pull, smooth, or otherwise manipulate a digital object as if it were made of a real-life substance such as clay. It can introduce details to surfaces that would otherwise have been difficult or impossible to create using traditional 3D modeling techniques. The downside is that in order to achieve detail with sculpting the models must have a high number of polygons. [22]

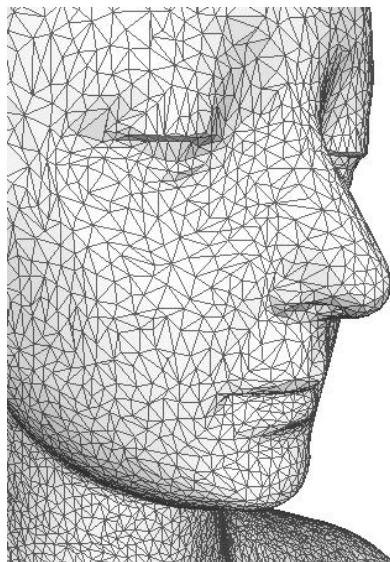


Figure 2.3 An example of
polygonal modeling.

2.4.2 Layout

After the 3D models are created they must be placed in the scene. This defines spatial relationships between objects, including location and size and involves multiple objects interacting with one another. For example, a solid object may partially hide an object behind it. This stage also includes animation, which refers to the temporal description of an object, how it moves and deforms over time.

2.4.3 3D Rendering

Converting information about 3D objects into a graphics image that can be displayed is known as rendering. It usually requires considerable memory and processing power. It is the process of adding realism to computer graphics by adding three-dimensional qualities such as light, shadows and variations in color and shade. This process is usually performed using 3D computer graphics software. There are many rendering methods that have been developed, each one appropriate for specific applications. There is the non photorealistic rendering, which gives the effect of painting, drawing or cartoons, and the rendering methods aiming to achieve high photorealism.

Another categorization is suitability for real time rendering and non real time rendering. Non real time rendering is implemented in non interactive media such as films and video. In this case, the rendering process can take huge amounts of time. That is because non real time rendering has the advantage of very high quality even with limited processing power due to the absence of real time response, which makes the time for the rendering process not considerable. A method suitable for non real time rendering is ray tracing, which simulates the path of a single light ray as it would be absorbed or reflected by various objects in the scene. Real time rendering is implemented in interactive media such as games and simulations. The calculations and the display are happening in real time. The primary goal is to achieve an as high as possible degree of photorealism at an acceptable rendering speed. This is 24 frames per second, as that is the minimum the human eye needs to see to successfully create the illusion of movement.

A method that is quite popular is radiosity. Radiosity is a global illumination algorithm, in the sense that the illumination arriving on a surface comes not just directly from the light sources, but also from other surfaces reflecting light. It is used not only for non real time rendering but also for real time with the use of pre computed calculations on the scene. [23]

2.5 Definitions

2.5.1 Polygon Mesh

A polygon mesh consists of vertices, edges and faces that define the shape of a 3D object. The vertices are points which represent positions along with other information such as color and texture coordinates, an edge is a connection between two vertices and a face is a closed set of edges. The faces can be represented by triangles (three edges), quads (four edges) or convex polygons (five or more edges). A vertex can be shared by many adjacent faces, and an edge can be shared by no more than two faces. The faces share edges and form the three dimensional surface of an object like ‘tiles’.

A polygon mesh is fairly easy to render, however most rendering hardware supports only triangle or four sided faces (quads). It should be noted that most computer graphics software require that objects be composed entirely of triangles or quads. [24]

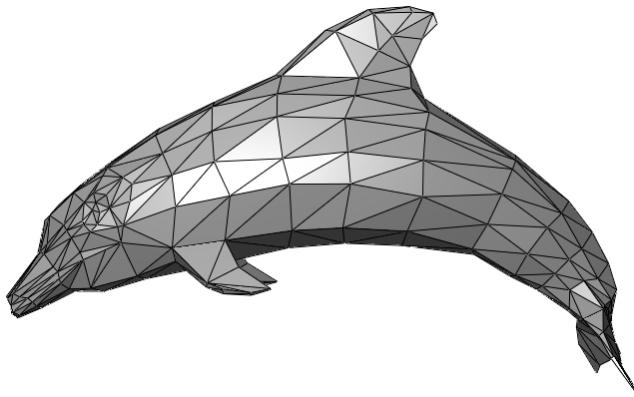


Figure 2.4 Example of a triangle mesh representing a dolphin.

2.5.2 Texture Mapping

There are two ways of adding detail to a surface. One method is to add details via modeling. That means that extra polygons need to be created in order to form the details. This results in increasing the scene complexity and consequently the rendering speed. In addition there are some fine details that are very hard to model. The other method, which is a more popular approach is to map a texture to the surface. Texture mapping is making possible to simulate photorealism in real time, by vastly reducing the number of polygons and lighting calculations needed to create a realistic 3D scene.

A texture map is a bitmap image that is applied to the surface of a polygon. Every vertex in a polygon is assigned a texture coordinate, which is known as a UV coordinate. Image sampling locations are then interpolated across the face of the polygon to produce a visual result that seems to have more details than could otherwise be achieved with a limited number of polygons.

Texture mapping was originally a method of simply mapping pixels from a texture to a 3D surface. Nowadays many complex mapping methods have been developed in order to add photorealism and detail to a flat surface. Some of the most used mapping methods are presented below. [25]

Diffuse mapping: A diffuse map is a texture you use to define a surface's main color. It is the most frequently used texture mapping method. It wraps a bitmap image onto the surface while displaying its original pixel color. Any bitmap image, such as images captured by digital camera, can be used as diffuse map. It sets the tint and intensity of diffuse light reflectance by the surface.

Bump mapping: is a texturing technique for simulating small displacements on the surface of an object, while the surface geometry is not modified. Instead only the surface normal is modified as if the surface had been displaced. This is achieved by perturbing the surface normals (the imaginary vectors perpendicular to the surface that are used in lighting or shading calculations) of the object and employing the perturbed normal during lighting calculations. It uses the values of a grayscale image to simulate abnormalities on the surface. Black areas recede and white areas protrude.

Normal mapping: is considered a variation of bump mapping. A common use of this technique is to greatly enhance the appearance and details of a low polygon model by generating a normal map from a high polygon model or a height map. (a height map is a raster image used to store values such as surface elevation data, for display in 3D computer graphics). Normal maps consist of red, green, and blue. These RGB values translate to x, y, and z coordinates, allowing a 2D image to represent depth. The 2D image is applied to the surface. This way, a surface simulates the lighting and color associated with 3D coordinates. In the normal map, each pixel's color value represents the angle of the surface normal.

Other types of mapping include specular mapping, which allows parts of an object to have a specular effect, reflection mapping which is a technique for approximating the appearance of a reflective surface by means of a pre computed texture image and opacity mapping which determines which parts of the object should be transparent and how much.

Multitexturing is the use of more than one texture at a time on a polygon. For instance, in order to achieve realism a diffuse map is inadequate. Nowadays it is used in addition with normal maps, height maps or bump maps. Employing an appropriate multitexturing technique is crucial, because is an easy, low-cost method of achieving high surface detail with minimal computational cost.

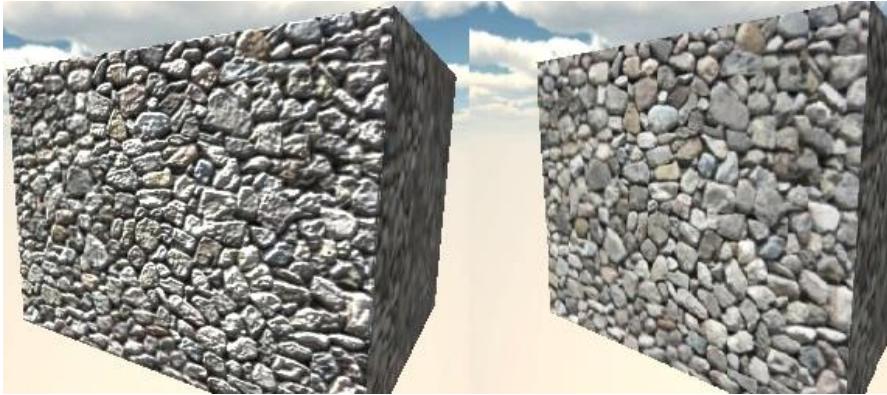


Figure 2.5 Right cube textured with only diffuse map. Left cube textured with normal map along with diffuse map.

2.5.3 Shader

A shader is considered a program that is executed during the rendering procedure and it refers to the process of altering the color of a surface in the 3D scene, based on its angle to lights and its distance from lights to create a photorealistic effect. Shading is also dependent on the lighting itself that is used in the scene. Shaders are most commonly used to produce lighting and shadow. Beyond that more complex uses include altering the hue, saturation, brightness or contrast of an image, producing blur, normal mapping, distortion, and a wide range of others.[\[26\]](#)

2.6 Surface Reconstruction

The 3D data acquisition methods generate data that are three dimensional points. These 3D points describe the shape or topology of the surface of an object. The goal is to produce a 3D model of the studied object from these points. Surface reconstruction algorithms deal with this problem. The input of these algorithms is a set of 3D points and the output is a 3D model. More specifically, the goal of the surface reconstruction algorithms can be stated as follows:

"given a set of sample points P assumed to lie on or near an unknown surface S , create a surface model S' approximating S "(Fabio,2003) [\[27\]](#).

This is a well studied problem in the computer graphics community and it continues to be the theme of research. Since in most cases there is information about a surface only through a finite set of sample points, the surface recovery cannot be guaranteed exactly. As the point density increases the surface is more likely to be topologically

correct, usually a good output is dense in detailed areas and sparse in featureless areas. The usual input of these algorithms is a point cloud (a dense set of 3D points).

There is a vast variety of algorithms which reconstruct the topology of a surface from 3D sample points. The classification of those algorithms can be quite complicated. This categorization can range from the type of input data, to the way the procedure is executed and the resulted surface. According to the type of input data the surface reconstruction algorithms can be categorized as the algorithms that deal with unorganized point clouds and structure point clouds. When the data are unorganized point clouds the algorithms dealing with them do not use any assumptions on the object geometry and the only information they have is the spatial position of the input data. The algorithms dealing with structured point clouds take into account additional information of the data. Another distinction is algorithms that deal with open or closed surfaces.

Taking into consideration the way the surface is reconstructed, there are two main categories. The algorithms that compute geometry and those that use implicit functions.

Computational geometry methods: They use geometric structures such as triangulation, alpha shapes or the Voronoi diagram to form shape based on the input points. The reconstructed surface interpolates all or most of the input points. That means that the algorithms fit the created mesh on the existing points. One of the most popular algorithms in this category is the powercrust algorithm (Amenta et al., 2001) [28] which uses Voronoi diagrams. Another is the ball pivoting algorithm patented by the IBM (Bernardini et al., 1999) [29] which uses alpha shapes.

Implicit functions based methods: They fit functions on the point cloud and they extract the mesh from these functions. Popular algorithms are the Hoppe's algorithm (Hoppe et al., 1992) [30] which estimates the surface using tangent planes and a signed distant function of the points to the tangent planes, and the Poisson algorithm (Kahzdan et al., 2006) [31] which utilizes a function that has value one inside the surface and zero outside. These algorithms approach the problem by approximating the surface and the initial set of input points may not be included in the resulting mesh.

The choice of the appropriate algorithm is important. A lot of parameters should be taken under consideration. The type of data and their properties are a major factor in that choice. If the input data does not satisfy certain properties required by the algorithm the reconstruction produces incorrect results. [32],[33],[34]

2.7 Subdivision Surface

In some cases, the object and their surfaces can be coarse and quite sharp. A usual technique is to implement a subdivision surface. A subdivision surface is a method of representing a smooth surface via the specification of a coarser polygonal mesh. The process starts with a given polygonal mesh. A refinement scheme is then applied to this mesh. This process takes that mesh and subdivides it, creating new vertices and new

faces. The positions of the new vertices in the mesh are computed based on the positions of nearby old vertices. That technique is used by the designers to create smooth and nicely rounded models, giving a more realistic effect. [38]

There are two kinds of subdivision schemes; approximating subdivision surfaces and interpolating subdivision surfaces. Approximating means that the result surfaces approximate the initial meshes and that after subdivision, the newly generated points are not included in the initial surfaces. Examples of such subdivision algorithms are Catmull–Clark (Catmull and Clark, 1978) [35], and Doo-Sabin subdivision surface (Doo and Sabin, 1978) [36]. Interpolating subdivision means that after the subdivision, the control points of the original mesh are interpolated to form the resulting surface. A popular method of interpolating subdivision is the butterfly subdivision (Zorin et al., 1996) [37] which uses triangle meshes.

2.8 Game Engines

A game engine is a system that is made to aid the development and creation of video games. They are used by developers to help create games for consoles, mobile devices and personal computers. Not only do they provide the necessary software framework to develop the game but also they are able to render 2D or 3D graphics in real time, bringing the developers idea onto the screen. The engine is also responsible for such things as Physics, Artificial Intelligence, collisions and particle effects. Game engines provide platform abstraction, which can hide platform differences during the stage of developing. This creates platform independence. It should be noted that a huge advantage of the game engine is hardware abstraction.

Game engines have a component based architecture. Each component is a software package that implements a functionality. The main components are:

The rendering engine, which is the component that deals with the rendering. It provides the rendering of the 3D models as quickly as possible, often with interactive element in real time. It also provides support of various visual effects such as shadows, particle and light effects. The rendering component is built upon a graphics application programming interface (API) such as Direct3D or OpenGL providing abstraction to different video cards.

The physics engine, which is responsible of giving a realistic sense of the laws of physics. It simulates certain physical systems such as soft body dynamics, fluid dynamics and collusions. It handles how objects interact with their environment by simulating and controlling concepts such as gravity, acceleration, mass etc.

The audio engine, which is the component which consists of any programs related to sound. It is built upon libraries providing software abstraction to sound cards.

Game engines also provide abstraction to input devices such as keyboard or mouse by having the appropriate components build upon low-level libraries.

Game engine technology has evolved and provided a general abstraction that has made game engines very user-friendly. They have ‘outgrown’ their initial purpose, which is game developing and are used for a vast variety of applications. They are employed by fields such as medicine, training providing simulations, science and education offering interactive applications. The fields that have been greatly benefited are architecture and archaeology. Visualization of buildings, monuments and the environment are implemented through game engines giving not only a visual but also an interactive element to the applications. Nowadays virtual environment and game engines have interconnected, giving users accessibility to many fields and applications.[\[39\]](#)

2.9 Popular Game Engines

With the proliferation of game engines in the last years, the choice for the right game engine can be tricky. Many powerful game engines have been released to the public, giving the software framework and technological tools to developers to create their own projects. The most popular game engines are presented below.

2.9.1 Unreal Engine / Unreal Development Kit

Unreal Development Kit (UDK), is the free edition of Unreal Engine. Although primarily developed for first-person shooters, it has been successfully used in a variety of other genres. With its code written in C++, the Unreal Engine features a high degree of portability and is a tool used by many game developers today. UDK has support for iOS, Mac OS, Windows applications. It provides advanced lighting and shadowing and rendering systems including global illumination system. It offers the ability of scripting in C/C++ or UnrealScript, UDK’s own scripting language. [\[40\]](#)

2.9.2 Unity 3D

Unity 3D is a flexible and powerful development platform for creating multiplatform 3D games and interactive applications. Developed by Unity Technologies, it offers a software development kit (SDK) and an integrated development environment (IDE) for cross-platform applications. Developers have control over delivery to mobile devices, web browsers, desktops, and consoles. It offers support to more than 15 platforms, including Windows, OS X, Linux, Android, iOS. It includes Nvidia's PhysX physics engine, and Unity Web Player, which is a browser plugin that is supported in Windows and Mac OS X. Unity allows the use for a wide variety of texture mapping , dynamic shadows and includes Beast a global illumination system that offers realistic lighting in all applications. The game engine's scripting is developed on MonoDevelop, which is an IDE and combines the operation of a text editor with additional features for debugging and other project management tasks. Unity offers the ability to program in javascript, C#

and Boo. The user friendly environment and the vast platform compatibility has made Unity one of the most popular options among developers. [41]

2.9.3 CryEngine

CryEngine is a development solution for the creation of games and interactive applications. It offers support for Windows, Linux and game consoles. Tools are also provided within the software to facilitate scripting, animation, and object creation. Its scripting language is lua, but it also supports C++. CryEngine is known for its graphical capabilities. [42]

2.9.4 Comparison

Unreal Engine and Unity are arguably two of the most popular game engines available to the public today. CryEngine may have a high quality graphics output but its limited platform support and lack of a robust community for development help has excluded it from our consideration. Both the Unreal Engine and Unity are implementing great graphics and have large community support. Both of them have expanded in applications beyond gaming and many designers are using them both as a way for creating interactive experiences and architectural and environmental visualizations. However, Unity's ease of use and insuperable platform compatibility more than any other game engine, has made it the ideal game engine for our project.

Chapter 3 Technological Background

3.1 Unity 3D

3.1.1 What is Unity 3D?

Unity 3D is a powerful cross-platform 3D game engine and a user friendly development environment. Unity 3D helps developers to create games and applications for mobile, desktop, the web, and consoles. Its 3D environment is suitable for laying out levels, creating menus, doing animation, writing scripts, and organizing projects. Unity's primary goal may be the development of 3D video games, however, it is also suitable to create other kinds of interactive content, such as animations or 3D visualizations.

Unity is a fully integrated development engine that provides functionality to create interactive 3D content. With Unity the developer can assemble assets into scenes and environments, add lighting, audio, special effects, physics and animation, simultaneously play, test and edit the application, and when ready, publish to chosen platforms, such as Mac, PC and Linux desktop computers, Windows, the Web, iOS, Android, Windows Phone 8, Blackberry 10, Wii U, PS3 and Xbox 360. Unity's complete toolset, intuitive workspace and rapid, productive workflows help users to drastically reduce the time, effort and cost of making interactive content.

3.1.2 Project Structure in Unity 3D

Unity is defined by its component based architecture. Its workflow builds around the structure of components. Each component has its own specific job, and can generally accomplish its task or purpose without the help of any outside sources.

Each game or application created in Unity is called a **project**. Each project is consisted of one or more **scenes**. Scenes contain the objects of the game. They can be used to create a main menu, individual levels, and anything else. Every scene is considered as a unique level. In each scene, the user can place the 3D environment, position the 3D models and essentially designing the game in pieces. Every object in a scene is a **GameObject**. GameObjects are consisted of **components**. Components are the basic elements with which the user builds the project. GameObjects don't do anything on their own. They need special properties in order to add behavior and characteristics. These properties are attained through the components. The user can add a wide variety of components in a GameObject. Unity provides a large number of components, but the user can also create components using scripts.

3.1.3 Components

Some of the most commonly used components in Unity are presented next.

Transform Component

Every GameObject contains a Transform Component. When creating a GameObject a transform component is added automatically. It is impossible to create a GameObject without that component.

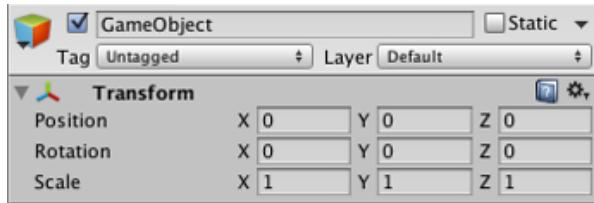


Figure 3.1 An empty GameObject with a transform component.

The Transform Component is one of the most important Components. It defines the GameObject's position, rotation, and scale in the game world. If a GameObject did not have a Transform Component, it would be nothing more than some information in the computer's memory. It effectively would not exist in the game world.

Asset Component

Assets are any resource the game uses. They are the models, materials, textures, sounds and all other "content" files that can exist in a game or application. Other than a few simple objects, Unity can't create most of these assets. Instead, they must be created externally using 3D modeling applications and painting tools and then imported into Unity. Unity's asset importing supports a wide variety of files. It accepts file formats from 3D modeling applications such as 3DS Max, Maya, Blender. Unity also supports all common image file formats, including PNG, JPEG, TIFF and even layered PSD files directly from Photoshop. When it comes to audio, Unity supports WAV and AIFF, ideal for sound effects, and MP3 and OGG for music.

Mesh Components

3D meshes are the main graphic object primitive of Unity. Various components exist in Unity to render meshes. The most commonly used components are the **mesh renderer** and the **mesh filter**. They are used in collaboration in order to display an object. The mesh filter takes a mesh from your assets and passes it to the mesh renderer for rendering on the screen. The mesh renderer takes the geometry from the mesh filter and renders it at the position defined by the object's transform component. When importing mesh assets, Unity automatically creates a mesh filter along with a mesh renderer. Another component is the text mesh. It generates 3D geometry that displays text strings.

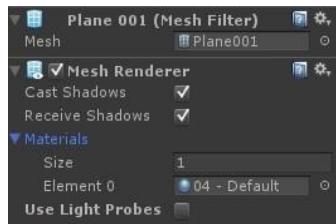


Figure 3.2 An example of a mesh filter and a mesh renderer component.

Physics Components

Physics components allow the user to give objects realistic motion and reaction to collisions. Unity has NVIDIA PhysX physics engine built-in. A physics engine is computer software that provides an approximate simulation of physical systems. This allows for unique realistic behavior and has many useful features. A rigidbody component makes the object that is attached to be affected by gravity or forces and collide with other objects. A character controller component is usually used when the developer wants to achieve a first person perspective. This component doesn't follow the rules of physics but instead it performs collision detection to make sure your characters can slide along walls, walk up and down stairs, etc. There is also a variety of collider components (mesh, box, sphere collider) which surround the shape of an object for the purposes of detecting physical collisions.

Rendering Components

These are the components that have to do with rendering in-game and user interface elements, as well as lighting and special effects. The camera component is essential as it is used to capture and display the world to the player. It can be customized and manipulated to fulfill the requirements of the user's application. The GUI Texture and GUI Text components are made especially for user interface elements, buttons, or decorations as well as displaying text on screen. Another important rendering component is the light component. It brings a sense of realism. Lights can be used to illuminate the scenes and objects, to simulate the sun, flashlights, or explosions just to name a few.

Audio Components

These components are used to implement sound.

Materials and Shaders

Materials and shaders are crucial components that are categorized in the asset component group. There is a close relationship between materials and shaders. Materials are used in conjunction with mesh renderers and other rendering components used in Unity. They play an essential part in defining how the object is displayed. The properties that a material's inspector displays are determined by the shader that the material uses. A shader is a specialized kind of graphical program that determines how texture and lighting information are combined to generate the pixels of the rendered object onscreen. In other words, it tells the graphics hardware how to render surfaces.

The user can select which shader each material will use. Specifically, a material defines which texture and color to use for rendering, whereas the shader defines the method to render an object.

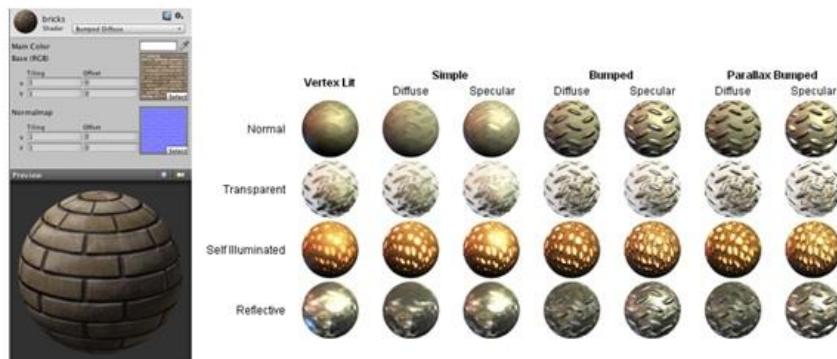


Figure 3.3 Left: a material component. **Right:** the built-in shaders that Unity provides.

3.1.4 Scripting

Scripting is an essential part of Unity as it defines the behavior of the game or application. Even the simplest game will need scripts to respond to input. Beyond that, scripts can be used to create graphical effects, control the physical behavior of objects or how characters and the player interact with the 3D environment. Unity supports three programming languages: C# (C sharp) which is similar to C++, UnityScript, a language designed specifically for use with Unity and modeled after JavaScript and Boo a .NET language with similar syntax to Python. The scripts are written and edited in MonoDevelop, which is an integrated development environment (IDE) supplied with Unity. An IDE combines the familiar operation of a text editor with additional features for debugging and other project management tasks.

Scripting is integrated with the component based architecture Unity uses. As it was mentioned above, the behavior of GameObjects is controlled by the components that are attached to them. Although Unity's built-in components can provide a wide variety of features, each developer needs to create his/her own functionalities by creating custom components using scripts. These allow the user to trigger events, modify existing component properties and respond to user input in any way.

Each script makes its connection with the internal workings of Unity by implementing a built-in class called `MonoBehaviour`. This class refers to the component that can be enabled or disabled. Javascript automatically derives from the class without the need to be declared, whereas the other two languages have to explicitly declare the class.

When a script is created, there are two functions automatically declared in it, `Start()` and `Update()`. `Start` is called when a script is enabled and it is called exactly once in the

lifetime of the script. The Start function is the place where any initialization happens. Update is the function that implements game behavior. It is called in every frame and is crucial for checking the state of various parts of the application. From a programming standpoint each game or application runs in loop, which allows it to run smoothly regardless of a user's input or lack thereof. For a change to occur an 'event' must be activated. The update function checks every frame for such events, which can be changes to position, state and behavior and determines the outcome. The start function is called by Unity before the Update function is called for the first time.

```
#pragma strict

function Start () {
}

function Update () {
}
```

There are other kinds of event functions that can trigger a change. There are the input event functions, that track the mouse movement and input. These functions allow a script to react to user actions with the mouse. Some examples of those functions are OnMouseDown, which is called when the user has pressed the mouse button, OnMouseUp which is called when the user has released the mouse button, OnMouseEnter which is called when the mouse enters a GUI element, etc.

Another important function is the OnGUI function. Unity has a system for rendering Graphical User Interface (GUI) controls over the main action in the scene and responding to clicks on these controls. The code handling those events is treated somewhat differently from the normal frame update and is placed in the OnGUI function, which is called multiple times per frame update.

Apart from the functions provided by Unity, the developer can create his/her own functions in order to control or determine the behavior of a GameObject, change the properties of a component or altering the overall state of the application. In order for these custom functions to be executed, they have to be called inside a Unity event function, like Update.

The most commonly used functions were presented briefly above, as well as the concept of how they are used. The basic notion of the Unity scripting is that the scripts are components that can control the GameObject. Each component property corresponds to a script variable and the scripts can access not only the components of the GameObjects they are attached to , but also other GameObjects.

Unity's scripting API is extensive. The basic ideas were briefly analyzed above. Further scripting details will be explained along with the implementation. For more information, the reader can visit the Unity documentation:
<http://docs.unity3d.com/Manual/ScriptingSection.html> or
<http://docs.unity3d.com/ScriptReference/index.html>.

3.1.5 Interface

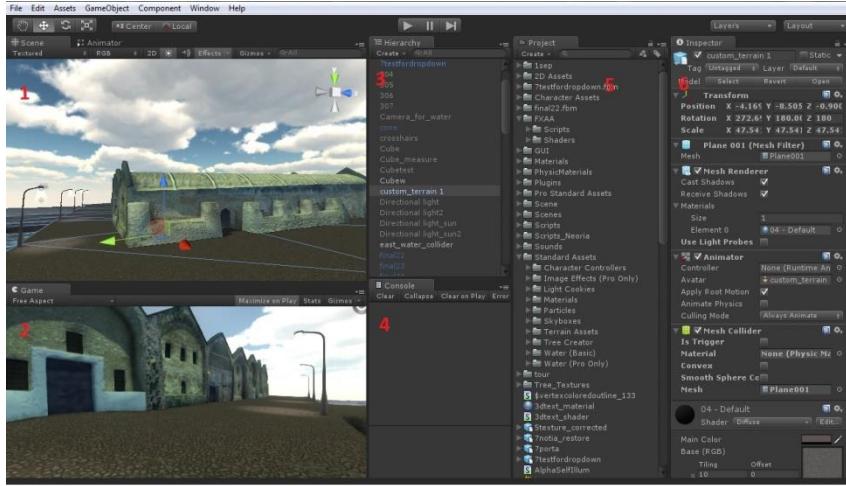


Figure 3.4 Unity's interface with a scene loaded.

Figure 3.4 displays Unity's interface. Each part is described below.

- 1 - The Scene View is the interactive viewport. In Scene View the user positions environments, the player, the camera, and all other GameObjects, as well as maneuvering and manipulating them.
- 2 - The Game View is rendered from the camera(s) in the application. It is representative of the final, published application and displays what the user sees.
- 3 - The Hierarchy contains every GameObject in the current Scene.
- 4 - The Console shows messages, warnings, and errors.
- 5 - The Project Browser contains the assets that belong to the project.
- 6 - The Inspector displays detailed information about your currently selected GameObject, including all attached Components and their properties. [43],[44]

3.2 Autodesk 3ds Max

Autodesk 3ds Max is a 3D computer graphics program for making 3D models, animations and images. It is developed and produced by Autodesk Media and Entertainment. It has modeling capabilities and a flexible plugin architecture. It is used not only by video game developers, but also architectural visualization studios and the movie industry. In addition to its modeling and animation tools, it also features shaders, normal map creation and rendering, a customizable user interface, and its own scripting language.

3ds Max's interface is shown in figure 3.5. The main viewport is displayed. On the top there is the main toolbar and the ribbon which contains many tools for modeling. On the

right there is the command panel which gives access to tools for creating and modifying geometry, adding lights, controlling animation, and adding complexity to geometry.

