POLITECNICO DI TORINO

College of Computer Engineering, Cinema and Mechatronics

Master's Degree

in Cinema and Media Engineering

Master of science Thesis

Extended Reality Data Visualization for the Energy Transition

Tutors

Bottino Andrea

Bompard Ettore Francesco

Francesco Strada

Candidate

Benfante Antonio

Acknowledgments

"Muscles aching to work, minds aching to create - this is man."
— John Steinbeck, The Grapes of Wrath

"It occurs to me that our survival may depend upon our talking to one another."
— Dan Simmons, Hyperion

"The most important thing about a technology is how it changes people."
— Jaron Lanier

This thesis is an account of struggle against limitations, where software optimization does not aim to improve what is already there, but to make new things possible. To the academic reader, it may represent a modest step forward in the state of the art. To me, it is a thrilling journey of problem solving and learning. I want to remember to my future self that I did not choose it for its flashiness or prestige, but for the chance to expand what is possible with the technologies of my craft.

I extend my heartfelt thanks to the professors of my master's degree for the rigor, courage and vision of their work. Reflecting on my journey I appreciate their efforts to prepare us to ever-changing world.

I thank the staff of the Energy Security Transition Lab, for having focused my enthusiasm on a challenge where I could sharpen my skills.

Finally, I am especially thankful to my family, who made great sacrifices for me.

Summary

Introduction

Context

The Energy Security Transition Lab (EST) at the Energy Center serves as a hub for "Politecnico energy thought", focusing on energy transition and security. EST addresses the needs of institutional, industrial, and research stakeholders, both nationally and internationally, by offering a vision and in-house developed technologies. These solutions aim to create tangible societal impact.

EST embodies the concept of a "dynamic" think tank, distinguishing itself from traditional ones by emphasizing the development of methodologies and tools that enable continuous tracking and monitoring of system evolution, rather than producing static reports.

The lab adopts a holistic approach to model and analyse energy systems, which are inherently complex, dynamic, and multi-layered. This approach is "instance-based," meaning it directly addresses real-world stakeholder demands.

EST actively contributes to national and international discussions on energy transition and security, offering original insights through its models and analyses. It has established significant collaborations with various stakeholders, formalized through specific agreements that have delivered concrete outcomes.

As part of its efforts, EST has developed and continues to refine the "platform of energy" software ecosystem. This system comprises a suite of thematic applications designed for planning and scenario analysis across different and specific dimensions of the energy system, all within a multi-layered perspective.

Objectives

In this context, the need arose to effectively interface with local decision makers, providing accurate and readable perspectives for different backgrounds. In this regard, an environment designed for immersive data visualization called Decision Theatre has been created in the EST. The environment consists of an amphitheatre-shaped room equipped with a cylindrical projection surface that extends for 220° around the centre of the room.

To enhance participation and make full use of its innovative facilities, we were tasked to develop a proof-of-concept for an extended reality decision support system, whose role is to demonstrate the possibilities of the new infrastructure.

The Decision Theatre has already been successfully using this environment for two-dimensional data visualizations based on maps and graphs, but due to the great similarity of the structure with a VR dome it was decided to explore the possibilities of new types of three-dimensional visualization. This thesis deals with the experiments carried out in this area, the user experience design and software development that were carried out.

Results

The result was a Unity application that allows to consult all the data in the database Edifici 3D 2017 from ARPA Piemonte[1], as a proof of concept for the possibility of visualizing georeferenced data in 3D in real time with good performance. Among the important results achieved in the software development we report:

- fast conversion of the original JSON database into objects interactable in the application (approximately 4 minutes to import 300.000 building descriptions);
- cross-referencing between the 3D buildings dataset and the DTM (digital terrain model) dataset of Piedmont to also show the altitude information;
- graphic optimization to display a large number of buildings with greater performance than the tools available up to that point (web libraries or GIS programs);
- panoramic rendering on the projected screen of the Decision Theatre.

These results, and a rich set of possible future developments, allow us to state confidently that the current technology is mature to start developing feature rich and effective 3D data visualization frameworks.

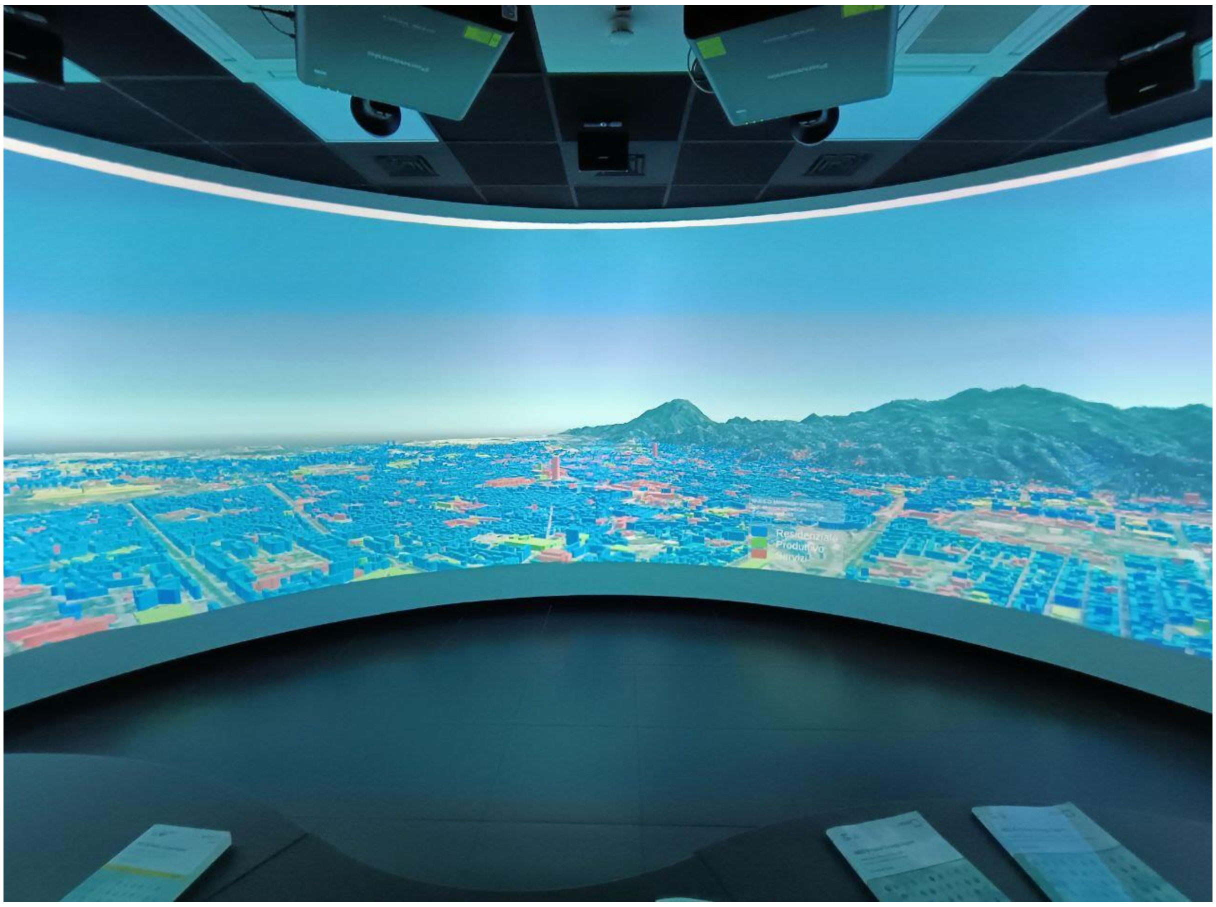


Figure 1: Prototype running inside the Decision Theatre

Decision theatres and decision support systems

Decision theatres are part of an effort to advance the state of the art of Decision Support Systems (DSS), a class of sophisticated computer applications designed to assist stakeholders in the decision making progress by analysing large volumes of data and simulating scenarios. A notable example of a very comprehensive DSS is the Energy Transition Model[3] developed by Quintel Intelligence. This model allows users to tune many different energy system parameters at a state or regional level, and formulates projections for the following years.

