



ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

**ΔΙΑΔΡΑΣΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ
ΠΟΛΙΤΙΣΤΙΚΟΥ ΠΕΡΙΕΧΟΜΕΝΟΥ ΜΕ ΤΗ
ΧΡΗΣΗ ΥΒΡΙΔΙΚΩΝ ΤΕΧΝΟΛΟΓΙΩΝ,
ΓΕΩΓΡΑΦΙΚΟΥ ΣΥΣΤΗΜΑΤΟΣ ΚΑΙ
ΤΡΙΣΔΙΑΣΤΑΤΩΝ ΤΕΧΝΟΛΟΓΙΩΝ**

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εισηγητής: Χαράλαμπος, Κολοκούρης, ΤΠ4504

Επιβλέπων: Αθανάσιος, Μαλάμος, Καθηγητής

©

2022



HELLENIC MEDITERRANEAN UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

**INTERACTIVE PRESENTATION OF
CULTURAL CONTENT USING HYBRID
TECHNOLOGIES, GEOGRAPHICAL
SYSTEMS AND THREE-DIMENSIONAL
TECHNOLOGIES**

DIPLOMA THESIS

Student : Charalampos, Kolokouris, TP4504

Supervisor : Athanasios, Malamos, Professor

©

2022

Υπεύθυνη Δήλωση: Βεβαιώνω ότι είμαι συγγραφέας αυτής της πτυχιακής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και αναφέρεται στην πτυχιακή εργασία. Επίσης, έχω αναφέρει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επίσης, βεβαιώνω ότι αυτή η πτυχιακή εργασία προετοιμάστηκε από εμένα προσωπικά ειδικά για τις απαιτήσεις του προγράμματος σπουδών του Τμήματος Μηχανικών Πληροφορικής του ΕΛ.ΜΕ.ΠΑ.

ΠΕΡΙΛΗΨΗ

Η βασική ιδέα είναι: η απεικόνιση πολιτιστικού περιεχομένου, προσεγγίζοντας τον χρήστη στον τρισδιάστατο ψηφιακό χώρο, μέσω του χάρτη όπου απεικονίζεται ως βασικό μέσω πλοήγησης. Η γενική πλατφόρμα συγκομιδής πληροφορίας, περί πολιτισμικής κληρονομιάς, χρησιμοποιεί τη Διεπαφή προγραμματισμού εφαρμογών της EMIAC, σε συνεργασία με το ΙΤΕ, που έχει ως σκοπό την ψηφιοποίηση εκκλησιαστικών κειμηλίων και Ιερών Ναών/Χώρων στην Κρήτη.

Η γενική διαχείριση της πληροφορίας πραγματοποιείται από ένα σύστημα back-end γραμμένο σε κώδικα PHP. Η εμφάνιση της πλατφόρμας είναι γραμμένη σε HTML και JavaScript και η οπτικοποίηση των διεπαφών χρήστη είναι υλοποιημένη με CSS, πλαισιομένη στο Bulma framework.

Η οπτικοποίηση 3D γραφικών και της μεικτής πραγματικότητας έχει γίνει με την χρήση THREE.js, ένα framework της Javascript βασισμένο σε WebGL και του Model-Viewer, από την Google, για τα οποία θα γίνει περαιτέρω ανάλυση στα επόμενα κεφάλαια, όπως και για τα λογισμικά διαχείρισης γεωγραφικών συστημάτων MapBox και Leaflet.

Μέσα στην εργασία αυτή θα αναλυθούν οι μέθοδοι υλοποίησης, διεκπεραίωσης όπως και τα διαγράμματα UML και διάδρασης χρήστη, αναφερόμενα στο μοτίβο σχεδίασης που χρησιμοποιήθηκε.

Ο σκοπός είναι η δημιουργία μίας πλατφόρμας η οποία θα δώσει την ευκαιρία σε ανθρώπους, απομακρυσμένα, να έχουν επαφή σε πολυπολιτισμικό περιεχόμενο δωρεάν, καθώς και με πολύ μικρές προσαρμογές μπορεί να προβληθεί οτιδήποτε μέσω της υπηρεσίας αυτής.

Λέξεις Κλειδιά : επαυξημένη πραγματικότητα, εικονική πραγματικότητα, γραφικά, τρισδιάστατος χώρος, γεωγραφικό σύστημα, χάρτες

ABSTRACT

The general idea is: the visualization of cultural content, approaching the user in the 3D digital space, through the map which is used as a basic means to navigate throughout the main interface. The main platform for collecting information on cultural heritage uses the EMIAK application planning interface, in collaboration with FORTH, which aims to digitize Church relics and temples/sites in Crete.

General information management is performed by a back-end system written in PHP code. The appearance of the platform is written in HTML and plain JavaScript as the visualization of the user interfaces is implemented with CSS, framed by the Bulma framework.

The visualization of 3D graphics virtual, mixed and augmented reality has been implemented using THREE.js, a framework of Javascript for 3D graphics based on WebGL and Model-Viewer, by Google, for which further analysis will be made in the following chapters, as well as for the management software of MapBox and Leaflet geographical systems.

In this thesis we will analyze the implementation methods, processing as well as the UML diagrams and user interaction, referring to the design pattern used.

The aim is to create a platform that will give people the opportunity, remotely, to have access to multicultural content for free, and with very small adjustments anything can be viewed through this service.

Key Words : augmented reality, virtual reality, graphics, 3d space, geographic system, maps

Contents

| | |
|---|-----------|
| IMAGE LIST | 5 |
| ABBREVIATION | 6 |
| ACKNOWLEDGEMENTS | 8 |
| INTRODUCTION | 1 |
| Graphics | 2 |
| 1.1 Wireframe | 3 |
| 1.2 Sign Shading | 3 |
| Textures & Materials | 3 |
| Bump mapping | 4 |
| Normal mapping | 4 |
| Displacement mapping | 4 |
| 1.3 Simple Sphere | 4 |
| 1.4 Normal + Bump | 4 |
| 1.5 All applied | 4 |
| Rendering | 5 |
| Real-Time Rendering | 5 |
| Virtual Reality | 6 |
| Augmented Reality | 7 |
| Augmented Reality vs. Virtual Reality | 7 |
| Mixed Reality | 7 |
| Geographic Information System (GIS) | 11 |
| Advantages and Disadvantages | 11 |
| Implementation | 13 |
| Web XR | 14 |
| Case studies | 14 |
| Video | 15 |
| Object/ data visualization | 15 |
| Experiences in the arts | 15 |
| Three.JS | 16 |
| 2.2 Layout of a simple three.js application | 16 |
| Mapbox | 18 |
| Approach | 19 |

| | |
|---|-----------|
| Setting Up the Interface | 19 |
| Why SQL? | 19 |
| 3.1 Main Interface | 19 |
| The Interface on first Sight | 20 |
| Bonanza | 21 |
| 3.2 Example on an AR compatible device | 21 |
| 3.3 The List of found Relics | 22 |
| 3.3 Magnifying glass mode | 22 |
| The Map | 23 |
| 3.4 3d Buildings | 26 |
| The 3D Virtual Environment | 36 |
| Setting Up the Cube | 36 |
| 4.1 Results of the scenery functions | 38 |
| Creating the XR experience | 38 |
| Movement | 41 |
| Motion Sickness in Virtual Reality | 42 |
| Locomotion | 43 |
| 4.1 Line illustration | 43 |
| 4.2 Raycaster we made | 48 |
| On Press/ Head movement based navigation (Head Tilt Navigation) | 48 |
| BIBLIOGRAPHY | 50 |
| A. FOREIGN | 50 |
| B. NATIVE | 51 |

IMAGE LIST

| | |
|---|----|
| 1.1 Wireframe | 3 |
| 1.2 Sign Shading | 3 |
| 1.3 Simple Sphere | 4 |
| 1.4 Normal + Bump | 4 |
| 1.5 All applied | 4 |
| 2.2 Layout of a simple three.js application | 16 |
| 3.1 Main Interface | 19 |
| 3.3 The List of found Relics | 22 |
| 3.3 Magnifying glass mode | 22 |
| 3.4 3d Buildings | 26 |
| 4.1 Results of the scenery functions | 38 |
| 4.1 Line illustration | 43 |
| 4.2 Raycaster we made | 48 |

ABBREVIATION

VR _Virtual Reality

AR _Augmented Reality

MR _Mixed Reality

XR _Experience

API _Application Programming Interface

GIS _Geographical Information System

CSS _Cascading Style Sheets

JS _JavaScript

HTML _Hyper Text Markup Language

3D _Three Dimensional

ΕΜΙΑΚ _Επικοινωνιακό και Μορφωτικό Ίδρυμα Ιεράς Αρχιεπισκοπής Κρήτης

_Communication and Educational Foundation of the Holy Archdiocese of
Crete

ACKNOWLEDGEMENTS

I would like to express my gratitude and appreciation for my Supervisor, professor Athanasios Malamos. Whose guidance, support and encouragement has been invaluable throughout this study. I also wish to thank the team in the CM Lab who have been a great source of support, and helped throughout this project practically and mentally.

I would also like to express my gratitude to the Communication and Educational Foundation of the Holy Archdiocese of Crete and Cyprus that generously provided this kind of information and gave access to all these datasets, so that we can advance this thesis to a whole project in the future.

INTRODUCTION

Progressing to the main body of this thesis, I would like to introduce you to the thought process that will be followed throughout its entirety, from the first chapter to the last.

Beginning our journey we will be introduced to some fundamental definitions in graphics and the basis of the technologies used to complete our tasks, such as GIS definition and WebXR or Flutter and web-based 3D Frameworks.

For the time being some of the aspects of the project are not fully developed, because of the lack of resources available, but the usage of state-of-the-art technologies was crucial for the user experience and of course the functionality of this project's entirety.

All technologies used for the implementation will be pictured and analysed in detail. Some of the maths required in certain situations will be explained and pictures will hopefully illuminate the practical vision of certain technologies such as AR, VR or even the three-dimensional representation of the Map.

Graphics

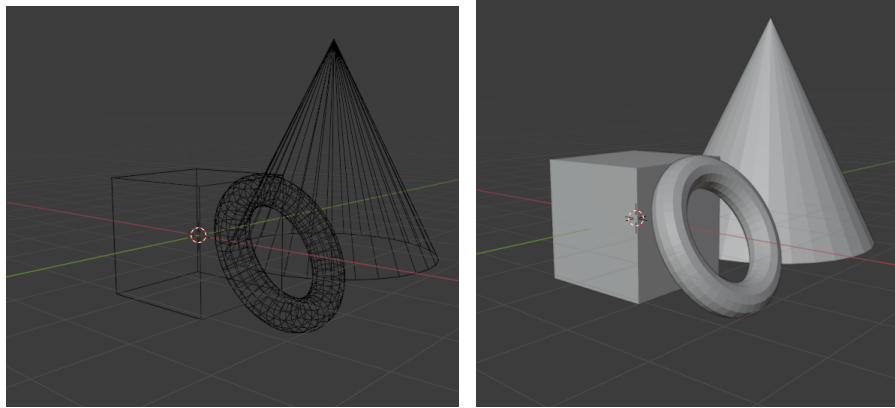
Computer graphics handles -as a definition - almost everything on a computer that is not referred to as text or music. Nearly every computer nowadays can generate a fair amount of decent graphics, and users have come to desire to be able to handle their computers via the use of icons and images instead of typing.

When we refer to computer graphics, we think about illustrating visuals on computers, commonly known as rendering. Drawings, photographs, movies, and simulations can all be used to generate things that do not exist yet or may never exist. Alternatively, they could be photos from locations we cannot see, such as medical snapshots from the inside of your body.