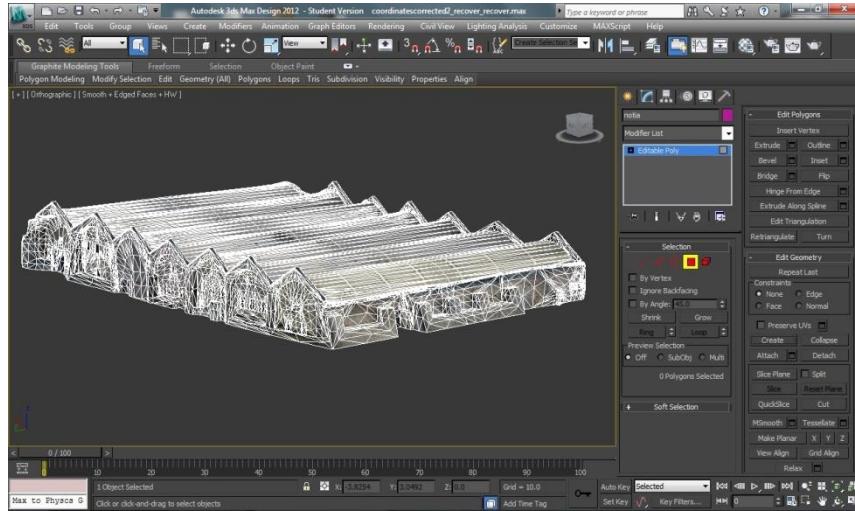


Figure 3.5 3ds Max interface.

The user can choose between two modeling techniques. Polygon modeling which is more common with graphic design than any other modeling technique as the very specific control over individual polygons allows for extreme optimization. Usually, the modeler begins with one of the 3ds max primitives (box, sphere, cylinder, cone, pyramid, plane) and using the provided tools adds detail and refines the model. The other technique is NURBS or non-uniform rational B-spline, which is a mathematical model for generating and representing curves and surfaces.

There is a variety of features for creating, editing, manipulating and refining a 3D model. There are also features for texture assignment. The texture workflow includes the ability to combine an unlimited number of textures and maps. 3ds max also provides tools for UV mapping. [45]

3.3 AutoCAD Civil 3D

AutoCAD Civil 3D is a civil engineering design software. It is used by civil engineers and topographers. It is a software that helps you organize and manage survey and geographical data as well as model earth surfaces, visualize geographical and spatial data and aid civil design.

Figure 3.6 shows the Civil 3D interface. On top there is the ribbon which is the primary user interface for accessing commands and features. On the left there is the toolspace survey tab in which the user can manage, organize and edit survey data. [46]

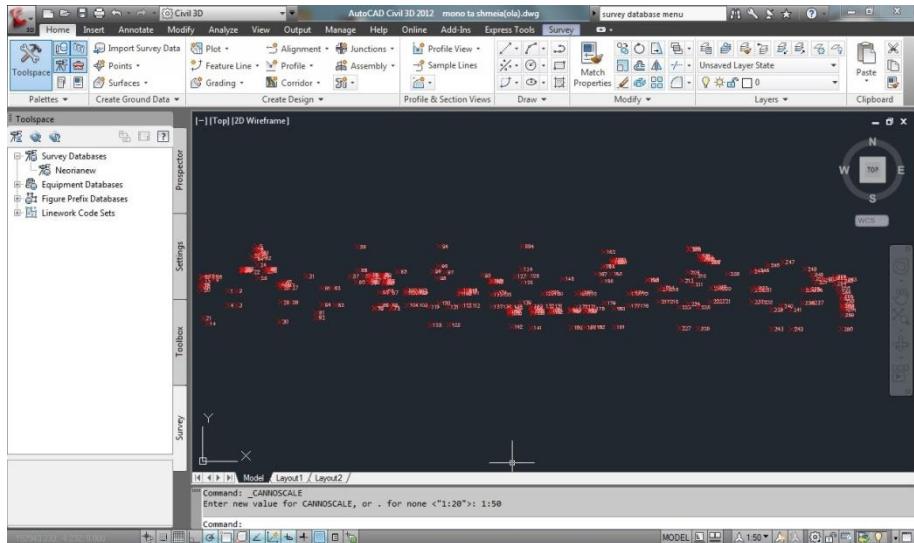


Figure 3.6 AutoCAD Civil 3D interface.

3.4 MeshLab

MeshLab is an open source, portable, and extensible system for the processing and editing of unstructured 3D triangular meshes. The system is aimed to help the processing of the typical not-so-small unstructured models arising in 3D scanning, providing a set of tools for editing, cleaning, healing, inspecting, rendering and converting this kind of meshes. [47]

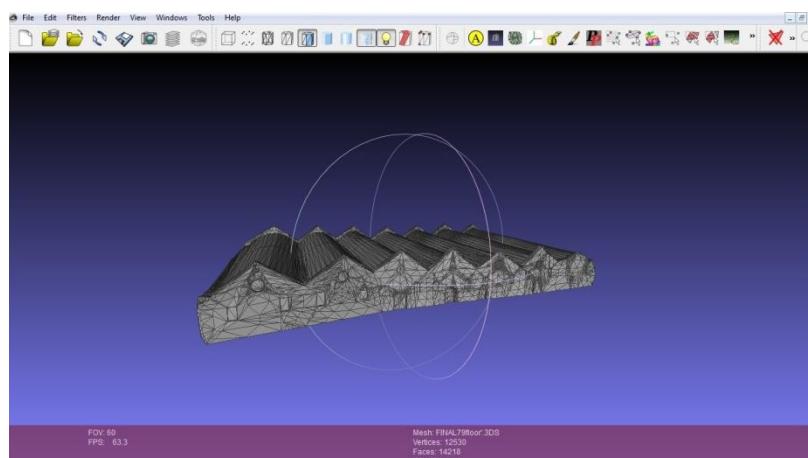


Figure 3.7 Meshlab Interface

Chapter 4 Monument Survey And Data Acquisition

4.1 Introduction

In this chapter we are presenting the object to be digitally reconstructed, 'Neoria', a massive historic building located at the old harbor of the old city of Chania, Greece. It is important to study the object in question beforehand, its history, structural details and architectural features in order to choose the appropriate method of acquiring the data. The current state of the monument should also be considered.

Next we are describing the monument survey. Our work on the field, in order to gather the spatial information we need, is presented. The equipment that we had in our disposal is described. Taking into consideration the previous facts, we demonstrate the detailed process of the digital measurement of the building using a total station.

Furthermore , the size, structure and damaged state of the building has influenced our decision of which features and damaged areas to measure as well as how dense or sparse our measurements throughout the monument should be. The final quantity of the data is something we take into account, because our goal is to gather the appropriate amount of information we need to digitally construct a geodetic mesh, without being a computually heavy task.

At last, we are depicting the raw data that we acquired and the initial process of converting them in our desired type, so that can be used in the extraction of the digital surface.

4.2 Venetian Neoria

Neoria is a beautiful venetian building located at the old harbor of the city of Chania. During the Venetian occupation (1204 - 1669), the need for the closer presence of Venetians in Crete made them construct a large number of "Neoria" (arsenali in Italian) in Chania, where the ships would be repaired during the winter.



Figure 4.1 Venetian Neoria at the old harbor of Chania.

In order to understand the original use of the arsenals, one must know that they were open to the sea, which would flow into them up to a certain point. They were interconnected by way of arched openings, while one could enter them via one of two gates; the main entrance was located roughly centrally. A part of this gate survives today, at the end of Daskalogianni Street. The construction of the first two "Neoria" in Chania was completed in 1526. In 1593, sixteen "Neoria" had already been constructed.

In 1599, the south "Neoria" complex was completed with the construction of the 17th "Neorio". In 1607, during the expansion of the northeast rampart, begins the construction of 5 more "Neoria" at the heart of the port to the east. Two of them were completed, however, only the walls to the arch of the third one were constructed. During the Turkish period, the lack of maintenance works in the port and the degradation of the role of "Neoria" also resulted in the alteration of the original function of "Neoria" which were now used as military storage spaces. From the initial "Neoria" complex with 17 "Neoria", nine were demolished.

Nowadays, a group of 7 continuous domes is preserved-along with another one further to the west, the "Grand Arsenal" (today, the Centre of Mediterranean Architecture). [48], [49]



Figure 4.2 The remaining Neoria. The seven continuous domes and the Grand Arsenal (image taken from Google maps).

Our focus will be centered on the seven consecutive domes, as they are in the most need of restoration. A close inspection reveals the deteriorating state of the structure. The monument stands a few meters from the water facing, at times, adverse climate conditions. Parts of the monument's stone walls have fallen, entire sections of walls are loose and certain edges of the building are partially or entirely destroyed. Significant structural elements are impaired and, as a result, Neoria is slowly losing its original beauty. Cracks are now appearing and restoration efforts had been limited or wrongly conducted. For instance, in order to repair cracks, craftsmen had often applied unsuitable modern materials such as cement. The outcome of such badly managed restorations was that at times, repairs had destroyed neighboring elements to the restoration areas rather than preserving them.



Figure 4.3 Damaged areas.



Figure 4.4 Inappropriately restored areas.

4.3 Monument Survey

4.3.1 Introduction

In order to produce accurate on site measurements, we use a geodetic total station. As it was presented in chapter 2, other methods of 3D data acquisition, such as photogrammetry and laser scanning produce high-resolution results with great detail, generating high-quality geometric meshes. However, these two methods require expensive equipment, are time-consuming and produce large amounts of data which have a high computational cost, and the post processing is quite complicated.

Having considered the volume and the structural state of the building, our data acquisition will be implemented with the use of a total station.

4.3.2 Geodetic Total Station

The raw data were acquired with the Nikon total station.



Figure 4.5 Total Station.

The total station measures angles and distances with precision from the instrument to points to be measured. To be more specific, the horizontal angle, the vertical angle and the slide distance are measured.

The horizontal angle is the angle on the horizontal plane that is determined by the total station , the point to be measured and a reference point. This is shown in the figure 4.7 where T is the total station , B is the reference point and A is the point to be measured.

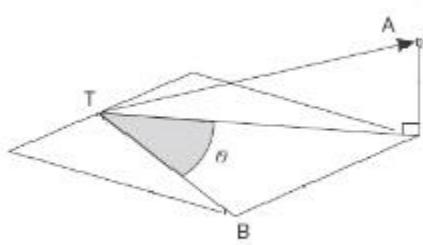


Figure 4.6 Horizontal angle θ .

The vertical angle is the angle between the point to be measured and the axis determined by an imaginary point directly above the position of the total station (zenith) and the ground. It is displayed in the figure 4.8

The slide distance is the distance from the total station to the point to be measured.

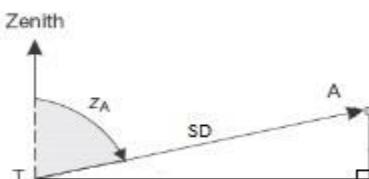


Figure 4.7 The vertical angle Z_A and the slide distance SD .

4.3.3 Field Work

It is important to study the building to be surveyed, as well as explore the surrounding area beforehand. We visited the area days before the measurements started, so that we could organize the way the measurements would take place.

An essential purpose of those early visits was to establish the positions from which we will gather our data. These positions, called 'stations' were selected to be around the monument so that we could gain all the spatial information we needed. The stations are the points where we set up the total station, in order to acquire our data.



Figure 4.8 The selected positions (stations) from which we took the measurements.

Taking into consideration, the volume and the state of the building, we initially decided to measure the contours of the seven domes. That would give us the basic outline of the building. So from every station we measured points on the roof outline and on the ground that were within its field of view. We also took measurements on the edges and corners of the doors and windows , so that we could determine their exact position and approximate their shape. An example of the basic measurements is shown below.

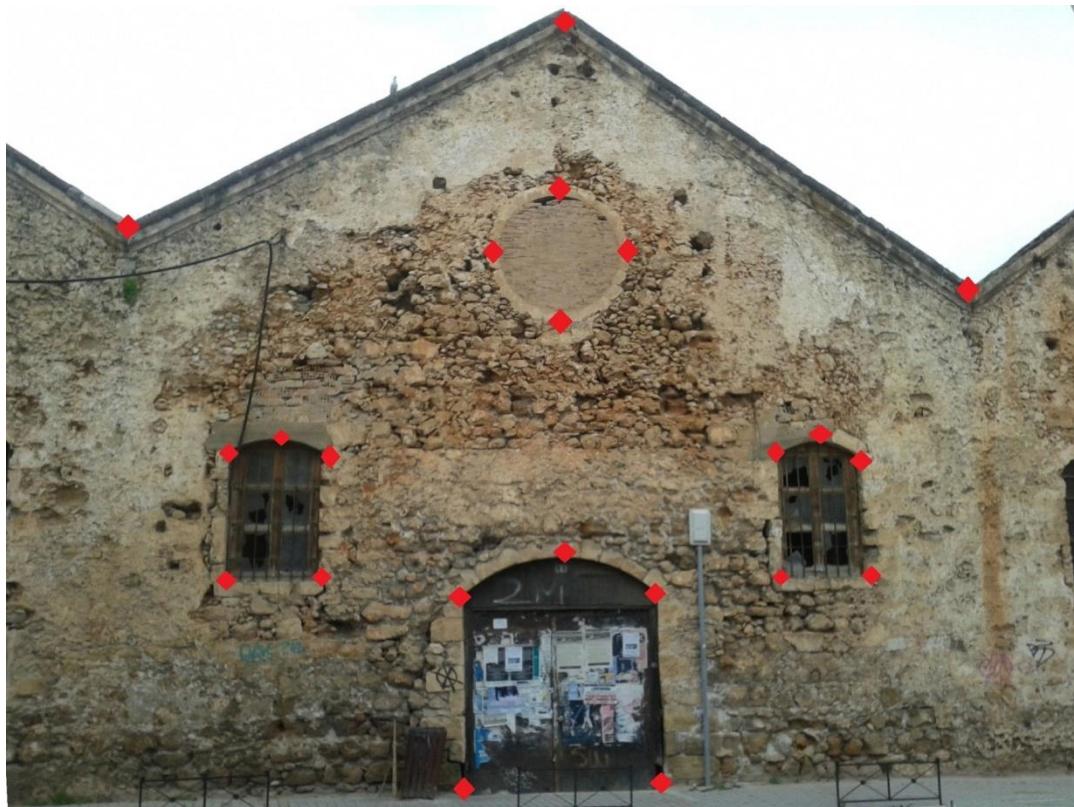


Figure 4.9 The measured points (in red) of a section at the south part of the monument.

Due to the extent of the erosion, it is impractical to document all the damages and the impaired areas. It can be seen clearly that there are whole walls and large surfaces that are hugely eroded and the detailed digital recording of those would be time consuming and complicated. We decided to document only the more serious damages , or the parts that the surface changes abnormally. Our interest was in getting information mainly on their outline (shape) and their depth.



Figure 4.10 An example of the measurements we acquired of some of the damages. We focused on measuring their outline and their depth.

The initial part of our field work lasted several days and we measured about 550 points from 10 different stations. One of the advantages of the total station is that you can easily later acquire additional points or take further measurements. Our framework provides the option to add data to the project. In our case further data were acquired at a later time. We measured 340 additional points, mostly on the west side.

During the field work we created the field sketch. We drew the basic outline and features of the monument in parts, marking on paper all the measured points. We gave each point an index in ascending order the same that was given at the digital measurement. The field sketch is invaluable, because it helps us distinguish the points and find easily their position on the building. It was very useful during the 3d modeling process, it provided consult for element position and error checking , as well as an easy interpretation of the measurements for a third party.

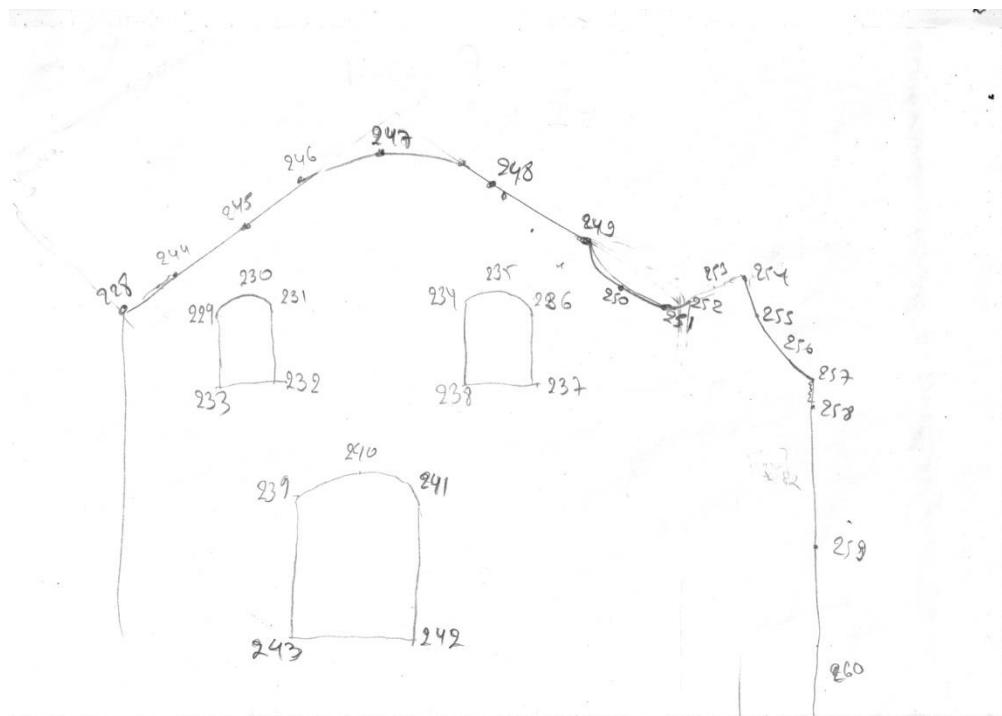


Figure 4.11 The sketch of the south side of the seventh dome. The measured points are marked with their respective numbers.

In conclusion, surveying with a total station provides a fast and relatively easy method of acquiring data, as well as giving the opportunity to stop the process at any time, and resuming it at a later time without complications.

4.4 Geodetic Data

The final results are in the form of Greek Geodetic Reference System '87 (GGRS87) coordinates. The GGRS87 is a geodetic system commonly used in Greece. It is considered a projection system [50].

Each measured point is depicted with 3 coordinates. The x coordinate is the easting which represents the eastward distance, the y coordinate represents the northing which refers to the northward distance and the z coordinate refers to height above the sea level [51]. All the coordinates are measured in meters.

```
1 1;501739.4804;3930190.498;8.487771258
2 2;501740.5993;3930190.605;8.492490225
3 3;501740.5836;3930190.649;6.634337957
4 4;501739.4669;3930190.495;6.617794298
5 5;501743.2364;3930190.679;14.29157799
6 6;501742.9629;3930190.723;13.97565644
7 7;501743.2483;3930190.936;13.79708272
8 8;501743.3665;3930190.948;13.87485332
9 9;501737.5716;3930190.077;9.962964551
10 10;501737.1727;3930190.24;10.32874388
11 11;501737.664;3930190.135;10.30618266
12 12;501736.7092;3930190.175;10.462952
13 13;501736.2236;3930191.143;10.37430159
14 14;501736.0123;3930191.248;10.27063506
15 15;501736.1294;3930190.43;9.677746177
16 16;501736.7847;3930190.13;4.255473642
17 17;501736.0242;3930190.544;9.266847961
18 18;501736.0814;3930191.117;9.183605279
19 19;501735.811;3930191.16;8.868774947
20 20;501736.0387;3930190.215;8.668279167
21 21;501736.2523;3930190.646;4.969751865
22 22;501742.4813;3930190.8;10.98394692
23 23;501744.2278;3930191.179;11.04766864
24 24;501743.3034;3930190.884;11.88349578
25 25;501743.4099;3930190.942;10.15484894
26 26;501746.3396;3930191.373;8.917457478
27 27;501747.4609;3930191.533;8.927301158
28 28;501747.4686;3930191.57;7.015345416
29 29;501746.3333;3930191.394;7.0125916
```

**Figure 4.12 A sample set of the final GGRS87 coordinates
(id;easting;northing;height).**

Chapter 5 3D Reconstruction

In this chapter the 3D Reconstruction process is described. We present the challenging part of transforming the 3D data into surface as well as processing the surface to form the final 3D model. The 3D modeling process is a collection of steps that lead to the creation of the 3D object suitable to be imported in the game engine. We present the surface creation, the post processing, the texturing, the materials and the shaders used, all of which converge to create a realistic result.

5.1 Surface Creation

5.1.1 Data evaluation

The first step to create surface is to evaluate the spatial data. Having gathered together all the spatial data in their final form, they are displayed in three dimensional space in order to acknowledge their properties.

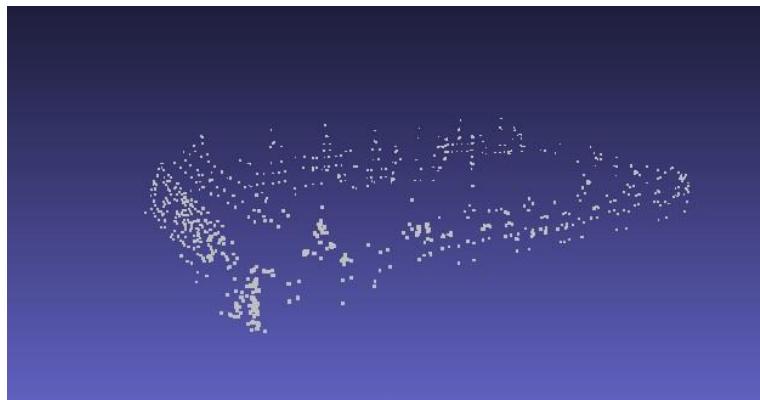


Figure 5.1 The points acquired displayed in 3D space.

We are going to refer to the four walls of the building as the north side, the south side, the west side and the east side. These four walls are depicted with less than 1000 points. This number is extremely small considering the millions of points that laser scanning or photogrammetry would have acquired for a building this size. Another observation that plays a huge role in surface reconstruction is the varying density of the sample points. Dense sets of points in a small area correspond to detailed parts of the building and blank spaces correspond to featureless parts. Our data acquisition method and our measurement process have created a distinctive collection of points that must be taken under consideration for the surface creation process.

5.1.2 Study of the known surface reconstruction algorithms

The next step in the process is to find an appropriate and correct way to connect the 3D points in order to create a 3D mesh. One of the most difficult problems in surface reconstruction is understanding how to connect the points in order to form the surface that has the same topological characteristics as the real one.

The usual input for the surface reconstruction algorithms are point clouds. Point clouds are commonly created by laser scanners and photogrammetry and are considered a dense collection of a huge amount of 3D points. Laser scanners can produce up to millions of points with a high and usually homogenous density. The quantity of information depicted with them is huge and can be directly inspected. The data properties in this case are suitable for the implementation of surface algorithms. Our set of data is far from the homogeneousness of the usual point cloud and it is easy to assume that there will be problems regarding its suitability for the algorithms.

As it was described in chapter 2 the known surface reconstruction algorithms are categorized in algorithms employing implicit functions and computational geometry algorithms. The basic difference of these categories is that in the case of implicit function algorithms the output surface approximates the original and it may not include the input data, in contrast with the computational geometry algorithms which are forming the surface interpolating the data points. In our project the interpolation of our measurements is crucial, because they represent geographical coordinates and the model needs to be georeferenced. Therefore the measured points are required to be on the final mesh so that they can act as reference points in our further implementation. For this reason it is concluded that the implicit function algorithms are unsuitable for our case. Furthermore, and considering the two popular algorithms of this category, the Hoppe's algorithm and the Poisson algorithm, it should be noted that both require very dense point clouds and additional information apart from the spatial information given by the data. Hoppe's in particular requires knowledge of the sampling process and the Poisson algorithm requires consistently oriented surface normals at the input points.

Having ruled out the implicit functions algorithms, we focus our research on the computational geometry algorithms. They seem suitable for our case because they create geometric structures based on the input points. The usual outcome of those algorithms is a triangle mesh. We focus our interest on the powercrust and the ball pivoting algorithm.

The powercrust algorithm uses Voronoi diagrams and subsets of weighted poles that lie inside and outside of the object. These sets of poles, like the Voronoi diagram, divide the space into polyhedral cells. The boundary of the union of the interior cells forms the polygonal output surface, or the Power Crust. There is a free implementation of the algorithm provided by its authors (<https://code.google.com/p/powercrust/>). We tested the algorithm with our data and it could not return results. Specifically, it could not compute the required poles. This is due to the sparsity of our data. This algorithm to have efficient results requires an adequate sampling density. According to its Principal

researcher Nina Amenta “the Power Crust algorithm should be quite useful for automatically building object models from laser range data.” [52]

The ball pivoting algorithm creates a triangle mesh with the given input points. Its principle is very simple: Three points form a triangle if a ball of a user-specified radius touches them without containing any other point. (Bernardini et al., 1999) [29]. This algorithm is time efficient because it doesn’t need to compute triangulations or Voronoi Diagram to form a mess. However, it is not suitable for point clouds with varying point density, as it will output the surface with holes when the sample points are not dense enough. The ball pivoting algorithm can be implemented via the Meshlab software. It was applied in a subset of our data set, the and it produced the expected result considering the input.

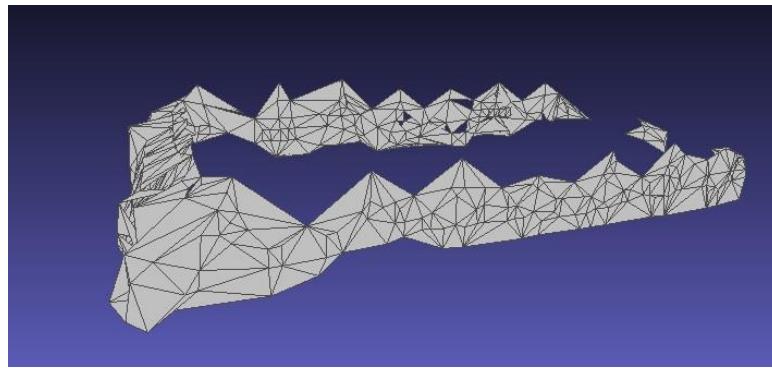


Figure 5.2 Implementation of the ball pivoting algorithm. Holes and discontinuities are shattering the mesh.

As it is clearly seen, the results are not sufficient. There are holes and the discontinuities are splitting the mesh. Spaces that are larger than the radius are not filled. The mesh is incoherent and some information has disappeared.

In conclusion, the previous algorithms are not suitable in our case. The quantity and type of our data must be taken under consideration in order to have good results.

5.1.3 TIN Surfaces and Delaunay Triangulation

As it was previously explained, given the input data, the implementation of algorithms did not return satisfying results. The data can have quite different properties that must be considered in the solution of the surface reconstruction problem. Thus, the characteristics of our data set have to be inspected, in order to determine the best way to form a surface.

In addition, we must not forget the process of our measurements. We measured points that form the outline of the building and the areas of interests that represent features, such as doors, windows and damages. Featureless parts of the building were not measured. This leads us to the acknowledgment that parts of the building that contain information that must be depicted, are represented by groups of points in close

proximity. These points must be connected with each other and not be interfered with other edges, in order to shape the topology. The isolated points that were measured, e.g. the tops of the roofs, should connect with the closest points which are the edges of the roofs. These realizations are making us lean towards a nearest neighbor based technique.

Furthermore, the data are geographic coordinates originating from a total station, which has lead us to examine ways of representing geographic data. Our focus was aimed at the Triangular irregular networks. Triangular irregular networks (TIN) have been used by the geographic information system (GIS) community for many years and are a digital means to represent surface morphology. A TIN is a representation of the physical land surface, made up of irregularly distributed nodes with three-dimensional coordinates (x , y , and z) and lines that are arranged in a network of non overlapping triangles. The nodes (vertices) are connected by lines (edges) to form the triangulation. TINs are usually associated with three-dimensional data (x , y , and z) and topography which fits the characteristics of our data. The triangles in those networks are formed with the Delaunay triangulation.[\[53\]](#)

In consideration of the previous points, we have decided to represent our data as TIN surfaces. Therefore the Delaunay triangulation is going to be used as the surface reconstruction method.

The Delaunay triangulation is a nearest neighbor based technique, something that makes it suitable for our data. A Delaunay triangulation for a set P of points in a plane is a triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$. This entails the "Delaunay condition", which means that no vertex lies within the interior of any of the circumcircles of the triangles in the network. If the Delaunay condition is satisfied, the minimum interior angle of all triangles is maximized. The result is that long, thin triangles are avoided as much as possible. In other words, given a set of data points, Delaunay triangulation produces a set of lines connecting each point to its natural neighbors.[\[54\]](#)

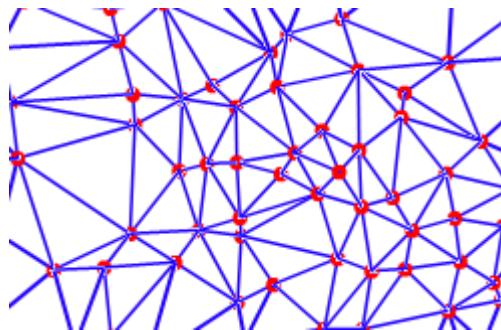


Figure 5.3 A part of a TIN surface created with Delaunay triangulation.

In conclusion, TIN surfaces created by Delaunay triangulation are most useful for mapping highly variable surfaces with irregularly distributed sample data, something

that identifies with our data's inconsistent density. The input data used to create a TIN remain in the same position as the nodes or edges in the TIN. This allows a TIN to preserve all the precision of the input data while simultaneously modeling the values between known points. That property of this methodology fulfills not only the condition of preserving the spatial positions of our geographical data, but also means that the three dimensional values of any point in the surface are defined by interpolating the values of the vertices of the triangles that the point lies in. In addition, TINs can have a higher resolution in areas where a surface is highly variable or where more detail is desired and a lower resolution in areas that are less variable. [55]

All the above make the adopting of the methodology of the TIN surfaces created by Delaunay triangulation ideal.