While providing very good comprehensive energy modelling and multicriteria decision analysis, such a DSS doesn't completely cover the needs of a smaller system such as a city, where geospatial data is desirable for studying energy networks and public transportation, and where shorter term estimates and more frequent consultation of data is desirable. This perspective validates the need for increasing the use of XR in our local decision support systems beyond the current trends.

Around the world Extended Reality is being applied to support or train[7 decision making in many settings, from navigation assistance to seafarers[4] to training and presenting information more effectively to first-responders[5]. This surge in application can be attributed to the latest advancements in this technology, and to its alignment with our current culture: it has been pointed out that we increasingly hunger for extending our sensory experiences in the digital world[6].

These benefits are not to be reaped without challenges, data integration and technological barriers due to the new visualization technologies and novel equipment are to be considered carefully when adopting and designing such a system. For this purpose, our proof of concept focused on leveraging the DT hardware at its best, handling correctly a large volume of data, and ease of use.

Development of a proof of concept

A subset of the Arpa Piemonte - Edifici 3D 2017[1] dataset was chosen for its detail, relevance to our area of study, and completeness, as it contains more than 300.000 buildings only in the chosen area including and surrounding Turin. To improve visualization, widely available Digital Terrain Model data and aerial photographs were added.

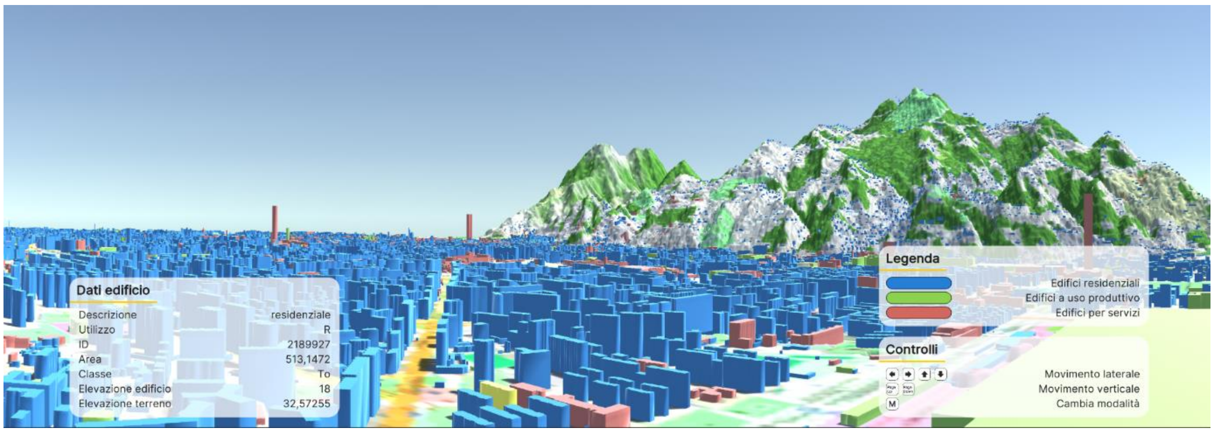
A special rendering system akin to those used in VR domes was developed for the cylindrical projection surface of the decision theatre to correctly fit the environment, and approximate a "Window effect", that we will explain in detail later on.

Prototype features and architecture

In this section, we share an overview of our prototype, before delving in the implementation details in the next chapters.

Usage

The prototype features a very simple 3D navigation system to easily move through the city. The first chosen controls were the tried and proven "WASD" along side Q and E for moving vertically; later with arrows and page scroll keys was added for improved user friendliness. The user can rotate the camera by moving the mouse. After the panoramic rendering was implemented, the possibility to look up and down was removed, because it was potentially jarring in a panoramic environment, especially when the controls would be handled by an user inexperienced with videogames. By pressing T the user locks their point of view and is able to consult the data of nearby buildings by hovering on them with their mouse. The buildings are highlighted when hovered on. Data and the control scheme is shown in a simple window overlaid on the view.



Dati edificio

Descrizione	residenziale
Utilizzo	R
ID	2189927
Area	513,1472
Classe	To
Elevazione edificio	18
Elevazione terreno	32,57255

Figure 2: Screenshot of final prototype

The following list presents its main features:

- 3D navigation;
- Panoramic rendering;
- Database preprocessing (chunking, mesh generation, serialization);
- Representing terrain from height map data;
- Rendering of buildings in accordance with heightmap data;
- Visualizing 3D building information.

The whole project is written in C# using the Unity game engine, whose features were found suitable this kind of industrial application. Among the most useful for this project we identify:

- Flexible 3D rendering pipeline
- Physics engine
- Terrain system
- User Interface Toolkit.

The software architecture can be seen as split in a backend and a frontend. The backend pre-processes the raw data and saves it in a way that is efficient to load in real time for the frontend, whose role is to display it with good performance and ease of use.

Frontend

Designing for rendering performance

The frontend has to contend with rendering all the buildings and the terrain, amounting to more than 24.000.000 polygons. Rendering performance in Unity depends upon several factors:

- Polygon count
- Material count
- Material complexity
- Texture sizes
- Draw calls count
- Game Object number

These factors are interrelated and are all possible intervention areas when attempting to improve performance. Many techniques have been developed to improve 3D rendering performance over the years, such as:

- Draw call batching
- Occlusion culling
- Levels of detail
- Texture mipmaps
- Texture atlasing
- Impostors
- Resource polling

Before moving on to explain the prototype's architecture, we will go over these techniques and explain how they were applied to our application, either by explicitly implementing them, or by leveraging the Unity's engine capabilities.

Draw Call Batching and Texture Atlasing

In modern 3D applications, the CPU delegates rendering calculations to the GPU. Unlike the CPU, GPUs are optimized for handling large numbers of parallel, independent calculations required in 3D graphics. As a result, their assembly is closely tied to the microarchitecture of the specific board family, making it highly complex and revealing of the details of the GPU manufacturer's design. For these reasons, developers don't interface with GPUs at the same low level as they do with CPUs; instead, they use graphics libraries. These libraries, such as DirectX or Vulkan, provide higher-level abstractions for interacting with the GPU, managing tasks like memory allocation, shaders, and draw calls.

However, every time a draw call is made (an instruction for the GPU to render an object), it incurs overhead as the CPU must prepare data, issue commands, and synchronize with the GPU. This back-and-forth can significantly slow down rendering if there are too many individual draw calls. Draw call batching addresses this problem by grouping multiple rendering commands into a single draw call. This reduces the number of CPU-GPU interactions, improving performance. By batching similar objects that share the same textures, shaders, or states, developers can minimize the draw call overhead, allowing for more efficient use of the GPU and smoother rendering in complex scenes.

In addition to draw call batching, texture atlasing is another technique that complements and enhances performance. Texture atlasing involves combining multiple smaller textures into a single, larger texture, often called a "texture atlas." By doing this, multiple objects that would normally require separate texture bindings can now reference different parts of the same atlas. This reduces the need to switch textures between draw calls, which is an expensive operation. When used alongside draw call batching, texture atlasing allows many objects to be rendered in a single batch without the need for frequent texture changes, further minimizing CPU-GPU communication and improving overall rendering efficiency. Together, these techniques reduce rendering overhead and improve frame rates in scenes with numerous small, textured objects.