Humanity worked hard to improve computer graphics' ability to emulate real-world conditions. We want computer graphics to be more naturalistic in terms of lighting, colouring, and the build of various materials, not simply in terms of appearance.

An assemblage of geometric objects may be displayed, or rendered, on a computer. A lusterless surface having three or more distinct edges is considered on each side of an object. The computer is used to determine how each item appears in perspective view before drawing the silhouettes on the screen.

The following image was constructed in wireframe mode as if the structure were assembled of straight wires.



1.1 Wireframe

1.2 Sign Shading

Only the domains of the borders visible to the viewer are formed using a process known as "hidden line removal". By reducing overlapping lines and causing items to appear solid, the effect assists in the comprehension of arcane situations.

"Sing shading" on revealed item surfaces enhances our understanding of their shapes and locations.

Textures & Materials

Textures and Materials are the apparatus that the rendering engine uses to generate the model. You can designate materials for your model and notify the rendering engine on how to manage the light when it irradiates the surface. Textures are used to add colouring to a material utilizing a colour or an "albedo" mapping, or sometimes to add colour to a surface feature using a bump map or normal map. It can also be used to transform the model itself using a displacement map.

Bump mapping

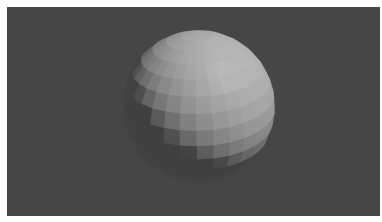
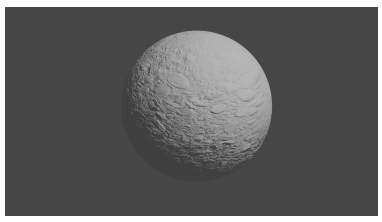
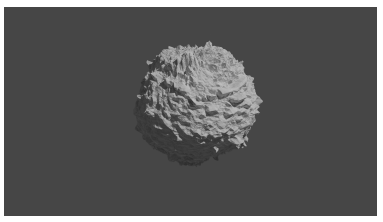
These are textures that hold the vehemence, which is the proximate height of the pixels as seen by the camera. The pixels seem to be shifted towards the surface normals by the necessary distance. (A "bump" is just an expulsion along the existing unaffected normal vector of the face.) You can use a greyscale image or an RGB texture intensity value.

Normal mapping

These are pictures that keep the direction and normal direction directly to the RGB values of the picture. It's much more precise because you can simulate pixels shifting in any direction, rather than affecting pixels moving along a line out from the face. However, The downside of normal maps is that unlike bump maps, which are easily drawn by hand, you are usually required to generate a normal map from the geometry that has a higher resolution than the geometry on which you apply the map.

Displacement mapping

Displacement mapping permits you to utilize the position of vertices on the geometry developed by the texture input. Unlike standard mappings and bump mappings that deform shading to construct the illusion of bumps, displacement maps create real bumps, creases, etc. on the actual model, itself. Therefore, mesh deformations can cast shadows, mask other objects, and alter everything you can change in the actual geometry.

| | | |
|---|---|---|
|  |  |  |
| <i>1.3 Simple Sphere</i> | <i>1.4 Normal + Bump</i> | <i>1.5 All applied</i> |

Rendering

Three-dimensional rendering is a 3D computer graphics procedure that transforms a 3D mesh into a 2D image on your monitor output. 3D rendering may contain photorealistic or non-photorealistic styles.

Rendering is the definitive process of creating a true two-dimensional image or animation from the set scene. This can be analogized to taking a photo or setting a scene after the actual formation is finished. An assortment of special rendering methods has been conceived throughout the years. These scope from the admirably unrealistic wireframe rendering to more refined techniques such as scan line rendering, polygon-based rendering, ray tracing, and radiosity.

Rendering can take from several days to split seconds for a single image/frame to be generated completely. Normally, we use different methods for real-time rendering or photorealistic.

Real-Time Rendering

Real-time rendering is a branch of computer graphics that focuses on creating and analyzing pictures in real time. The word was mostly usually applied to immersive 3D computer graphics created using a graphics card (GPU). A video game that speedily produces shifting 3D scenes to create the sensation of motion is one version of this notion.

Throughout their inception, computer systems have been capable of producing 2D visuals such as basic pictures, polygons, and lines in real-time. Nonetheless, for conventional Von Neumann architecture-based systems, drawing complicated three - dimensional objects fast remains a formidable undertaking. Icons (2D pictures) which might imitate 3d computer graphics were an early solution for this problem.

There are also additional visualization approaches available now, including such as ray tracing and B. Screening.

Computer systems nowadays can construct graphics quickly enough to provide the sensation of motion whilst also collecting human input using such approaches and improved technology. This means that viewers may reply to the generated image in real time, creating an immersive experience.

Virtual Reality

A three-dimensional picture or environment which is generated by digital means to simulate a type of reality that may be interacted with in a seemingly real or material way by a person wearing the characteristic electronic equipment, such as a helmet with a screen inside or gloves with detectors, is called a VR experience. Virtual reality (VR) is an experience comparable to or different from reality. Applications for virtual reality enclose entertainment, education, and business.

Currently, traditional virtual reality systems employ virtual reality headsets or multi-projected environments to give realistic sights, sounds, and other sensations that simulate a user's physical presence.

A virtual reality user may see the virtual world, move everywhere permitted inside it, and interact with virtual objects or products.

One method of experiencing virtual reality is through simulation. Driving simulators, for instance, can provide the onboard operator the idea that he or she is steering a real automobile by mimicking automobile motion caused by user involvement and supplying visual, motion, as well as audio and visual inputs.

Avatar image-based vr technology allows individuals to participate in the simulated environment through the use of genuine video together with an avatar. To participate in the 3D networked virtual environment, a regular avatar or a video can be utilized. Users can select the amount of engagement they desire depending on the system's capabilities.

One method of experiencing virtual reality is through simulation. To participate in the 3D distributed virtual environment, a regular avatar or a video can be utilized. Users can select the amount of engagement they desire based on the system's capabilities.

Augmented Reality

A technique that superimposes a computer-generated picture over a user's viewpoint of the factual world, resulting in a synthesized view.

Augmented reality (AR) is a type of experience in which designers supplement features of users' physical surroundings with computer-generated input.

Designers create digital inputs that respond in real-time to changes in the user's environment, frequently through movement, ranging from music to video, graphics to GPS overlays, and more.

Augmented Reality vs. Virtual Reality

On the other hand, virtual reality engages individuals in a whole different environment, one created and illustrated by computers. Users may be immersed in an animated set or a picture of a real-world place entrenched in virtual reality software. Using a virtual reality viewer, users may look up, down, or in any other direction as if they were genuinely there.

| AUGMENTED REALITY | VIRTUAL REALITY |
|--|---|
| Layers on top of real-world elements. | An Immersive experience that alters your perception to make you feel like you are in another world. |
| Enhances what you see with computer-generated images or graphics to your view. | Provides a fully digital environment with no real elements. |

| | |
|---|---|
| Requires AR-Compatible device. | Requires a VR headset or/and additional equipment. |
| Users can interact and change their perception of the world in real-time, while they can easily distinguish reality from virtual enhancement. | Users might experience mobility problems while remaining fully immersed in the virtual environment. |

Mixed Reality

Mixed Reality is the merging of the physical and virtual worlds to create new habitats and depictions in which actual and computer generated elements live and interact together in real time. Cyberspace is sometimes described as a connected virtual reality. Simulated reality is a potential virtual reality that is as totally immersing as real life, enabling for a close-to-lifelike experience or maybe virtual eternity.

A mixed reality environment is built utilizing a range of digital technologies, whereas Mixed Reality refers to the high-level interweaving of the physical and virtual environment. They can be anything from small portable devices to whole rooms, and each has a distinct set of uses in a variety of disciplines.

- Cave Automatic Virtual Environment
 - It is a tiny space within a larger outer room where people are encircled by projected panels above, below, and around them. Surround sound and 3D glasses are used in conjunction with the projections to provide the user with a feeling of perspective that is intended to mimic the existing reality. CAVE systems were being used by engineers building and testing prototype products since their beginning. Following the launch of the CAVE, this very same researcher developed the CAVE2, which improves on the original CAVE's inadequacies. The original projections have been changed with 37 megapixels 3D LCD screens, network connections have been fitted to connect the CAVE2 to the internet, as well as a more precise camera system which allows the environment to alter as the user wanders around it.
- Head-up display
 - A head-up display (HUD) is a display that is displayed into a user's field of vision and gives extra information without disturbing or forcing them to look away from the area in front of them. A common HUD contains three parts: a projector for overlaying the HUD's visuals, a combiner for displaying the

graphics, and a computer for combining the two additional components and doing any real-time computations or alterations. HUD concepts were primarily applied in military applications to aid fighter pilots in combat, but they were later made to help in all areas of flying, not just fighting. Pioneer's Heads-up system, which replaces the driver-side mirror, was one of the earliest implementations of HUD in automobile transport. Pioneer's Heads-up system, which substitutes the driver-side window shade with a display that projected navigation data onto the route in front of the operator, was one of the pioneering usages of HUD in vehicle transportation. Since then, auto manufacturers such as GM, Toyota, Audi, and BMW have added some type of head-up display to their products.

- Head-mounted display
 - A head-mounted display (HMD), which may be worn over the entire head or in front of the eyes, is a device that projects a picture right in front of the user's eyes using one or two lenses. Its uses include medical, entertainment, aviation, and engineering, and it provides a level of visual immersion that standard displays do not. Consumers in the entertainment business want head-mounted displays, and major technology firms are creating HMDs to complement their existing offerings. These head-mounted displays, however, are virtual reality displays that do not incorporate the physical environment. Popular augmented reality HMDs, on the other hand, are more suitable for business situations.
- Mobile devices
 - Mobile devices, especially smartphones and tablets, have gradually improved in terms of CPU power and mobility. While initial smartphones displayed a computer-generated interface on

an LED screen, recent smartphones have quite a toolset for making augmented reality apps. These capabilities encourage developers to combine multiple computer images over footage of the physical world. Pokémon GO, which was introduced in 2016 and has 800 million downloads, was the first massively acclaimed augmented-reality smartphone game. Google Maps has been updated with AR navigation directions superimposed into the streets in front of the user, as well as an expansion of their translation program to overlay translated text over actual lettering in over 20 different languages.

Geographic Information System (GIS)

GIS or Geographic Information System is a widespread technology that is actually pretty popular for many parts of scientific or engineering culture of research and developing. It is a digital system that creates, stores, interprets, and maps different kinds of information about anything contained along with meta-data about its location. GIS links information to maps by combining location with other sources of distinguishing data.

This lays the framework for analysis and mapping, which are used in research and almost every other industry. Users may utilize GIS to better grasp patterns, linkages, and geographical context. Among some of the benefits of employing these sorts of technologies are effective communication and efficiency, which leads to better organization.

Advantages and Disadvantages

There is a variety of data that can be viewed and inventoried using GIS or a geographic information system.

From

- natural resources,
- wildlife,
- cultural resources,
- wells,
- springs,
- water pipes,
- facets,
- roads,
- streams,
- and homes.

You can view and calculate the quantity, or density, of a particular item in a particular area. However, there is still much you can do with GIS technology.

Here are a few advantages of utilizing Geographical information systems:

- They are capable of enhancing organizational cohesiveness. In order to collect, analyse, organize, and display all sorts of spatially relevant data, GIS incorporates hardware, software and even data.
- With the use of GIS, you can also explore, query, visualize, comprehend, and analyse data in a variety of ways to identify trends, connections, and patterns that may then be represented in the form of globes, graphs, maps, and reports.
- By presenting data in a style that can be swiftly and readily shared, the geographic information system hopes to assist in providing answers to queries and solving issues.