5.1.4 Reconstruction of TIN surfaces

Having decided the best approach to reconstruct the data, we started looking for the appropriate software to import them in order to organize and manage the data volume. AutoCAD has developed Civil3D, directed mainly at civil engineers and topographers. It is great for managing GIS data as well as any kind of geodetic data. It works with multiple coordinate systems and has tools for creating and organizing survey databases. It is ideal for viewing the spatial data in 3D space along with imported information about each point, something that few other CAD programs do. Spatial information can be processed and used in creating surfaces, design civil projects and document topographical surveys.

Due to the scarcity of information it is important to utilize the documentation we created during our field work, in order to identify every point measured and where it corresponds in the real surface. Modeling software, like 3ds Max or Maya does not have that kind of feature, nor they can import and display simple 3D points. AutoCAD Civil3D provides that functionality, by displaying survey data and coordinates along with identification information. It can import spatial information from simple text files.

Unfortunately, that kind of software which recognizes spatial data and constructs TIN surfaces, acknowledges data as Digital Terrain Models (DTM), which are topographic model of the bare earth. That means that it only constructs surfaces 'looking from above'. It recognizes easting as the x axis, northing as the y axis and elevation as the z axis.

After all, TIN surfaces represent earth surfaces.

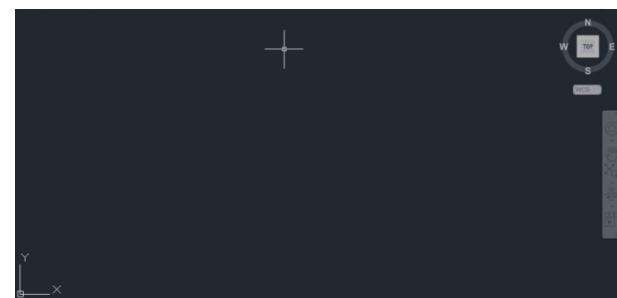


Figure 5.4 Civil3D's viewport. It processes the data with a 'top' view.

In order to visualize and process our data, we need to treat them as earth surfaces and alter them accordingly. We separated our data in four groups, according to which side of

the building they belong. So, all the data acquired from the north side of the building belong to one group, the data acquired from the south side to another group, another group consists of the data of the west side and the last one of the east side data. Now we have four groups of data each one representing a side of the building. In order to visualize them we need to make the software perceive them as earth terrain, i.e. geographical information representing earth ground. Therefore, we have to rotate each group in the three dimensional space accordingly.

As it was mentioned above the software perceives as the x axis the easting coordinate, i.e. the west-east information, as the y axis the northing coordinate, i.e. the north-south information and as the z axis the elevation (height) information, processing the spatial information with a ‘top’ view, as it is shown in figure 5.4. We need to process each group, in order to convert them to this form.

South group data

The data belonging to that group are facing south. In order to have a top view, they have to be rotated 90 degrees around the x – axis (Figure 5.5).

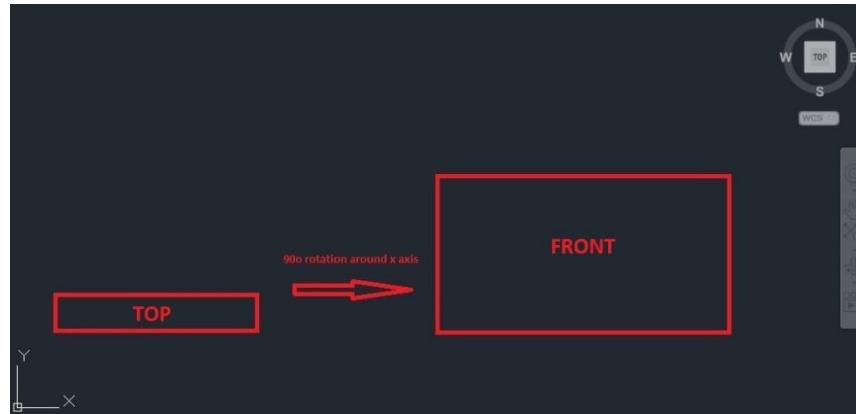


Figure 5.5 Rotation of south group data.

Each point is represented with x, y, z values. We use rotation matrices to calculate our data. The rotation matrices for rotation around the x axis, y axis and z axis are shown in the right.

$$\mathbf{A}_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & \sin \phi \\ 0 & -\sin \phi & \cos \phi \end{bmatrix}$$

$$\mathbf{A}_Y = \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix}$$

$$\mathbf{A}_Z = \begin{bmatrix} \cos \psi & \sin \psi & 0 \\ -\sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For the south side we have:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{aligned} x' &= x \\ y' &= y \cos \theta + z \sin \theta \\ z' &= -y \sin \theta + z \cos \theta \end{aligned} \xrightarrow{\theta=90^\circ} \begin{aligned} x' &= x \\ y' &= z \\ z' &= -y \end{aligned}$$

That means that we are going to import the x values as easting, the y values as negative elevation and the z values as northing.

North group data

Those data have to be rotated first 270 degrees around the x axis and then 180 degrees around the z axis.



Figure 5.6 Rotation stages of north data.

Rotation 270° around x axis:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{aligned} x' &= x \\ y' &= y \cos \theta + z \sin \theta \xrightarrow{\theta=270^\circ} y' &= -z \\ z' &= -y \sin \theta + z \cos \theta & z' &= y \end{aligned}$$

Rotation of those results 180° around z axis:

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \Rightarrow \begin{aligned} x'' &= x' \cos \theta + y' \sin \theta \xrightarrow{\theta=180^\circ} x'' &= -x' \\ y'' &= -x' \sin \theta + y' \cos \theta & y'' &= -y' \\ z'' &= z' & z'' &= z' \end{aligned}$$

$$\begin{aligned} x'' &= -x' & x'' &= -x \\ y'' &= -y' \Rightarrow y'' &= z \\ z'' &= z' & z'' &= y \end{aligned}$$

That means that we are going to import the x values as negative easting, the y values as elevation and the z values as northing.

West group data

We rotated those data first 270° around z axis and then 90° around x axis.



Figure 5.7 Rotation stages of west group data.

Rotation 270° around z axis:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{aligned} x' &= x \cos \theta + y \sin \theta & \theta = 270^\circ & x' = -y \\ y' &= -x \sin \theta + y \cos \theta & & y' = x \\ z' &= z & & z' = z \end{aligned}$$

Consecutive rotation 90° around x axis:

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \Rightarrow \begin{aligned} x'' &= x' & x'' &= x' \\ y'' &= y' \cos \theta + z' \sin \theta & \theta = 90^\circ & y'' = z' \\ z'' &= -y' \sin \theta + z' \cos \theta & & z'' = -y' \end{aligned}$$

$$\begin{aligned} x'' &= x' & x'' &= -y \\ y'' &= z' & \Rightarrow y'' &= z \\ z'' &= -y' & z'' &= -x \end{aligned}$$

That means that we are going to import the x values as negative level, the y values as negative easting and the z values as northing.

East group data

We rotated those data first 90° around z axis and then 90° around x axis.



Figure 5.8 Rotation stages of east group data.

Rotation 90° around z axis:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{aligned} x' &= x \cos \theta + y \sin \theta & \theta = 90^\circ & x' = y \\ y' &= -x \sin \theta + y \cos \theta & & y' = -x \\ z' &= z & & z' = z \end{aligned}$$

Consecutive rotation 90° around x axis:

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \Rightarrow \begin{aligned} x'' &= x' & x'' &= x' \\ y'' &= y' \cos \theta + z' \sin \theta & \theta = 90^\circ & y'' = z' \\ z'' &= -y' \sin \theta + z' \cos \theta & & z'' = -y' \\ &&x'' = x' & x'' = y \\ &&y'' = z' & \Rightarrow y'' = z \\ &&z'' = -y' & z'' = x \end{aligned}$$

That means that we are going to import the x values as level, the y values as easting and the z values as northing.

After doing the necessary changes, we stored our data in .csv (comma separated values) files. Then we imported the data into the Civil3D software. Civil3D can process geodetic data in any coordinate system. The data were imported as GGRS87 coordinates. The software also gives us the opportunity to create and store survey databases which can hold the raw data.

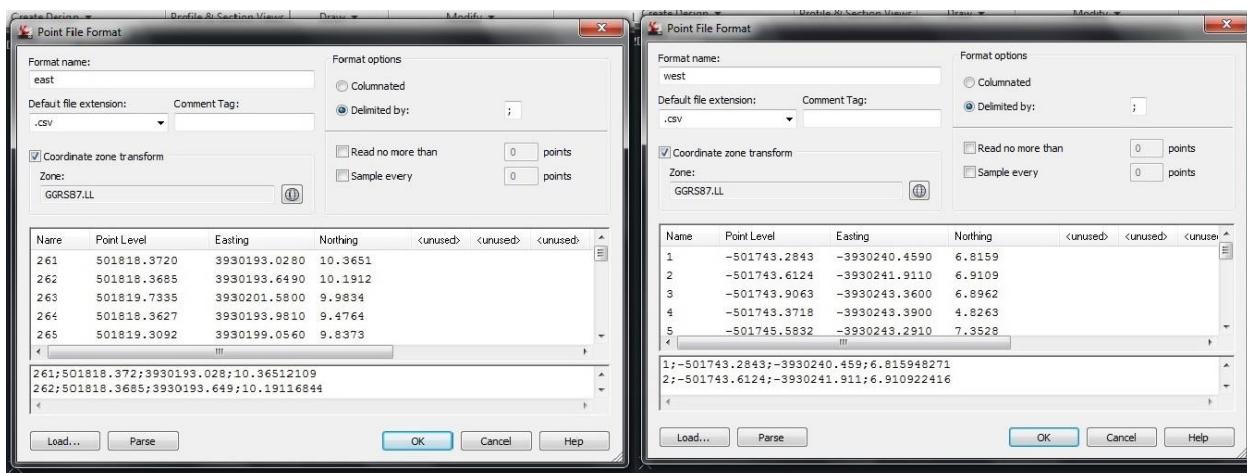


Figure 5.9 Import of csv files for east and west data.

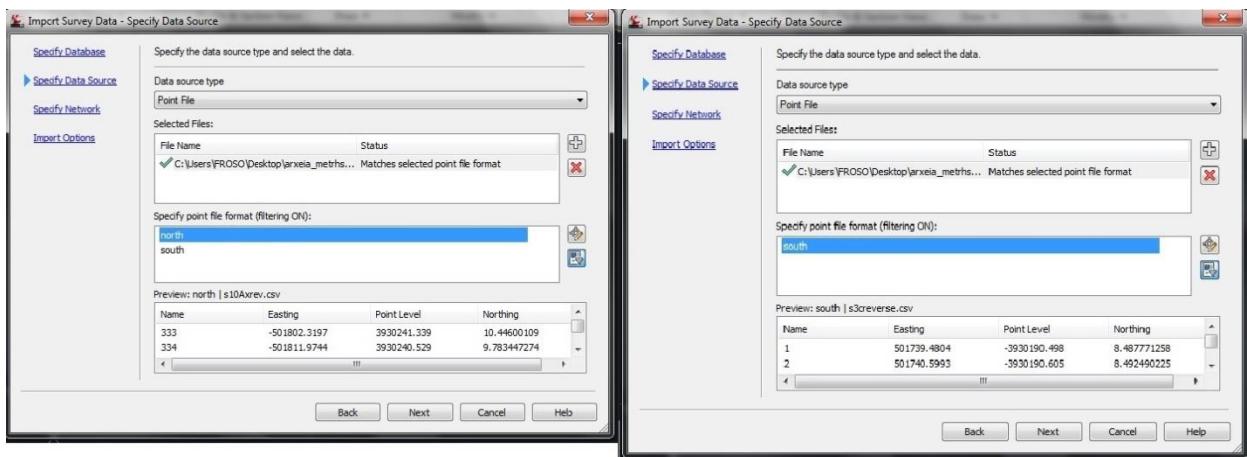


Figure 5.10 Insertion of the data files in survey database.

Now we can visualize our data as 3D points in space.



Figure 5.12 Above the measured 3D points of the north side. Below the measured 3D points of the east side.

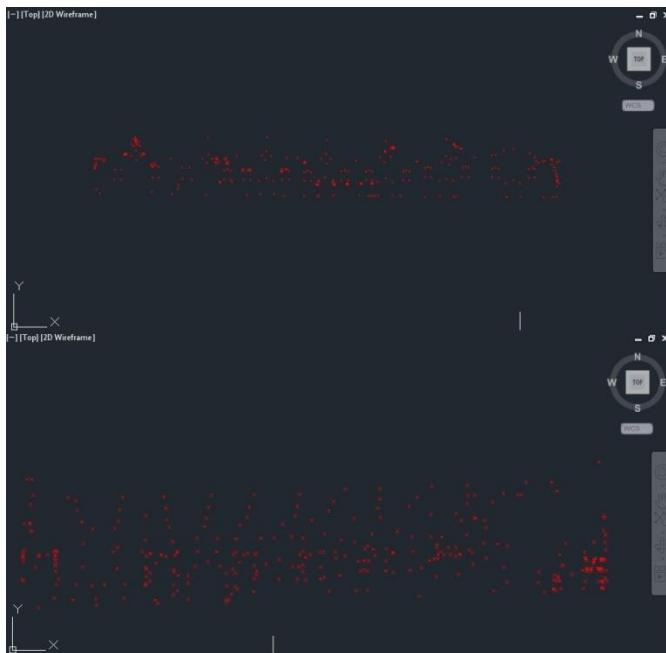


Figure 5.11 Above the measured 3D points of the south side. Below the measured 3D points of the west side.

Before we triangulate the point groups, we have to connect the outer points so that we can create boundaries. These boundaries, also called breaklines, give an outer limit to

the triangulation and define the TIN surface. Civil 3D has a feature for creating a TIN surface with Delaunay triangulation. The user can select which point group to triangulate and define boundaries. We triangulated each point group, after defining the outer boundaries by connecting the outer points. The resulting surfaces are shown below (the green lines define the outer boundaries).

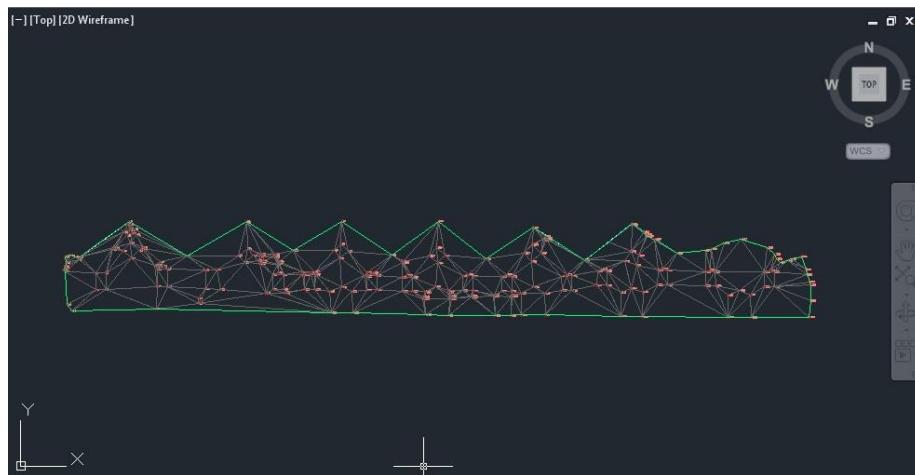
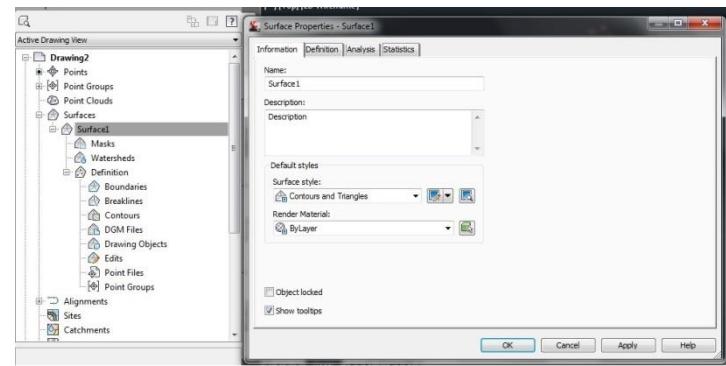


Figure 5.13 The triangulated surface of the south side.

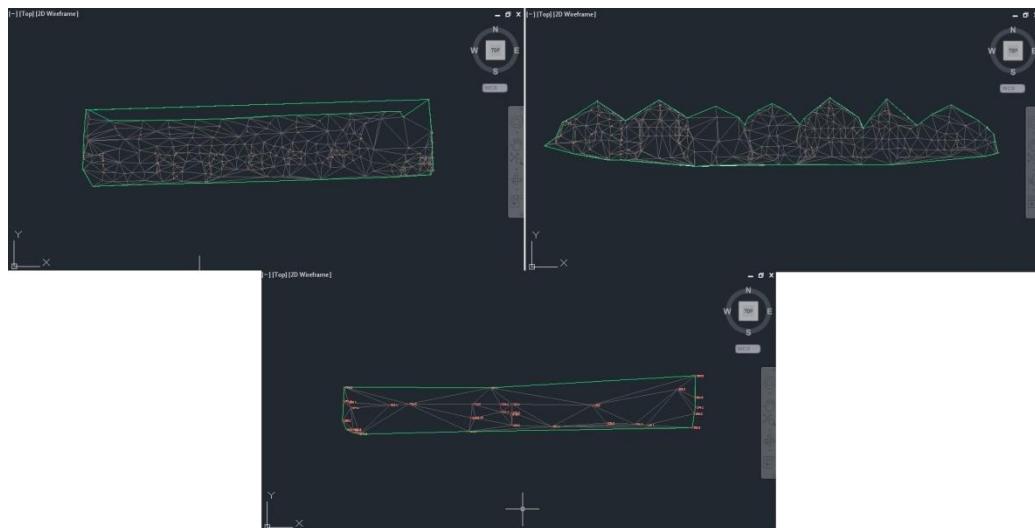


Figure 5.14 The triangulated surfaces of (clockwise from top left) the west, north and east sides.

Having triangulated the 4 surfaces, it can be observed that only the basic information is visible. There is the basic outline, window and door details and sparse surface information. At this point the field sketch we created during the measurements, becomes important because it is the means that will help us distinguish the points and give further shape to the surfaces. The next step is to import those surfaces to a modeling software program in order to process them further and extract more information and detail.

5.1.5 Processing of the TIN surfaces

Autodesk provides a streamlined method for transferring data from Civil3D to 3ds Max. It is the Civil View feature set and runs inside 3ds Max. It imports files or geometry data from Civil3D and can help enhancing a Civil3D project with 3D content.

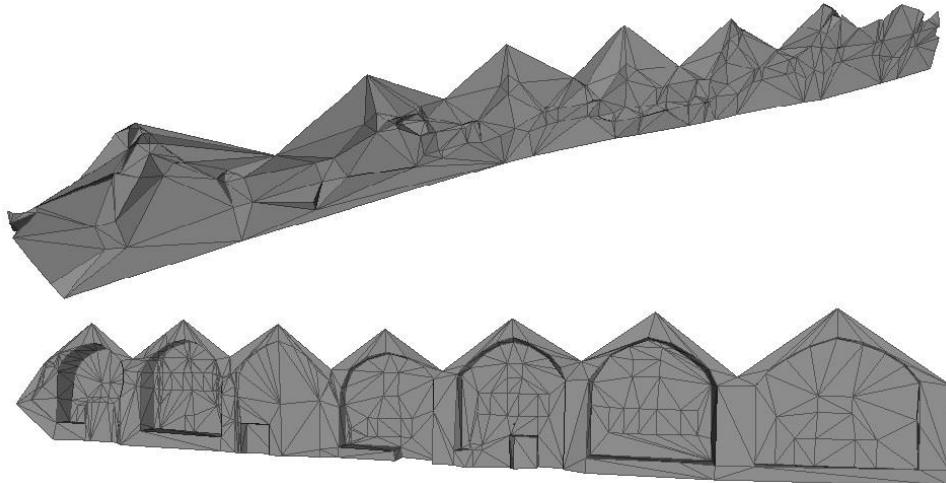


Figure 5.15 On top: the south side as it was imported from Civil3D (i.e. the TIN surface). On bottom: the north surface after some process.

Using the Civil View feature we imported the 3D geometry we created in 3ds Max. At this point and with the help of our field sketch, we processed the TIN surfaces by hand in order to correct mistakes, distinguish features and add detail where it was needed in order to make the surface morphology recognizable. The south surface did not need any particular changes the features were recognizable from the triangulation. In figure 5.15 the south surface is shown exactly as it was imported from the Civil3D software. The north surface needed some corrections. There were visible mistakes in measurements, some wrongly measured points and we needed to clearly distinguish the depth of the arcs.

The most time consuming surface at this step was the west side. Due to the complexity of the surface and the many protruding features a lot of points were measured, resulting in a complex TIN surface. We corrected the morphology by keeping the points steady and deleting the unnecessary triangles and connections. Some of the stages of the west side correction are shown below.

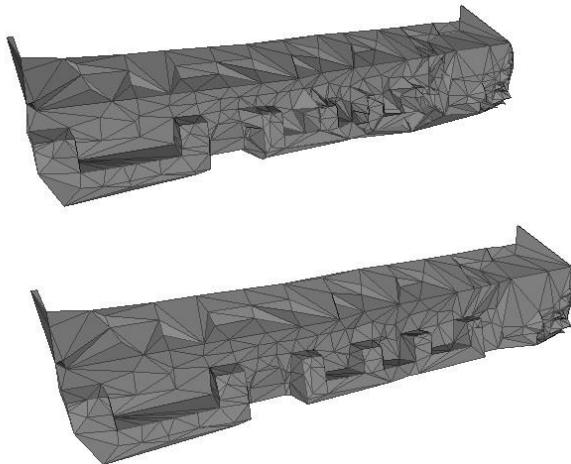


Figure 5.16 The stages of the west side correction.

We use these surfaces as base surfaces. Further processing is required in order to approximate the real surface. The first step is to add depth in the windows and doors. The depth was measured on the field, wherever it was possible. Due to the height of the building it was impossible in some cases to take accurate measurements, so we assumed an approximating value, taking in consideration and for comparison the neighboring parts. The depth was added with the extrude feature, in specified units. Having as a reference point a measured distance and knowing its respective value in units in the modeled surface, we calculated the units of each depth based on that analogy. The surfaces were ‘tweaked’ were they were needed, in order to correct small discontinuities or mistakes that are visible in comparison with the real surface.

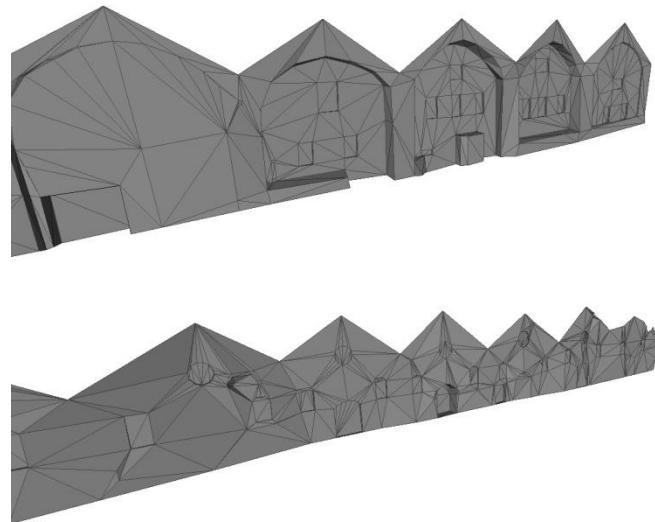


Figure 5.17 Depth details in doors and windows.

Each surface was rotated, aligned and attached accordingly in order to form the shape of the building.

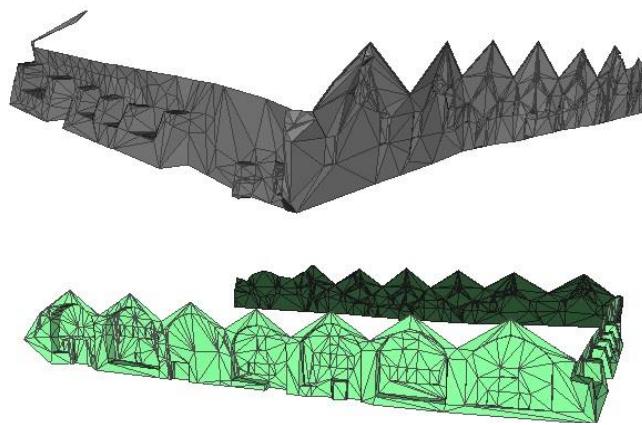


Figure 5.18 Stages of attachment. Each surface was attached accordingly to form the final shape of the building.

One by one the surfaces were positioned in their correct spatial places and the shape of the final model begins to form. Up at this point we have not mentioned the reconstruction of the roof. In our initial research, before we began our project, we were unable to find documented architectural information about the roof. Furthermore, during our measurements it was impossible to access the roof. Due to the limited space around the monument, the laser beam of the total station could not reach the roof. Therefore we created the roof based on satellite photographs (via Google Maps), and

following the pattern of the peripheral parts. The roof consists of seven consecutive long domes. It was created in 3ds Max and was attached to the model.

At this point, a basic shape of the of the 3D model of the monument including details of its structure was completed. We have a basic graphic representation of the building but it still needs further refinement.

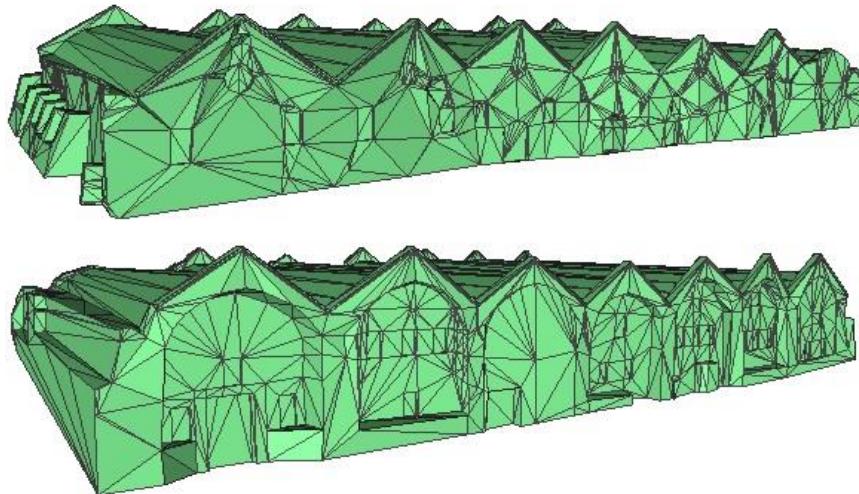


Figure 5.19 The final form of the 3D model.

5.2 Surface Processing

We have acquired a triangle mesh. It can be observed that while it accurately represents the monument, it is still coarse and sharp. It has the appearance of artificiality and lacks realism. As a consequence further refinement is required. In order to integrate further detail, and have a more 'rounded' result, the surface was subdivided. As it was mentioned in the chapter 2, surface subdivision is a method of representing a smooth surface via the specification of a coarser polygonal mesh. The subdivision creates new vertices and new faces and their position is calculated based on the positions of nearby old vertices. The aim of this technique is the creation of a smooth and nicely rounded model, giving it a more realistic effect.

The two categories of subdivision schemes that were mentioned, were examined. The first one, the approximating subdivision schemes, in which the newly generated points are not included in the initial surfaces were dismissed. As it was stated during the surface reconstruction stage, it is significant to maintain the original position of the vertices so that we will have an accurate geographical reference. Consequently, an interpolating subdivision scheme should be deployed, in which the points of the original mesh are interpolated to form the resulting surface.

Taking into consideration the shape and quality of the surface, it can be seen that some segments of the monument included insufficient spatial information, there were sparse blank spaces, while others were represented by dense groups of points. The model is also quite sharp and coarse. While approximating schemes are known to perform well on highly irregular surfaces such as ours, interpolating schemes are considered to be sensitive to irregularities and sharp features, deforming the surface and negatively affecting the geometric fidelity of the model [37]. Without wanting to quit the interpolating route, we decided to apply a simple midpoint subdivision to the model. Every edge was split on its midpoint and connected to its four neighboring edge-midpoints [56]. This step was implemented in Meshlab. A finer mesh with no distortion was acquired at the end of this process.

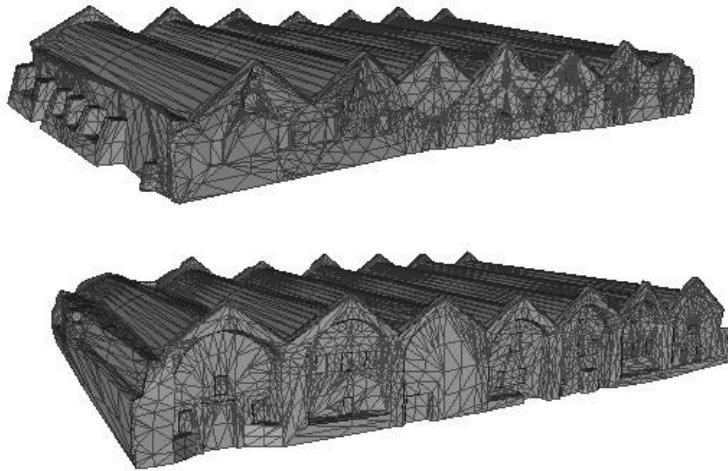


Figure 5.20 The 3D model after implementing midpoint subdivision.

To further approximate the real-world surface, the subdivided mesh was more refined in parts where it was needed. The problematic areas such as the circular windows and doors were initially brought into focus. To accomplish a realistic circular window, the edges and faces of the windows were further subdivided by applying a Meshsmooth modifier in 3ds Max, which subdivides the geometry while interpolating the angles of new faces at corners and edges. The effect it accomplishes is to round over corners and edges as if they had been filed or planed smooth. This process resulted in the desired geometric shape around the problematic areas. The triangle mesh that represents our 3D model is completed.