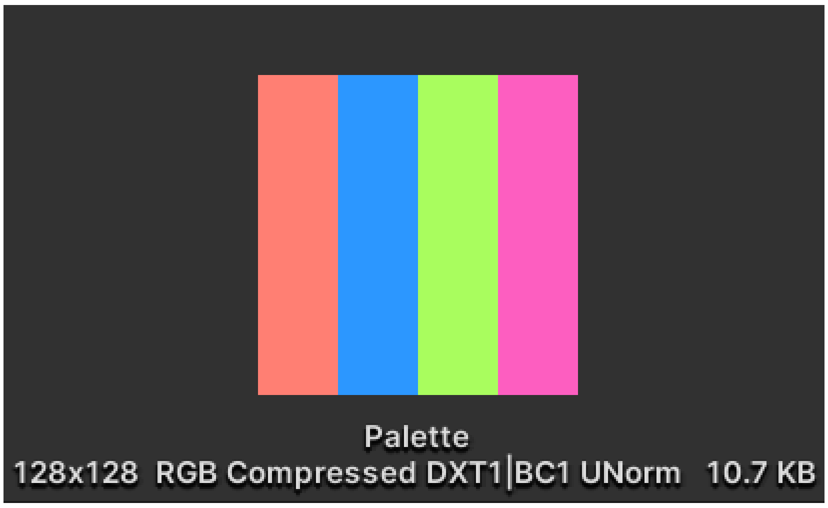


Figure 3: the simple texture atlas used to join several monochrome materials

Draw call batching is a built-in feature in Unity. To leverage it, one only has to make sure to share the same material across many meshes. Our app renders buildings in three different colors, according to the data regarding their usage. Inside each quadrant, at first buildings were grouped according to their color, to each corresponding a different material. This introduced complexity in the code and impeded draw call batching of all buildings together. Later a different approach was chosen, were a single, simple texture atlas (Figure 1) containing all colors was used to create a single material for all buildings. This way, we can choose the color of the building via its UV coordinates and allow Unity to freely batch all draw calls as it sees fit. Moreover, if more colors need to be added, it can be done at no additional expense.

Occlusion Culling

Occlusion culling is an optimization technique used in 3D rendering to improve performance by not rendering objects that are fully obscured (occluded) by other objects in the scene. The basic idea is that if an object is not visible from the current camera's point of view because it is hidden behind other geometry, there is no need for the GPU to waste resources rendering it.

The process of occlusion culling typically involves checking which objects in the scene are visible and which are blocked from view. Various algorithms and hardware-based methods can be used to determine visibility. For example, bounding volume hierarchies (BVH) or potentially visible sets (PVS) can be used to pre-determine which objects might be visible or occluded. Additionally, modern GPUs often perform occlusion culling in hardware using specialized tests that quickly assess whether parts of the scene are fully hidden by others.

By reducing the number of objects that need to be drawn, occlusion culling helps improve rendering performance, especially in complex scenes with a large number of objects where many are not visible. This allows the GPU to focus its resources on rendering only what the user can see, resulting in better frame rates and more efficient use of system resources.

Occlusion culling was not found to be very useful in our application, as the scene features only big open space with long distance visibility.

Frustum culling

Frustum culling is a technique used in 3D rendering to optimize performance by excluding objects that are outside the camera's view from being rendered. The camera's field of view in a 3D scene is represented as a view frustum, which is a pyramid-shaped volume that defines what the camera can see. Any object outside this volume is not visible to the camera and therefore doesn't need to be processed or rendered.

Frustum culling works by testing whether an object or its bounding box (a simple geometric shape that encompasses the object) intersects with the frustum. If the object is entirely outside the frustum, the rendering engine skips it, saving GPU and CPU resources. Objects partially inside the frustum are still rendered, but if large parts of a scene fall outside the camera's view, frustum culling can significantly reduce the number of objects that need to be processed.

This technique is particularly important for improving performance in large scenes with many objects such as ours, as it ensures the GPU focuses only on rendering objects that are visible to the camera. Frustum culling is usually done on the CPU before the draw calls are sent to the GPU, making it a fundamental step in efficient scene management. By excluding off-screen objects from the rendering pipeline, frustum culling helps maintain smooth frame rates and better resource utilization in 3D applications.

The effectiveness of this feature is reduced in our application by the panoramic camera rendering system, that covers 270° degrees around the player. However, it can be estimated that the rendering cost would increase by around 30% if this feature was turned off, because the rendering angle would go from 270° to 360°.

Levels of detail

Levels of Detail (LOD) is a technique in 3D rendering designed to optimize performance by adjusting the geometric complexity of objects based on their distance from the camera.

Objects that are far from the viewer typically require less detail, as fine features are less perceptible at greater distances. By employing LOD, rendering systems can dynamically substitute different versions of an object's model, with each version containing a varying number of polygons or details.

In practice, for each object, several representations are precomputed, ranging from highly detailed models for close-up views to simplified models for distant perspectives. During the rendering process, the appropriate level of detail is selected according to the object's proximity to the camera. This ensures that high-resolution models are used only when necessary, while less detailed models suffice for objects further away. This approach conserves computational resources, particularly reducing the load on the GPU, without significantly compromising visual fidelity.

LOD is particularly beneficial in large-scale environments, such as open-world games or virtual simulations, where numerous objects are present in the scene, but only a few are in close proximity to the camera. By employing less detailed models for distant objects, the system minimizes rendering overhead while maintaining a visually coherent scene. This technique can also be applied to textures, where lower-resolution versions are used for objects at a distance, further enhancing performance efficiency.

Whether to implement levels of detail for the buildings was evaluated, but good performances were reached earlier with simple optimizations, rendering it unnecessary. Conversely, the terrain was created using Unity's Terrain System, which features LODs that dynamically adjust the complexity of terrain geometry and textures based on the camera's distance from different sections of the terrain. This approach helps optimize performance in large outdoor environments such as ours by reducing the rendering workload while maintaining visual quality where it's most needed—close to the camera.

Unity's terrain is divided into smaller sections called terrain chunks or patches, which are rendered at different levels of detail depending on their distance from the camera. Chunks that are closer to the camera are rendered with more vertices, providing high geometric detail. As the camera moves further from certain chunks, Unity reduces the vertex count by simplifying the mesh. This ensures that distant terrain uses less GPU and CPU resources, while nearby terrain remains detailed. In our application, the terrain gathers heightmap and color info from georeferenced satellite images and aerial LIDAR scans.

The system uses continuous LOD methods, meaning that the transitions between different LOD levels are smooth and gradual to avoid noticeable "popping" as the detail changes. The terrain's heightmap resolution, which defines the elevation data, is also simplified in the distant patches.

In addition to simplifying the geometry, Unity's terrain system employs mipmaps for texture LOD. Mipmaps are precomputed, smaller versions of the terrain textures that are used for objects or surfaces that are further from the camera. When rendering distant terrain, lower-resolution textures from the mipmap chain are used, reducing the memory and processing requirements while still providing adequate visual quality at a distance.

Game Object number optimization

Performance of the app also depends on the number of Game Objects in the Unity scene. Each Game Object is an object designed to be moved independently than the other, therefore has its own Transform component to determine its position and rotation during the game. Each Game Object belongs to a hierarchy of transforms, where each object has its reference system determined by his parent. So to render each object correctly each child object's position, rotation and scale must be transformed to its parent's space, then its parent's parent, and so on until the world space.

Therefore, having a big number of Game Objects in our scene will increase the memory footprint due to the data each object stores, and the computational cost due to all the matrix multiplications that need to be performed. Both the memory footprint and the matrix multiplications are highly optimized, making it possible to run scenes with hundreds of objects on high-end machines. However, our application needs to render an amount of objects that is 3 order of magnitude greater than that (300.000), making it impossible to instance each building with a Game Object of his own.

Once tests showed that such a number of Game Object would degrade performance, Unity's Data Oriented Technology Stack was evaluated for implementing better performing rendering of such a high number of entities. The Entity Component System (ECS) in Unity is a core part of the Data-Oriented Technology Stack (DOTS) designed to enhance performance by separating data and behaviour. However, DOTS doesn't grant full VR compatibility, that is a desired future feature of this project.

The transform system is designed to move many 3D objects independently from each other, but the buildings of our application never move relative to one another, nor the terrain. This means that the scene can be optimized by joining multiple buildings in a single Game Object. We would therefore would aim for the biggest mesh possible, however a single mesh always in view would make any form of culling useless for obvious reasons, and distance-based rendering relies on separate Game Objects. Distance-based rendering is still necessary for the application to be supported by machines with lower performances.

To optimize rendering of buildings their individual meshes should be combined into larger ones, associated to fewer Game Objects. However sizing them appropriately is essential to reap the most benefit. These larger meshes must be big enough to keep the number of Game Objects relatively low, but small enough to maintain the effect of the culling, and to be loaded fast from disk. Here the chunking explained in the backend chapter comes into play. The frontend combines the meshes from each chuck (quadrant) into a small number of bigger meshes and renders it on screen. This gives us big but manageable, uniformly sized meshes.