- GIS technology can also be integrated into any enterprise information system framework.
- And there will be countless job opportunities.

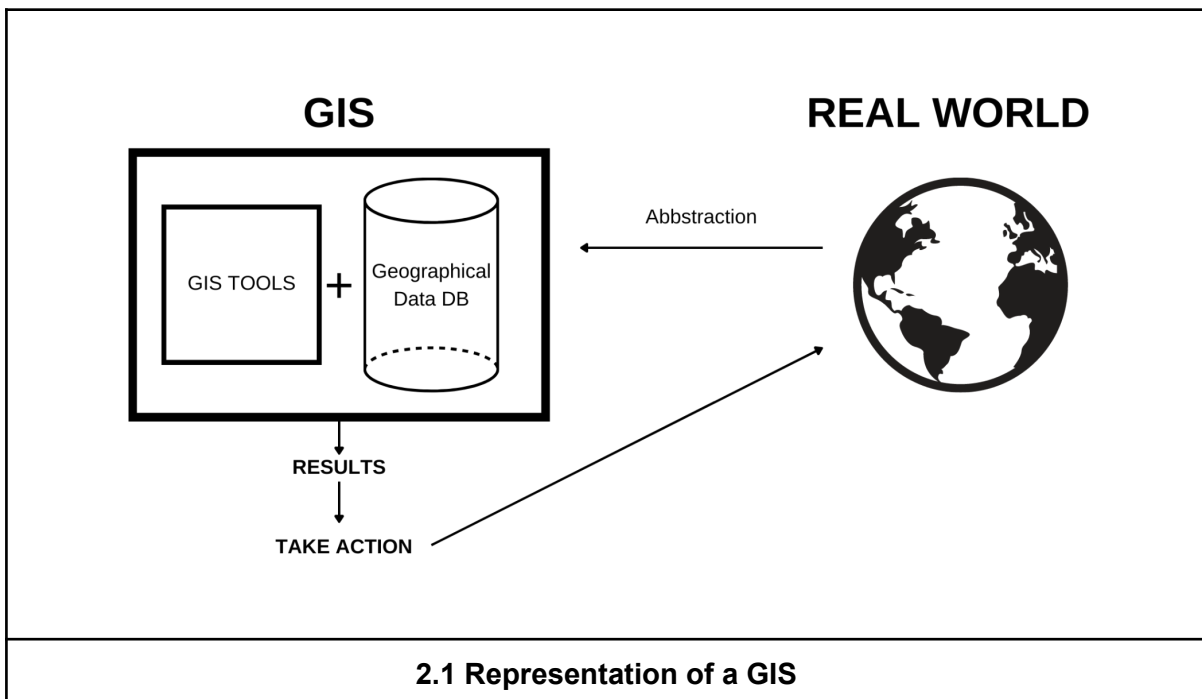
These are some of the benefits that can be achieved by using GIS technology. Given the use of the above techniques, this can be seen as a major decision.

On the other hand, there are some drawbacks that can occur with the use of GIS technology. And a few of these negatives are:

- GIS software can be considered quite expensive.
- In order to do several other activities effectively, you are also required to enter a lot more data.
- Because the Earth is spherical, geographical inaccuracy increases with increasing scale.
- The GIS layer costs a lot more when logical errors occur when it comes to realtors trying to understand an engineer's design for a GIS map or GIS utility.
- You also may fail to start or start additional efforts to fully implement GIS.

These are one of the pitfalls of using GIS technology and may or may not occur in some cases. The above disadvantages can be considered on a case-by-case basis, depending on how efficiently GIS technology is being used.

In general, it is true that GIS technology has the ability to offer both strengths and weaknesses to everyone. Apart from the advantages and disadvantages mentioned above, there are definitely many possibilities when using GIS technology, except for a few drawbacks.



Implementation

In this project, we try to make distant visits to monuments more accessible to the public by only having a laptop, a smartphone or a tablet. To achieve this we are using a big variety of technologies to produce our front-end and back-end as well. The initialisation started by thinking about how would users first interact with our application. Is it going to be Web Based or Mobile Based? The truth is that both the user Experiences should be accessible because some of the users' devices might not have the capacity to achieve such heavy-duty as 3D and VR or AR immersions, in the first place.

Web VR is not new to the world but the amount of clever and useful frameworks does not, at least not yet, really cover the needs of the developing demands.

In testing and early development, I started creating 3D worlds in various ways that seemed fitting by that time being.

Secondary came the user interface and the CSS implementation. In contrast to the common approach of HTML interfaces, I did not choose to use Bootstrap or React as my main course for UI production. As time goes by, people tend to use their smartphones more than their regular personal computers. The capacity of a website or an application, being able to change its appearance to the most efficient and functional way for the users' experience, is essential. Further beyond we will discuss the CSS framework I chose to be the most convenient and functional in manners of complexity and adaptability.

Additionally, addressing the problem of mobile devices and compatibility I also made a distinction between PC use and mobile devices, by programming, almost the same App for Android and iOS. Cross-platform applications are the state of the art in programming. Different frameworks have been developed throughout the previous decade. For our implementation of this task, we chose **Flutter** as our main framework of concept.

Addressing the previous considerations and making my way through the project, I took advantage of the main concept of Geolocation systems and I made it so that everything is around or on a map in the middle of the page. Users will be able to navigate through a map, which we will combine with the 3D concept. Information on the subject such as geographical location is acquired by using an **API** developed by EMIK. The map was provided after several trials and errors on different systems by **Mapbox**, which will be extensively described as a tool later on.

In conclusion, 3D worlds - as mentioned before - are considered to be developed in **three.js**, for convenience. Further applications of 3D frameworks were discussed to be used, but most of them couldn't fit the level of workability for further exploration and development in terms of evolution.

Web XR

There are a lot of "Reality" words floating around these days, considering the 3D World. Even though there are many commonalities among them, it might be impossible to keep track of Virtual, Augmented, and Mixed Reality. This exact API seeks to offer the building blocks for everything mentioned. And, because they do not want to be confined to one aspect of virtual and augmented reality, they use "X" which represents "Your Reality Here," not as part of a boring acronym.

The WebXR Device API gives you the ability to input (pose info from headsets and control systems) and output (hardware projection) features that are often related to Augmented Reality and Virtual Reality devices. It enables you to create and host VR and AR experiences on the web.

The advantages of performing XR on the web include instant deployment to any XR platform using a WebXR-enabled Web Browser.

- Future-proof applications should remain available on devices without requiring the deployment of new code, as new VR and AR technologies are often introduced.
- An engagement may support portable and head-mounted VR and AR devices with a single release. Just a few small changes to the code are needed to support AR and VR concurrently.
- There is no need for the use of big downloads or app stores; visitors may enjoy your experience without leaving your website.
- Because WebGL, that has been available since 2011, handles the processing, you have accessibility to its comprehensive ecosystem of tools as well as its sizable and vibrant developer community.

At this certain point a lot of groundbreaking technologies benefit from Web XR such as

- [A-Frame](#)
- [Babylon.js](#)
- [model-viewer](#)
- [p5.xr](#)
- [PlayCanvas](#)
- [React-XR](#)
- [Sumerian](#)
- [Three.js](#)
- [Unity](#)
- [Verge3D](#)
- [Wonderland Engine](#)

Case studies

Some might assume that this API will be primarily used for game development given the promotion of early XR devices to gamers. Although there will probably be a lot more "long tail"-style material than big-budget games given the history of the WebGL API, which is extremely equivalent. The majority of XR content will likely be available online in regions that don't cleanly fit into the app-store models of all of the

major VR/AR hardware vendors are using as their primary distribution methods, or where the material itself is not authorized by the store restrictions.

Video

The web has already demonstrated to be a wonderfully effective platform for the delivery of video, and Three-Sixty Degrees and 3D video are both quite popular. Similar to the "Full screen" buttons present in current video players, a "View in VR" button will be visible when an XR-enabled video player detects the presence of XR equipment. When the user touches the button, a film that mimics natural head movement shows in the headset. A more immersive experience may be achieved by displaying standard 2D video in the headset as though the user were seated in front of a theatre sized screen.

Object/ data visualization

Simple 3D visualizations can be delivered on websites using WebXR, often as a progressive improvement to their more traditional representations. A more realistic feeling of scale may be achieved while viewing 3D architectural pre-visualizations, maps, medical imaging, models, and basic data visualization in VR and AR. When internet information is readily available with only a link or click, few people would even think about installing a native software for such use scenarios.

Demos for shopping applications do incredibly well. Depending on the gear and software, websites might be anything from a simple photo carousel to an interactive 3D model that lets users experience a walkthrough in virtual reality, giving users the impression that they are actually in the property.

Giving customers a low-friction experience avoids the need to persuade people to install a huge (and potentially hazardous) executable, which is advantageous for both users and developers.

Experiences in the arts

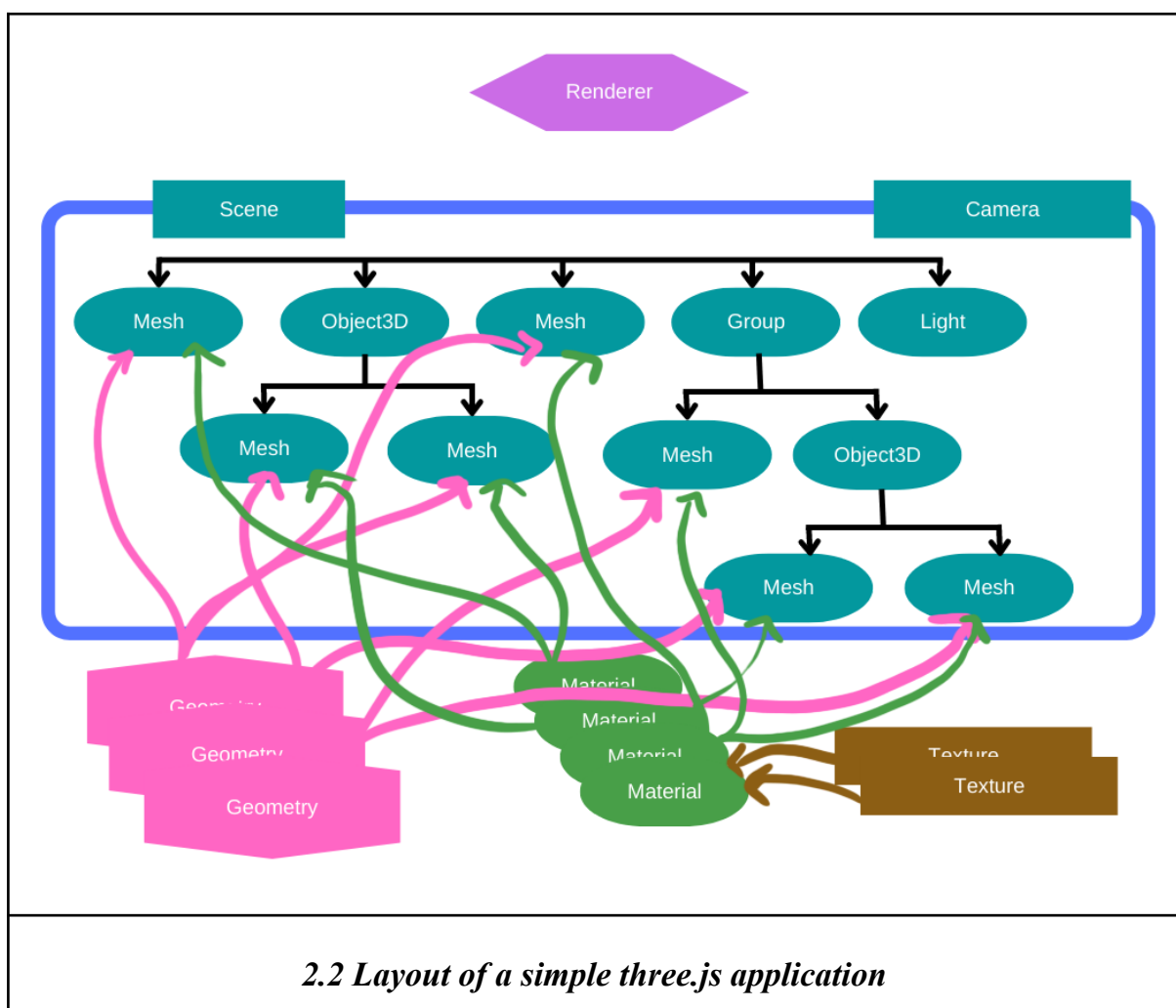
For creatives interested in exploring the possibilities of a different medium, VR presents an exciting canvas. Shorter, hazier, and more imaginative experiences are frequently inappropriate for an app-store strategy since the perceived effort required to download and install a native executable may be excessive compared to the content provided. These kinds of apps are much more appealing since they provide a seamless way to view the experience because the web is fleeting. Additionally, by using a single code base, designers may target the greatest number of devices and platforms, which makes it simpler to attract viewers to the content.