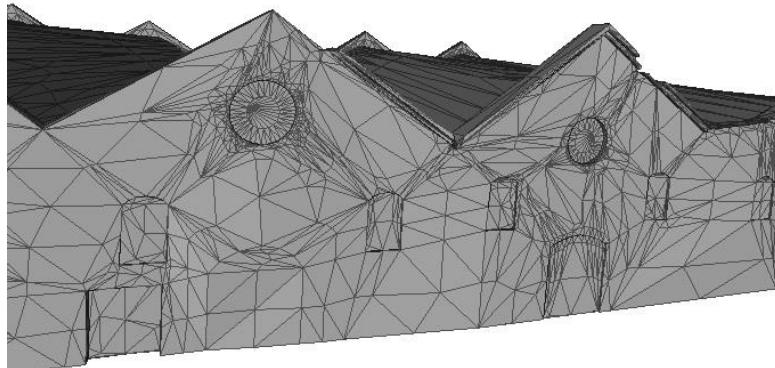


Figure 5.21 Perfectly rounded windows after further refinement.

It is essential to have a nicely smooth surface. A finer and smoother model enables the application of detailed texturing without making it look artificial. A refined surface with a nice texture mapping, can give a more realistic effect.

5.3 Texturing

The notion of texturing was explained in the second chapter. Texture mapping is a method of adding photorealism and detail to a flat surface without the use of extra triangles. A texture map is a bitmap image that is applied to the surface of a polygon creating a visual result that seems to have more details than could otherwise be achieved with a limited number of polygons.

The texturing process at its basic form requires three steps: unwrapping the mesh (UV mapping), creating the texture, and applying (rendering) the texture.

5.3.1 UV Mapping

In order to apply textures in our model it is essential to implement UV mapping first. UV maps are the first step in correct and realistic texturing. UV mapping is the 3D modeling process of making a 2D image representation of a 3D model's surface. This process projects a texture map onto a 3D object. The letters "U" and "V" denote the axes of the 2D texture.: UV texturing permits polygons that make up a 3D object to be painted with color from an image. The image that UV mapping produces is called a UV texture map, and is very useful for the further texturing of the object. The UV mapping process involves assigning pixels in the image to particular surface places on the polygon, usually done by programmatically copying a triangle shaped piece of the image map and pasting it onto a triangle on the object. UV is the alternative to XY, it only maps into

a texture space rather than into the geometric space of the object. But the rendering computation uses the UV texture coordinates to determine how to paint the three-dimensional surface. [57] UVs are essential in that they provide the connection between the surface mesh and how the image texture gets mapped onto the surface mesh. That is, UVs act as marker points that control which points (pixels) on the texture map correspond to which points (vertices) on the mesh.

When a model is created as a polygon mesh, UV coordinates can be generated for each vertex in the mesh. One way is to unfold the triangle mesh at the some designated seams (edges), automatically laying out the triangles on a flat page. Once the model is unwrapped, the artist can paint a texture on each triangle individually, using the unwrapped mesh as a template. When the scene is rendered, each triangle will map to the appropriate texture from the paint unwrapped mesh.

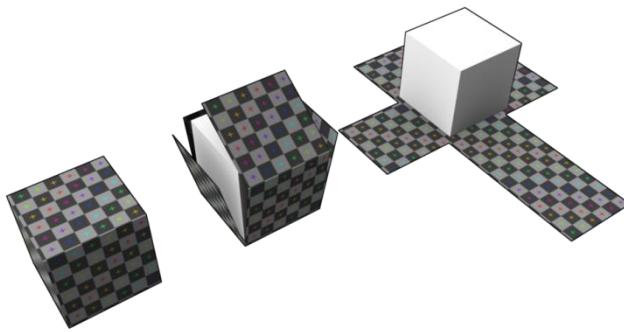


Figure 5.22 An example of UV mapping of a cube. The flattened cube can be painted to texture the cube.

UV mapping (just like the whole texturing process) was implemented in 3ds Max. It provides the appropriate tools and has features that the user can utilize. The main feature for UV mapping is the Unwrap UVW modifier.

Unwrap UVW modifier

The Unwrap UVW modifier lets you assign mapping (texture) coordinates to objects and sub-object selections, and to edit those coordinates by hand as well as with a variety of tools. You can also use it to unwrap and edit existing UVW coordinates on an object. You can adjust mapping to fit on Mesh, Polygon, and NURBS models using any combination of manual and several different procedural methods.

In using Unwrap UVW, you usually break up the object's texture coordinates into smaller groups known as clusters. That way you can position the clusters precisely over different areas of the underlying texture map for optimal mapping accuracy. Each of

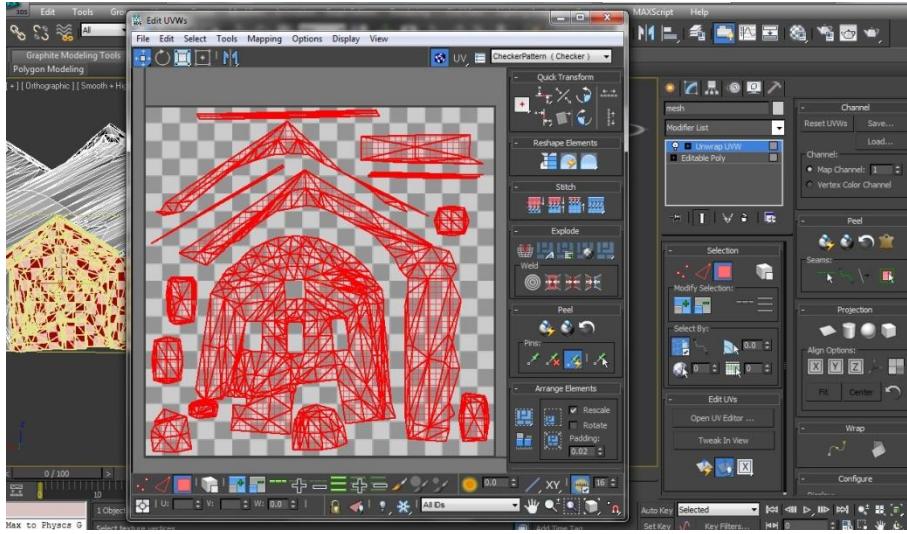


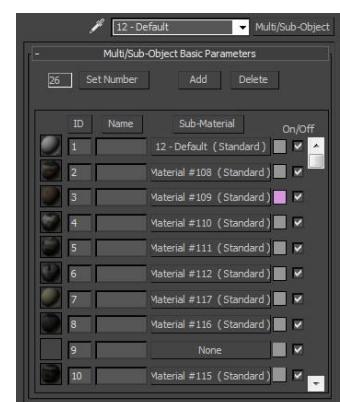
Figure 5.23 The Unwrap UVW modifier interface. Its menu is on the right with the UV editor open in the middle.

these clusters has an outline called a map seam which appears superimposed over the object in the viewports. This helps you visualize the locations of mapping clusters on the object surface.

To begin the unwrapping process we need to consider the complexity of the surface and the size of the model. The texturing is going to be made with photographs taken on the field. That means that there is great detail to be depicted and we need to be careful to correctly place the textures to have an accurate result, responding to reality. It is impossible to unwrap the whole model due to not only the complexity of the surface but also the size of the model. It is difficult to place all the unwrapped triangles in an image but more importantly, impractical for us to 'paint' the UV texture map.

Therefore, it is required to separate the triangle mesh in clusters for a smooth and easier workflow. 3ds max gives to the user that ability with the use of special materials. 3ds Max uses materials to add texture to objects. By applying them, you can add images, patterns, and even surface texture to them. They can be manipulated via the Material Editor, 3ds Max's design studio for materials and maps.

The material ideal for this is the Multi/Sub-Object material, which lets you assign different materials at the sub-object level of your geometry. You can create a multi-material, assign it to an object, and then you can select faces, group them and choose which of the sub-materials in the multi-material are assigned to the selected faces. [58]



The geometric mesh was split into clusters by selecting parts and faces that have a spatial connection and can be manipulated easily in order to apply texture on them. The groups are consisted mainly of each façade of the seven domes. The north surface is split in seven groups each one for each façade, the same as the south surface. The east and west surfaces were split in convenient groups, as well as the roof. Some of the groups are shown in the figure 5.24.

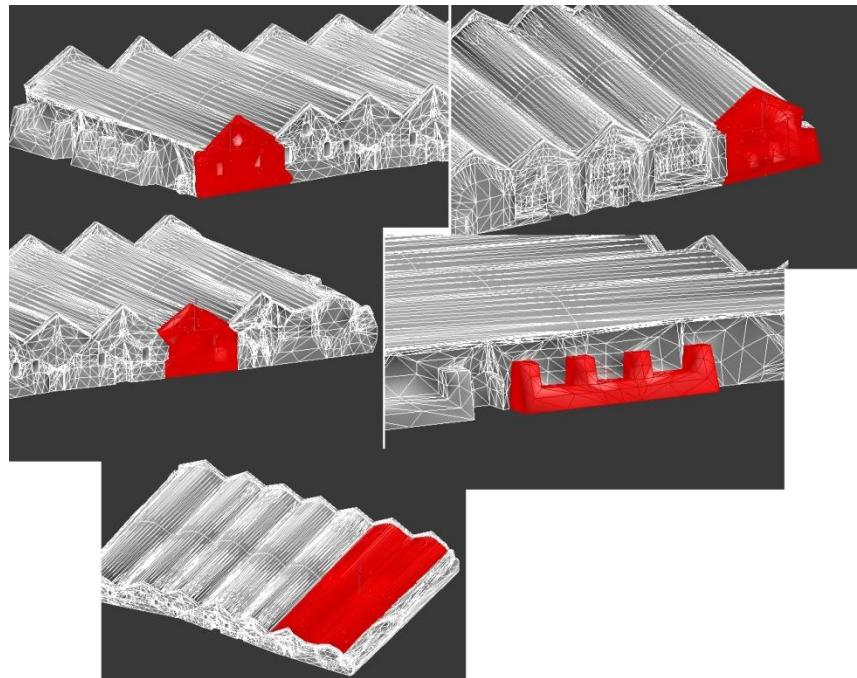


Figure 5.24 Some examples of the face groups that were created for texturing.

Every group is assigned a material ID which determines which sub-material to use when you apply a Multi/Sub-Object material. Now the process of UV mapping begins. We process the groups one by one. For each group we apply an Unwrap UVW modifier. Our objective is to unwrap, flatten the mesh in order to appear in a 2D image. Considering the complexity (the mesh is not as simple as a cube), we have to divide it. Each mesh is separated based on similar parts. We created seams for subdividing the mesh. The seams are existing edges and are what separates the clusters of the mesh. Each face cluster that is defined by the seams is textured separately. Sometimes this creates visible discontinuities on the texture and are damaging realism. For this reason we have to create seams in places that are not visible or in places that there is a natural difference in appearance.

We chose our seams in places that can be hidden, like the inside of doors or windows. After we determine the seams we peel each cluster in order to flatten it. The process and the results are shown on the UV editor. With the help of a checkered pattern we can check if the texture is applied without disturbances. If we found disturbances we can correct them by moving around the appropriate vertices in the UV editor.

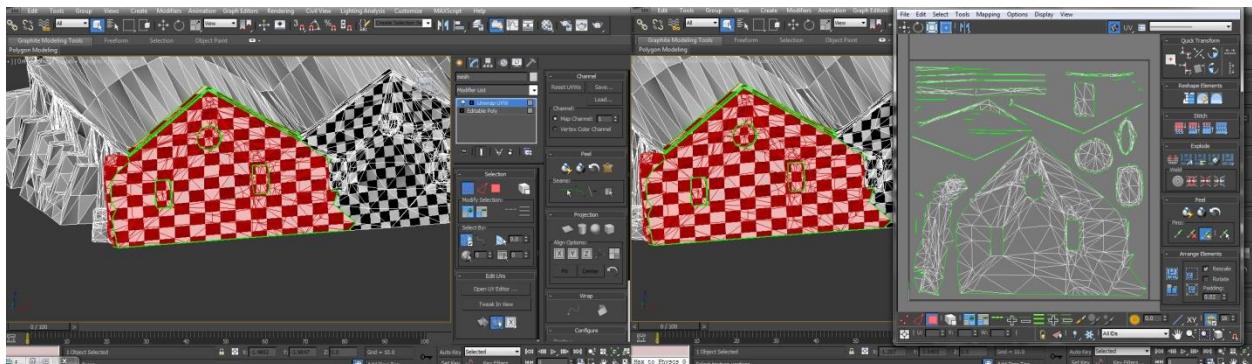


Figure 5.25 Left: the seams of the mesh that separate the clusters. **Right:** the created clusters are shown on the UV editor window.

Once we've laid out the mesh's texture coordinates in the Unwrap UVW modifier editor, we can export them to a paint program for creating the texture map. In the UV editor there is an option 'Render UVW Template' , where we can render the template as a bitmap. We can inspect the output, and if changes are necessary, like if there are overlapping faces we can correct them and render again. If the results are satisfying we save the template as a bitmap image. We can use that image in a painting program to create the texture. Some examples of the UV templates are shown in figure 5.26.

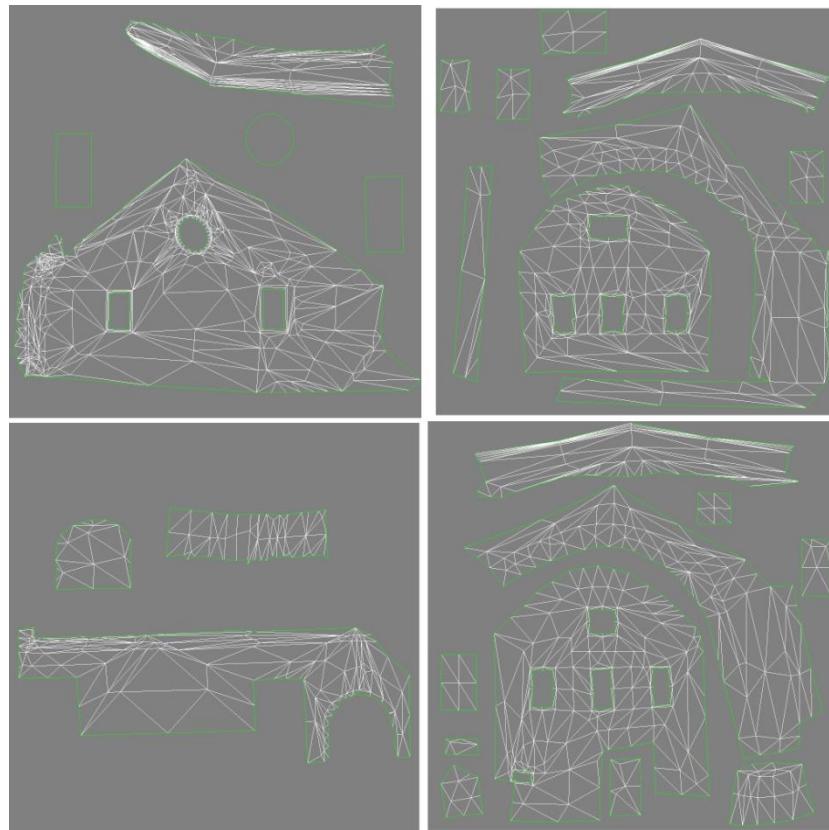


Figure 5.26 UV texture coordinates rendered in bitmap images.

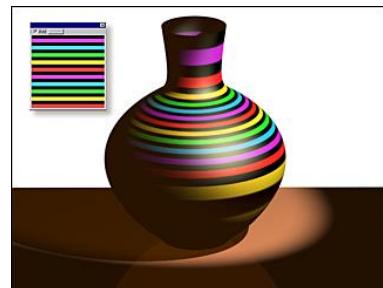
This process is continued for all the groups we created.

5.3.2 Texture mapping

Now that we have laid out the UV coordinates it is time to apply texture to the triangles. Because we are trying to digitally document a real monument, it is infeasible to create the textures ourselves. It is impossible to reproduce the level of detail that exists, especially in a building such as this with eroded stone walls and damages. In order to have a result that corresponds to reality, we need to acquire texture information from photographs.

During our measurement process we took photographs of the building from a standard camera. The photographing process was also repeated on later days. Whenever we were missing texture information or the photographs were not usable, we went to the field to acquire new ones. We photographed every side of the building, covering all the exposed surfaces. Those images were treated as textures.

We use these images as a diffuse map. As it was stated before, a diffuse map is a texture that defines the surface's main color. Diffuse mapping takes the images and wraps them on the object's surface according to the UV mapping, displaying the original pixel color. It sets the tint and intensity of diffuse light reflectance by the surface.



The image textures can be applied to the UV templates that were created on any painting or image processing software, like Photoshop. The workflow that we followed is this: we take small patches from an image and we apply it to its corresponding place on the UV template. We are careful to place it accurately and then we process it to cover its attachments to the neighboring patches. An example of the texture workflow is presented next.



Figure 5.27 Photographs of the westernmost façade of the north side taken on the field.

We took multiple photographs of each wall, to acquire as much information as possible. After processing the photographs in order to have the same colors, brightness and hue

we took the best looking patches from each one and we filed the UV template. The image was saved as a bitmap image.

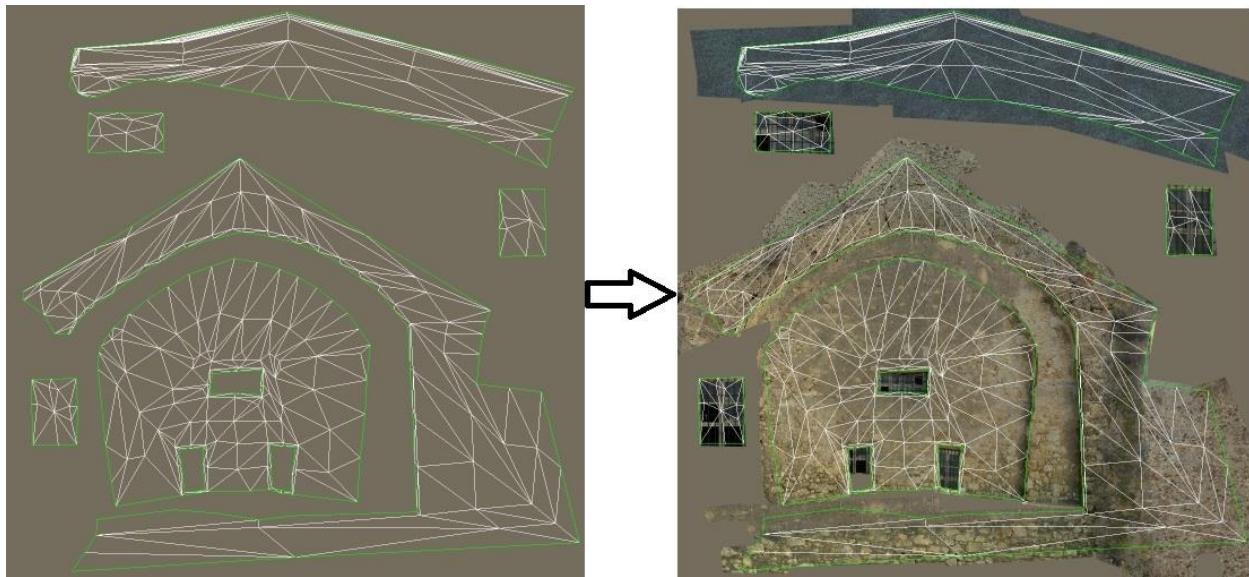


Figure 5.28 The UV template we rendered on the left, and the texture we created on the right.

The texture is ready to be placed on the model. We go back to 3ds Max and we place the bitmap image on the diffuse slot on the appropriate submaterial of the Multi/Sub-Object material we created earlier. It can be viewed on the 3ds Max viewport.

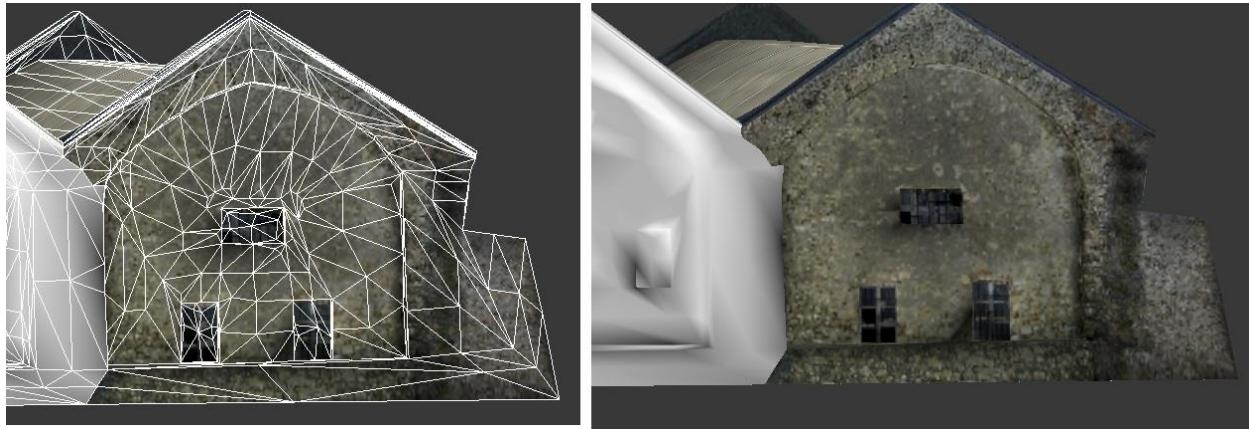


Figure 5.29 The textured group. Displayed with (left) and without (right) the triangle mesh.

The same workflow was deployed in order to texture the whole model. It should be noted that it was a very time consuming process, because we wanted to replicate in the best way possible the real surface.



Figure 5.30 The textured model displayed from several sides.

5.3.3 Normal mapping

As is obvious the application of images as textures, i.e. the diffuse mapping is not adequate for a realistic result and it does not communicate photorealism. Taking into account the surface condition of the monument, we see that there are damages, alterations on the surfaces and each wall has different appearance and uneven surface. Those details were impossible to be digitally documented. Additionally, creating these details with polygons raise the polygon count, making it harder to run in systems with limited processing power. We want to create complex effects in the available polygons.

The perception of realism as well as 3D depth is attained by normal mapping. Normal mapping simulates small displacements of the surface while the surface geometry is not modified. Instead, only the surface normal is modified as if the surface had been displaced. This is achieved by perturbing the surface normal of each object and using the perturbed normal during lighting calculations. Normal mapping is commonly used to capture the detail of a high resolution mesh with the geometry of a low resolution model.

Normal maps consist of red, green, and blue. These RGB values translate to x, y, and z coordinates, allowing a 2D image to represent depth. The 2D image is applied to the surface. This way, a 3D surface simulates the lighting associated with 3D coordinates. In the normal map, each pixel's color value represents the angle of the surface normal.

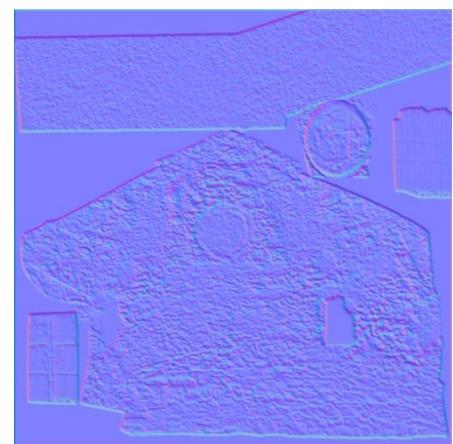


Figure 5.31 An example of a normal map.

We use normal maps to create details and give the illusion of an eroded and uneven surface. We created the normal maps from the textured UV templates, so that they have the same texture coordinates with the diffuse maps. The normal maps were generated in Photoshop with the help of nDo, which is a normal map creation script and it works as a plugin in Photoshop. (information about nDo <http://www.philipk.net/ndo.html>).



Figure 5.32 A normal map (right) generated from the texture (left).

We are going to employ the use of multitexturing in our model, that means that we are going to use more than one maps on the polygons. We are going to texture the surface of our model with diffuse maps in addition with normal maps. In that way we have a low cost method of achieving high surface detail. The normal maps are going to be applied inside Unity after we will have imported the model.

The 3D model is completed. We began with the 3D coordinates and we constructed a realistic 3D model, representing accurately the Venetian monument Neoria. We textured it and it is ready to be imported in Unity.

Chapter 6 Presentation of the 3D interactive visualization and the Graphical User Interface

6.1 Concept

The aim of this project is the 3D digitization of a historical building. The 3D reconstruction of the monument was presented via a visualization tool which offers expert users the capability of navigating and manipulating the model. A fully interactive application was created based on game engine technologies. Unity3D offers sophisticated tools for the development of 3D interactive applications and a programming interface utilized for the development of complex geometry behaviors, user interaction, etc.

The interactive visualization tool that was implemented offers the user the ability to photorealistically visualize the actual monument from all sides. The user can navigate the geometric model and tour the perimeter. The manipulation of the monument is featured via advanced interactive functionalities which are offered to the user. The surface can be visualized as is currently, with erosion and damages, cracks and inappropriately use of modern materials as well as how it will look if those damages or surface textures were restored. There are damages that are corrected and rendered as such after user interaction. The user can also calculate distances on the monument surface and have details about them displayed on the screen.

The data acquisition process is fully utilized in the application. The user can visualize the actual measurements that were acquired in the monument surface and attain information for each one of them. A database was also created and displayed with all the measured coordinates. The user can choose a 3D coordinate and see in which point in the surface corresponds.

The monument can be displayed under different lighting conditions. Each part of the surface can be visualized in lighting or shadow. The user can manipulate the lighting and position it in a desired place, simulating the sun conditions. Additionally there is a transition from day lighting conditions to night lighting conditions. Night and day lighting are simulated and the user can smoothly change the atmosphere to replicate any time of the day and see the monument under those simulated lighting conditions.

The fully interactive application will be introduced in the following sections of this chapter. The functionalities will be presented and we will explain how to use them. We will analytically show how the user can utilize the abilities that the application offers and present the graphical user interface.

6.2 User Navigation and Perspective

The application is carried out with a first-person perspective, that is a graphical perspective rendered from the viewpoint of the player character (as it is called in games). There are two perspectives that games or applications are rendered through. The first person perspective where the user experiences the action through the eyes of the character and the third-person perspective, in which the player can see (usually from behind) the character they are controlling. In our case we consider it suitable to have a first person perspective because we want the user to experience the environment from his/hers own point of view.



Figure 6.1 The perspective of the user. It is a first person view.

As is shown in figure 6.1 the user experiences the environment in first person and no character is visible. The user sees directly through the camera placed on the scene. The user can rotate around the camera and see in a 360° angle from the point he/she stands, on the horizontal level. The camera can also be tilted up or down. In order to do these actions the user should hold down the right mouse button. The view from the camera changes only while the right mouse button is pressed and follows the motion of the mouse. When the mouse goes left, the camera also rotates in the left angle. Same goes for all the other directions. When the user releases the right mouse button, the camera movement stops.

The navigation is implemented with the arrow keys. The user can move forward, backwards and right or left. Wherever the camera is turned the movement direction follows (e.g. if the user goes forward and then the camera turns right, the forward movement will be in the direction of the camera). The user is prohibited from falling in

the sea that is placed on the front of the environment, due to the placement of invisible colliders and going through the walls of the building. The user also cannot move past a perimetrical limit around the monument (defined by the trees which are placed around it).

6.3 Graphical User Interface

The Graphical User Interface when the main scene is loaded is presented in the figure below.



Figure 6.2 The main GUI.

The menu is positioned on the top left. It is consisted of buttons, on/off toggle switches and a slider. The function of each button will be explained next.

6.3.1 The Main Menu

Explore Side menu

On the top left side there are four buttons named ‘North’, ‘South’, ‘East’, ‘West’. These buttons are implementing a transportation action. When one of them is pressed the user is transported in the corresponding side. This is created so that the user can have easier access on the sides and not navigating all the way around. The spots where the user is transported are shown in figure 6.3. When the user presses the north button he/she transports on the spot shown on the top left image, when he/she presses the south button he/she transports on the top right image and so on.

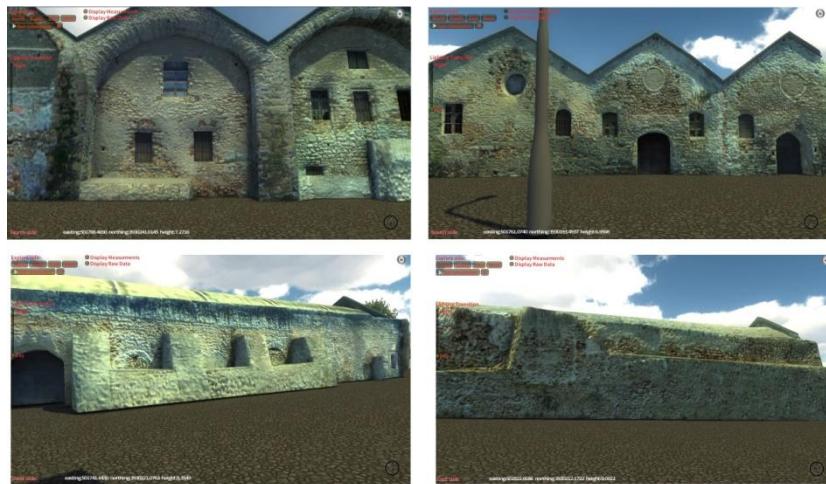


Figure 6.3 The display when the user presses each transport button. Clockwise: north, south, east, west.

Sun Simulation

Below the explore side menu there is a button called sun simulation. This activates the lighting simulation. When it is pressed a light source starts moving around and above the building simulating the sun. The movement of the light source is animated, that is, it moves continuously in the round. It casts light (or shadows) on the building and depending on its position it lights each side when it passes above it. The user can also stop the simulation on a desired position. The light source stops at a chosen place so that the user will be able to see a surface lighted. When the button is pressed again the simulation continues from where it stopped.



Figure 6.4 Part of the building with the light source stopped above it (left) and in the shadows (right).