Because quadrants are defined spatially, they each hold a different amount of meshes of varying sizes. Unity by default uses 16-bit integers to store the indexes of the vertices array of each mesh. This limits the vertex count of each mesh to 65.536, that is exceeded in some quadrants. The issue could be solved by raising the limit to 4.294.967.296 vertices bit using 32-bit indexes for all meshes, or splitting the bigger quadrants up in more meshes. Since very few quadrants would exceed the 16-bit limit and raising it would increase the memory footprint of all meshes, the former solution was chosen.

The result was a system of rendering based on the chunking done in the backend, where individual building meshes are combined right after loading and joined into few Game Objects that are culled when not in view. This system can easily handle a database more than 300.000 static buildings in real-time on a high-end machine. However, because of the big number of Game Objects, interaction with the buildings has to be managed in a separate system.

Interaction

The default interaction mode of the application is navigation mode, that consists in being able to hover over the city using the arrow keys and looking around with the mouse. From here the user can transition into data mode, where the cursor appears on screen and the data regarding the building on which it is hovering appears on screen. Like for the rendering, a naïve implementation of this system could not reach satisfying performance, because, as discussed before, too many Game Objects in the scene would use up too much resources. This is even more true for a big amount of colliders, the unity component necessary to support the desired hover interaction.

Colliders

Colliders are components attached to Game Objects that define the physical shape for an object in the physics engine, making them essential for interactions like detecting collisions or raycasts. When implementing hover interactions with 3D objects, colliders play a crucial role in detecting the player's input and determining when the object is being "hovered" over.

When a collider is attached to an object, it acts as a boundary that can register when other objects, such as a mouse pointer (via a raycast), come into contact with it. In the case of hover interactions, a common approach is to use a raycast that is projected from the user's camera towards the 3D scene. The raycast detects if it hits any colliders, and if it does, Unity's physics engine calculates which object was hit based on the collider's shape. Once the collider is "hit" by the ray, hover-specific behaviors, such as highlighting, displaying UI elements, or triggering animations, can be executed.

Colliders come in various shapes (e.g., BoxCollider, SphereCollider, MeshCollider), allowing developers to approximate the 3D object's geometry for precise or efficient detection. For hover interactions, the type of collider used depends on the object's shape and performance considerations.

We need to grant to the user the ability to pick a potentially small building among many others, so we need our colliders to precisely match the buildings shapes. This can't be done with primitive colliders like a box or cylinder collider, as buildings vary greatly in shape and size; moreover trying to match meshes with primitive colliders would require more computation and management of many edge cases, as it is impossible to do manually.

So mesh colliders are needed, which are the more expensive but also the most accurate choice of colliders. As anticipated, these can not be used in the amounts that our application requires, so a technique to reduce their numbers is needed. The colliders with which can user can have a meaningful interaction are only the closest, because the furthest in the horizon can't be distinguished properly. Consequently, a system of distance-based instancing was developed also for the colliders.

This means that when entering data mode, only the closest buildings will become interactive until transitioning back to navigation mode.

Pooling of interactable buildings

Instancing the interactable buildings, albeit being a cheap operation, threatens the performance of the application if repeated, because our scripts run on C#, which has a managed memory system. This means that the programmer doesn't need to worry about deallocating memory, however, dereferencing often big chunks of memory will slow down the program.

For each data structure allocated on the heap, .NET tracks how many variables are referencing it. Based on the settings of the garbage collector, every time a certain condition is met (such a threshold memory usage is reached, or a periodic garbage collection is scheduled) the collector claims back all memory from unreferenced data structures, appropriately called garbage.

To correctly handle the scenario in which data structures with a short lifecycle are created often, object pooling needs to be implemented. In Unity, object pooling is a design pattern used to optimize the instantiation and destruction of Game Objects. Instead of creating and destroying objects continuously, object pooling maintains a pool of inactive Game Objects that can be reused, reducing overhead and improving performance. We follow with a breakdown of usual phases of object pooling:

1. Pre-instantiation of Game Objects: At the start of the game or when the pool is initialized, a predefined number of Game Objects (typically of the same type, like projectiles, enemies, or power-ups) are created and stored in a pool (usually a list or queue). These objects are initially inactive, meaning they are not visible or interacting in the scene.
2. Reusing Objects from the Pool: When an object is needed (e.g., a bullet is fired or an enemy spawns), rather than instantiating a new Game Object, the system checks the pool for an available inactive object. If one exists, it is activated, repositioned, and reused in the game. This avoids the performance cost of creating a new object.
3. Returning Objects to the Pool: After the object has served its purpose (e.g., the bullet hits something or an enemy is defeated), instead of destroying it, the object is simply deactivated and returned to the pool. It remains in memory but is no longer active in the scene, ready to be reused when needed again.
4. Pool Expansion (Optional): If the pool runs out of available objects and more are needed, the pool can be expanded by instantiating additional Game Objects, but this is usually avoided in favour of keeping the pool at a predefined size for efficiency.

In our case, the objects managed with this pattern are all buildings in a chosen radius around the player. Their number is not known at the beginning, so we implemented a more dynamic object pool for interactable buildings, where we don't do a pre-instantiation, our pool starts with zero objects, and we expand it each time a new object is needed. No longer needed objects are not destroyed but just returned to the pool until new objects are needed. This way the pool size doesn't grow beyond the need of the application, and no garbage is created.

Panoramic rendering

As anticipated in previous chapters, the decision theatre possesses a 220° panoramic screen. The goal of the panoramic rendering system is to render the scene for an optimal view when standing or sitting at the center of the decision theatre. To this end, measurements of the environment were taken to calibrate the rendering accordingly, then a system that simulates a panoramic camera was devised.

Theoretically, one could create ex novo a system that renders the images directly with a cylindrical projection. Rather than using a virtual camera with a flat sensor and a pyramidal frustum, a cylinder sensor and a cylindrical frustum could be used for rendering, but it would prove very difficult, as the computer graphics legacy has always worked mainly with flat screens. Such a system would require such heavy customization of the rendering system that is unnecessary and is beyond the scope of this project.

Another more viable solution would be to render the cylindrical image with many virtual cameras arrayed radially, each rendering a thin vertical slice of the cylinder, that is closer to a perfect result the more cameras are employed. This doesn't avoid distortion altogether but would make us face a trade-off between performance and quality, where more cameras yield better pictures at the cost of a higher overhead.

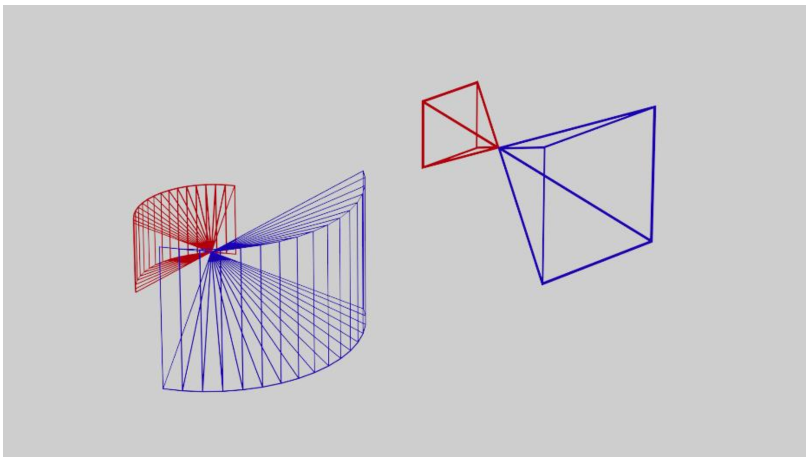


Figure 4: Arrayed cameras vs. single camera

We found out that our problem was only a particular case of the projection mapping used in non-head mounted virtual reality, such as CAVE[7] systems. Cave systems and projection mapping in general don't override regular render pipelines, but rather use them to render cube maps that are then projected to the shape of the walls.