Three.JS

Three.js is a 3D framework that aims to create exhibiting 3D material on a browser as simple as possible.

Three.js is sometimes misguided with WebGL since it frequently, but not always, utilises WebGL for drawing 3D. WebGL is a very low-level method for generating simply lines, points, and triangles. You'll want a significant amount of code to execute anything fascinating with WebGL, and three.js can help with that. If you utilised WebGL explicitly, you would have to write each of the features it offers, such as reflections, scenes, lighting, materials, texturing, and 3D maths.

A three.js program necessitates the establishment of several elements and their interconnectivity. Here's a layout of a simple three.js application.



What to look for in the diagram above.

- A renderer is present. Undoubtedly, this is three.js's main objective. A 2D image of the portion of the 3D scene that lies within the camera's

frustum is rendered to the canvas using the Renderer, Camera, and Scene that you supply.

- Several elements, including several Mesh objects, a Scene object, Group, Light objects, Object3D, and Camera objects, are included in a scene graph, which is a structure that resembles a tree. A Scene object defines the basis of the scene graph and includes details like the colour of the background and the presence of fog. These elements provide us a parent-child tree-like hierarchical arrangement and show us where as well as how events happen. In respect to their parents, children are positioned and oriented.
- Mesh objects offer the ability to draw a certain Geometry using a specific Material. Both Geometry and Material elements may be combined using multiple mesh objects. We could need two mesh objects to specify the position and orientation of each blue cube, for example, if we wanted to generate two blue cubes in two different places. We would only require one Geometry and one Material to describe the colour blue in order to keep the vertex information for a cube. The same Geometry and Material objects may be related to both mesh objects.
- Vertex data of a geometric object are represented by geometry objects such as a dog, cube, plane, sphere, cat, person, tree, building, and so on. Three.js has a wide range of geometric primitives. You may also import geometry from files and generate your own geometry.
- Surface parameters are constituted by Material objects so that geometry to be created, such as the colour to use and the shine. One or more texture objects may also be referred to as Materials; they may be used, for instance, to wrap an image around the surface of a geometry.
- Texture objects are used to be pictures that are imported from picture files, which are produced by a canvas element, or generated from another scene.
- Many types of lightning are represented by Light objects.

*Take note that the camera in the scene is split in and out of the scene graph in the diagram. This is to show that, unlike the other objects in three.js, a Camera does not need to be in the scene graph to operate. A Camera, like everything in the scene, will be oriented and moved relative to its parental object, just like any other object.

Mapbox

Mapbox GL JS is a JavaScript client-side framework for creating online maps and web apps using Mapbox's cutting-edge technology. Mapbox GL JS allows you to show Mapbox layouts in either a internet browser or mobile, add user interaction, and personalize the map interface in any project.

Mapbox GL JS applications include:

- Geospatial data visualization and animation.
- Real - time data queries and filtration map elements.
- Using a Mapbox style to place your data within layers.
- Customized client-side data is dynamically displayed and styled on a map.
- Sequences and representations of 3D data.
- Algorithmically inserting pop-ups and markers to maps.

The "GL" in Mapbox GL JS relates to Mapbox GL, a graphics framework that uses OpenGL to generate two-dimensional and three-dimensional Mapbox maps as dynamic visualizations in any suitable internet browser.

Client-side processing is used by Mapbox GL JS. Mapbox GL JS maps are dynamically drawn in the computer instead of on a server by mixing vector tiles with rules and guidelines, allowing the maps' style and presented data to vary in response to the user activity.

Mapbox GL JS maps may be made up of many layers that contain both visual components and map information. Each layer specifies how the renderer may depict particular information in the user's computer, and the renderer employs several layers to display the map on the display.

The `addLayer` Mapbox GL JS function adds a Mapbox style layer to the map's style. The only argument necessary for `addLayer` is a Mapbox style layer entity. It also supports an explicit `before` argument, which specifies the ID of an existing layer that should be inserted before the new layer. If this option is omitted, the renderer will create the layer upon the map's surface.

Approach

Setting Up the Interface

Considering the use case scenarios of a user navigating through a map, there has to be easy and instant access to the users' demand just by being redirected to the page.

The first things first were to decide if the CSS layout for the main interface and the easiest way to integrate an API with functions and new endpoints for the page. The first idea was to use Node.JS for the back-end operations and the Ejs framework to manage JSON data with JavaScript.

Considering that in future integration, there will be a user-related system for up-loadable content by users, I chose to frame it with PHP, so I can use SQL for the main interface, as long as the entity relations are not that complex.

Why SQL?

MongoDB and SQL databases are on different ends of the backend spectrum. The former works with large unconstructed datasets that are chaotic, whilst the latter works with structured data that is orderly. Both worlds have benefits and weaknesses, and they are intended for distinct sorts of use cases. In this post, we will compare MongoDB to SQL databases (specifically, the MySQL database), as well as discuss how we can do MongoDB analytics with the same ease that we can with SQL databases.

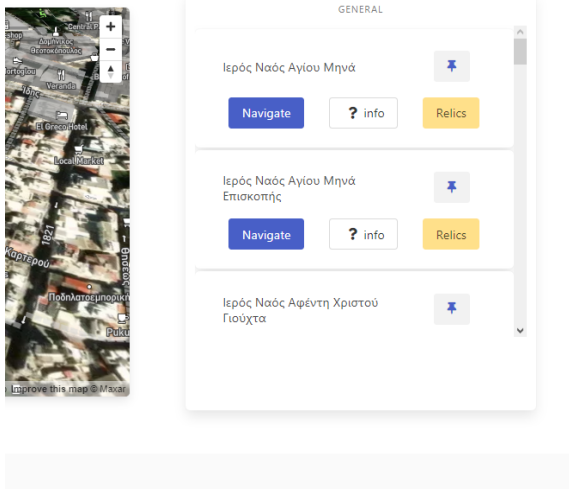


3.1 Main Interface

The Interface on first Sight

The main functions that are integrated into the Users' interface are defined with simple buttons and icons, so it is made clear what they do, and the solution is a mostly simple design which makes the use cases more accessible to different ages or levels of comprehension.

In each tile of the monuments we will find four buttons considering Map navigation and page redirections:



- **Navigate:** Redirects us to the 3D generated space and AR/VR immersion page

- **Info:** Reveals a sidebar with information about the monument and certain pictures from it.
- **Relics:** Redirects us to the page of the monument for us to see the relics that were found inside.
- **Go to (Pin):** Triggers the `goTo(x,y)`, which brings the pin of a certain monument to the centre of the Map View.

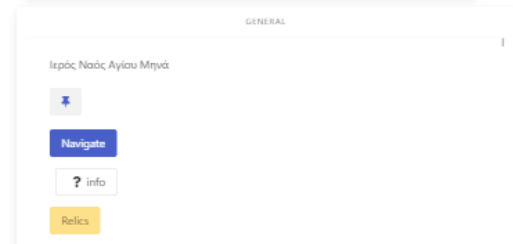
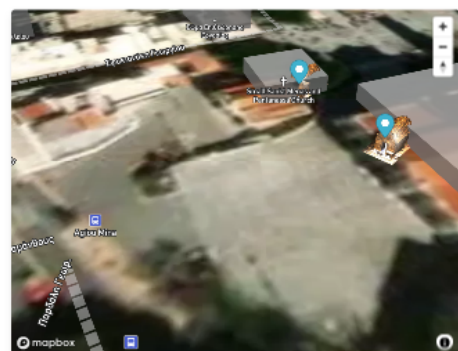
```
function goTo(x, y) {map.setCenter([x, y]);}
```

That is all considering the right part of the screen and the main Functionalities that are accessible in a blink of an eye.

As we mentioned before, most modern users use their smartphones to browse the internet, so we made sure that can be used with the same efficiency in both portrait and landscape mode.

By using columns and rows from the CSS framework, we control the position of every HTML element with ease and make it possible to change to a horizontal view as well as use the responsiveness of the Map that comes integrated with the MapBox API, without destroying the main interface's appearance or functionality.

Buttons have the same responsiveness, but the float state is disabled because of the touchscreen usage.