Lighting Transition

On the left of the screen there is a slider which simulates a day/night transition. The user can move the slider up or down and change the atmosphere to replicate day or night. This way a transition from day lighting conditions to night lighting conditions is implemented. Night and day lighting are simulated and the user can smoothly change the atmosphere to replicate any time of the day.

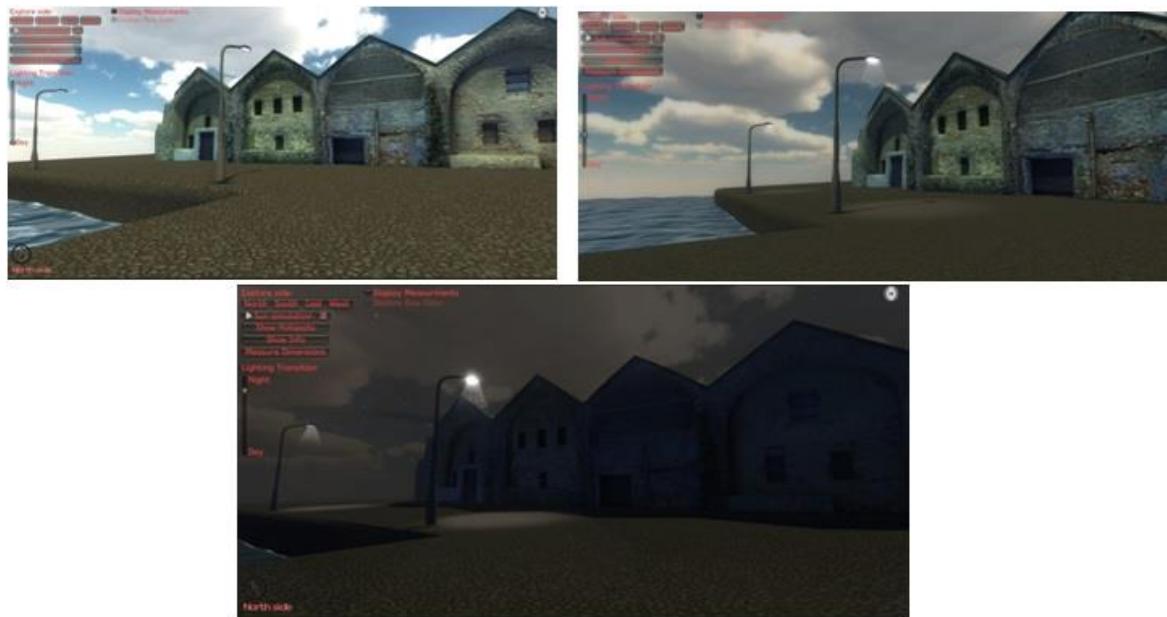


Figure 6.5 The day/night transition. On the top left the day simulation is shown , on top right a middle time of day is shown and on the bottom the night simulation is displayed.

On figure 6.5 it is shown that when the slider is on the far bottom the day simulation is displayed. As the slider goes upwards the atmosphere slowly changes and the night simulation smoothly transits. At around the middle point of the slider, a middle time of day is simulated as is shown in the top right picture above. When the slider goes on the top the night time is simulated.

Display Measurements

One of the aims that we wanted to achieve is to connect and relate the 3D model with the initial geographical data. We did not want the measurement process to be considered as something used only in the beginning of the modeling process but also as something that can be included in the interactive application. It would be interesting to show how the measurement process was conducted and where the measured 3D points correspond to the surface. The visualization of the measurements on the monument surface is also a means to georeferencing the model. It is also a way to provide scientific information about the original data acquisition process and relate the model to it, instead of having an isolated 3D graphical model.

To this aim a functionality that displays the measured geodetical points on their corresponding spots on the monument surface, as well as geographical information about them was implemented. It is activated with the on/off toggle button that is on the top of the screen, appropriately named ‘Display Measurements’. When the button is activated small green cubes appear on the spots where each measurement was contacted with the geodetic total station.

Display Measurements



Figure 6.6 The display measurements functionality. On the right picture the measurements that were conducted are displayed on the surface (The left picture is without that functionality enabled).

Moreover, the user is given the ability to click on a green cube and information about the specified point appear. More specifically, the easting, northing and the elevation of the point measured appear on 3D text. When a cube that displays information is clicked again the data disappear.



Figure 6.7 When a green cube is clicked the corresponding geographical data from the measurements appear next to it.

The visualization of the coordinates of the original geodetical measurements can help the user to understand the method of the data acquisition, see which areas were measured in detail and identify areas where the points are dense or scarce.

Display Raw Data

The opposite action of what was described above was also implemented. In the previous functionality the user could click on a displayed point on the monument's surface and information about it was provided. It would be also useful to have the geographical information provided and choosing a coordinate to be directed to the surface point to which corresponds. The latter is implemented with the display raw data functionality.

The ‘Display Raw Data’ toggle switch activates the  implementation. When the button gets activated a scroll plane

appears with all the data from the measurement process. The point ID, which is a number used to discern the points, easing, northing and level are displayed. The point ID of each point is a button and next to it are its spatial data.

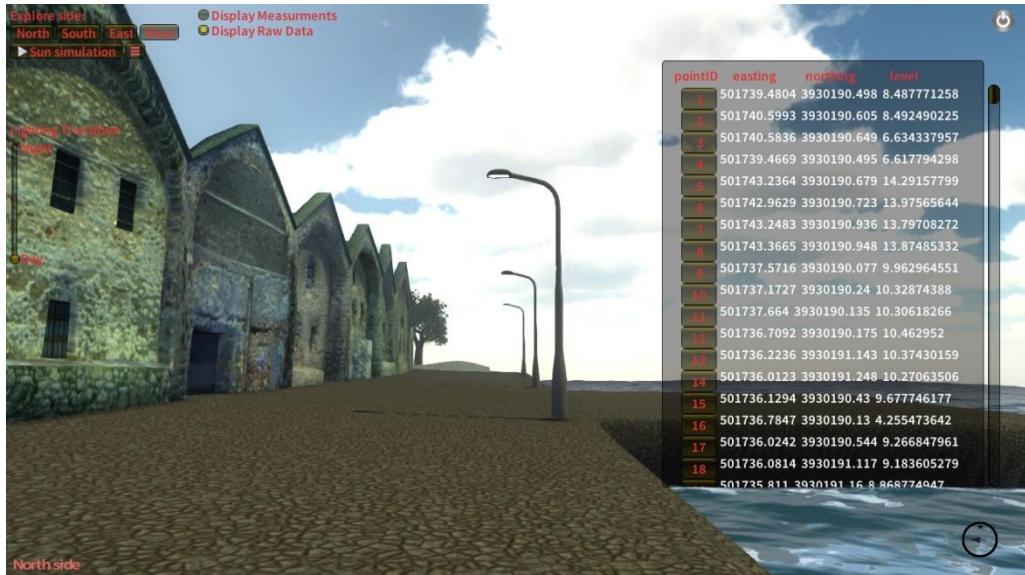


Figure 6.8 The display raw data toggle activated. A scroll plane appears on the screen with the geographical information acquired.

The raw data that are displayed are stored in a local database, which was created in order to eliminate the file dependencies. The database was created with SQLite locally and it can be accessed via SQL commands. In the next chapter it is explained more analytically. It should be mentioned that the information in the previous functionality, 'Display Measurements' are also derived from the database.

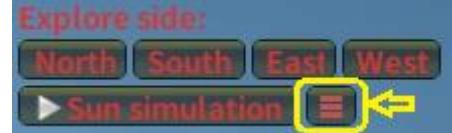
The user can scroll through the information and can click each point ID button. When one of those buttons is clicked a text label appears informing the user in which side that point is. Then the user can use the transportation buttons and reach the side to find the point. The point is displayed as a blinking sphere. When that same button is pressed again the point disappears from the surface.



Figure 6.9 When a point ID button is pressed, information about its position appears on screen and the point is displayed on the surface.

6.3.2 The Submenu

Next to the Sun simulation button there is a small rectangular button. That button corresponds to the Submenu. When it is pressed a submenu appears. That submenu consists of three buttons, each one implementing a functionality. Those are the ‘Show Hotspots’ functionality, the ‘Show Info’ and the ‘Measure Dimensions’ functionality.



Show Hotspots

The goal of this work was to develop an interactive visualization tool which can offer users the capability of interacting with a 3D reconstruction of an existing monument. One way that interaction can be achieved, is if the user had the ability to alter the geometry provided. As it can be observed the building has a lot of damaged areas and spaces that have been wrongly altered with modern or inappropriate materials. A functionality has been implemented that provides the ability to change the geometry in order to resemble the original non damaged surface.

There are a lot of areas in need of restoration. Parts of the roof have fallen, doors have been blocked up and the texture of the surface has been changed. The user can visualize those areas as if they were repaired.

When the user clicks on the ‘Show Hotspots’ button, several blinking spheres appear on the walls throughout the building. These spheres, i.e. hotspots, are placed in areas that are impaired and each one corresponds to a different alteration.



Figure 6.10 Two hotspots as they appear on the surface monument.

There is a hotspot that is on the north side in front of the easternmost dome. As it can be observed that dome has modern materials applied to it. It has been altered with modern white paint. When the user clicks on the hotspot, this modern material is removed and replaced with textures derived from non-altered areas. By clicking the hotspot again the user can return to the old surface.



Figure 6.11 The material of the dome changed in order to cover the white paint. The user can toggle between surfaces.

On the north side there is also a facade which is heavily covered with cement and the original surface has almost disappeared. A hotspot has been placed on that spot changing the surface’s texture when clicked. Although architectural information about

the monument's original appearance could not be found, the appearance of uncovered parts was taken into consideration to create the new texture.



Figure 6.12 On the left added cement can be seen on the monument's facade. The user can toggle between materials in order to visualize the original surface.

The user toggles between materials and visualizes the original surface in contrast with the existing one.

On the easternmost dome on the south side, it can be seen that the roof has collapsed. The capability to visualize to original roof is offered to the user. As it has been said before, due to the lack of architectural information, the restored roof is created based on the appearance of the non-damaged parts, as well as the measurements of structural elements of neighbouring areas. A hotspot appears on that spot and by clicking it the user can toggle between the monument surface as it exists today and the restored surface rendered in red color.



Figure 6.13 The user can alter the geometry provided, by toggling between the existing one and the 'restored' one.

A similar procedure is implemented in a dome on the south side, which has a missing roof edge. A hotspot was placed there and the restored roof can be visible by clicking it.



Figure 6.14 The missing roof edge can be visualized with the toggle hotspot button.

The easternmost façade on the south side apart from the missing roof has also a blocked door, the door via the toggle hotspot button can be replaced with an unblocked one.



Figure 6.15 The blocked door on the south side can be replaced with a regular one.

Via that particular functionality the user can alter the geometry and materials provided. A restored appearance in the damaged areas is recommended and the visual impact of such restoration is rendered after user interaction.

Show Info

Underneath the ‘Show Hotspots’ button, there is a button called ‘Show Info’. When it is clicked a plane containing historical facts about ‘Neoria’ appears on the screen, providing the user with historical information about the monument.

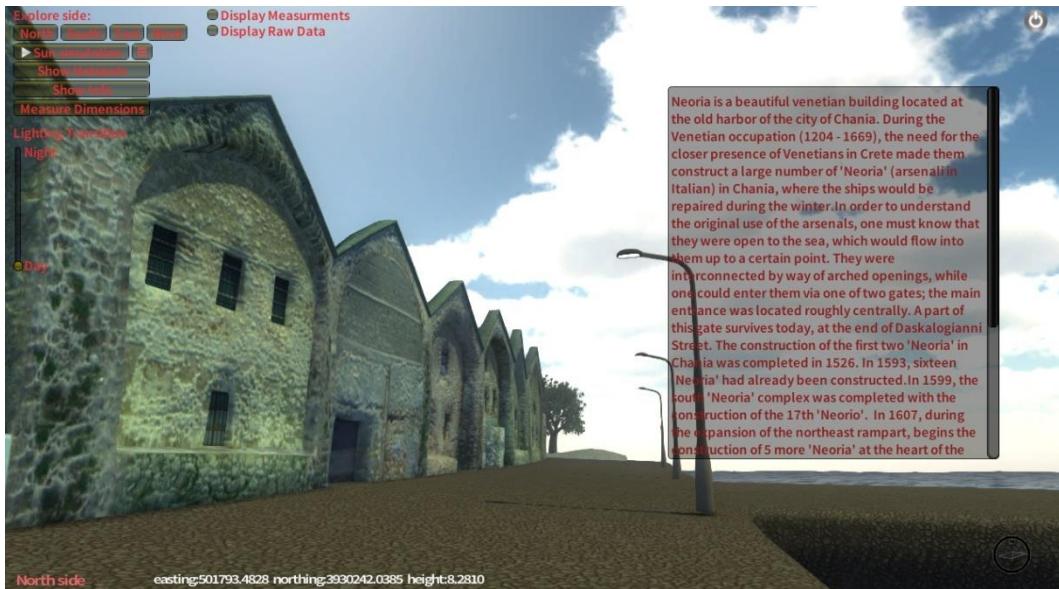


Figure 6.16 When the 'Show Info' button is clicked, a scroll plane with historical information appears on the screen.

Measure Dimensions

Scientific information can be offered to the user in relation to measurements between areas, existing and restored. This action is implemented with the 'Measure Dimensions' button. The user can select specific points on the surface and the distance between them is calculated. When the user clicks on the button, a text label appears on the screen with the message "Click on a starting button". Then a yellow dot appears whenever the mouse cursor is on the monument surface, making it easier for the user to pick a spot. When a particular spot is clicked, a small green pyramid marks the spot and a message "Click on a second point" appears. When the second point is clicked, another pyramid marks the spot and a line between the two pyramids appears, so that their distance can be visualized. Their distance in 3d space is calculated, as long as their depth difference on the vertical surface and both are displayed on the screen.

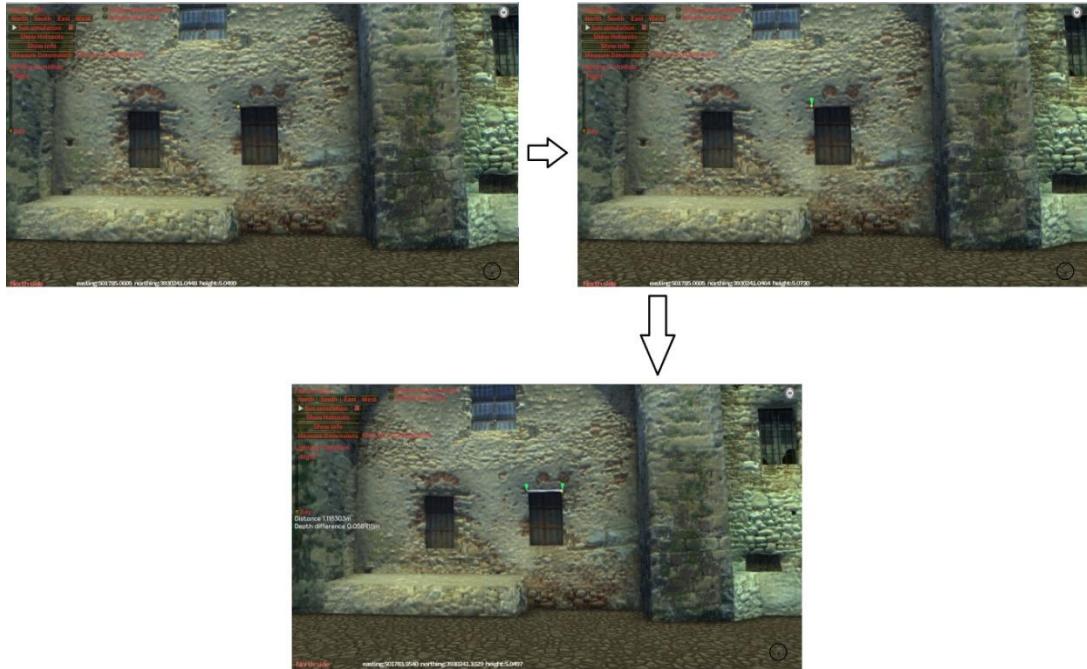


Figure 6.17 The steps for measuring the dimension between two points.

This operation can help the user gain information regarding the position and relation of chosen points. The distances are measured in meters. In that way each user can measure the dimensions of a particular feature or can gain information on the size and exact place of a restored part.

6.4 Additional Features

As it was mentioned before, it is crucial to relate the geographical measurements with the 3D graphical model. The model gains purpose by associating it with the real physical space. Unity's world coordinate system, which references the space that the models are placed on a scene, albeit in real world scale, may be useful for working and associating objects within the scene but is making the created 3D environment isolated from the real world.

A way of connecting and associating the environment with the real space is by aligning geographic data to the local coordinate system that Unity uses. This process called georeferencing assigns the position of the model in the physical space, giving to the application the ability to be used, analyzed or compared with other geographical data.

This was accomplished in our application with a feature that displays dynamically the geographical information on the bottom of the screen. Whenever the mouse cursor is on the vicinity of the monument's surface, the geographical information of the spot that the cursor points at, are displayed on the bottom. The display is continuous throughout the application's operation.



Figure 6.18 The dynamic display of the coordinates. Wherever the mouse points, the corresponding coordinates of the spot appear on the bottom of the screen.

A compass is also displayed on the bottom of the screen. It is a way of preventing the user from being disoriented.

Spatial recognition is also implemented. On the bottom left of the screen a message is displayed informing the user on which side he/she is. This is created due to the size of the model and for easier navigation around the monument especially for users that have no knowledge or experience regarding the real monument.

On the next chapter, the implementation of the functionalities is described analytically, as well as the features and tools used for them.

Chapter 7 Implementation

In this chapter we are going to present how the final 3D interactive visualization was created. Each functionality will be presented and its implementation will be explained thoroughly. The programming process will also be explained.

Every object that there is in the application has multiple components, most of which are custom made via scripts. That means that the behavior of the objects is controlled by multiple scripts. Scripts access other scripts and gameObjects that may even not be connected with them. We ended up having a large amount of interactivity between the scripts and the components.

The implementation was developed in Unity. The programming parts were developed using javascript. The graphic features as well as the scripting IDE that provides, can give the ability of developing advanced games and applications.

7.1 3D model import

After we created a Unity project, we must import our created 3D model into it. Fbx is a file format owned by Autodesk and it is used to provide interoperability between digital content creation applications. It is widely used by modeling software like 3ds Max or Maya to exchange data with other software applications, like game engines.

We export our model from 3ds Max in fbx format. In our newly created game, we create a new scene in which we are going to build our environment. We import the model as an asset. The import settings dialog is shown on the figure 7.1. During the import process the user can generate colliders or generate lightmaps. By hitting apply the model is transported into the project panel from where we can place it in the scene.

The imported object remembers the materials, their names and their place and reserves slots for them. As developers we have to import as material assets the bitmap images that we have created as textures and their respective normal maps. By choosing which shader we want to use with which material the appropriate slots are opening, and we choose the bitmap images we want to insert on them.

After we place it on the scene, we can view it on the viewport and change its position, rotation or scale via the transform component. The model is ready now to be

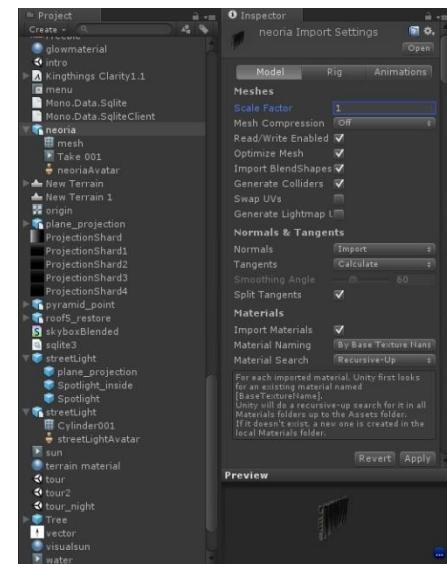


Figure 7.1 Import settings for assets.

manipulated. The same procedure was applied in all the 3D models we used in our project.

7.2 Creation of the 3D environment

At this section we are going to describe all the models we used, how we placed them and how we manipulated them in order to create our basic scene.

The most important model is the 3D model that we created based on our geographical data. It is the basic element around which we are going to build our application. After we imported it in our project and we placed it in our scene, we have to choose and apply the materials. The first step is to choose which shader to use. Shaders, like we explained in the third chapter are assets that contain code and instructions that tell the graphics hardware how to render. In our case we use the bumped diffuse shader.

The bumped diffuse shader computes a simple Lambertian lighting model, which means that the apparent brightness of a surface is the same regardless of the observer's angle of view. In other words the surface is not specular, it doesn't shine, it is matte. This shader has two slots. One for a diffuse map for color and one for normal map. The textures we created serve as the diffuse maps and the normal map slot is

assigned the appropriate normal map. In that way we can simulate small surface details without added polygons. After designating all the materials, we can observe the result. It is a high quality model with a lot of detail depicted.



Figure 7.2 Bumped diffuse shader.

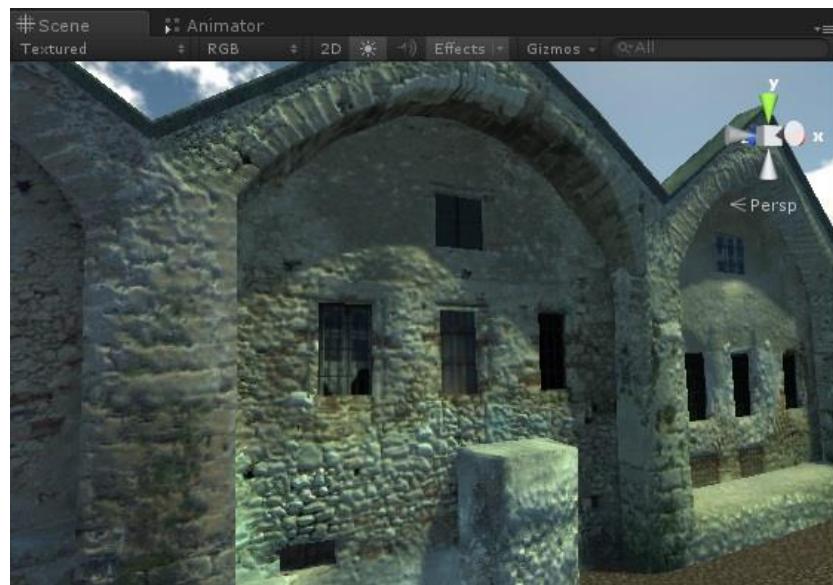


Figure 7.3 The final model imported in Unity. The normal mapping depicts a high level of surface detail.

Next we have to create a terrain, representing the ground. Unity provides features that can create terrains. Unity's Terrain system allows the user to add vast landscapes and sculpt them. Unfortunately it doesn't provide a tool for rotation of the terrain. Our model just like the actual monument has a slope and we need the ground to fit that. So we decided to create our own terrain in 3ds Max. We modeled it beginning with the primitive shapes that 3ds Max provides and based on the actual ground on the harbor in which the monument is. We textured it with a diffuse map, we imported it in the project and we assigned the fitting rotation. Now we have the monument on its ground.

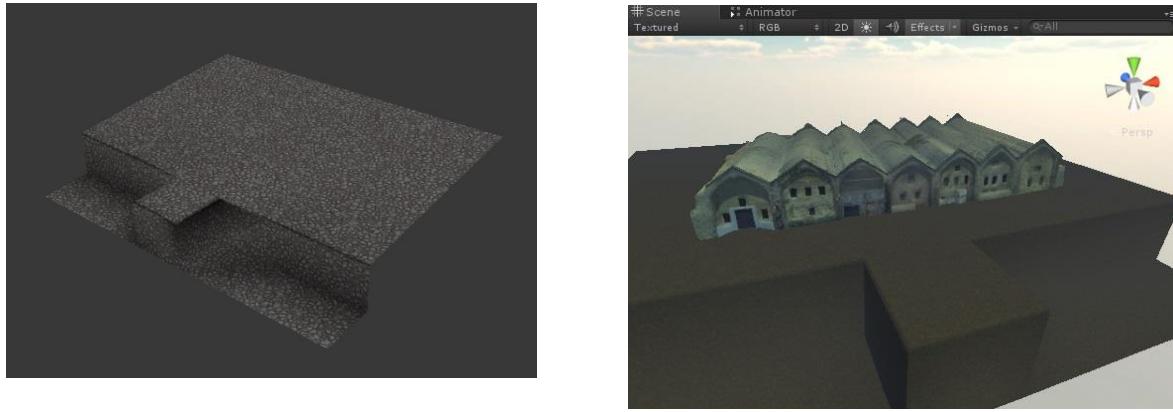


Figure 7.4 The terrain we created in 3ds Max (left) and the monument with the terrain (right).

To create atmosphere and beautify the environment we created street lights in order to position them around the model. After the actual area has streetlights. We modeled them in 3ds Max and, textured them and we imported them in Unity we placed them around the model as prefabs.

The Unity prefab asset type allows the user to store a GameObject complete with components and properties. The prefab acts as a template from which the user can create new object instances in the scene instead of duplicating the object, something that means that each one will have to be edited independently. Whereas, any edits made to a prefab asset are immediately reflected in all instances produced from it.

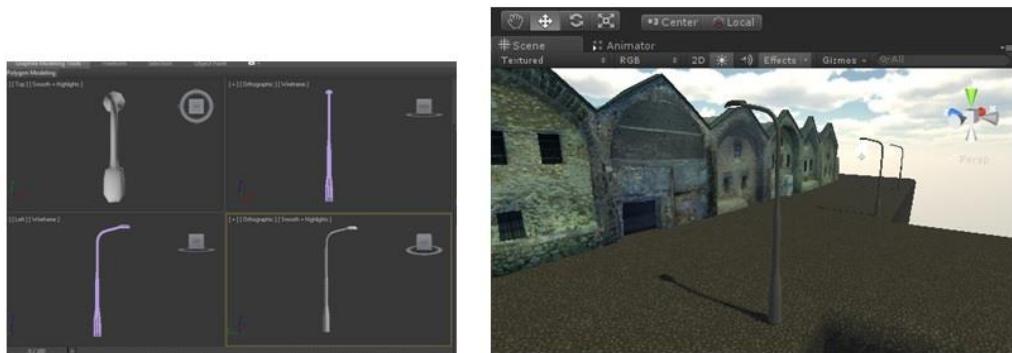


Figure 7.5 Street lights as created in 3ds Max (left) and places in the scene (right).

The streetlight functionality as well as how they were lighted will be explained in later sections.

Unity provides assets for water simulation. It is imported as a package and used as a regular gameObject by positioning and manipulating it on the scene. Its surface is animated and can provide a realistic simulation for lakes, rivers, ocean. We used it as the sea that there is on the harbor in front of the monument.

At this point we are just going to set the basic visual elements in our scene. The Render Settings menu in Unity contains features that can help control the visual style in our environment. First we use a skybox in order to have a nice realistic look around our environment. Skyboxes are a wrapper around the entire scene that shows what the world looks like beyond the geometry. They are rendered around the scene in order to give the impression of complex scenery at the horizon. In other words they simulate the sky. To implement a Skybox we import a skybox material and apply it on the designated slot. The skybox material is visible in figure 7.5.

Another feature that can be controlled at the render settings is the ambient light. Ambient light refers to the uniform illumination that lights the entire scene. The user can adjust the color of the ambient light to give the desirable effect. At this point we are going to give a light color in order to make our scene visible. It will be further adjusted when the scene is properly lit.

7.3 User Navigation and Camera Control

For interactive 3D environments Unity provides two main navigation methods. The first and third person perspective. Unity comes with packages that provide assets to common tasks. Such package is the Character Controller. It includes the first person controller and the third person controller. In our application we want to implement a first person perspective that's why we are going to use the first person controller.

The first person controller has some important information such as size. As it can be seen in the inspector the controller's height is 2 meters. This is something important in the scaling process, because we are going to use the controller as our scaling guideline. By keeping the controller intact and scaling everything around it, we stay true to the world scale and all the physics simulations will work normally.

By placing the first person controller in our scene, we see that there is a huge difference with the rest models. That means that we have to scale up the rest of the models to fit the real world sizes. Taking in account the height of the controller (2 meters) and knowing the real height of monument we came to the conclusion to scale up the monument 47.5415 units. The same goes for the terrain. We appropriately scale the street lights. Now we have a scene respective to reality.

Another part that is very important in the interactivity in Unity are the colliders. They are components that are attached to GameObjects and they detect collisions. They detect when the objects interact with one another, they prevent objects from walking through doors, or as is this case they prevent objects from falling through the floor. For this reason we have to add a mesh collider component in the terrain we have created. Mesh collider means that the collider takes the shape of the object. There are other colliders such as cube, sphere or capsule.

Now we have placed the first person controller in our scene. It has two child objects Graphics and the Main camera and provides functionality for moving the controller and rotating the camera. We have named the first person controller ‘walker’.

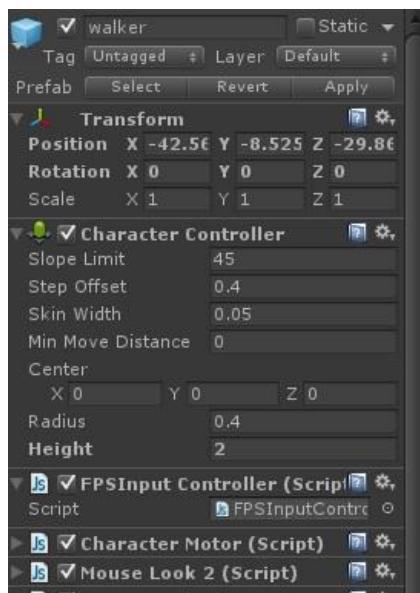


Figure 7.6 The components of the parent of the first person controller.