The system in our decision theatre is composed of an array of three image cameras that span 270° around the player position without visible seams.

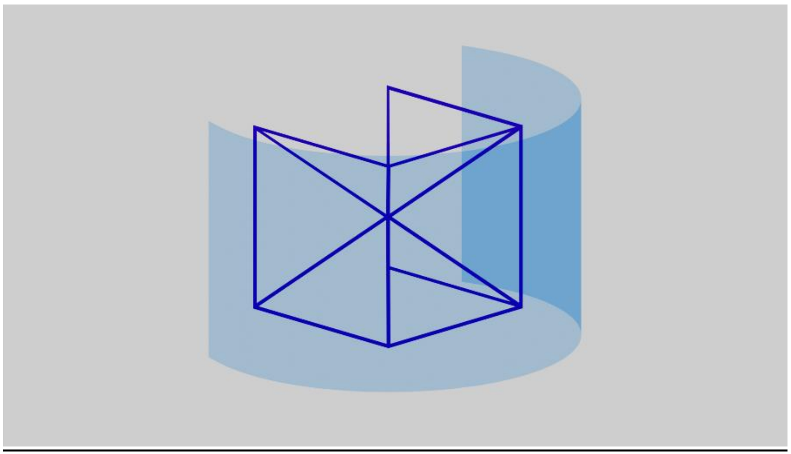


Figure 5: a 3-camera array surrounded by a reference cylinder

The images captured by these cameras are stitched together onto a render texture.

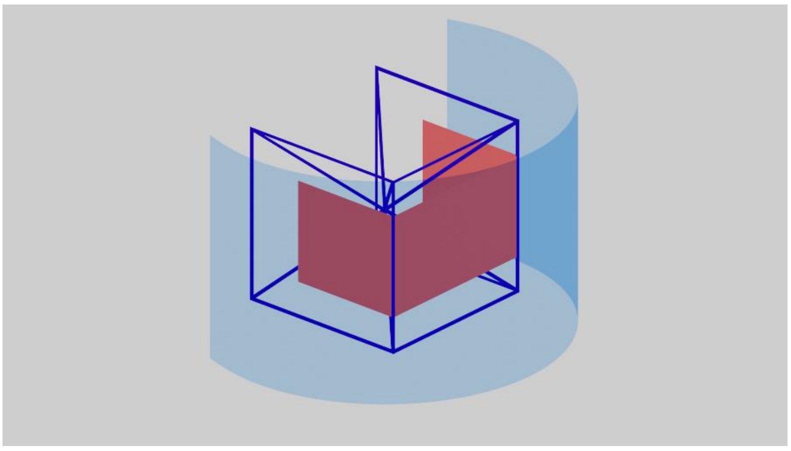


Figure 6: Render texture, original

The resulting render texture is then projected by a shader program to the cylindrical shape of the panoramic screen.

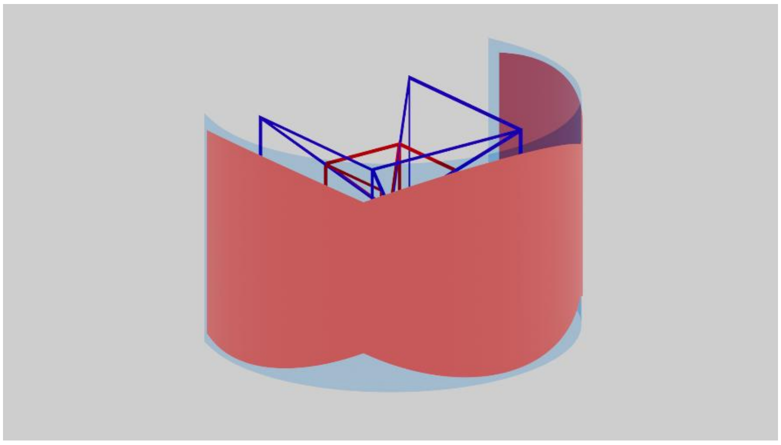


Figure 7: Render texture, projected

Finally, only a rectangular slice of the final result is rendered.

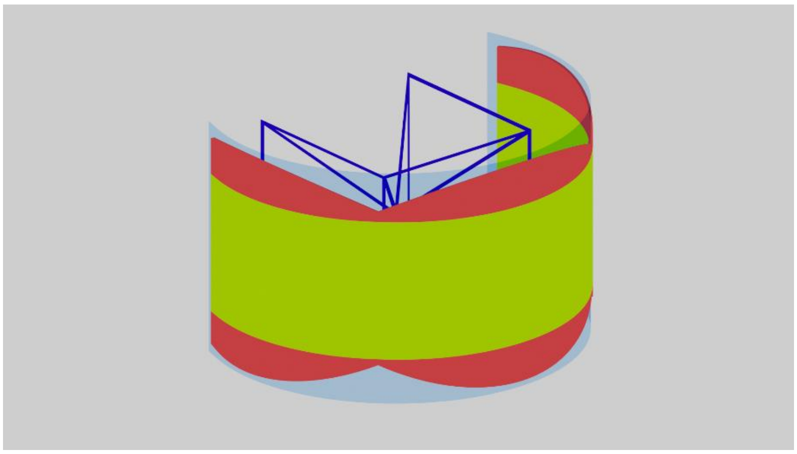


Figure 8: render texture, cut

Backend

Overview

The role of the backend is to feed the front-end data that is easily loaded and rendered. This data is derived from 2 Datasets:

- Edifici 3D 2017 from Arpa Piemonte [1]: a comprehensive list of the buildings in Piedmont;
- RIPRESA AEREA ICE 2009-2011 - DTM 5[8] : Digital terrain model of Piedmont, made of large georeferenced raster images containing the value of the terrain height;

The fundamental dataset is Edifici 3D 2017, that provides information about each building's geometry (that is needed to represent it in 3D), and general information about the building that will be shown in the UI.

The beginning JSON dataset is not fit to be rendered directly, as the geometry data is comprised of just the height of the building and a list of geodetic coordinates points representing the footprint of the building on the terrain as a polygon. The dataset does not contain info about the height of the terrain that the building stands on, that we need to provide a better representation of the landscape. Towards this purpose, a terrain heightmap has been used to complement the geometry present in the dataset. Once the mesh is obtained, a building can be rendered or made interactable as described in the front-end chapter.

The backend pre-processes the dataset and the heightmap to derive necessary data for rendering, then splits it into chunks and serializes it with a high-performance binary serialization library to facilitate real-time loading. The chunks consist of even quadrants aligned with the latitude and longitude dimensions. The so obtained serialized quadrants at the end are then loaded and rendered dynamically at runtime.

The Edifici 3D 2017 dataset

The Edifici 3D 2017 is the main dataset used by the application, containing detailed information about every building that will be rendered. Here is a list of the information we have about each building:

- Building footprint on the terrain represented as a polygon made of latitude-longitude points;
- Elevation: the height of the building in meters;
- A Unique Id;
- Area of the footprint;
- Building height derivation procedure;
- Usage: whether the building is considered residential, dedicated to production or to services.

Heightmap

RIPRESA AEREA ICE 2009-2011 - DTM 5 [8] is a digital terrain model derived from a LIDAR scan of the terrain and covers the whole region of Piedmont.

A Digital Terrain Model (DTM) is a 3D representation of the Earth's surface, capturing the precise elevation data for a given area. Used extensively in fields like cartography, geographic information systems (GIS), and civil engineering, DTMs represent the "bare earth" by excluding surface objects like buildings and vegetation. This data is typically derived from technologies such as LiDAR, photogrammetry, or satellite imagery, offering accurate elevation and slope information essential for applications in flood modelling, urban planning, and infrastructure development. DTMs are crucial for analysing terrain characteristics, aiding in more informed decision-making and resource management.

It is a raster image collection with a grid resolution of 5 meters, and an error of 0.6 meters at worse. It's used to shape the digital terrain of our application and to place buildings at the right height when generating their meshes.