Monument

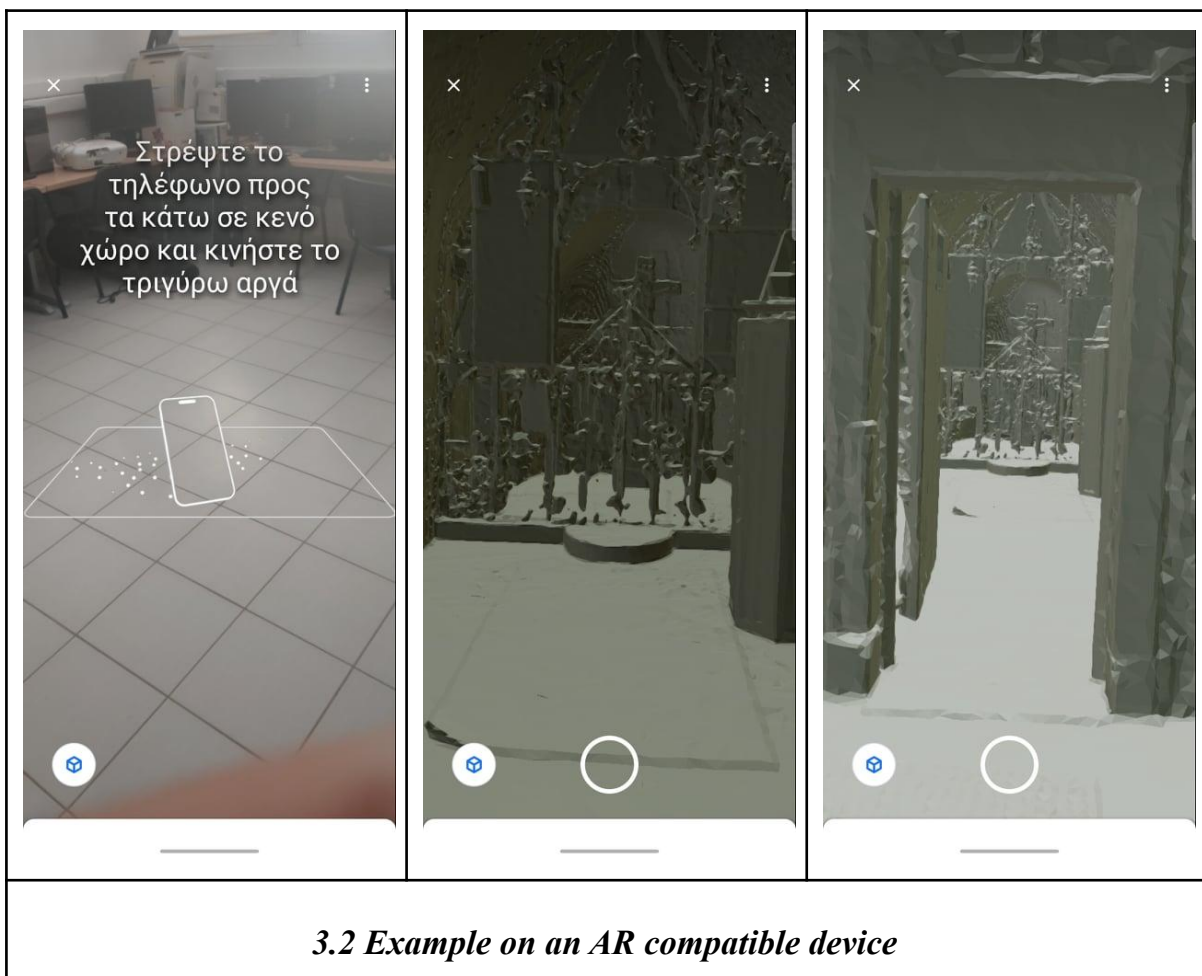


Bonanza

If a user decides to press on the relics button, then he will be redirected to the page dedicated to the monuments' relics, that are found until our present time.

On the left side of the page, the user will be viewing a 3D model of the monument which is turnable, 360° available for viewing, and it is also zoomable. We achieve that by using Google's model-viewer, and by using this plugin, and also if our device is capable of doing so, we can view the monument in our own phone via AR camera.

Throughout the AR camera, we can place the model wherever we want on a flat surface and adjust it's size so that we can view it as we like to.




After this, on the left side of the page we can see a list of the relics that were found with every and each one of the names and Identification code given.

We retrieve this kind of information via the API that we mentioned earlier, and we arrange them with simple PHP code, which makes a single tile for every element of the JSON we get.


Ιερός Ναός Αγίου Μηνά

things we found there



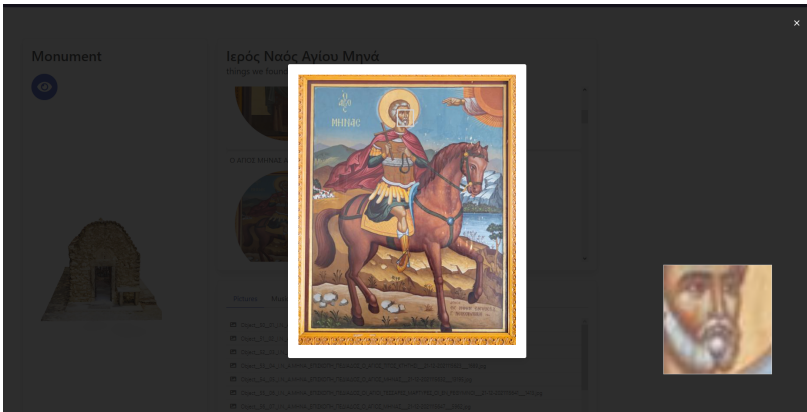
Ο ΑΓΙΟΣ ΜΗΝΑΣ ΑΡ ΕΙΚ.5

IAK-IN-AME-00002-K-EIK-00005



3.3 The List of found Relics

And to top it off we added a magnifying glass that's working with plane CSS and javascript.



3.3 Magnifying glass mode

The Map

On the right side of the screen we can see a Map. This Component is brought by MapBox.

We initialize the map with this simple piece of JavaScript code:

```
const map = new mapboxgl.Map({
  container: 'map',
  style: 'mapbox://styles/mapbox/satellite-streets-v11',
  zoom: 18,
  center: [25.134144396166306, 35.34073423355571],
  pitch: 50,
  antialias: true // create the gl context with MSAA antialiasing, so
  custom layers are antialiased
});
```

We define the ID of the canvas component in our HTML markup how much zoomed we want it to be, as well as the pitch of angle on the map view. The style argument refers to the image that will be used. For this project, I chose that the satellite view is well-fitted, and I centred it in the centre of Heraklion, keeping in mind that the coordinates are inversely proportional to the ones on Google Maps.

A Mapbox style is a document that defines the visual look of a map, including what data to show, how to represent it, and then how to design the info when rendering it. A style document is a JSON object containing nested and root-level attributes. These attributes are defined and described in this standard.

Layers in Mapbox GL JS are asynchronous since they are remote. As a result, code that connects to Mapbox GL JS frequently uses event binding to alter the map at the appropriate moment.

We separate the layers from the map in a list element, so we can add new features as layers one above the other with lists' functions, such as push and pop. Most of the layers load on the map once it is declared during the loading session with `map.on('load')`.

To call map, use `map.on('load', function())` in this sample code. After the map's resources, including the style, have been loaded, `addLayer` is called. If you ran `map.addLayer` immediately away, it would throw an error because the style to which you want to add a layer does not yet exist.

```

map.on('load', () => {

  // Insert the layer beneath any symbol layer.

  const layers = map.getStyle().layers;

  const labelLayerId = layers.find(

    (layer) => layer.type === 'symbol' && layer.layout['text-field']

  ).id;

```

The Mapbox Streets layer titled "building." Data on building heights from OpenStreetMap is included in the vector tileset.

```

map.addLayer(

  {

    'id': 'add-3d-buildings',

    'source': 'composite',

    'source-layer': 'building',

    'filter': ['==', 'extrude', 'true'],

    'type': 'fill-extrusion',

    'minzoom': 15,

    'paint': {

      'fill-extrusion-color': '#aaa',

      // 'fill-extrusion-pattern': 'raster',

```

As the user zooms in, provide a seamless transition effect to the buildings by using a "interpolate" phrase.

```

    'fill-extrusion-height': [

      'interpolate',

      ['linear'],

```

```

        ['zoom'],
        16,
        0,
        16.06,
        ['get', 'height']
    ],
    'fill-extrusion-base': [
        'interpolate',
        ['linear'],
        ['zoom'],
        16,
        0,
        16.06,
        ['get', 'min_height']
    ],
    'fill-extrusion-opacity': 0.7
  }
},
labelLayerId
);
));
map.addControl(new mapboxgl.NavigationControl());

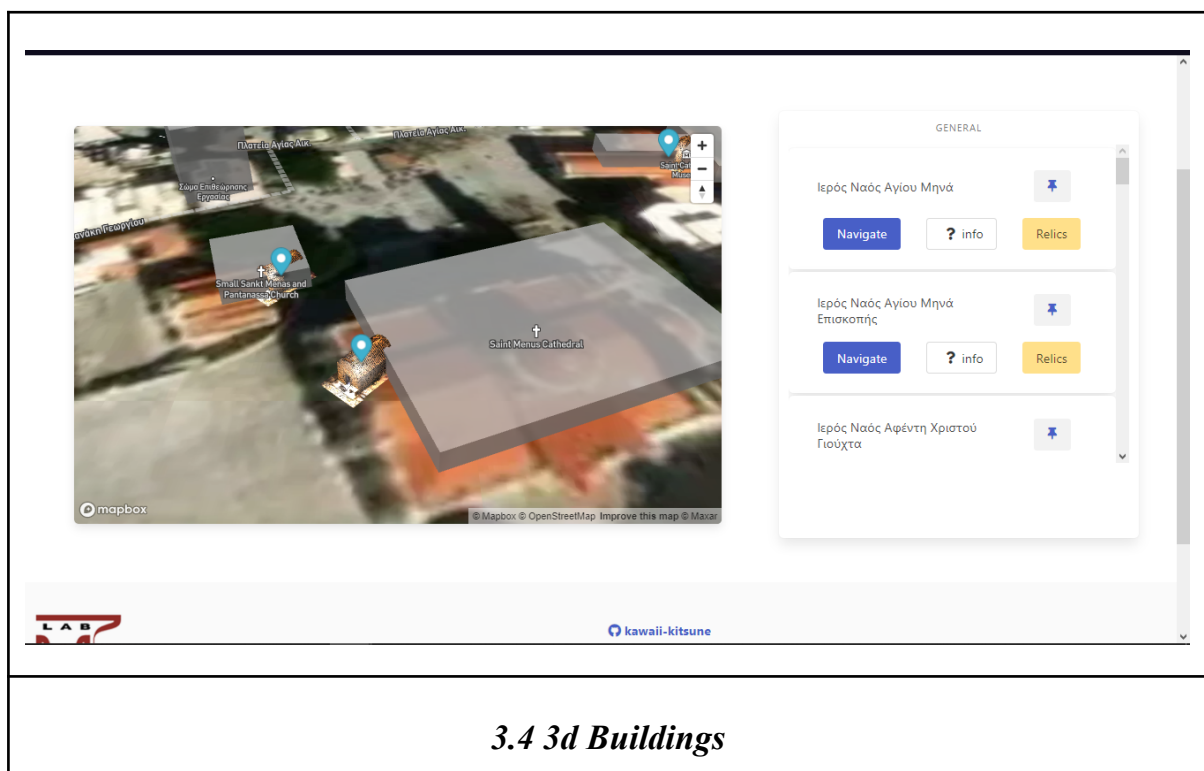
```

We use `addLayer` to add a `fill-extrusion` layer that displays building heights in 3D.

A fill-extrusion style layer creates a map with one or more filled (and optionally stroked) extruded (3D) polygons. A fill-extrusion layer can be used to customize the extrusion and visual appearance of polygon or multipolygon features.

The data provided for the polygons to be made are provided by the Mapbox API, and they give us a number of polygons which later are converted to 3D by adding height vectors and creating faces from the points created.

And we add Navigation controls so we can change our view by rotating our compass. The compass and the Zoom buttons can be found on a NavigationControl control.