The transform component controls the place, the rotation and the scaling of the object. The Character Controller is mainly used for player control. It is simply a capsule shaped Collider which can be told to move in some direction from a script. The Controller will then carry out the movement but be constrained by collisions. The FPSInput Controller script and the Character Motor scripts come with the controller. The FPS Input Controller controls left right forward backwards movement. The character motor script contains a class that uses the character controller. The mouseLook script that comes with the asset controls the whole rotation of the controller and the camera.

The Graphics child object represents the physical presence of the controller and has a mesh renderer and a mesh filter component.

The main camera child object is our eyes for the scene. It determines our view of the environment. The components of the main camera are briefly explained below.



Figure 7.7 The components of the main camera.

- The camera component controls the setting of the camera.
- The GUI Layer Component enables the rendering of 2D GUIs, i.e. menus.
- The Flare Layer Component can be attached to make Lens Flares appear in the image.
- The audio listener receives input from any given audio source in the scene and plays sounds through the computer speakers.
- The FXAA script (Fast Approximate Anti Aliasing) is provided for free and is attached to the camera in order to perform anti aliasing in the game play.
- The Fast Bloom is a script that provides the optical effect where light from a bright source appears to leak into surrounding objects.
- The Screen Space Ambient Occlusion (SSAO) effect approximates Ambient Occlusion, i.e. how exposed each point in a scene is to ambient lighting.
- The Bloom and Lens Flares image effect adds image based bloom and automatically generates lens flares.
- The Glow Effect can dramatically enhance the rendered image by making over bright parts “glow”.
- The mouseLook component controls the rotation.

Having tested the application so far we have a basic movement with the arrow keys that satisfies us but the rotation of the camera is very unsteady and not practical. The functionality that this component provides has the camera following the mouse around. It doesn't stop and it may prohibit the development of other functionalities because of the mouse use. So we decided to alter it. The rotation of the camera now works only if the user holds down the right mouse button down.

The mouseLook script was further altered to implement an additional functionality, for that reason it will be analyzed as a whole in the following sections.

7.4 Spatial Recognition

Due to the size of the model and for future reference, we need to implement a way of recognizing on which side of the monument the controller is. This will help the user to navigate easily around the monument and prevent disorientation. Additionally, we need to store information regarding each side of the building, something that is helpful for other implementations.

To do this we created cube gameObjects one for each side. We scaled them in order to cover the whole side they were placed and we defined their collider as trigger. A trigger doesn't register a collision. Instead, it sends the function `OnTriggerEnter()`, a message when an object enters or exits the trigger volume. We disabled the mesh renderer so that it won't be visible. Their only use will be the object detection.

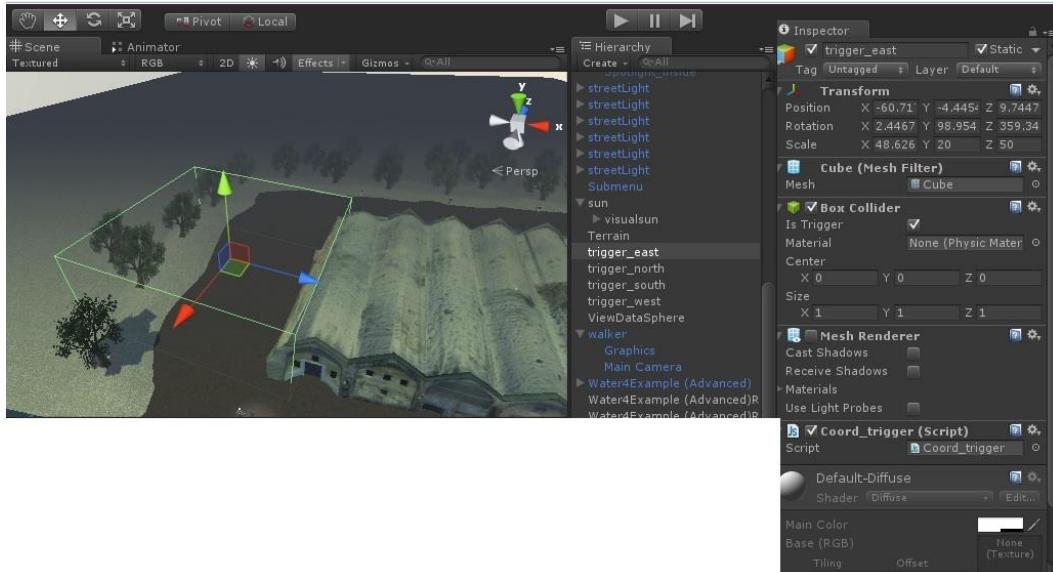


Figure 7.8 The east side trigger. In the inspector we have defined as trigger and we have disabled the mesh renderer.

In each one we attached a script called **Coord_trigger.js**. That script has the function `OnTriggerEnter()` and recognizes when an object enters the collider. We also created a `GuiText` object because we want to display a message and attached the script

Coordinates_script.js to it. In that we have created functions that implement actions depending on the side.

```

function Awake () {
    guiTextObject = GameObject.Find("GUI Text_coord");
}

function OnTriggerEnter (other:Collider){
    if (name == "trigger_south"){
        guiTextObject.GetComponent(Coordinates_script).south_coord();
    }
    else if (name == "trigger_north"){
        guiTextObject.GetComponent(Coordinates_script).north_coord();
    }
    else if (name == "trigger_east"){
        guiTextObject.GetComponent(Coordinates_script).east_coord();
    }
    else if (name == "trigger_west"){
        guiTextObject.GetComponent(Coordinates_script).west_coord();
    }
}

```

The script **Coord_trigger.js** is shown above. When the walker enters a trigger the script checks which trigger is based on its name and calls the right function from the **Coordinates_script.js**.

There we have created four functions one for each side. Each function when is called, displays on the screen a text, informing the user in which side he/she is, then keeps on a variable the a rotation angle for each side, something that is useful for other functionalities.

This is a way to provide to the user information for where he/she is, especially when he/she doesn't know the monument beforehand.

```

function south_coord(){
    guiText.enabled = false;
    guiText.enabled = true;
    guiText.text = "South side";
    rot=Quaternion.Euler(15,210,10);
}

function north_coord(){
    guiText.enabled = false;
    guiText.enabled = true;
    guiText.text = "North side";
    rot=Quaternion.Euler(0,60,0);
}

function west_coord(){
    guiText.enabled = false;
    guiText.enabled = true;
    guiText.text = "West side";
    rot=Quaternion.Euler(0,350,10);
}

function east_coord(){
    guiText.enabled = false;
    guiText.enabled = true;
    guiText.text = "East side";
    rot=Quaternion.Euler(15,140,8);
}

```

7.5 The Main Menu

The Graphical User Interface is the means for the user to interact with the application through graphical icons. Our application has a GUI that consists of buttons, sliders, toggle switches and is displayed on the screen throughout the application run. The GUI activates all the functionalities of the application.

Most of the buttons are displayed on the left top, and have descriptive names on them that explain their functionalities. There is the “Explore side” label and underneath it four buttons (“North”, “South”, “East”, “West”), that when pressed we are transporting on the respective side. Underneath those there is a button which controls the sun simulation and another one that when pressed displays a submenu. Next to those are two button on/off toggles that activate or deactivate the functionalities of the measurements display and the raw data display. The appearance of the GUI as well as the functionalities of the buttons are controlled with the **menu_script.js**.

Initially an empty gameObject was created, we named it Menu and the script menu_script.js was attached to it. The rendering and handling of the GUI items is happening in the **OnGUI()** function. Inside it there are other functions that render screen elements. The **GUI.Button()** function makes a press button, the **GUI.Label()** prints text on the screen and the **GUI.Toggle()** creates an on/off toggle button. These trigger actions that are either other functions or give pass variables to the **Update()** function for implementing behavior. On the **Start()** function we initialize the variables and the state of the objects or the components. More specifically, inside the **OnGUI()** function:

```

GUI.skin =menuSkin;
GUI.backgroundColor=Color(1,0.84,0,1);
GUI.BeginGroup(menuAreaNormalized);
GUI.Label(Rect(t),"Explore side:");
GUI.Label(Rect(m),"Display Measurements");
var controller = GameObject.Find("walker");

if(GUI.Button(Rect(nbutton), "North")){////// the user transferes to the north side with the correct rotation
    audio.PlayOneShot(beep);
    isClicked=true;
    target=Vector3(-2.1,-8.8,-33);
    target_rotation=Quaternion(0,0,0,1);
    controller.transform.position = Vector3.Lerp(controller.transform.position,target,1);
    controller.transform.localRotation = Quaternion.Slerp(controller.transform.localRotation,target_rotation,1);
}

if(GUI.Button(Rect(sbutton), "South")){////// the user transferes to the south side with the correct rotation
    audio.PlayOneShot(beep);
    isClicked=true;
    target=Vector3(18,-5.2,47.45);
    target_rotation=Quaternion(0,1,0,0);
    controller.transform.position = Vector3.Lerp(controller.transform.position ,target ,1);
    controller.transform.localRotation = Quaternion.Slerp(controller.transform.localRotation,target_rotation,1);
}

```

We set the appearance of the GUI by determining the **GUI.skin**, which sets its properties on the inspector panel via the **menuSkin** in which the image and the colors of the buttons and the labels are defined. After setting the color and position on the screen via the variable **menuAreaNormalized**, we display text via the **label** function.

The **GUI.Button** takes as arguments the position and a phrase to be displayed on the button and returns true if the user clicks the button. When each button is clicked there is a sound played via the **audio.PlayOneShot()** function that activates the **Audio Source** component. For each button there is a functionality that is implemented.

7.5.1 Transportation functionality

The “North”, “South”, “East”, “West” buttons when pressed they transport the controller to the respective side in order to have faster access and avoid walking around the monument. As is shown in the segment of code above, when the user clicks one of those buttons a sound is played and the variable “isClicked” is given a value. That variable is accessed by the script “MouseLook2.js” to signal that the button is pressed. Then we give the controller the desired position and rotation via the Lerp() and Slerp() functions which interpolate between the first two arguments by the fraction in the third argument. In our case by giving it 1 we transporting the position and the rotation in the second argument’s value. The same actions are applied to all four buttons.

The change in the position is remembered when we are transporting to the sides of the building. We also want each time we transport to a side to always face the building, otherwise it can become very disorienting. Giving the correct orientation to the controller in the button functions above, is not enough because the script that controls the camera rotation which was imported in the first person controller does not remember the rotation given by the buttons. Therefore we have to change it. We are naming in **MouseLook2.js**. In the Start() function we initialize the variables that we use appropriately. In the Update() we read the input from the mouse and depending on the axis we rotate the camera appropriately. Specifically :

The Input.GetMouseButtonDown() function detects the mouse click. 0(zero) is for left click and 1 is for right click. When the mouse is pressed for a long time (*click==true && (Time.time - temps) > 0.3*), i.e. pressed down continuously, we check around which axis to do the rotation and if one of the transport button is pressed (we check the *clickCheck* variable which has been passed a value from the *menu_script.js*), we save the rotation angle on a variable. Then we read the value of the input axis continuously and we rotate around the chosen axis, while updating with the *originalrotation* variable the current rotation angle of the camera.

If we release the button (*Input.GetMouseButtonUp(1)*) quickly, that is we click only once then the rotation of the camera remains the same and doesn’t change. The function *Adjust360andClamp()* prevents the rotation angle from going beyond 360 degrees and also clamps the angle to the min and max values set in the Inspector.

```

function Update () {
    if ( Input.GetMouseButtonDown (1) ){// activates rotation with right click
        temps = Time.time ;
        click = true ;
    }
    if ( click==true && (Time.time - temps) > 0.3){ // checks if the button is pressed down continuously
        var clickCheck=GameObject.Find("Menu").GetComponent("menu_script").isClicked;// checks if the transport buttons are clicked
        var controller= GameObject.Find("walker");
        if (Axis == Axes.MouseXandY){
            if (clickCheck){
                GameObject.Find("Menu").GetComponent("menu_script").isClicked=false;
                // takes the current rotation (it changes when the buttons are pressed)
                // the rotation is determined in the menu_script
                originalrotation=controller.transform.localRotation;
            }
            // Read the mouse input axis
            rotationX += Input.GetAxis("Mouse X") * sensitivityX;
            rotationY += Input.GetAxis("Mouse Y") * sensitivityY;
            // Call our Adjust to 360 degrees and clamp function
            Adjust360andClamp();
            var xQuaternion = Quaternion.AngleAxis (rotationX, Vector3.up); // rotates rotationX degrees around y axis
            var yQuaternion = Quaternion.AngleAxis (rotationY, Vector3.left); // rotates rotationY degrees around x axis
            transform.rotation = originalrotation*xQuaternion*yQuaternion;
        }else if (Axis == Axes.MouseX) {
            if (clickCheck){
                GameObject.Find("Menu").GetComponent("menu_script").isClicked=false;
                originalrotation=controller.transform.localRotation;
            }
            // Read the mouse input axis
            rotationX += Input.GetAxis("Mouse X") * sensitivityX;
            Adjust360andClamp();
            transform.localRotation = originalrotation*Quaternion.AngleAxis (rotationX, Vector3.up); // rotates rotationX degrees around y axis
        }else{
            if (clickCheck){
                GameObject.Find("Menu").GetComponent("menu_script").isClicked=false;
                originalrotation=controller.transform.localRotation;
            }
            // Read the mouse input axis
            rotationY += Input.GetAxis("Mouse Y") * sensitivityY;
            transform.localRotation = originalrotation*Quaternion.AngleAxis (rotationY, Vector3.left); // rotates rotationY degrees around x axis
        }
    }
    if( Input.GetMouseButtonUp(1)){
        click=false;
        if ( (Time.time - temps) <= 0.3){// if we press a quick right click then the walker stays firm
            var cam2= GameObject.Find("Main Camera");
            var controller2= GameObject.Find("walker");
            if (Axis == Axes.MouseX) {
                transform.localRotation.x=controller2.transform.localRotation.x;
            }else if (Axis == Axes.MouseY) {
                transform.localRotation.y=cam2.transform.localRotation.y;
            }
        }
    }
}

```

Now when the buttons are pressed we are transporting correctly in the designated place with the correct rotation angle and maintaining our navigation and camera rotation smooth.

7.5.2 Sun simulation functionality

In order to achieve photorealism and give an essence of realism to the building, we need to light it. The proper illumination is what enhances the textures and makes the normal mapping visible. The lighting makes the surfaces look realistic and emphasizes the details on them. Additionally, the lighting creates shadows which are important in the real life look we are trying to achieve. It is nice to have the opportunity to observe the building under different lighting conditions, to see the walls under shadows or light and this is what is implemented in this functionality.

When the button is pressed the simulation begins and the light source goes in a trajectory around the building. When is pressed again the simulation stops. The button

changes image depending on the state. If the simulation is not activated the button displays the play sign, when it is active the pause sign is displayed.

To achieve this we created an empty gameObject the ‘sun’ and we place it in the centre of the scene, then we created a sphere object and we made it a child of the sun. Children objects are linked to their parent object and follow their parents transformation. We positioned the sphere object above the sun object and slightly sideways. Lastly we created a Directional light and made it child of the sphere object.

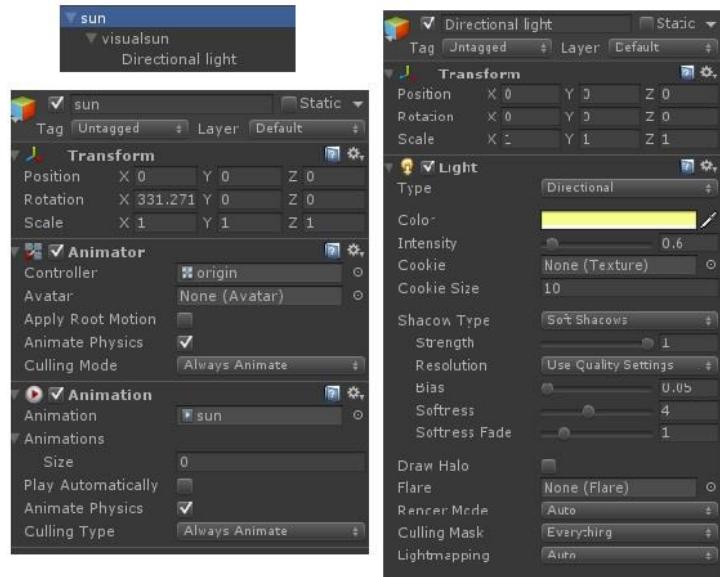


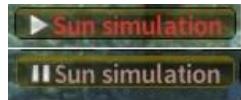
Figure 7.9 Top left: sun parent object with the children objects. the components of the sun and directional light object are shown.

We gave the directional light appropriate settings and we attached to the parent object an animation component. Animations are used to automate movement in a game. We created an animation clip named ‘sun’ and added it in the animation component. The animation clip was created in the animation tab and we defined the y rotation of the sun to change from 0 to 360 in 20 seconds. We set it to run on loop on the inspector tab.

Now that we have the components ready, we need to connect this with the appropriate button. Inside the menu script on the OnGUI function we add the following code:

```
if(GUI.Button(Rect(sim),content)){// button for sun simulation
    audio.PlayOneShot(beep);
    var origin = GameObject.Find("sun");
    var anim = origin.GetComponent();
    if (content.image==image_play){// toggle image on the button
        content.image=image_pause;
    }
    else{
        content.image=image_play;
    }
    anim.enabled=!anim.enabled; // start/stop the animation
}
```

The button displays a GUIcontent on it, that means that it can contain elements such as images or text. If the button is pressed, a sound is played and the image content changes depending on the state. The animation is enabled or disabled appropriately.



We have implemented a virtual sun that goes in circle above and around the building at our command. When the simulation stops, then the next time it begins it continues from where it has stopped.

7.5.3 Display measurements functionality

One of the things we considered ideal for our application is to be able to display the measurements we took on the field upon the monument. In this way the user will be able to see the data on the surface, and which measurement represents which point on the surface. It is a nice visual way of connecting the geographical data with the 3D model.

The measurements are displayed with little green cubes. Each cube represents the point where the total station shoot its beam and took measurements. Each one depicts 3D geographic coordinates. Additionally, the user is given the ability to click on any cube and the respective geographical coordinates be displayed.

7.5.3.1 Data Storage

In order to access and manipulate the data we have to find a way of storing or importing them in Unity. The data are now on a csv file. We have to find a way of storing them locally and be independent of files.

SQLite provides that way. SQLite is a software library that implements a self-contained, serverless, SQL database engine. [59] SQLite provides local data storage for individual applications and devices. It is a convenient way of implementing a simple database in Unity. Rather than a client-server based implementation of SQL, SQLite uses a single local file to store the database, and provides access to that file via standard SQL commands.

We decided to store our data in a SQLite database. We pass the data from the file to the database (once) and thus our application interacts only with the database and is made independent of the file that is stored on a computer. This way the application is portable and can run on other computers while maintaining the data.

In order to implement the database in our project we have to add the javascript class in our project. [60] The class is in the dbAccess.js script. The commands run by the dbAccess class are IDbCommand commands, and those commands return an IDataReader object. The IDbCommand objects represents an SQL statement that is

executed while connected to a data source and the `IDataReader` object provides a means of reading one or more forward-only streams of result sets. The class functions are presented next. All of them are in the `dbAccess.js` script.

```
class dbAccess {
    // variables for basic query access
    private var connection : String;
    var dbcon : IDbConnection;
    private var dbcmd : IDbCommand;
    private var reader : IDataReader;

    function OpenDB(p : String) {
        connection = "URI=file:" + p; // we set the connection to our database
        dbcon = new SqliteConnection(connection);
        dbcon.Open();
    }
}
```

We create a `IDbConnection` object which represents a connection to a data source. The function `OpenDB()` opens and connects to a database specified by a name.

```
// This function deletes all the data in the given table.
function DeleteTableContents(tableName : String) {
    var query : String;
    query = "DELETE FROM " + tableName;
    dbcmd = dbcon.CreateCommand();
    dbcmd.CommandText = query;
    reader = dbcmd.ExecuteReader();
}

function CreateTable(name:String, col:Array, colType:Array) { // Create a table, name, column array, column type array
    var query : String;
    query = "CREATE TABLE "+name+"("+col[0]+")"+colType[0];
    for(var i=1; i<col.length; i++) {
        query += ", " + col[i] + " " + colType[i];
    }
    query += ")";
    dbcmd = dbcon.CreateCommand(); // create empty command
    dbcmd.CommandText = query; // fill the command
    reader = dbcmd.ExecuteReader(); // execute command which returns a reader
}
```

The function `DeleteTableContents()` creates and executes a query to delete all rows from a table specified by `tableName`. The `CreateTable()` function creates and executes a query that creates a table with a specified name and column names and types which are defined by given arrays.

```
function InsertInto(tableName : String, values : Array) { // basic Insert with just values
    var query : String;
    query = "INSERT INTO " + tableName + " VALUES (" + values[0];
    for(var i=1; i<values.length; i++) {
        query += ", " + values[i];
    }
    query += ")";
    dbcmd = dbcon.CreateCommand();
    dbcmd.CommandText = query;
    reader = dbcmd.ExecuteReader();
}
```

The function `InsertInto()` creates and executes a query that inserts an array of values on a specified table.

Now we have a class in our project with the appropriate functions to implement the database.

7.5.3.2 Initialization of the database

To work with the database we have to initialize it first. We access the file with our raw data, we read it and insert the data in the database. This happens only once, in the development stage when we know the datapath of the file and is on the same computer. When the standalone application is build the database is already filled and we can run the application on any computer.

The script **Read_data.js**, which is a component to the ‘neoria’ gameObject, contains the function to initialize the database.

```
function Start() {// Database and table creation
    db = new dbAccess();
    db.OpenDB( Application.dataPath + "/StreamingAssets/" + DatabaseName );
    var tableName = TableName;
    var columnNames = new Array("pointID", "easting", "northing", "level");
    var columnValues = new Array("text", "text", "text", "text");
    try {
        db.CreateTable(tableName, columnNames, columnValues);
    }
    catch(e) {
    }
}

function InitializeDB()// initialization of the database. Reads the data from the file and inserts them in the database
{
    db.DeleteTableContents(tableName);
    var num_of_lines :int;
    var lines : String[];
    var prefab :GameObject;
    var fileData : String = System.IO.File.ReadAllText("C:/Users/FROSO/Desktop/arxeia_metrhsewn/OLES_OI_METRHSEIS_ME_THN_SEIRA.csv");
    num_of_lines = fileData.Split("\n"[0]).length-1;
    lines = fileData.Split("\n"[0]);
    for (var i = 0; i < num_of_lines;i++){
        var lineData : String[] = (lines[i] .Trim()).Split(";"[0]);
        var values = new Array("'" +lineData[0] +"'", "'"+lineData[1] +"'", "'"+lineData[2] +"'", "'"+lineData[3] +"'");
        db.InsertInto(tableName, values);
    }
}
```

In the Start() function we create a new dbAccess class object and we call the OpenDB() to create the database. Then we create arrays with the names of the columns and their types and pass them in the createTable() function in order to create the table. In our case the columns are the point id , the easting coordinate, the northing coordinate and the level (height). The function InitializeDB() reads the file and inserts the data in the database. More specifically, after we clear the database with the DeleteTableContents() function, we access the file on the specified datapath, we separate each line and then for each line in the file we separate the values and pass them in the database in the appropriate columns with the InsertInto() function. The InitializeDB() is called in the Start() function on the menu_script. After the database is initialized it is placed in comments.

7.5.3.3 Display measurements

In the Read_data.js we created a function called InstantiateMeasurements(). In that we read the data from the database and for each row a cube is created and positioned in the specified place by the geographic coordinates that are represented by the data. As can be seen in the section below we create an query, we executed it and we are returned the resulting records. Each record represents a position in space whose

absolute values are very big. In order to fit our scene's coordinate system we have to translate it. We add or subtract the right absolute values in order for the 3D point to be suitable for the scene's coordinate system.

```
function InstantiateMeasurements(mylist:List<GameObject>){/// displays the measured data as cubes
    var prefab:GameObject;
    prefab=GameObject.Find("Cubew");
    var dbcmd = db.dbcon.CreateCommand(); // create empty command
    var sqlCommand = "SELECT * FROM CoordinatesTable";
    dbcmd.CommandText = sqlCommand; //define the command
    var reader = dbcmd.ExecuteReader(); // executes command which returns a IDataReader object with all the records
    while (reader.Read()){// reads a row at the time
        prefab.active=true;
        var pos = Vector3 (-reader.GetFloat(1)+501695.7833+87.23, reader.GetFloat(3)-10.72520026, -reader.GetFloat(2)+3930158.887+63.2);
        var ob=Instantiate(prefab, pos, GameObject.Find("GUI_Text_coord").GetComponent(Coordinates_script).rot);
        ob.name= reader.GetString(0);
        mylist.Add(ob); //adds the created objects in a list
    }
}
```

Then we create clones of the Cubew object with the instantiate function on the places defined by the read data. We name the created clone objects by their id and keep them on a list for future reference.

Back to the menu_script.js.

```
if (GUI.Toggle(Rect(checkbox), ButtonToggle, "") != ButtonToggle){// toggle for display measurements
    ButtonToggle = !ButtonToggle;
    if (ButtonToggle){
        var link=GameObject.Find("neoria").GetComponent(Read_data); // accessing the Read_from_csv_file script
        link.InstantiateMeasurements(mylist); //calls the InstantiateMeasurements function
    }else{
        for (var i=0;i< mylist.Count;i++){// toggle off: destroy the objects
            Destroy(mylist[i]);
        }
        var gameObjects : GameObject[]; // if there are any open 3D text coordinates, they are destroyed
        gameObjects = GameObject.FindGameObjectsWithTag("cube text");
        for(var f = 0 ; f < gameObjects.length ; f ++){
            Destroy(gameObjects[f]);
        }
    }
}
```

We create an on off toggle button. When the button is activated, it accesses the Read_data.js script and calls the InstantiateMeasurements() function which creates the cubes in the correct places. If the button is switched off then we take the list with the created objects and delete its content, in order to stop displaying the objects. The rest of the code will be explained in the next section.

7.5.3.4 Clickable measurements

Having displayed the measured points, we want to give the ability to click on a point and display the spatial information, i.e. the easting, northing and level coordinates. To do this we created a script called **clickMeasurement.js** and we attached it to the Cubew object. In the Update() function of that script we detect when the object is clicked, we find that object's information from the database and we display it. When that same object is clicked again the information disappears.

First we need to detect when the mouse clicks on the cube. With the function OnMouseOver() we activate the commands when the mouse is on the spatial area of the cube, then when we click it a boolean variable is updated and sends that information in the Update()

```
function OnMouseOver(){
    if ( Input.GetMouseButton(0) ){
        clickmeasure=!clickmeasure;
    }
}
```

function.

```

function Update () {
    if (clickmeasure){// displays the data when a green cube is clicked once
        if (once){
            var prefabtext :GameObject;
            prefabtext=GameObject.Find("3dText");
            var link=GameObject.Find("meoria").GetComponent(Read_data).db;
            var dbcmd = link.dbcon.CreateCommand();
            var sqlQuery = "SELECT * FROM CoordinatesTable where `pointID`='"+this.name;
            dbcmd.CommandText = sqlQuery;//query returns the data of the clicked measurement
            dbcmd.ExecuteNonQuery();
            var reader = dbcmd.ExecuteReader();
            check_true=true;
            reader.Read();
            var pos = Vector3 (-reader.GetFloat(1)+501695.7833+87.23, reader.GetFloat(3)-10.72520026, -reader.GetFloat(2)+3930158.887+63.2);
            prefabtext.GetComponent(TextMesh).text=reader.GetString(1)+"\n"+reader.GetString(2)+"\n"+reader.GetString(3);
            obtext=Instantiate(prefabtext, pos, GameObject.Find("GUI Text_coord").GetComponent(Coordinates_script).rot);
            obtext.name= "mes"+reader.GetString(0);
            obtext.tag="cube text";
            k=reader.GetString(0);
            k=reader.GetString(0);
            once=false;
        }
    }else{// if is clicked again then the display disappears
        if (check_true){
            once=true;
            var textob=GameObject.Find("mes"+k);
            Destroy(textob.gameObject);
            check_true=false;
        }
    }
}

```

In the Update() when the first click on a object occurs, we connect to the database and run a query searching for its information based on the point id. As it was mentioned beforehand the objects were named by their id, so the query will return that id's coordinate information. Having created an object to display 3D text, we clone it in the position provided by the coordinates, we define the text to be displayed and we give it the appropriate rotation which we have taken from the Coordinates_script.js as it was described in the section 7.4. We need a different rotation angle for each side in order to make the 3D text visible and not be hidden in the model. We give each text object a name based on its point id and we tag it appropriately for future reference. Tags are intended to identify GameObjects for scripting purposes. When the cube is clicked a second time, the boolean variable on the function OnMouseOver() changes and different actions are triggered. In this case we look for the object based on the name we gave it beforehand and we destroy it. In this way we have created an activity for opening and closing the displayed information.

When the displayed measurement functionality is closed, apart from the cubes we want every displayed information to be closed too. We do this by finding all the 3D text objects, which we have tagged appropriately and destroy them. This piece of code is in the last four lines of the menu_script.js part, shown in the previous section (7.5.3.3).

The objects are cloned with their scripts, each object has its own script and its functions run for that particular object only. Because of this, the events regarding each object do not get confused and we have a smooth implementation.