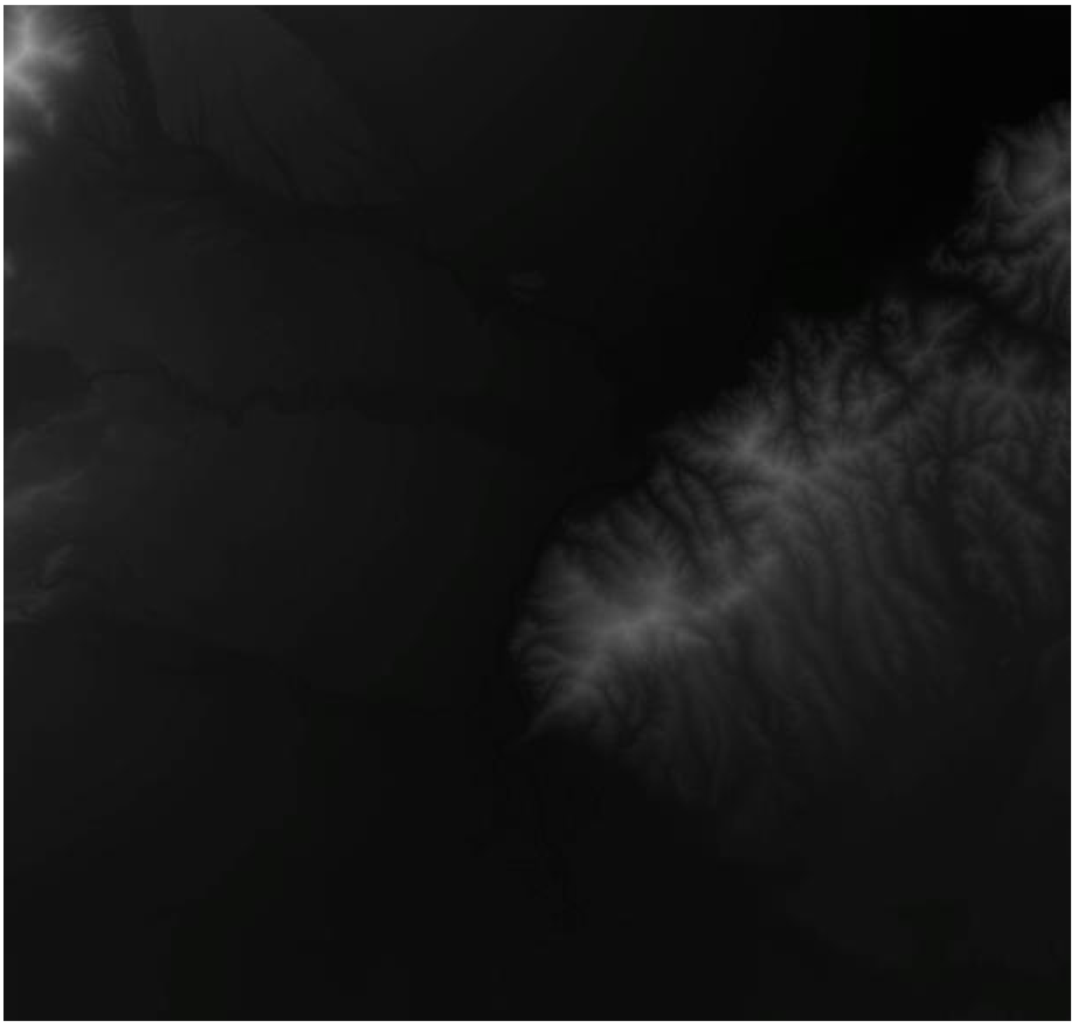


Figure 9: Heightmap of the area of interest

Pre-processing Pipeline

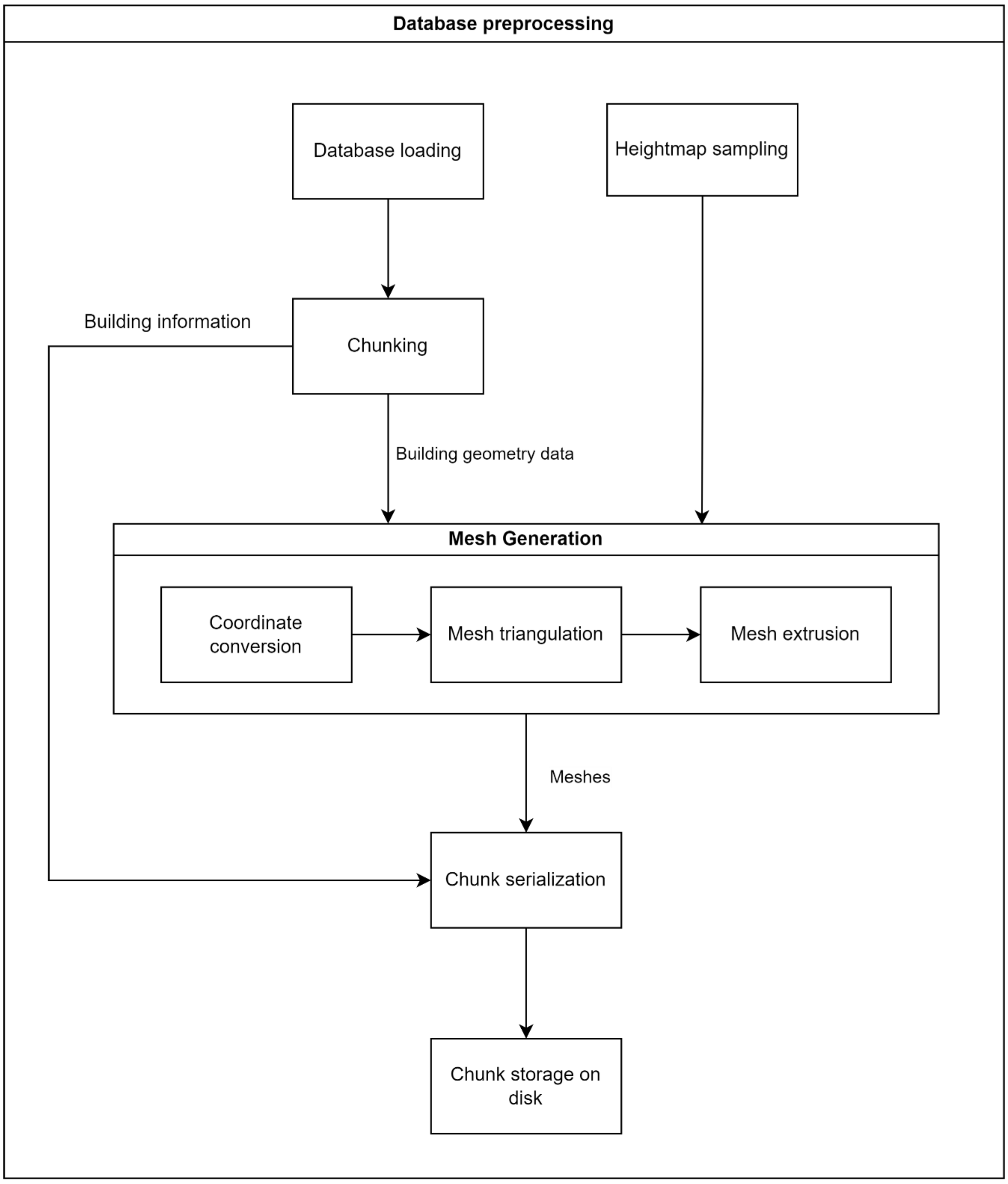


Figure 10: database preprocessing diagram

Due to the intensive operations necessary to make the dataset fit to be rendered, the application pre-processes and crosses the datasets. A number of naive attempts have been made to execute these operations at runtime, but no matter how much that design would be optimized, one would still be performing a big amount of calculation to obtain always the same result. Given the relative staticity of the reference dataset, there's no reason to perform all calculations all over at every startup of the application. Therefore a pre-processing approach was preferred.

The pre-processing pipeline chunks the database to allow for fast pre-processing, constructs the meshes used in the data visualization, and finally serializes the resulting pre-processed chunks in a format that can be loaded fast and asynchronously.

In case of a change in the dataset, redoing the preprocessing only takes 4 minutes of computation, and it could take less if a more parallelized algorithm would be written. During development the pre-processing was run several times to find a good chunk size, an the application has been using the same pre-processed dataset since.