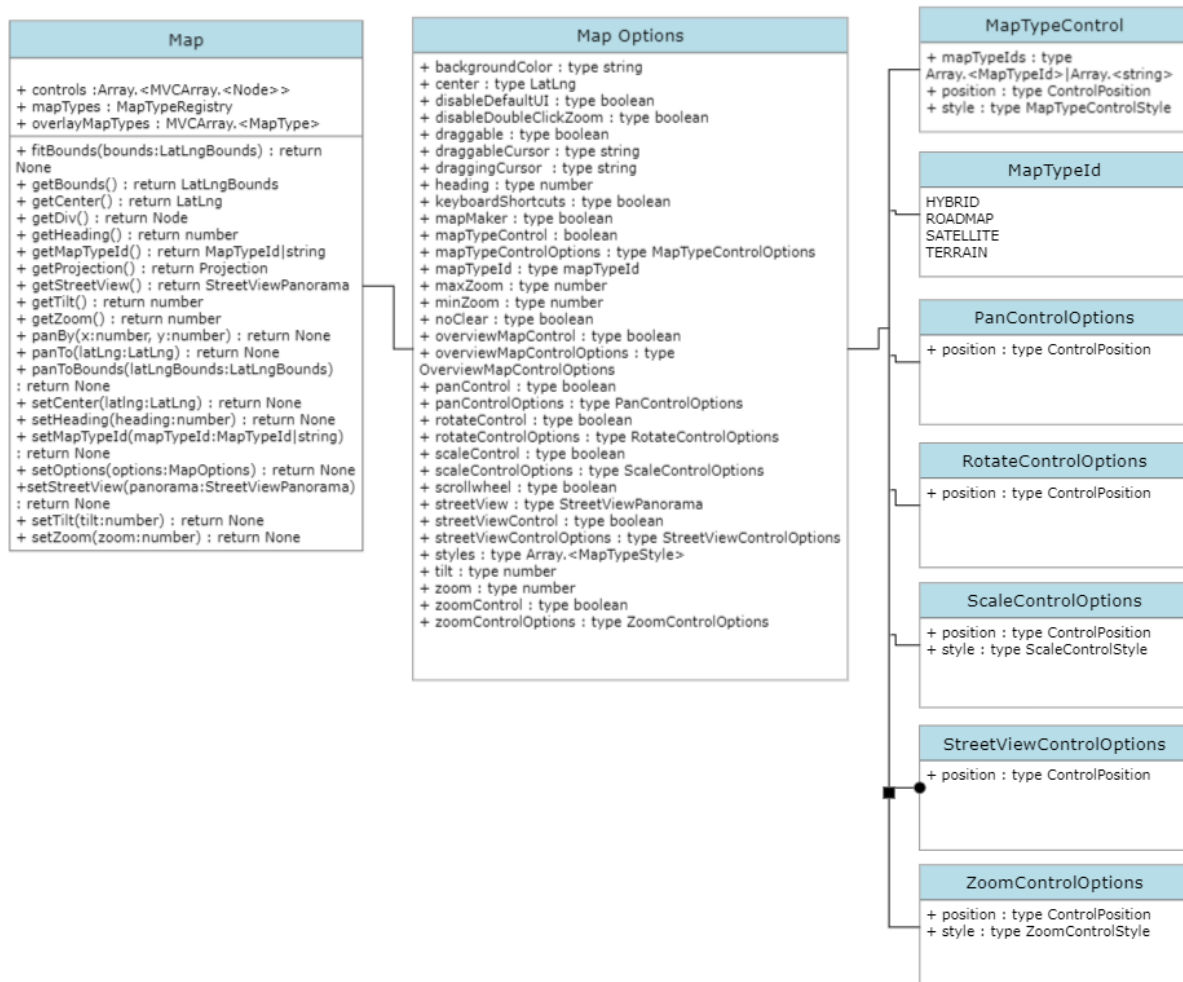


3.4 3d Buildings

We can actually represent the Map class in a class diagram, in which we can see the Attributes and Structures that are affected by the Methods of our parent class, the Map.

- Map options: affect the appearance of the map.
- MapTypeControl: Affects Controls that navigate you throughout the terrain.

- MapTypeId: The type of the Map that is used to display the data or the Map Tile that is displayed.
- PanControlOptions: Position of the options' control panel about the point of view.
- RotateControlOptions: Position of the options' control panel about the rotation on the map.
- ScaleControlOptions: Position of the options' control panel about the scale of the map.
- StreetViewControlOptions: Position of the options' control panel of the map's location display.
- ZoomControlOptions: Position of the options' control panel about the zoom in/out.



The whole representation process is actually represented by markers on the map loaded by a function that manages the data that is coming from the API.

A Marker is another component that we use to represent a location on the Map, and its constructor looks like this.

```
new Marker(options: Object?, legacyOptions: Options?);
```

| Name | Description |
|---|--|
| options.anchor String default: 'centre' | a string describing which area of the Marker should be placed closest to the coordinates specified by Marker#setLongLat. |
| options.clickTolerance Number default: 0 | The most pixels a user's mouse cursor can move when they click a marker for it to be recognized as a legitimate click (as opposed to a marker drag). The default is to take the click from the map. The capacity for clickTolerance. |
| options.color String default: '#3FB1CE' | If options.element is omitted, the default marker's color will be used. By default, light blue is used. |
| options.draggable Boolean default: false | a boolean value indicating if dragging a marker to a different spot on the map is possible. |
| options.element HTMLElement? | Marker DOM element to utilize. |
| options.offset PointLike? | The distance in pixels that a PointLike object should be displaced from the center of the element by. Negatives point up and to the left. |
| options.pitchAlignment String default: 'auto' | Map centers the Marker on the map's plane. The Marker is aligned to the viewport's plane. Auto automatically equates to rotationAlignment's value. |
| options.rotation Number default: 0 | The marker's angle of rotation with relation to the selected rotationAlignment, determined in degrees. The marker will turn |

| Name | Description |
|--|---|
| | clockwise if the positive value is present. |
| options.rotationAlignment String default: 'auto' | In order to keep a bearing when the map rotates, the map aligns the Marker's rotation with respect to it. Regardless of map rotations, viewport aligns the marker's rotation with respect to the viewport. Viewport is identical to auto. |
| options.scale Number default: 1 | The scale that will be applied to the default marker, if any. No element is offered. The usual scale translates to a width and height of 27 and 41 pixels, respectively. |

Seeing the Map Component without its function makes it seem less useful, but in any case, this is the main mean of navigation throughout the map and webpage as well.

loadMarker() is a custom function that we made to serve the purposes of the project. In our function you will notice that we are using setLang(), addTo() and setPopup() which are methods() of the Marker Class. In the end, we set a new Pop-up component on the marker which is another class that is bound to our Marker anew.

| Name | Description |
|---|---|
| options.anchor string? | a string set by Popup#setLngLat that designates the area of the popup that needs to be located closest to the location. |
| options.className string? | CSS class values divided by whitespace to add to a popup frame. |
| options.closeButton boolean default: true | If true, a close button will show up inside the popup's upper-right corner. |

| | |
|--|--|
| options.closeOnClick boolean default: true | If true, popup closes when the map is to be clicked. |
| options.closeOnMove boolean default: false | If true, the pop-up will close when the map moves. |
| options.focusAfterOpen boolean default: true | If true, the pop-up tries to focus the first shown focusable element in the pop-up's internal. |
| options.maxWidth string default: '240px' | A string (for instance, "300px") that specifies the popup's maximum width as a CSS attribute. Set this parameter to "none" to guarantee that the popup will resize to suit its content. For a list of the possible values, go to the MDN documentation. |
| options.offset (number PointLike Object)? | <p>the popup's position defined as a pixel offset which is applied as</p> <ul style="list-style-type: none"> • an individual number indicating the distance to the popup's location • a PointLike specifying a constant offset • a Points object specifying an offset for each anchor position. <p>Negative offsets indicate up and left.</p> |

We are using the function `setHTML()` from the set of methods that this class carries.

setHTML(HTML){

Sets the pop-up's content to the HTML provided as a string.

This method does not perform HTML filtering or sanitization and must be used only with trusted content.

Parameters

HTML(string) A string representing HTML content for the pop-up.

Returns

Popup: Returns itself to allow for method chaining.

}

```
function loadMarker(x, y, URL, id, fldname) {
    var marker = new mapboxgl.Marker().setLngLat([x, y]).addTo(map);
    marker.setPopup(new mapboxgl.Popup().setHTML(
        '<div class="columns is-centered">' +
        '<div class="column">' +
        '<span class="is-size-6" id="name-' + id + '">' + fldname +
        '</span>' +
        '</div>' +
        '<div class="column is-centered">' +
        '<a href="/OpenWorld/php/views/3dXdom/model-view.php?modelId=' +
        id + '" target="_blank" class="is-centered" rel="noopener noreferrer">' +
        '<button class="button is-link is-centered" value="' + id +
        '">Navigate</button>' +
        '</a>' +
        '<button class="button is-outlined mx-2 model-info-button"
        onclick="openNav(this)" ' +
        'value=' + id + '><i class="fas fa-question mx-2"></i>
        info</button>' +
        '<a href="/OpenWorld/php/views/bonanza.php?modelId=' + fldname +
        '" target="_blank" class="is-centered" rel="noopener noreferrer">' +
        '<button class="button is-warning is-centered" value="' + id +
        '">Relics</button>' +
        '</a>' +
        '</div>' +
        '</div>') .setMaxWidth("330px"));
}
```

One of the most important features of our interface is the 3D Elements! For the Map interior, we created an extra Tile on which 3d models are established.

We firstly need the exact location of the model to load on the map and the origin of the model on the system, or an online file URL. We set the altitude on the tile which is always 0 and the rotation degree on the map.

```
function loadModel(x, y, url, id) {  
  
  const modelOrigin = [x, y];  
  
  const modelAltitude = 0;  
  
  const modelRotate = [Math.PI / 2, 0, 0];  
  
  const modelAsMercatorCoordinate =  
mapboxgl.MercatorCoordinate.fromLngLat(  
  
    modelOrigin,  
  
    modelAltitude  
  
  );  
}
```

Settings for positioning, rotating, and scaling the 3D object on the map

```
const modelTransform = {  
  
  translateX: modelAsMercatorCoordinate.x,  
  
  translateY: modelAsMercatorCoordinate.y,  
  
  translateZ: modelAsMercatorCoordinate.z,  
  
  rotateX: modelRotate[0],  
  
  rotateY: modelRotate[1],  
  
  rotateZ: modelRotate[2],  
  
  /*
```

Since the CustomLayerInterface requires units to be in MercatorCoordinates and the 3D model is in actual world meters, a scale conversion must be used.

```

*/

scale: modelAsMercatorCoordinate.meterInMercatorCoordinateUnits()

};

const THREE = window.THREE;

```

Setting up a 3D model's custom layer according to the CustomLayerInterface

```

const customLayer = {

id: '3d-model'+id,

type: 'custom',

renderingMode: '3d',

onAdd: function (map, gl) {

this.camera = new THREE.Camera();

this.scene = new THREE.Scene();

```

Create two three.js lights in the scene for illuminating the model

```

const directionalLight = new THREE.DirectionalLight(0xffffff);

directionalLight.position.set(0, x, y).normalize();

this.scene.add(directionalLight);

const directionalLight2 = new THREE.DirectionalLight(0xffffff);

directionalLight2.position.set(0, 70, 100).normalize();

this.scene.add(directionalLight2);

```

Use the three.js GLTF loader to add the 3D model to the three.js scene

```

const loader = new THREE.GLTFLoader();

loader.load(

```

```

url,

(gltf) => {

this.scene.add(gltf.scene);

}

);

this.map = map;

// use the Mapbox GL JS map canvas for three.js

this.renderer = new THREE.WebGLRenderer({

canvas: map.getCanvas(),

context: gl,

antialias: true

});

this.renderer.autoClear = false;

},

render: function (gl, matrix) {

const rotationX = new THREE.Matrix4().makeRotationAxis(

new THREE.Vector3(1, 0, 0),

modelTransform.rotateX

);

const rotationY = new THREE.Matrix4().makeRotationAxis(

new THREE.Vector3(0, 1, 0),

modelTransform.rotateY

);

```

```
const rotationZ = new THREE.Matrix4().makeRotationAxis(  
    new THREE.Vector3(0, 0, 1),  
    modelTransform.rotateZ  
);  
  
const m = new THREE.Matrix4().fromArray(matrix);  
  
const l = new THREE.Matrix4()  
    .makeTranslation(  
        modelTransform.translateX,  
        modelTransform.translateY,  
        modelTransform.translateZ  
    )  
    .scale(  
        new THREE.Vector3(  
            modelTransform.scale,  
            -modelTransform.scale,  
            modelTransform.scale  
        )  
    )  
    .multiply(rotationX)  
    .multiply(rotationY)  
    .multiply(rotationZ);  
  
this.camera.projectionMatrix = m.multiply(l);
```

```

this.renderer.resetState();

this.renderer.render(this.scene, this.camera);

this.map.triggerRepaint();

}

};

map.on('style.load', () => {

map.addLayer(customLayer, 'waterway-label');

});

}