7.5.4 Display Raw Data functionality

Having implemented the functionality where we click at a point and the coordinates are displayed, we want to create the reverse activity where we have the coordinates available to us and when we click on a coordinate, the point corresponding to it is displayed on the model. What we want to do, is display the contents of the database on the screen, create one clickable button for each coordinate and when we click it to

inform us in which side the corresponding point is and display it on the surface. For that functionality we created a script called **viewRawData().js** and we attached it to our main model. That script contains only an **OnGUI()** function because we want to have the information displayed on screen. Inside that function we put our code.

```

GUILayout.BeginArea(Rect(840, 100, 400, 500)); //we define the gui display area
scrollPosition = GUILayout.BeginScrollView(scrollPosition, GUILayout.Height(500)); //we create a scroll window
var dbcmd = db2.dbcon.CreateCommand();
var sqlQuery = "SELECT * FROM CoordinatesTable"; // we access data from the database
dbcmd.CommandText = sqlQuery;
var reader = dbcmd.ExecuteReader();
var prefab :GameObject;
prefab=GameObject.Find("ViewDataSphere");

while (reader.Read()){
    GUILayout.BeginHorizontal(); //in every line there is a button and the information next to it
    if (GUILayout.Button (reader.GetString(0), GUILayout.Width(44))) { //the button has the point id on it
        k=reader.GetInt32(0); //if its clicked a sphere appears in the correct position on the model
        if (GameObject.Find("sphere"+reader.GetString(0))==null){
            prefab.active=true;
            var pos = Vector3 (-reader.getFloat(1)+501695.7833+87.23, reader.getFloat(3)-10.72520026, -reader.getFloat(2)+3930158.887+63.2);
            var ob=Instantiate(prefab, pos, GameObject.Find("GUI_Text_coord").GetComponent(Coordinates_script).rot);
            ob.name= "sphere"+reader.GetString(0);
            mylist.Add(ob);
        }
        else{
            var textob=GameObject.Find("sphere"+k);
            Destroy(textob.gameObject);
        }
    }
    //the information displayed next to the button
    GUILayout.Label(reader.GetString(1)+" "+reader.GetString(2)+" "+reader.GetString(3) ,localStyle,GUILayout.Width(325));
    GUILayout.EndHorizontal();
}
GUILayout.EndScrollView();
GUILayout.EndArea();
}

```

As is shown above we created a GUI area on screen. We connected with the database and executed a query retrieving all the rows. We have created a scroll window and inside it, for each retrieved row we create a button with the point id on it. Next to the button we display the corresponding information. When a button is clicked we check to see if the sphere object representing the point exists, and if not we clone a sphere in the position appointed by the button information. The sphere is appointed a name for reference and stored on a list. When we click a button again and the object already exists we destroy it, creating an open/close activity.

In order to be more helpful to the user, when a button is clicked we display text informing the user in which side the clicked point is. This way the user can quickly transport to that side. This is implementing by storing in a variable the id of the retrieved row, and depending on its value, we recognize in which side it belongs and display the appropriate information.

pointID	easting	northing	level
63	501753.4333	3930192.504	6.672056271
64	501751.7389	3930192.327	6.725322618
65	501759.2264	3930193.577	8.332736531
66	501759.8912	3930193.716	8.540181318
67	501760.4688	3930193.816	8.281400678
68	501760.4546	3930193.867	6.67588849
69	501759.2386	3930193.591	6.645607224
70	501758.546	3930193.506	6.293892145
71	501760.7225	3930193.925	6.33706209
72	501760.7242	3930193.904	6.248273271
73	501758.5545	3930193.506	6.200280275
74	501757.7996	3930193.329	10.58570091
75	501758.0325	3930193.32	10.5889474
76	501758.0671	3930193.336	10.344435
77	501757.8502	3930193.292	10.3681485
78	501759.3104	3930193.569	10.5747853
79	501759.5864	3930193.62	10.6001626
80	501759.5657	3930193.63	10.31465413
81	501759.3434	3930193.576	10.32241426

Figure 7.10 The GUI of the database as is displayed on screen.

```

GUI.Label(Rect(835, 75, 400, 30), "pointID      easting      northing      level");
if (k>=1 && k<=252){// depending on which button is clicked a label informing in which side the point is appears
    GUI.Label(Rect(550, 570, 400, 30), "This point is on the south side.",localStyle);
}else if((k>252 && k<=296) || k==300 || k==302){
    GUI.Label(Rect(550, 570, 400, 30), "This point is on the east side.",localStyle);
}else if((k>296 && k<=528 && k!=300 && k!=302) || (k>=553 && k<=559)){
    GUI.Label(Rect(550, 570, 400, 30), "This point is on the north side.",localStyle);
}else if((k>528 && k<=552) || (k>=560 && k<=838)){
    GUI.Label(Rect(550, 570, 400, 30), "This point is on the west side.",localStyle);
}

```

In the menu_script.js we have created an on/off toggle button implementing that functionality.

```

var raw =GameObject.Find("neoria").GetComponent(viewRawData);
GUI.Label(Rect(e),"Display Raw Data");
if (GUI.Toggle(Rect(checkbox_data), ButtonToggle_data, "") != ButtonToggle_data){// toggle for display raw data
    ButtonToggle_data = !ButtonToggle_data;
    if (ButtonToggle_data){
        raw.enabled=true;// activates view raw data script
    }else{
        raw.k=0;
        raw.enabled=false;
        for (var j=0;j< raw.mylist.Count;j++){// destroy displayed objects
            Destroy(raw.mylist[j]);
        }
    }
}

```

When the toggle button is on the viewRawData.js script is activated and the database is displayed. When it is off the script deactivates and every object that may have been created is destroyed (by accessing the object list).

In order to enhance the sphere's visibility and making it easier to be detected, we implemented a pulsing functionality. The sphere has a blinking lighting effect around it and draws the attention of the user. In order to achieve this we attach a halo component on it. It creates a halo effect around the object. we also created a script called **pulse.js** to implement the blinking activity.

```

function Update () {
    ch!=ch;
    PulseLight();
}

function PulseLight(){
    if (ch){
        (this.GetComponent("Halo")as Behaviour).enabled=true;
    }else{
        (this.GetComponent("Halo")as Behaviour).enabled=false;
    }
}

```

In that script we created a function called PulseLight() which enables or disables the halo based on a Boolean variable. In the Update() change that variable and call the latter function. Because the Update runs in every frame, the variable changes in every frame and the light blinks with the frame rate.

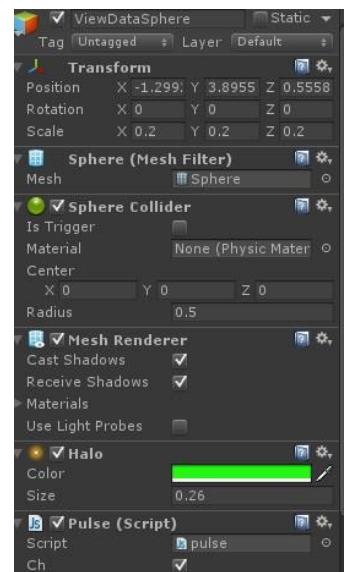


Figure 7.11 The sphere object components.

7.6 Display coordinates dynamically

A feature that the application provides to the user is the continuous display of the geographical coordinates on the bottom of the screen. When the mouse point is on the vicinity of the monument, then the coordinates of the point that the mouse shows appear on the bottom of the screen. Those coordinates change dynamically whenever the mouse moves.



Figure 7.12 The mouse points at a spot and its coordinates are displayed on the bottom.

The creation of that feature is achieved with the method of raycasting. In Unity any point in the camera's view corresponds to a line in world space. Unity provides a mathematical representation of that line in the form of a Ray object. The ray corresponds to a point in the view and is implemented with the ScreenPointToRay() function which returns a ray going from camera through a screen point. The ray consists of a point of origin and a vector which shows the direction of the line from that origin.

The most common use of a ray is to perform a raycast into the scene. A raycast sends an imaginary “laser beam” along the ray from its origin until it hits a collider in the scene. Information is then returned about the object and the point that was hit in a RaycastHit object. This is a very useful way to locate an object or a point based on its onscreen image.

This method helped us implement that feature. We cast a ray from the camera into the scene, we detected when it hit a collider and we were returned the point of impact in world space.

This was implemented in the **functionality.js** script. That script was attached to the walker.

```

function Update () {
    coord=GameObject.Find("x/y coordinates");
    var s=GameObject.Find("Menu").GetComponent(menu_script).submenu_obj;
    var guilTextObject = GameObject.Find("GUI Text_coord");
    ////////////////////COORDINATES DISPLAY IN RUNTIME
    var hit: RaycastHit;
    var ray : Ray = camera.main.ScreenPointToRay (Input.mousePosition);
    Debug.DrawRay (ray.origin, ray.direction * 20, Color.yellow); // a ray is drawn for development purposes
    if (Physics.Raycast(ray, hit)) { // returns true when the ray inserts a collider
        // checks if the ray hits the building or one of the imported objects
        if (hit.collider.gameObject.name=="necria" || hit.collider.gameObject.name=="Stexture_corrected"
            || hit.collider.gameObject.name=="7notia_restore" || hit.collider.gameObject.name=="7porta"
            || hit.collider.gameObject.name=="7testfordropdown" || hit.collider.gameObject.name=="rooft5_restore"){
            resx=501695.7833+87.23-hit.point.x;
            resy=hit.point.y+10.72520026;
            rezx=3930158.887+63.2-hit.point.z;
            coord.guilText.text="easting:"+resx.ToString("F4")+" northing:"+rezx.ToString("F4")+" height:"+resy.ToString("F4");
        }else{
            coord.guilText.text=" ";
        }
    }
}

```

Inside the `Update()` function, we created a ray with the `ScreenPointToRay()` function going from the camera through the mouse position x, y pixel coordinates on the screen. The resulting ray is in world space. We drew that ray on the scene view for development purposes. Using the `Raycast()` function we detect if the ray intersects any collider. The information is sent to us with the `hit` variable which is used to get information back from a raycast. When the ray hits a collider that is distinguished by name we take the point impact information with the `hit.point` variable. That information is properly transformed to depict the corresponding geographical coordinates and is displayed on the screen via a `GUIText` object.

This provides a way to have a continuous dynamic depiction of the coordinates of the mouse point at any given time.

7.7 The Submenu

The final functionality that the main menu provides, is the appearance of another menu, the submenu. When the button of the submenu is pressed, new buttons appear which implement new functionalities. The submenu is implemented in the same way as the main menu. We created an empty object called the submenu and attached a script to it. In that script we implement an new GUI with buttons. The script which controls the submenu is the `submenu_script.js`.

Inside the `menu_script.js` we detect when the submenu button is pressed and in the `Update()` function we enable or disable the submenu object, depending on that detection.

```

if(GUI.Button(Rect(submenu), submenu_content)){ // submenu button
    audio.PlayOneShot(beep);
    isClickedsub=!isClickedsub;
}

function Update(){
    if (isClickedsub){ // activates submenu
        submenu_obj.active=true;
    }else{
        submenu_obj.GetComponent(submenu_script).isClicked=false;
        submenu_obj.GetComponent(submenu_script).isClickedinfo=false;
        submenu_obj.active=false;
    }
}

```

The submenu consists of three buttons and provides three functionalities. A functionality that displays hotspots (which are clickable objects) around the building in special places. When clicked the appearance of that part of the monument changes. A functionality that displays information about the monument and a functionality that measures distances on the building.

7.7.1 Alteration functionality

As it was mentioned when this functionality is enabled hotspots appear. These hotspots are clickable objects that when clicked, the surface of the monument on the area they are, changes. In that way the user can alter the geometry provided and visualize it in different ways.

In the **submenu_script.js** inside the `OnGUI()` function we created a button named Show Hotspots. When clicked the appropriate actions are triggered on the `Update()` function.

In the `Update()` function when the button is clicked once the hotspots are activated when is clicked again the hot spots are deactivated along with the alterations that have happened in the building.

The hotspots are sphere objects that we have created and we have positioned them in places where we want to implement a change. There are five hotspots.

```

        GUI.BeginGroup(menuAreaNormalized);
        if(GUI.Button(Rect(hotspotsbutton), "Show Hotspots")){
            audio.PlayOneShot(beep);
            isClicked!=isClicked;
        }

        function Update(){
            if (isClicked){// activates the hotspots
                hotspot.active=true;
                hotspot2.active=true;
                hotspot3.active=true;
                hotspot4.active=true;
                hotspot5.active=true;
            }else {
                hotspot.active=false;// hides the hotspots along with the objects that have appeared
                hotspot.GetComponent(hotspots).plan.renderer.enabled=false;
                hotspot2.GetComponent(hotspots2).plan2.renderer.enabled=false;
                hotspot3.GetComponent(hotspots3).plan3.renderer.enabled=false;
                hotspot4.GetComponent(hotspots4).plan2.renderer.enabled=false;
                GameObject.Find("neoria").renderer.materials[0].shader=Shader.Find("Bumped Diffuse");
                hotspot5.active=false;
            }
            if (isClickedInfo){// activates the info plane
                info_obj.active=true;
            }else{
                info_obj.active=false;
            }
        }
    
```

Hotspot1

The first hotspot is represented by the object called Sphere and it is placed on the north side of the building in front of the eastern dome. That dome is painted in parts white with modern materials. We want to replace those materials with other ones identical to the neighbouring parts.

In order to do that we created an object representing the façade of that dome. That object consists of the faces of the façade of the seventh dome, but is textured differently. The white parts are covered with texture similar to the rest of the surface.



Figure 7.13 The object used to alter the appearance of the dome. The white parts are gone, and are textured with similar looking material.

We imported that object in our project and placed it exactly where the corresponding part of the monument is. We disabled the mesh renderer so that it would not be visible.

In the sphere object we attached a script called **hotspots.js**. In the `OnMouseOver()` function of that script we detect when the mouse is on the vicinity of the sphere and when the sphere is clicked a variable informs the `Update()` function whether the imported object's mesh renderer should be enabled. When a click happens in the already open hotspot, the object disappears by disabling its mesh renderer.

The hotspots is blinking in order to be more distinguished. The blinking function was implemented in the same way as it was mentioned in the previous chapter by the `PulseLight()` function. It blinks in the same speed as the frame rate.

Hotspot2

The second hotspot was placed on the south side of the building in the eastern dome. That part of the building is heavily damaged. The roof has collapsed and the whole top of that dome is missing. We want to give the ability to the user to visualize the missing roof and toggle between the current and 'restored' surface. It should be mentioned that due to the lack of architectural information, the

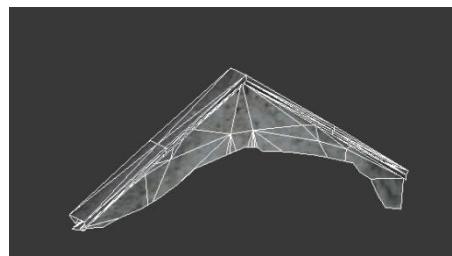
```

function Start () {
    plan=GameObject.Find("testfordropdown");
    ch=true;
}

function Update () {
    ch=!ch;
    PulseLight();
    if (clickhot) {
        plan.renderer.enabled=true;
        plan.renderer.material.color.a=0.7;
    }else{
        plan.renderer.enabled=false;
    }
}

function OnMouseOver(){
    if ( Input.GetMouseButtonDown (0) ) {
        audio.PlayOneShot(beep);
        clickhot=!clickhot;
    }
}

```



restored surface was created based on the appearance of the non-damaged parts, as well as the measurements of structural elements of neighboring areas.

We created an object of the missing roof based on the aforementioned information. It is designed to complement the current surface the best way possible. We imported it into the scene and placed it in a way that fits the current surface. We disabled the mesh renderer in order to make it invisible. When is visible is rendered in red color.

Following then same workflow as the first hotspot, we created a sphere object and placed on the spot mentioned above. We attached a script called **hotspots2.js** to it. The same implementation as the first hotspot was followed. By detecting the click on the sphere object the mesh renderer is enabled or disabled making the roof appear or hide.

Figure 7.14 The object representing the missing roof.

```
function Start () {
    plan2=GameObject.Find("7notia_restore");
    ch=true;
}

function Update () {
    ch!=ch;
    PulseLight();
    if (clickhot){
        plan2.renderer.enabled=true;
    }else{
        plan2.renderer.enabled=false;
    }
}

function OnMouseOver(){
    if ( Input.GetMouseButtonDown (0) ){
        audio.PlayOneShot(beep);
        clickhot!=clickhot;
    }
}
```

Hotspot3

The third hotspot deals with the missing roof edge of the fifth dome (counting from the west) on the south side. Exactly as it was implementing for the second hotspot, the clicks on the corresponding sphere trigger the appearance or disappearance of the missing roof edge. The missing roof edge was created in the same way as the roof previously and imported and placed in the correct place in the scene.



Figure 7.15 The part of the building as is currently (left) and with the missing roof edge (right).

Hotspot4

The fourth hotspot has to do with the alteration of textures. There are places in the current monument that are covered with modern materials such as cement. One such surface is the façade of the third (counting form the east) dome on the north side. Its

appearance is heavily altered. The whole façade is covered with cement and modern bricks. It would be nice if the user can visualize it with textures similar to the neighboring surfaces.

The texture change is different from the procedure followed above. We want to keep the mesh the same and change the textured applied on it in realtime. Firstly, we painted a new texture with the same UV mapping for the damaged façade. It was created with painted details derived from the non-damaged areas. A normal map was also created to accompany that new texture. Inside Unity we created a new material with a bumped diffuse shader. The new texture and its normal map where placed on the corresponding slots of that material.

Our main 3D model has a list of materials with which is textured. That list is in its mesh renderer component. What we want to achieve is to change that list during runtime with another that has on the appropriate slot the new material we created.

Having created a sphere object to use as a hotspot on the correct spot we attached the script **hotspots4.js** to it. In that script we defined two material lists. One that holds the current materials and one that has the new material in it. The new list was filled with the appropriate materials in the inspector. When the script is first called, i.e. in the `Start()` function, we store the current materials on a list.

When the hotspot is clicked for the first time the material list of the mesh renderer changes with the new one. When is clicked for the second time the material list is replaced with the initial one. That way the texture appearance changes and the user can toggle between textures.

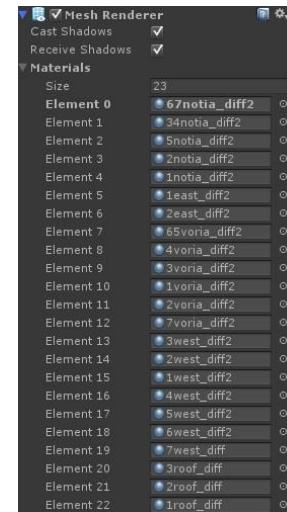


Figure 7.16 The material list of the 3D model.

```
var mater: Material[];
var intMaterials: Material[];

function Start () {
    plan=GameObject.Find("neoria");
    intMaterials=plan.renderer.materials;
    plan=GameObject.Find("Stexture_corrected");
    ch=true;
}

function Update () {
    ch=!ch;
    PulseLight();
    if (clickhot){
        plan.renderer.materials=mater;
    }else{
        plan.renderer.materials=intMaterials;
    }
}

function OnMouseOver(){
    if ( Input.GetMouseButtonDown ( 0 ) ){
        audio.PlayOneShot(beep);
        clickhot=!clickhot;
    }
}
```

Hotspot5

The fifth hotspot alters the appearance of the seventh dome in the south side. It is the dome that has the collapsed roof. We see that the door in that façade is blocked and covered with brinks. We want to create a door opening as we imagined was in the past. We created an object identical to that particular part of the building but with a door opening. We imported it and placed it in the spot of the current part.



Figure 7.17 The created object with the door opening. The newly created faces are rendered in red.

Following the previous steps we created a sphere object and placed it in front of that part. We attached a script called **hotspots5.js** to it. In this case a simple toggle of the mesh renderer of the imported object is not suitable, because there is the old surface underneath and intervenes with the one. The blocked door is still visible when the new surface is rendered. What we did to prevent that is a combination of the previous hotspot activities. When we click the hotspot we want the old surface disappear and the new object appear.

In order to do that we need to change the material of that part of the monument and make it completely transparent. The faces that are covered with that material become invisible and the new object can be rendered in that invisible part. The faces of the new object match the faces that are covered with the material that is going to be transparent.

Having kept the old material list in a variable, we develop the usual clicking functionality. When the hotspot is clicked, the first thing we do is to check if the fourth hotspot might be on. This is checked because the previous functionality changes the material list. If the fourth hotspot is on then the list is different. Depending on whether the previous hotspot is on or off we manipulate the corresponding list. We enable the mesh renderer of the new object and then we change the shader of the material on that part to a transparent/diffuse shader. The transparency of the shader depends on the value of the alpha channel of the texture. To be completely invisible we set that value to zero. Now the old part of the model is completely transparent and the new part is displayed over it.

```

function Start () {
    plan2=GameObject.Find("porta");
    ch=true;
    m=GameObject.Find("neocia");
    intMaterials=m.renderer.materials;
}
function Update () {
    intMaterials=m.renderer.materials;
    var hot4=GameObject.Find("Sphere4").GetComponent(hotspots4);
    var r=hot4.mater;
    ch=ch;
    PulseLight();
    var shaderNoOutline = Shader.Find("Transparent/Diffuse");// finds a new transparent shader
    if (clickhot){
        if (hot4.clickhot){//checks to see if the hotspot is activated and takes the appropriate material list
            plan2.renderer.enabled=true;//enables the new object
            r[0].shader=shaderNoOutline;// changes the shader of the appropriate material
            r[0].color.a=0;// the alpha channel is 0 in order to be completely transparent
        }else{// the same process but with other material list
            plan2.renderer.enabled=true;
            m.renderer.materials[0].shader=shaderNoOutline;
            m.renderer.materials[0].color.a=0;
        }
    }else{
        plan2.renderer.enabled=false;
        m.renderer.materials=intMaterials;
        var s = Shader.Find("Bumped Diffuse");//changes the shader of the material of both lists to bumped diffuse
        m.renderer.materials[0].shader=s;
        r[0].shader=s;
    }
}
function OnMouseOver(){
    if ( Input.GetMouseButtonDown ( 0 ) ){
        audio.PlayOneShot(beep);
        clickhot=!clickhot;
    }
}

```

When we click again the new object disappears, the shader of that part returns to bumped diffuse and the material list is replaced with the initial one.

When the ‘show hotspots’ button is clicked for a second time in the submenu the hotspots disappear, as well as the objects or the alterations that may be activated.

7.7.2 Information plane

The second submenu button is named ‘Show Info’. When it is clicked a GUI window pops up with information regarding the history of the ‘Neoria’ monument. As it is shown on the part of code of submenu_script.js in the previous section, when a click occurs on that button the info gameObject is activated. The info gameObject is an empty object with the **info.js** script attached to it.

In the script we create a GUI scroll window in which we display text regarding the history of the monument. The text is loaded in the longString variable. When the button is clicked again the GUI window disappears.

```
function OnGUI() {
    GUI.skin =menuSkin;
    GUI.BeginGroup(menuAreaNormalized);
    scrollPosition =GUILayout.BeginScrollView (
    scrollPosition, GUILayout.Width (400), GUILayout.Height (450));
    GUILayout.Label (longString);
    GUILayout.EndScrollView ();
    GUI.Box(Rect (0,0,400,450), "");
    GUI.EndGroup();
}
```

7.7.3 Measure Dimensions Functionality

The third button is named ‘Measure Dimensions’. This is a method of providing a way to measure dimensions on the monument surface. The user can click on two points on the surface monument and their distance and depth difference is calculated and displayed on the screen. To do this functionality the raycasting method that was explained in the section 7.6 was employed.

The code for this feature is in the functionality.js script. First we wrote a function that calculates the distance between two points called CalculateDistance(). It takes as arguments two 3D points and calculates their distance in the three dimensional space. The 3D distance between the points (x_1, y_1, z_1) and (x_2, y_2, z_2) is

```
// Function for calculating the 3D distance
function CalculateDistance(xe1:float,xe2:float,ye1:float,ye2:float,ze1:float,ze2:float){
    distance=Mathf.Sqrt((Mathf.Pow(xe2-xe1,2)+Mathf.Pow(ye2-ye1,2)+Mathf.Pow(ze2-ze1,2)));
    depthx=Mathf.Abs(xe2-xe1);
    depthy=Mathf.Abs(ye2-ye1);
    depthz=Mathf.Abs(ze2-ze1);
    callLabel=true;
}
```

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Inside the Update() function on the functionality.js script we check to see if the submenu object is active in order to be able to access its variables. If it is active we check if the button 'Measure Distance' is clicked. If it clicked we enable a gameObject we have created called point_highlight. That object is an empty object with a halo component attached to it that acts as a yellow dot pointing to the surface. It is a way of highlighting the mouse point's position and distinguish easily the point we want to click. We use the raycast we implemented in the dynamic display of the coordinates (it is in the same script). We take the information of that ray, which identifies with the mouse movement and we pass it in the point_highlight in order to follow the mouse. Now we a yellow dot following the mouse when it hits a surface. Depending on whether we are about to apply a first or second click we send information to the submenu_script (as is shown on the right) and a message appears. When we are about to click the first point the message "Click on a starting point" appears. Before we have the second click the message "Click on a second point" appears. It is a way of guiding the user during the action. When we first click we keep the information of the impact point in variables. Then we create a small pyramid object and position it in the spot we clicked, to mark the point. (If there are other pyramids from previous measurements they are destroyed) When we click again we also keep the information of that impact point in variables and we place a pyramid on the spot. We enable a line renderer which we have attached as a component to the walker. The line renderer draws a line between two (or more) points. In our case a line is drawn between the previous clicked points. Then we call the CalculateDistance() function and the 3D distance is calculated.

```

////////// MEASURE DISTANCES
var pyr=GameObject.Find("pyramid_point");
var line=GetComponent(LineRenderer);
if (s){//checks if the submenu has been activated
    var first_click=GameObject.Find("Submenu").GetComponent(submenu_script).first;
    var label=GameObject.Find("Submenu").GetComponent(submenu_script).labelActive;
    if(label)//checks if the button 'measure distances' is clicked
        var highlight=GameObject.Find("Point_highlight").GetComponent("Halo")as Behaviour;
        highlight.enabled=true;// enables the yellow dot to be displayed
        var point=GameObject.Find("Point_highlight");
        point.transform.position=hit.point;// the point where the ray hits the collider
        if ( Input.GetMouseButtonDown ( 0 ) )
            if (first_click==false){// when we click for a second time
                line.enabled=true;
                var x=hit.point.x;// stores the xyz dimensions of the point that was hit on the collider
                var y=hit.point.y;
                var z=hit.point.z;
                x2=x;
                y2=y;
                z2=z;
                //displays the pyramid that shows the clicked point
                ob2=Instantiate(pyr, Vector3(x2,y2+0.2,z2),Quaternion.Euler(90,180,0));
                //draws a visible line between the two clicked points
                line.SetPosition(0,Vector3(x1,y1,z1));
                line.SetPosition(1,Vector3(x2,y2,z2));
                CalculateDistance(x1,x2,y1,y2,z1,z2);// calculates the distance
                GameObject.Find("Submenu").GetComponent(submenu_script).first=true;
            }
            if (first_click){// when we click the first point
                line.enabled=false;
                var x=hit.point.x;
                var y=hit.point.y;
                var z=hit.point.z;
                x1=x;
                y1=y;
                z1=z;
                //destroys remaining pyramid objects
                Destroy(ob1);
                Destroy(ob2);
                //displays the pyramid
                ob1=Instantiate(pyr, Vector3(x1,y1+0.2,z1),Quaternion.Euler(90,180,0));
                calllabel=false;
                GameObject.Find("Submenu").GetComponent(submenu_script).first=false;
            }
        }
    }
}
else{// when the "measure dimensions" button is clicked a second time the objects disappear
    // and the variables are initializing
    if (ob1!=null || ob2!=null){
        Destroy(ob1);
        Destroy(ob2);
        line.enabled=false;
    }
    GameObject.Find("Submenu").GetComponent(submenu_script).labelActive=false;
    GameObject.Find("Submenu").GetComponent(submenu_script).first=true;
}
}

if(GUI.Button(Rect(dimensions), "Measure Dimensions")){
    audio.PlayOneShot(beep);
    labelActive=!labelActive;
}
if(labelActive){
    if (first){
        GUI.Label(Rect(message_space), "Click on a starting point.");
    }else{
        GUI.Label(Rect(message_space), "Click on a second point.");
    }
}
GUI.EndGroup();
}

```

When that function is called, the GUI label is activated and informs the user of the resulting distance. Additionally, the depth difference is displayed. We need to

distinguish what is considered surface depth on each side. The north and south side consider as surface depth the z axis, while the east and west side the x axis.

```
function OnGUI() { // displays the results
    GUI.BeginGroup(AreaNormalized);
    GUI.skin.font = MyFont;
    var guiTextObject = GameObject.Find("GUI_Text_coord");
    if (callLabel) {
        GUI.Label(Rect(info_extraction), "Distance "+distance+"m");
        // the depth difference on each side
        if (guiTextObject.GetComponent(GUIText).text=="North side" || guiTextObject.GetComponent(GUIText).text=="South side"){
            GUI.Label(Rect(depth_extraction), "Depth difference "+depthz+"m");
        }
        if (guiTextObject.GetComponent(GUIText).text=="West side" || guiTextObject.GetComponent(GUIText).text=="East side"){
            GUI.Label(Rect(depth_extraction), "Depth difference "+depthx+"m");
        }
    }
    GUI.EndGroup();
}
```

Because the monument is scaled inside Unity according to the world scale the results represent real life units and are in meters.

7.8 Lighting transition

We have developed a functionality in which the user can control a day night lighting transition. The user has the opportunity to observe the monument under different lighting conditions. A state representing the day and a state representing the night have been created and a smooth transition between them. The user can control the transition via a slider that is displayed on the screen. In order to implement that functionality we have created the **day_night_transition.js** script and attached it to an empty gameObject.