Chunking

The “chunking” technique involves dividing large datasets into smaller, more manageable units, or "chunks." This approach enhances the efficiency of data processing, improves accessibility, and optimizes both storage and network resources. In chunking, data is split into fixed-size or variable-size pieces, which are processed, stored, or transmitted independently. Each chunk typically includes a small identifier or metadata that marks its position within the dataset, allowing for accurate reassembly when needed.

Chunking contributes to improved performance and efficiency, as processing smaller chunks of data generally requires less memory and computational resources than handling a large dataset in its entirety. Applications can thus load data in pieces, reducing the demand on memory and CPU resources. Additionally, chunking enables parallel processing, where chunks are handled simultaneously across multiple threads or servers. This capacity for distributed handling is especially beneficial where processing large volumes of information in parallel significantly reduces computation times.

In our application, chunking optimizes data retrieval by allowing access to only the necessary chunks rather than loading the whole dataset. Consequently, it minimizes both retrieval time and system load, contributing to a more efficient runtime. The chosen chunking scheme is that of grid-based spatial indexing, where chunks span tiles of constant latitude and longitude. For this reason, in the following paragraphs, the term "chunk" and "quadrant" will be used interchangeably.

In future developments, chunking could facilitate incremental data synchronization, where only the chunks that have changed need to be updated, avoiding the transfer or duplication of the entire dataset.

Determining an optimal chunk size is essential, as it affects the balance between flexibility, efficiency, and potential overhead. Smaller chunks offer more adaptability but can increase management overhead, while larger chunks might reduce this overhead yet make data retrieval less efficient.

In our specific case, each chunk will be turned into a small set of Game Objects that each come with some overhead (more details in the front-end chapter). As the chunk size increases, less Game Objects will have to be managed inside our application making it faster, but the loading time of each chunk will increase. On top of that, if the meshes inside the quadrant have too many points, the chunk will be split into more Game Objects, increasing the overhead. Loading times were found to be always very fast, so optimization of the rendering performance was prioritized when choosing the chunk size. The current chunk size was chosen so that the great majority of chunks is under the unity mesh threshold size, with a few exceeding it. This way, chunks minimize the number of Game Objects while preserving the advantages of occlusion culling.

Once the whole dataset is loaded, it is split into chunks. Then, the unchunked version is unloaded, and mesh generation begins.

Mesh generation

In computer graphics, a mesh is a collection of vertices, edges, and faces that defines the shape of a 3D object. Typically, these components form polygonal structures, with triangles and quadrilaterals being the most common face shapes, though other polygons can also be used. The vertices represent specific points in 3D space, edges connect pairs of vertices, and faces are closed sets of edges that collectively create the surface of the object. By combining numerous polygons in a systematic arrangement, a mesh forms a continuous surface that can represent complex shapes and contours.

Meshes are essential in the modelling and rendering processes, as they serve as the primary framework for defining the visual form of objects in a 3D scene. Each component of the mesh—vertices, edges, and faces—holds data that defines the spatial structure and surface characteristics of the object. Additionally, meshes often include supplementary data, such as normals for lighting calculations, texture coordinates for applying surface details, and weights for animation and deformation.

To turn the geometrical information contained in the dataset into meshes that can be rendered in unity, the following steps are taken:

- Converting geodetic coordinates into unity coordinates;
- Triangulating the footprint of each building;
- Extruding the footprint into the final rendered prism;
- Shifting the prism at the right height.

Geodetic coordinates mapping

Geodetic coordinates map points on the Earth's surface to latitude and longitude values. Accurately mapping these in Unity would require addressing Earth's curvature, which would also curve the terrain under the buildings. However, Unity's terrain system does not natively support geodetic data or real-world scale; therefore, we used a planar approximation to convert geodetic coordinates to Unity's world space. To prevent the scene from becoming too large, a scale factor of 1:10 was applied, keeping the dataset manageable. Since the dataset covers less than half a degree in latitude and longitude, a first-degree planar approximation was used, disregarding Earth's curvature. This approach introduces a slight distortion, which remains unnoticeable at the scale of interest.

Polygon triangulation

Unity supports only triangular meshes at runtime due to the inherent advantages of triangles in computer graphics rendering and processing. In our case we need to triangulate a simple planar polygon with no holes which involves dividing it into a set of non-overlapping triangles, where the vertices of each triangle are vertices of the polygon. Here we present the method we used, called the ear clipping algorithm, chosen for its simplicity.

1. **Identify and Initialize Vertices:** We begin with a list of the polygon's vertices ordered clockwise or counterclockwise from our database. Let $V = \{v_1, v_2, \cdots, v_n\}$ represent the sequence of vertices that define the polygon's boundary.

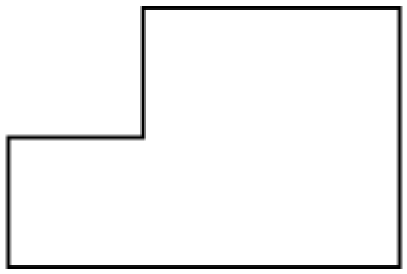


Figure 11: a generic concave polygon

2. **Check Polygon Convexity:** A polygon is convex if every interior angle is less than 180 degrees. If the polygon is convex, it can be easily triangulated by drawing diagonals from a single vertex to all non-adjacent vertices. However, for a non-convex polygon, we'll need to proceed with ear clipping.

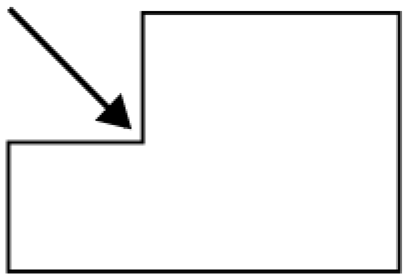


Figure 12: detected concavity

3. **Find "Ears" in the Polygon:** An ear of the polygon is a triangle formed by three consecutive vertices (v_i, v_{i+1}, v_{i+2}) where:

- The interior angle at v_{i+1} is less than 180 degrees.
- The triangle (v_i, v_{i+1}, v_{i+2}) contains no other vertices of the polygon within it.

Identifying ears is crucial because each ear can be safely removed without affecting the polygon's overall shape.

4. **Clip the Ear:** Once an ear is identified, remove the vertex v_1 (the middle vertex of the ear) from the list of vertices. The triangle formed by (v_i, v_{i+1}, v_{i+2}) is stored as part of the triangulation. This step effectively reduces the polygon by one vertex.

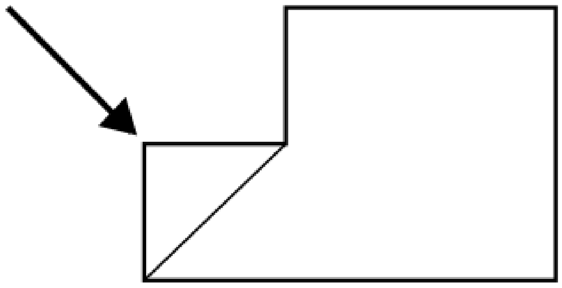


Figure 13: first ear clipped

5. **Repeat Until the Polygon is Reduced:** After clipping an ear, repeat the process of finding and clipping ears in the remaining polygon. Continue this loop until only three vertices remain, which form the last triangle of the triangulation.