```

The 3D Virtual Environment

Setting Up the Cube

We made a function that create a Plane Geometry of 10000x10000 tiles on which we assign a picture for each tile.

```

function addGround() {
    var groundTexture = new
THREE.TextureLoader().load('/OpenWorld/assets/images/floor.png');
    groundTexture.wrapS = groundTexture.wrapT = THREE.RepeatWrapping;
    groundTexture.repeat.set(10000, 10000);
    groundTexture.anisotropy = 16;
    groundTexture.encoding = THREE.sRGBEncoding;
    var groundMaterial = new THREE.MeshStandardMaterial({
        map: groundTexture
    });
    mesh = new THREE.Mesh(new THREE.PlaneBufferGeometry(10000, 10000),
groundMaterial);
    mesh.position.y = 0.0;
}

```



```

mesh.rotation.x = -Math.PI / 2;
mesh.receiveShadow = true;
mesh.name = 'floor'
scene.add(mesh);
}

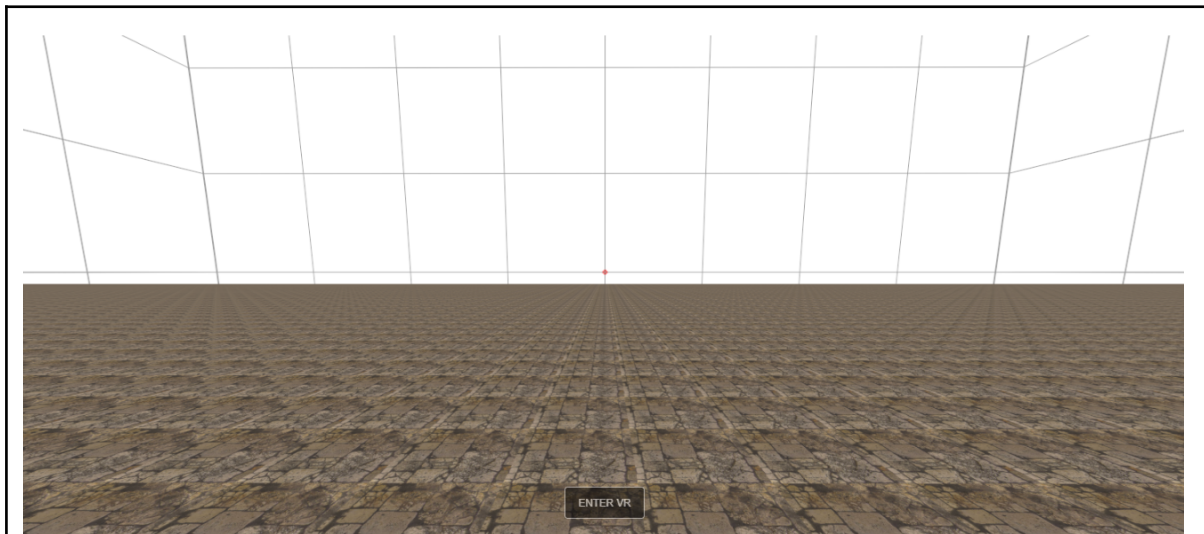
```

Following the same principal for the we create acube that include everything inside it and we assign a larger picture for each side of it.

```

function createCube() {
  const loader = new THREE.CubeTextureLoader();
  const texture = loader.load([
    'https://threejsfundamentals.org/threejs/resources/images/grid-1024.png',
    'https://threejsfundamentals.org/threejs/resources/images/grid-1024.png',
    'https://threejsfundamentals.org/threejs/resources/images/grid-1024.png',
    'https://threejsfundamentals.org/threejs/resources/images/grid-1024.png',
    'https://threejsfundamentals.org/threejs/resources/images/grid-1024.png',
    'https://threejsfundamentals.org/threejs/resources/images/grid-1024.png'
  ]);
  scene.background = texture;
}

```



4.1 Results of the scenery functions

Creating the XR experience

Then we make a function about when we have to enable the XR renderer and enable the VR button so we can access the immersion through the glasses or any immersive device.

VRButton.createButton() performs two critical functions: It generates a button indicating if your device is VR compatible. Furthermore, when the user presses the button, it starts a VR experience.

```
function enableXR() {  
    renderer.xr.enabled = true;  
  
    document.body.appendChild(VRButton.createButton(renderer));  
  
    document.getElementById('VRButton').style.background = 'rgb(0, 0,  
0) none repeat scroll 0% 0%'  
}
```

A big part of our VR experience is that we can interact with several things inside it, in the process of experiencing anything, we have to create the concept of hands in our 3D Scene. We cannot have real hands in the virtual world, that is why we have to create them anew. For this reasons, we will project our controllers in the scene.

To build the Controllers, we have to access the XRControllerModelFactory to recognize the 3D model for the controller model that have to be loaded for hands.

And following the API call, than we have to take these steps to add “our hands” to the scene:

- Assign the controllers an entity.
- We push the model on the entity.
- We give it an ID.
- Furthermore, we create the grip.
- Add it on the Dolly.
- And following repaint the Dolly on the scene.

```
function buildControllers() {

    const controllerModelFactory = new XRControllerModelFactory();

    const geometry = new THREE.BufferGeometry().setFromPoints([new
THREE.Vector3(0, 0, 0), new THREE.Vector3(0, 0, -1)]);

    const line = new THREE.Line(geometry);

    line.name = 'line';

    line.scale.z = 0;

    const controllers = [];

    for (let i = 0; i < 2; i++) {

        const controller = renderer.xr.getController(i);

        controller.add(line.clone());

        controller.userData.selectPressed = false;

        scene.add(controller);

        controllers.push(controller);

        controller.name = "controller-" + i;

        const grip = renderer.xr.getControllerGrip(i);

        grip.add(controllerModelFactory.createControllerModel(grip));

        dolly.add(grip);

        dolly.add(controller);

        scene.add(dolly);
    }
}
```

```

    }

    return controllers;
}

```

When the controllers have been built then we have to assign functions, in order to flag some of the states of the controller, for later use upon user's input insertions such as **Select** and **Squeeze**. The point of the flags is to realize if the user is continuously pressing the select button or squeezes the trigger.

```

const controllers = buildControllers();

function onSelectStart() {

    this.userData.selectedPressed = true;

}

function onSelectEnd() {

    this.userData.selectedPressed = false;

}

function onSqueezStart() {

    this.userData.squeezePressed = true;

}

function onSqueezEnd() {

    this.userData.squeezePressed = false;

}

controllers.forEach((controller) => {

    controller.addEventListener("selectstart", onSelectStart);

    controller.addEventListener("selectend", onSelectEnd);

    controller.addEventListener("squeezestart", onSqueezStart);

    controller.addEventListener("squeezeend", onSqueezEnd);

```

```
});
```

Movement

As we mentioned above, to make ourselves interact with the environment surrounding us in the VR session we needed hands, and we made clear that we have two inputs for sure on any controller:

- Squeeze and
- Select

For the computer to understand if the user is giving any input at all is simple, it just as simply checks if a button is pressed, but what if the user gives a continuous input?

The XR renderer shares the same frame rate as our immersion Device and refreshes our picture at this specific rate in time. That's why we made a function that keeps track of the **select** and **squeeze** flags, that we mentioned earlier, so we can configure the input's duration and continuity for each controller.

- controller: the controller we are seeing.
- delta: the time fractal.

```
function handleController(controller, delta) {  
  
    if (controller.userData.squeezePressed) {...}else{...}  
  
    if (controller.userData.selectedPressed) {...}else{...}  
  
}
```

But before any further elaboration on the movement subject, I would like to state some facts about movement and senses in the VR world, as I myself had some trouble getting accustomed to the testes and giving myself quite a number of nausea episodes.

Motion Sickness in Virtual Reality

Many users have stated that, even if they don't normally have motion sickness, they might feel unbearably uncomfortable after only a brief session upon trying out the groundbreaking technology for the first time.

What causes motion sickness, you ask? When you play a virtual reality game, your eyes register the motions generated around you. These can range from meteors zipping past in virtual space to riding on the back of a galloping horse. The inner ears may also detect virtual activity all around oneself.

Regardless of what is displayed in your Virtual reality headset, your joints and muscles perceive that you are still seated and not moving. These confusing

messages are sent to your brain by your eyes, inner ears, and body all at the same time. When you have motion sickness, your brain gets disoriented and confused.

The goal of VR game creators is to generate an occurrence known as presence. While engaging in a game, presence refers to both the physical and conceptual experiences of "being there" rather than where you are. It is the presence of the user that makes well-designed virtual reality sessions strong and lifelike. However, it is also what causes VR motion sickness to appear similar to motion sickness caused by physical movements. The sole distinction between VR and other kinds of motion sickness is that there is no genuine motion happening throughout a VR game.

So, avoiding motion sickness in our virtual immersion, I decided to move quite differently than the orthodox way of VR games. Most of the time we are introducing users in moving a body along with the camera, which is their head.

In our project, we move the head along with a body, but we actually decide where this body will go.

We, first, make the Camera and then add it to a 3D object we called Dolly, which will be our vehicle for the whole session.

```
var scene = new THREE.Scene();
const fov = 75;
const aspect = 2; // the canvas default
const near = 0.1;
const far = 50;
var camera = new THREE.PerspectiveCamera(fov, aspect, near, far)
const dolly = new THREE.Object3D();
dolly.position.z = 0.1;
dolly.add(camera);
var dummyCam = new THREE.Object3D();
camera.add(dummyCam);
```

Locomotion

Making a straight line from the controller and moving in the direction it suggests would be an easy way to navigate.

This is not ideal since you must constantly point the joystick downward, which is annoying, and you may move infinitely far by aiming towards the horizon, which doesn't feel natural.

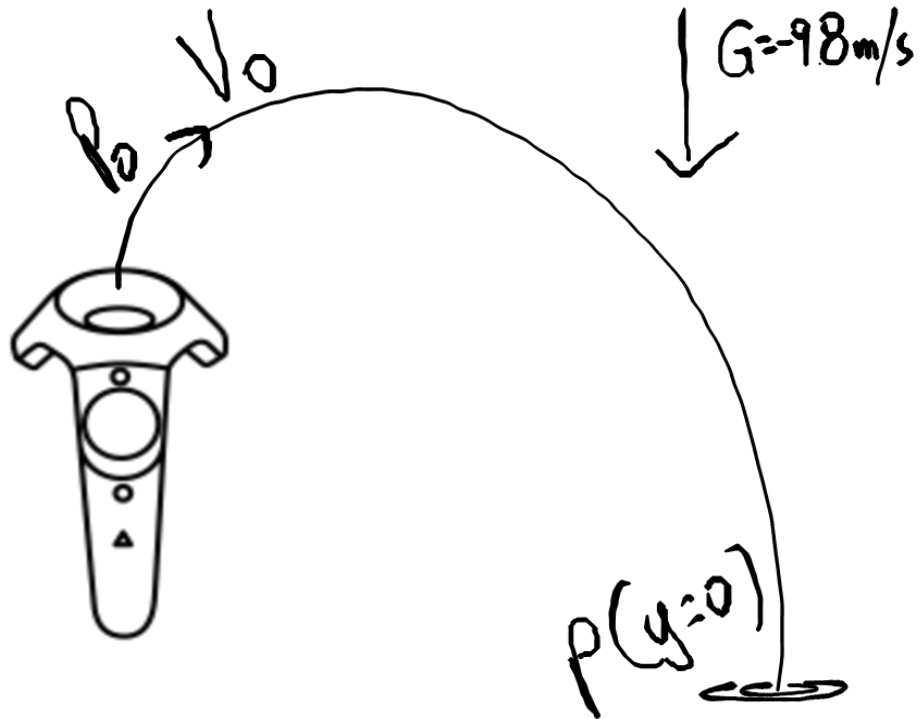
In this method, the controller tosses a ball, and we teleport to the location where the ball lands. You've undoubtedly seen this kind of VR movement before if you've ever worn a VR headset.

```
const lineSegs=10;
const lineGeo = new BufferGeometry();
```

```

const lineGeoVertices = new Float32Array((lineSegs + 1) * 3);
lineGeometryVertices.fill(0);
lineGeometry.setAttribute('position', new
BufferAttribute(lineGeoVertices, 3));
const lineMat = new LineBasicMaterial({ color: 0x888888, blending:
AdditiveBlending });
const guideline = new Line( lineGeo, lineMat );

```



4.1 Line illustration

To make the line in the 3D world, we need to figure out the line geometry: Since we don't have the curvature of the line, we've set all of its vertices to zero for the time being.