A vertical slider is a GUI element that the user can drag to change a value between a min and a max. The `GUI.VerticalSlider()` function returns the number that the slider shows at any given time. We take that number and depending on it we change the properties and settings of the components we want. Because we will work with a lot of different range values we want to normalize them in order to be on the same range. We used the following function.

```
function normalize(value:float ,min:float , max:float ,dmin:float , dmax:float ) {
    return Mathf.Abs((dmax-dmin)/(max-min)*(value-max)+dmax);
}
```



This function is given a value between a min and a max and returns the corresponding value in the `dmin, dmax` range. We made our implementation in the **day_night_transition.js** script and inside the `OnGUI()` function.

```

GUI.skin =menuSkin;
GUI.BeginGroup(menuAreaNormalized);
GUI.Label(Rect(lt),"Lighting Transition");
GUI.Label(Rect(d),"Day");
GUI.Label(Rect(n),"Night");
vSliderValue = GUI.VerticalSlider (Rect (slider), vSliderValue, 1.0, 0.0);
var watermat=GameObject.Find("Water4Example (Advanced)").GetComponent(WaterBase).sharedMaterial://change the color of the water
var base=watermat.GetColor("_BaseColor");

base.r=normalize(vSliderValue,0,1,normalize(daybase_r,0,255,0,1),normalize(nightbase_r,0,255,0,1));
base.g=normalize(vSliderValue,0,1,normalize(daybase_g,0,255,0,1),normalize(nightbase_g,0,255,0,1));
base.b=normalize(vSliderValue,0,1,normalize(daybase_b,0,255,0,1),normalize(nightbase_b,0,255,0,1));
base.a=normalize(vSliderValue,0,1,normalize(daybase_a,0,255,0,1),normalize(nightbase_a,0,255,0,1));
watermat.SetColor("_BaseColor",base);

var reflection=watermat.GetColor("_ReflectionColor");
reflection.r=normalize(vSliderValue,0,1,normalize(dayreflection_r,0,255,0,1),normalize(nightreflection_r,0,255,0,1));
reflection.g=normalize(vSliderValue,0,1,normalize(dayreflection_g,0,255,0,1),normalize(nightreflection_g,0,255,0,1));
reflection.b=normalize(vSliderValue,0,1,normalize(dayreflection_b,0,255,0,1),normalize(nightreflection_b,0,255,0,1));
reflection.a=normalize(vSliderValue,0,1,normalize(dayreflection_a,0,255,0,1),normalize(nightreflection_a,0,255,0,1));
watermat.SetColor("ReflectionColor",reflection);

```

We created the slider and it gives us a value between 0 and 1. Day corresponds to 0 and night to 1. First we change the color of the water. During the day it has a bright blue color, but during the night changes to a dark color. We accessed the water's base and reflection colors. For the red, green and blue values of those colors, we take the value of the slider and normalize it between a min and a max value that we have defined. Those min and max values represent the color of the day and night and must be normalized to be between 1 and 0. While the value of the slider changes, so does the color.

Another element that we should focus is the streetlights we created. It would be beautiful if as we change the state of the atmosphere the streetlights were smoothly lighted with a glow around them.

To create the lighting of streetlights we put a spot light on the top looking down, so it will cast light down and fake the real streetlights. A point light was also put inside the curve of the streetlight in order to give it a glow and halo like a light bulb.



Figure 7.18 The streetlight with the two lights attached to it and a plane with an image of a glow.

To achieve an even more realistic look we want to create a glow that begins from the light and as it is descending to disappear. To fake that we created a plane and put as a texture the image of a glow on it and a transparent shader. In order to be able to look at the light from all sides and see the glow , we made the plane turn towards the camera all the time. To achieve that we attached a script to the plane object with the code shown on the right. With that the plane always turns towards the target, which is the camera.

```

var target:Transform;
function Update () {
    target=camera.main.transform;
    transform.LookAt(target);
}

```

```

GameObject.Find("Directional light").GetComponent(Light).intensity=normalize(vSliderValue,0,1,0.8,0.4);

var dr:float=185;
var dg:float=236;
var db:float=249;
var nr:float=12;
var ng:float=28;
var nb:float=60;
var num_r=normalize(vSliderValue,0,1,normalize(dr,0,255,0,1),normalize(nr,0,255,0,1));
var num_g=normalize(vSliderValue,0,1,normalize(dg,0,255,0,1),normalize(ng,0,255,0,1));
var num_b=normalize(vSliderValue,0,1,normalize(db,0,255,0,1),normalize(nb,0,255,0,1));
RenderSettings.ambientLight=Vector4(num_r,num_g,num_b,1); //change the color of the ambient light
for(var obj : GameObject in GameObject.FindGameObjectsWithTag("street light")){
    if (obj.name=="Spotlight"){// change the intensity of the street lights
        obj.GetComponent(Light).intensity=normalize(vSliderValue,0,1,0,3.5);
    }
    if (obj.name=="Spotlight_inside"){
        obj.GetComponent(Light).intensity=normalize(vSliderValue,0,1,0,6.5);
    }
    if (obj.name=="plane_projection"){// how to render the streetlight halo depending on the slider value
        var flare = obj.renderer;
        if (vSliderValue>=0.33 && vSliderValue<0.45){
            flare.enabled=true;
            flare.material.SetTexture(0,tex3);
        }else if(vSliderValue>=0.45 && vSliderValue<0.6){
            flare.material.SetTexture(0,tex1);
        }else if (vSliderValue>=0.6){
            flare.material.SetTexture(0,tex2);
        }else{
            flare.enabled=false;
        }
    }
}
}

```

Having created the effect of a realistic streetlight, we want to change its illumination and appearance as the slider value changes. Continuing on the **day_night_transition.js** script, we find the street lights we created and we change the intensity of the spot and point light based on the slider value. To have a realistic transition from day to night the halo should not exist in the day and as the slider changes to slowly brighten. For that reason we created three textures with the image of a glow on each one of them. The first with a faint glow, the second with a more intense glow and the third with a very bright glow. We check the slider values and as they change, we change the texture of the plane object depending on if the values are towards the day or night. This gives us a smooth appearance or disappearance of the glow.

We also set the intensity of the directional light that represents the sun and change the color of the ambient light based on the slider value to make them suitable with the atmosphere.

We also set the properties of two camera effects. The glow effect and the bloom and lens flares. These effects help us enhance the atmosphere, especially at night.

```

var gloweffect = GameObject.Find("Main Camera").GetComponent(GlowEffect); // change the camera effects
var bloomFlares=GameObject.Find("Main Camera").GetComponent(BloomAndLensFlares);
bloomFlares.sepBlurSpread=normalize(vSliderValue,0,1,0,1.5);
bloomFlares.bloomIntensity=normalize(vSliderValue,0,1,0,1.8);
bloomFlares.useSrcAlphaAsMask=normalize(vSliderValue,0,1,0,0.5);
if (vSliderValue>0.63){
    gloweffect.enabled=true;
    gloweffect.glowIntensity=normalize(vSliderValue,0,1,0,1.1);
    gloweffect.blurIterations=normalize(vSliderValue,0.63,1,14,10);
    gloweffect.blurSpread=normalize(vSliderValue,0,1,0,0.7);
} else{
    gloweffect.enabled=false;
}
RenderSettings.skybox = sunny_skybox; // change the appearance of the skybox
RenderSettings.skybox.SetFloat("_Blend",vSliderValue);
GUI.EndGroup();

```

Finally, we need to change the skybox from a day sky to a night sky. With the help of a skybox shader that blends between two sets of skybox textures.

7.9 Compass

The functionality of a compass is implemented. A vector rotates to always point towards the north. It is displayed at the bottom right of the screen and can help the user be oriented. We created the script **compass.js** for that purpose and we attached it on the walker.

We store the y rotation of the camera at any given point



and we pass it in the RotateAroundPivot() function. This function rotates the GUI that follows, i.e. the image vector by the angles we have given around a pivot point (the centre point of the image).

```

function Update () {
    rot=Camera.main.transform.eulerAngles.y;
}

function OnGUI() {
    GUI.BeginGroup(menuAreaNormalized);
    GUI.DrawTexture(Rect(0, 0, 50, 50), image_compass);
    var pivotPoint = Vector2(25,25);
    GUIUtility.RotateAroundPivot (rot+180, pivotPoint);
    GUI.DrawTexture(Rect(0, 0, 50, 50), image_vector);
    GUI.EndGroup();
}

```

7.10 Application utilities

In order to close the application at any given time a quit button is implemented. A GUITexture object was created and placed on the top right corner of the screen. When it is clicked the application closes.

```

function OnMouseEnter () {
    guiTexture.color = Color (.7,.7,.7);
}

function OnMouseExit () {
    guiTexture.color = Color (.5,.5,.5);
}

function OnMouseDown(){
    audio.PlayOneShot(beep);
    guiTexture.color = Color (1,1,1);
    Application.Quit();
}
@script RequireComponent(AudioSource);

```

When the application launches a welcome screen appears with a start button. That button is created with a GUI Texture with a script attached to it.

```
function OnMouseEnter () {
    guiTexture.color = Color (.1,.1,.1);
}
function OnMouseExit () {
    guiTexture.color = Color (.67,.33,.33);
}

function OnMouseDown() {
    audio.PlayOneShot(beep);
    Application.LoadLevel("neoria");
}
@script RequireComponent(AudioSource);
```

When the button is clicked the scene ‘neoria’ is loaded. That scene contains all the functionalities of our application.

Chapter 8 Result and System Evaluation

The evaluation of the whole system is separated in two parts. The evaluation of the 3D reconstruction of the model and the evaluation of the interactive application.

8.1 3D Reconstruction Evaluation

The creation of the 3D model was a long and complex procedure. It began with the field work, the measurements, the data acquisition and continued with the data process, the surface reconstruction and post processing. From the beginning, the initial data were converted and processed. They were exchanged between several 3D software programs with different coordinate systems. Once the surface was formed, it was heavily processed and in some cases altered in order to simulate the real surface in the best way possible. It is natural to have errors throughout the framework that was followed.

Errors can appear in all the stages of the data acquisition process and surface creation, affecting the credibility of the outcome. The types of errors that can occur are:

- Total station errors - The instrument error has been mentioned on the fourth chapter in the total station's technical characteristics. Although total stations are highly accurate instruments, their accuracy can be compromised by temperature changes, vibrations, long transport periods and beam blockage.
- Errors in the surface creation - The data managing, the exchange between devices, the data process that occurred in order to create the TIN surfaces, as well as the import/export from the modeling software are procedures prone to errors.
- Surface processing errors - Similar with what was mentioned above. Errors can appear in the exchange between modeling software and subsequent coordinate system conversion as well as the surface processing itself. It should not be forgotten that the resulting surface although containing the original measurements is quite different from the initial TIN surface.
- Human error – Last but not least the human error is a significant factor in the credibility of the results. Human errors can occur in the data acquisition process. The operator of the total station can cause vibrations or laser beam deviations resulting in incorrect measurements. The human factor is heavily featured during the surface creation and processing. The developer operated the software and a big part of the surface creation procedure was implemented by hand.

The evaluation of the results consists of the simple visual evaluation and the accuracy evaluation. The model should have the same appearance with the real monument and display the details of the real surface, something that can be depicted with the comparison of the 3D model with the actual building or photographs. In the accuracy evaluation the resulting model should be compared with another model, surface or data

representing the ground truth, and calculate the deviation from them. Next, the accuracy evaluation that was performed is presented.

8.1.1 Accuracy Evaluation

Having no ground truth to compare our results, the final model is going to be compared to the initial measurements. This is to determine if the procedure of the 3D reconstruction has deviated or altered the points that were initially measured. This is also a way to provide an error calculation for the functionalities that have to do with the geographical coordinates.

The comparison was executed with the CloudCompare software. CloudCompare is an open source 3D point cloud and mess processing software. We imported the model in it as well as a file with the original raw data. The raw data were considered as a sparse point cloud and acted as a reference entity, meaning the calculations will be computed relatively to its points. The 3D model is the compared mesh the one on which distances will be computed. CloudCompare will compute the distances of each of its points relatively to the reference entity.

After they were aligned to each other, the comparison was executed. A 'nearest neighbor distance' method was applied. For each point of the compared cloud, i.e the 3D model, the nearest point in the reference cloud (the raw data measurements) was found and their euclidean distance was computed. The results are shown in figure 8.1. The Cloud to Mesh (C2M) absolute distances are calculated. A scalar field color scale is also displayed.

In order to have accurate results, only the parts of the model that could have a reference to the data were used. The roof was created without being based on geographical data and its presence on the comparison would alter the results.

The results are considered satisfying. Most of the points of the model have about less than 10 cm deviation from the measured points. As it can be observed from the colored model, most of the surface of the main structure, i.e. blue color, identifies with the geographical measurements. The parts where the surface is green corresponds with the areas that we did not take adequate measurements. Those are some bottom parts on the south side and the bottom of the outermost domes on the north side. These parts were inaccessible due to the limited space around the monument.

The east side was inadequately measured. This is the area where the most deviation exists. That side was inaccessible during the measurement process. Trees and plants were covering it, and the presence of a property and a parking lot next to it hindered the data acquisition. Few points representing the basic outline were measured.

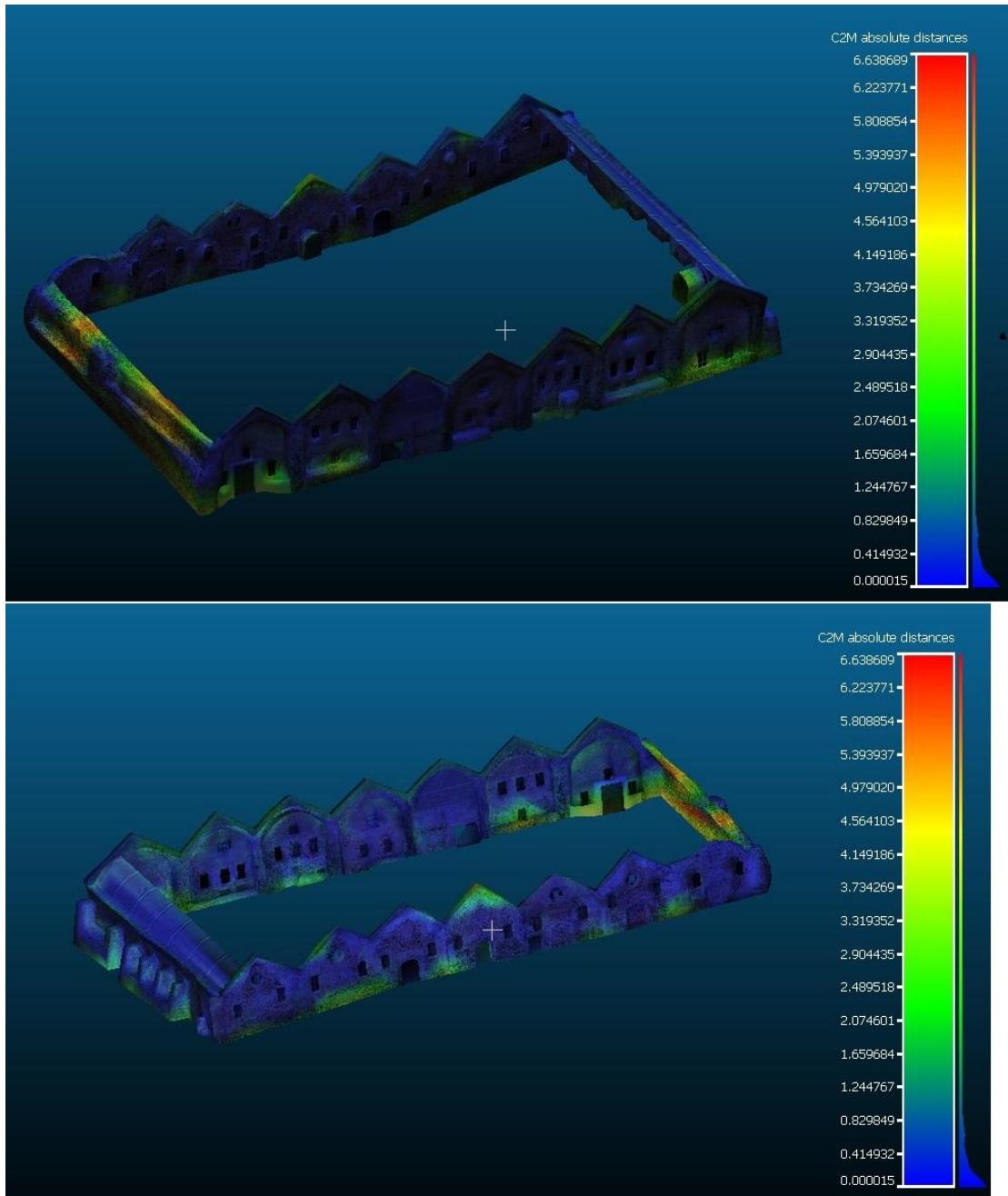


Figure 8.1 The comparison of the 3D model with the original raw data.

Taking into consideration the size of the monument and the scarcity of the measured points, we can conclude that we have a high level of accuracy.

8.2 Evaluation of the application

From the first stages of the development, the application was constantly tested and changed. Due to the level of detail that entails the appearance of the 3D model was assessed and corrected continuously. Unity provides the features and environment of application testing. During the development process the application was assessed during game play and along with the model was in a constant circle of changes and refinement. Although the first versions of the application included defects and flaws, it was constantly improved assessing the feedback.

The application was given to several users in order to be tested and evaluated. Their feedback was insightful and constructive. Their remarks ranged from comments about the 3D environment to the graphical user interface and the detection of bugs.

Regarding the 3D environment, there were comments about minor imperfections of some objects. Some streetlights were wrongly positioned or mistakenly moved during the development process, something that was corrected.

Concerning the user navigation, there were observations about minor flaws or defects. Although the character controller was prohibited to walk through the building, the camera could look inside the model when the walker hit a wall, something that wasn't desirable. This was corrected by adjusting the camera settings, particularly the clipping plane in order to render the environment from nearest distances. Another comment regarding the navigation was the inability to rotate the camera 360 degrees around itself, something that bothered some users. This was also corrected. An additional bug that was detected was that at a particular point in front of the harbor the controller could fall in the sea. This was caused due to the misplacement of the collider that stopped the controller's movement, something that was also corrected.

The most important comment was about the lack of instructions. The users that tested it took some time to familiarize with it and understand the functionalities. To that aid we created an instructions menu. A button was created on the top right called "Instructions". When it is clicked, other buttons appear each one regarding a functionality to be explained. When one of them is clicked, instructions about that particular functionality are displayed.



Figure 8.2 The instructions menu. When one of the buttons is clicked instructions about a particular functionality are displayed.

The instructions menu was implemented on a similar way as the other menus. On the `menu_script.js` we added the code to display the button. An empty `GameObject` was created called 'Instructions' and a script called **Instructions.js** was attached to it. That script is enabled when the instructions button is pressed.

The **Instructions.js** script contains a `OnGUI()` function displaying the individual buttons. When any of those buttons is clicked another script is enabled. That script is the **InstructionLabels.js** script and is also attached to the Instructions `GameObject`. A variable holding information about which button was pressed is passed to the **InstructionLabels.js** script.

In that script depending on the value of the variable a message appears providing information about the particular functionality.

```

function Update () {
    var c=this.GetComponent(Instructions).choice;
    if(c==1){
        instr_String="Use the arrow keys to move around. Hold down the right mouse button to rotate the camera.";
    }else if(c==2){
        instr_String="Click on the hotspots that appear through the monument surface to alter the monument's geometry. Click again to close the geometry.";
    }else if(c==3){
        instr_String="Click on two points wherever on the monument's surface to measure their distance.";
    }else if(c==4){
        instr_String="The measurements that were conducted are displayed on the monuments surface. Click on a green cube to reveal its "+ "measured coordinates. Click again to hide the coordinates.";
    }else if(c==5){
        instr_String="The raw data that were measured are displayed. Click on a point ID and go to the side that it is to see it displayed.";
    }
}

function OnGUI () {
    GUI.backgroundColor=Color(1,0.84,0,1);
    GUI.skin =menuSkin;
    GUI.BeginGroup(Rect(102,234,200,100));
    scrollPosition =GUILayout.BeginScrollView (
        scrollPosition, GUILayout.Width (200), GUILayout.Height (100));
    GUILayout.Label (instr_String);
    GUILayout.EndScrollView ();
    GUI.Box(Rect(0,0,200,100),"");
    GUI.EndGroup();
}

```

The users provided useful suggestions and comments that contributed to the optimization of the application.

Chapter 9 Conclusions and Future Work

The digital representation of physical objects is not only a means for storing data but also a way of capturing the world around us and making information accessible to a wider audience. The depiction of objects as digital models is becoming a significant part of many scientific fields and an aid in research. Computer graphics and 3D technology is utilized so it could be done in an accurate way. Cultural heritage is hugely aided by the new technologies. Three-dimensional digitization is a way of moving the data into the future, preserving invaluable artifacts and monuments for posterity. Objects of historic value face the danger of deterioration. A digital 3D representation can help preserving the original condition or the current state of a historical object, as well as track changes and regain information about them.

In this thesis the computer graphic technology was utilized to create a 3D representation of 'Neoria' a historic building in Crete, Greece. Tacheometry acquisition provided spatial and geographical data to base the digital reconstruction. The integration of tacheometry measurements and computer graphics puts forward framework for the scientifically accurate 3D reconstruction of the historical building. The framework presented was based on game engine technologies which provide powerful tools for photorealistic visualization and functionalities development. The main technical challenge of this work was the production of a scientifically accurate 3D mesh based on a rather small number of tacheometry measurements acquired fast and at low-cost. The combination of surface reconstruction and processing methods ensured that a detailed geometric mesh was constructed fast based on far fewer points available compared to laser scanning. A 3D interactive application was created involving the 3D model of the monument. The user can visualize and tour the monument and the area around it, as well as manipulating the model. Advanced interactive functionalities are offered to the user in relation to identifying restoration areas and visualizing the outcome of such works. Moreover, the user could visualize the co-ordinates of the points measured, calculate distances at will and navigate the complete 3D mesh of the monument.

Future extensions or improvements would be to produce an advanced visualization system operated on web or a mobile platform, a presentation of the original building fully restored and deprived of modern alterations by methods of determining what the appearance of the surface might have been, or an extended application on a mobile platform that would allow access to expert users in order to collect or add data regarding the monument, its visualization or restoration.

References

- [1] Boochs, F., Hoffmann, A., Huxhagen, U., and D. Welter, D., 2006. Digital reconstruction of archaeological objects using hybrid sensing techniques-the example Porta Nigra at Trier. *BAR INTERNATIONAL SERIES* 1568 : 395.
- [2] 3D Interactive Environments -
http://edutechwiki.unige.ch/en/3D_interactive_environments
- [3] Scientific Visualization - http://en.wikipedia.org/wiki/Scientific_visualization
- [4] Li, R., Luo, T., & Zha, H. (2010). 3D digitization and its Applications in Cultural Heritage. In *Digital Heritage* (pp. 381-388). Springer Berlin Heidelberg.
- [5] Rozenwald, G. F., Seulin, R., & Fougerolle, Y. D. (2010, February). Fully automatic 3D digitization of unknown objects. In *IS&T/SPIE Electronic Imaging* (pp. 753807-753807). International Society for Optics and Photonics.
- [6] Pavlidis, G., Tsirliganis, N., Tsiafakis, D., Arnaoutoglou, F., Chamzas, C., TSIOUKAS, V., ... & Mexia, A. (2006, June). 3D digitization of monuments: the case of Mani. In *Proc. 3-rd International Conference on Museology*, (Jun. 2006).
- [7] Pavlidis, G., Koutsoudis, A., Arnaoutoglou, F., Tsioukas, V., & Chamzas, C. (2007). Methods for 3D digitization of cultural heritage. *Journal of cultural heritage*, 8(1), 93-98.
- [8] Sifniotis, M., Jackson, B., Mania, K., Vlassis, N., Watten, P. L., & White, M. (2010, July). 3D visualization of archaeological uncertainty. In *Proceedings of the 7th Symposium on Applied Perception in Graphics and Visualization* (pp. 162-162). ACM.
- [9] Photogrammetry - <http://en.wikipedia.org/wiki/Photogrammetry>
- [10] Photogrammetry - <http://www.photogrammetry.com/>
- [11] Lingua, A. M., Piumatti, P., & Rinaudo, F. (2003). Digital photogrammetry: a standard approach to cultural heritage survey. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 34(Part 5), 210-215.
- [12] Ioannides, M., Arnold, D., Niccolucci, F., & Mania, K. Combination and Comparison of Digital Photogrammetry and Terrestrial Laser Scanning for the Generation of Virtual Models in Cultural Heritage Applications.
- [13] Schenk, T. (2005). Introduction to Photogrammetry, From
<http://www.mat.uc.pt/~gil/downloads/IntroPhoto.pdf>

References

- [14] Laser Scanning - http://en.wikipedia.org/wiki/Laser_scanning
- [15] 3D Scanner - http://en.wikipedia.org/wiki/3D_scanner
- [16] Staiger, R. (2003, December). Terrestrial laser scanning technology, systems and applications. In *2nd FIG Regional Conference Marrakech, Morocco* (Vol. 1).
- [17] Boehler, W., Bordas Vicent, M., Heinz, G., Marbs, A., & Müller, H. (2004, May). High quality scanning and modeling of monuments and artifacts. In *Proceedings of the FIG Working Week* (pp. 22-27).
- [18] Allen, P. K., Troccoli, A., Smith, B., Murray, S., Stamos, I., & Leordeanu, M. (2003). New methods for digital modeling of historic sites. *IEEE Computer Graphics and Applications*, 23(6), 32-41.
- [19] Geodetic Measuring Devices - <http://www.brighthubengineering.com/geotechnical-engineering/44583-geodetic-measuring-devices/>
- [20] Total station - <http://www.nikon.com/about/technology/life/others/surveying/>
- [21] 3D Computer Graphics - http://en.wikipedia.org/wiki/3D_computer_graphics
- [22] 3D Modeling - http://en.wikipedia.org/wiki/3D_modeling
- [23] 3D Rendering - http://en.wikipedia.org/wiki/3D_rendering
- [24] Polygon Mesh - http://en.wikipedia.org/wiki/Polygon_mesh
- [25] Texture Mapping - http://en.wikipedia.org/wiki/Texture_mapping
- [26] Shader - <http://en.wikipedia.org/wiki/Shader>
- [27] Fabio, R. (2003). From point cloud to surface: the modeling and visualization problem. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 34(5), W10.
- [28] Amenta, N., Choi, S., & Kolluri, R. K. (2001, May). The power crust. In *Proceedings of the sixth ACM symposium on Solid modeling and applications* (pp. 249-266). ACM.
- [29] Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., & Taubin, G. (1999). The ball-pivoting algorithm for surface reconstruction. *Visualization and Computer Graphics, IEEE Transactions on*, 5(4), 349-359.
- [30] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., & Stuetzle, W. (1992). *Surface reconstruction from unorganized points* (Vol. 26, No. 2, pp. 71-78). ACM.

References

- [31] Kazhdan, M., Bolitho, M., & Hoppe, H. (2006, June). Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing* (Vol. 7).
- [32] Alqudah, A. (2014). Survey of Surface Reconstruction Algorithms. *Journal of Signal and Information Processing*, 2014.
- [33] Tang, R., Halim, S., & Zulkepli, M. (2013). Surface reconstruction algorithms: review and comparisons. *ISDE2013*.
- [34] Bolitho, M., Kazhdan, M., Burns, R., & Hoppe, H. (2009). Parallel poisson surface reconstruction. In *Advances in Visual Computing* (pp. 678-689). Springer Berlin Heidelberg.
- [35] Catmull, E., & Clark, J. (1978). Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-aided design*, 10(6), 350-355.
- [36] Doo, D., & Sabin, M. (1978). Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6), 356-360.
- [37] Zorin, D., Schröder, P., & Sweldens, W. (1996, August). Interpolating subdivision for meshes with arbitrary topology. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (pp. 189-192). ACM.
- [38] Subdivision surface - http://en.wikipedia.org/wiki/Subdivision_surface
- [39] Game Engines - http://en.wikipedia.org/wiki/Game_engine
- [40] Unreal Engine - http://en.wikipedia.org/wiki/Unreal_Engine
- [41] Unity 3D - http://en.wikipedia.org/wiki/Unity_%28game_engine%29
- [42] CryEngine - <http://en.wikipedia.org/wiki/CryEngine>
- [43] Unity Manual - <http://docs.unity3d.com/Manual/index.html>
- [44] Scripting API - <http://docs.unity3d.com/ScriptReference/index.html>
- [45] Autodesk 3ds Max - http://en.wikipedia.org/wiki/Autodesk_3ds_Max
- [46] AutoCAD Civil 3D - <http://www.autodesk.com/products/autocad-civil-3d/overview>
- [47] MeshLab - <http://meshlab.sourceforge.net/>

References

- [48] The Venetian Port of Chania - <http://www.destinationcrete.gr/en/monuments-other/chania-monuments-other/chania-city-monuments-other/limani-faros-enetika-neoreia>
- [49] Venetian Neoria - <http://www.chaniatourism.com/see-do/archaeological-sites-historical-monuments/66-venetian-neoria.html>
- [50] Hellenic Geodetic Reference System 1987 -
http://en.wikipedia.org/wiki/Hellenic_Geodetic_Reference_System_1987
- [51] Easting and northing - http://en.wikipedia.org/wiki/Easting_and_northing
- [52] From point clouds to powercrusts -
<http://www.cgw.com/Publications/CGW/2001/Volume-24-Issue-6-June-2001-/From-Point-Clouds-to-Power-Crusts.aspx>
- [53] Triangulated Irregular Network (TIN) -
http://en.wikipedia.org/wiki/Triangulated_irregular_network
- [54] Delaunay Triangulation - http://en.wikipedia.org/wiki/Delaunay_triangulation
- [55] TIN surfaces -
<http://resources.arcgis.com/en/help/main/10.1/index.html#/006000000001000000>
- [56] Peters, J., & Reif, U. (1997). The simplest subdivision scheme for smoothing polyhedra. *ACM Transactions on Graphics (TOG)*, 16(4), 420-431.
- [57] UV Mapping - http://en.wikipedia.org/wiki/UV_mapping
- [58] 3ds Max Documentation - <http://knowledge.autodesk.com/support/3ds-max#/p=3ds%20Max&sort=score>
- [59] SQLite - <https://www.sqlite.org/>
- [60] SQLite class - <http://wiki.unity3d.com/index.php/SQLite>