Then we have to conduct some arithmetic to figure out precisely where it lands. We realize a few small details: we're operating under typical gravitational acceleration, which is approximately 10m/s or 9.8m/s down; we in addition know the speed, orientation, and also the beginning location of the bullet.

We may deduce that the velocity changes relatively to time as the original velocity + gravity is compounded by time, as well.

$$V(t) = V_0 + Gt$$

You can derive location from velocity by integrating it to time, which is what we'll do next.

$$P(t) = \int V(t)dt = \int V_0 + Gt = V_0 * t + \frac{Gt^2}{2} + C$$

In this scenario, the residual constant from the integration (c) tends to work into being our first starting point. So that we get the correct position formula, which happens to be a quadratic equation:

$$P(t) = 0.5 * Gt^2 + Vt + P$$

This method may be used to calculate the location at any moment in time. This may be used to form the arc.

For convenience, the result is saved to the vectors inVec in THREE.js vector terms such as this:

```
function positionAtT(inVec, t, p, v, g) {  
  inVec.copy(p);  
  inVec.addScaledVector(v, t);  
  inVec.addScaledVector(g, 0.5*t**2);  
  return inVec;  
}
```

Because our line has ten vertices, we must determine the location of the ball at each of those ten points until it reaches the ground. To do so, we must determine the value of t whenever it hits the ground.

To determine where the line ends, calculate the preceding equation, where $y = 0$.

$$0 = 0.5 * Gt^2 + Vt + P$$

The general solution, fortunately, is known as yielding:

$$t = \frac{-V_{oy} \pm \sqrt{V_{oy}^2 - 2 * P_{oy} * G_y}}{G_y}$$

There will be two solutions to this problem: one in the future and another in the past. We only care about the future one, so we can ignore the other, which then in JavaScript is:

Controller starting position

```
const p = guidingController.getWorldPosition(tempVecP);
```

Set Vector V to the direction of the controller.

```
const v = guidingController.getWorldDirection(tempVecV);
```

Set the initial velocity to 6m/s

```
v.multiplyScalar(6);
```

Calculate t

```
const t = (-v.y + Math.sqrt(v.y**2 - 2*p.y*g.y))/g.y;
```

Given our updated settings, we could now alter each arc geometry vertex to represent it in three dimensions. This is something we do every frame, so I included it in the loop that creates the animation.

```
const vertex = tempVec.set(0,0,0);  
for (let i=1; i<=lineSegs; i++) {
```

Set vertex to current position

```
positionAtT(vertex, i*t/lineSegs, p, v, g);
```

Copy to the Array Buffer

```
vertex.toArray(lineGeoVer, i*3);  
}  
guideline.geometry.attributes.position.needsUpdate = true;
```

And that is how we create the arced vertex.

There are two ways to collect user input in WebXR. First, each WebXRInputSource is associated with a gamepad object that performs similarly to the Gamepad API. Since this system depends on polling, you need to check it every frame to determine which keys are being pressed. Because different equipment has different button and movement mappings, using it might be challenging.

The select and squeeze actions on the session are the best approach to do it. Such events are triggered by whatever the VR system chooses and/or the corresponding squeeze/grab button if one is provided.

In THREE.js, these events are available on the controller objects. You may listen for them as follows:

```
const controller1 = renderer.xr.getController(0);
controller1.addEventListener('selectstart', onSelectStart);
controller1.addEventListener('selectend', onSelectEnd);
```

There are several methods for changing the user's location. We nest the camera and controllers in a single cluster and move the said cluster along.

This implies that we control a variety of 3D spaces relevant to the user:

- Our setting: This is the interactive virtual environment.
- Camera: This group is for devices that should remain relative to the user, such as interfaces and cameras.
- Use with caution because the user will not be able to gaze at or away from them.

All elements relevant to the user, such like controllers or local interfaces, move also with the camera group.

Three steps to teleportation:

- The user chooses a teleportation point.
- Determine the vector from the user's legs to the spot in question.
- That vector should be used to offset the cameraGroupPosition.

```
const feetPos = renderer.xr
  .getCamera(camera)
  .getWorldPosition(tempVec);
feetPos.y = 0;

const p = guidingController.getWorldPosition(tempVecP);
const v = guidingController.getWorldDirection(tempVecV);
v.multiplyScalar(6);
const t = (-v.y + Math.sqrt(v.y**2 - 2*p.y*g.y))/g.y;
const cursorPos = positionAtT(tempVec1,t,p,v,g);
// Offset
const offset = cursorPos.addScaledVector(feetPos,-1);
```

Upon simplifying the method above. We made this locomotion alternative method with a straight line, when we receive a select input from the user.

```
if (controller.userData.selectedPressed) {
```

```

    controller.children[0].scale.z = 10;

    tempMatrix.identity().extractRotation(controller.matrixWorld);
    raycaster.ray.origin.setFromMatrixPosition(controller.matrixWorld);

    raycaster.ray.direction.set(0, 0, -1).applyMatrix4(tempMatrix);

    intersects = raycaster.intersectObjects(scene.children, false);

    if (intersects.length > 0) {

        controller.children[0].scale.z = intersects[0].distance;

    }

} else {

    controller.children[0].scale.z = 0;

}

function onSelectEnd() {

    if(point!=null){

        dolly.position.x=point.x;

        dolly.position.y=point.y;

        dolly.position.z=point.z;

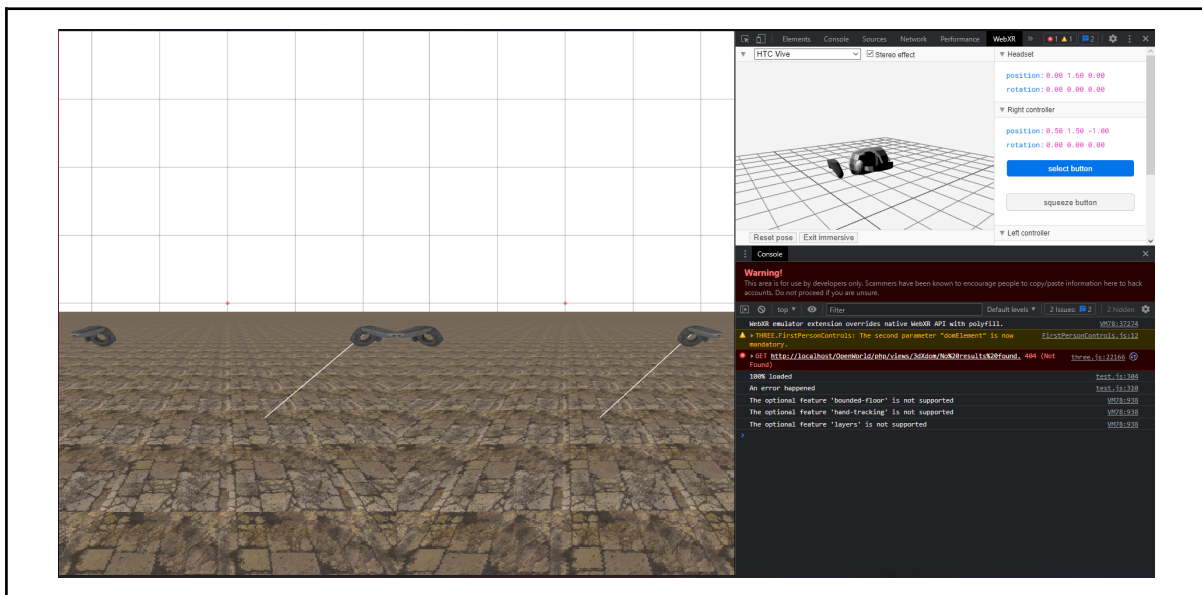
    }

    point=null;

    this.userData.selectedPressed = false;

}

```



4.2 Raycaster we made

On Press/ Head movement based navigation (Head Tilt Navigation)

The other way of moving in our character is by moving the dolly corresponding to our head's position and turning according to time.

Once we squeeze the trigger, Dolly will start to move slowly, taking us with it and will mimic our head movement on the x-axis, so it will turn each time we move our head on an angle.

```

if (controller.userData.squeezePressed) {

    const speed = 1;

    dummyCam.updateMatrixWorld();

    const quaternion = dolly.quaternion.clone();

    dummyCam.matrixWorld.decompose(

        dummyCam.position,

        dolly.quaternion,

        dummyCam.scale

    );
  
```

```
dolly.translateZ(-delta * speed);  
  
dolly.position.y = 0;  
  
dolly.quaternion.copy(quaternion);  
  
}
```

BIBLIOGRAPHY

A. FOREIGN

1. What is Computer Graphics?, Cornell University Program of Computer Graphics. Last updated 04/15/98. Accessed November 17, 2009.
2. Blender Docs, Bump and Normal Maps
3. Blender Docs, Displacement Maps
4. Balder, Norman I. "3D Object Modeling Lecture Series" (PDF). University of North Carolina at Chapel Hill. Archived (PDF) from the original on 2013-03-19.
5. "Fundamentals of Rendering - Reflectance Functions" (PDF). Ohio State University. Archived (PDF) from the original on 2017-06-11.
6. Malhotra, Priya (July 2002). Issues involved in Real-Time Rendering of Virtual Environments (Master's). Blacksburg, VA: Virginia Tech. Pp. 20–31. Retrieved 31 January 2007.
7. Haines, Eric (1 February 2007). "Real-Time Rendering Resources". Retrieved 12 Feb 2007.
8. Chang, Kang-tsung (2016). Introduction to Geographic Information Systems (9th ed.). McGraw-Hill. p. 2. ISBN 978-1-259-92964-9.
9. Milgram, Paul & Kishino, Fumio. (1994). A Taxonomy of Mixed Reality Visual Displays. IEICE Trans. Information Systems. vol. E77-D, no. 12. 1321-1329.
10. Tor Bernhardsen (2002). Geographic Information Systems: An Introduction, John Wiley & Sons
11. Three.js Fundamentals. <https://threejs.org/manual/#en/fundamentals>
12. WebXR Device API Explained. <https://github.com/immersive-web/webxr/blob/master/explainer.md#whats-the-x-in-xr-meaning>
13. What causes motion sickness in VR and what can you do to avoid it? <https://www.livescience.com/what-causes-motion-sickness-in-vr>
14. Using VR controllers and locomotion in THREE.js <https://ada.is/blog/2020/05/18/using-vr-controllers-and-locomotion-in-threejs/>

B. NATIVE

APPENDIX A

ROCK MODEL https://polyhaven.com/a/river_small_rocks

APPENDIX B