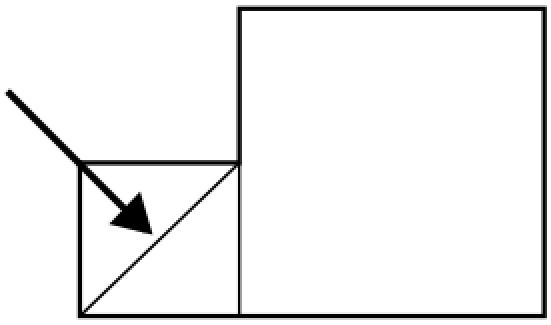


Figure 14: successive clipping (1)

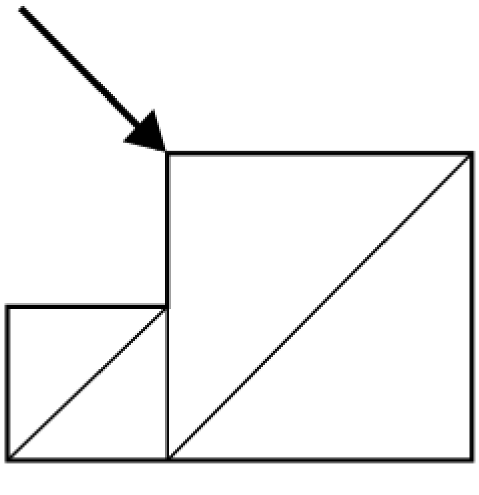


Figure 15: successive clipping (2), triangulation complete

6. **Store the Triangles:** The triangles identified during each ear clipping step, along with the final triangle, collectively represent the triangulation of the polygon.

Extrusion and shifting

Once a triangulated mesh footprint is obtained, to turn it into a prism we just need to duplicate it, shift it by the height of the building, and connect the duplicate edges with 2 triangles for each couple. The result is then shifted upward by the terrain elevation taken from the DTM dataset. The terrain elevation is sampled at the location of the first point of the list describing the polygon. This approximation was preferred over sampling the elevation at the centroid of the polygon, because such precision would be unnecessary, and not worth the additional computation.

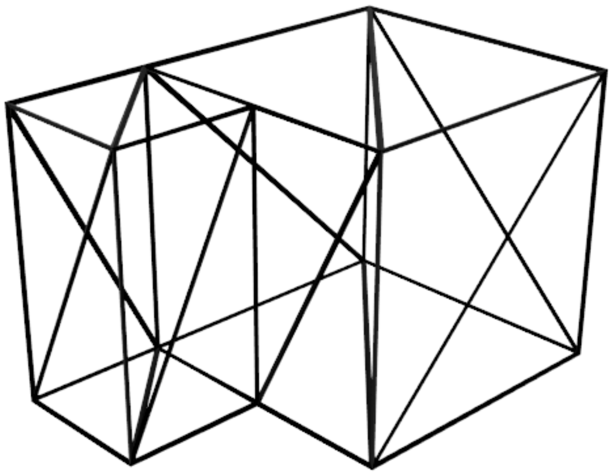
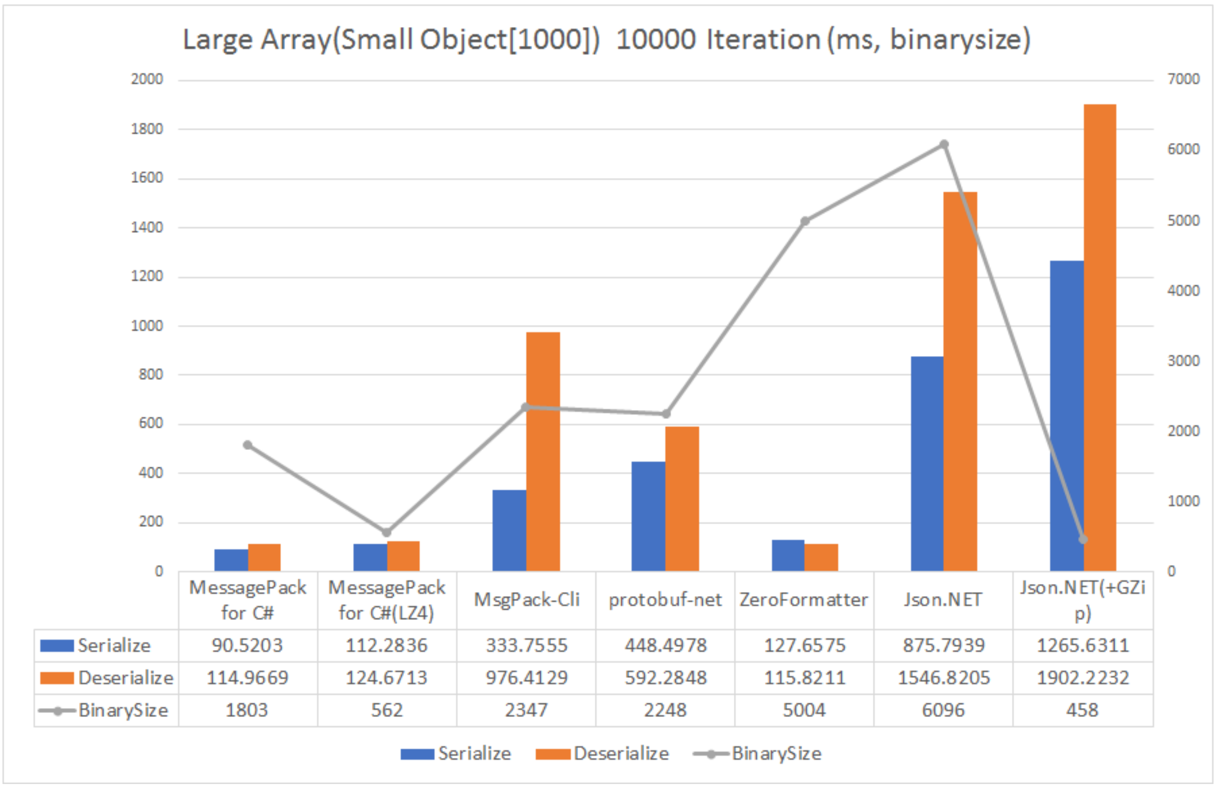


Figure 16: extruded polygon

Serialization

Once all meshes are computed, and buildings are chunked in quadrants, these quadrants need to be serialized for use by the frontend. A high performance serialization and deserialization library was needed to allow fast pre-processing and an acceptable loading time on the frontend.

At the beginning Dotnet's BinaryFormatter class was tried. Besides its obsolescence, it was not nearly fast enough to offer a good experience in the front end.



	MessagePack for C#	MessagePack for C#(LZ4)	MsgPack-Cli	protobuf-net	ZeroFormatter	Json.NET	Json.NET(+GZip)
Serialize	90.5203	112.2836	333.7555	448.4978	127.6575	875.7939	1265.6311
Deserialize	114.9669	124.6713	976.4129	592.2848	115.8211	1546.8205	1902.2232
BinarySize	1803	562	2347	2248	5004	6096	458

Figure 17: performance comparison of several serialization libraries, see [9]

Therefore MessagePack for C#[9] was used, allowing for surprisingly good results. However, it is not able to natively serialize unity meshes. Decomposing meshes into their triangles and vertices arrays was a sufficient workaround, as they are easily reconstructed when reloaded.

Notes on pre-processing performance

The first implementation of the above processing would freeze the machine due to its memory usage. From loading to serialization, all the data structures inside managed memory of which a reference was kept and unity engine data structures would not be disposed. This led to the application of the following memory management best practices:

- Deleting references to data that won't be used again;
- Explicitly ask the Unity engine to destroy data structures that become useless
- Process all data in batches manageable by the system resources.

This reduced dramatically the application's memory footprint, from 30Gb in RAM to less than 3Gb, allowing it to pre-process the whole database in 4 minutes and not saturate the main memory.

Serialized result

Once all buildings are processed and made serializable, we insert each building in a list representing a quadrant. This list is serialized and saved as a binary file, ready to be loaded in the application.

Besides a file for each quadrant, the backend also saves a metadata file that indicates how the database was split, from which the frontend can infer how many files were produced and with which names they can be retrieved.

Asynchronous loading

We discussed how the application pre-processes the dataset before runtime, now we move on to explain how the loading at runtime is performed.

The application stores each quadrant into a serialized file in a folder, and when the app is started up, it starts loading them. When the user moves into the map and other quadrants enter a certain radius, the new closer quadrants are loaded. If all of this was to be done synchronously, the application would stall for a long time at the beginning and would exhibit a lag spike for each quadrant entering the loading radius. Fundamental to designing such systems in Unity is knowing Unity's update cycles, and what it is affected by.

Unity has several update cycles, and they all reside on the main thread. The most important one is simply called "Update" and is called once at the beginning of each frame. For maintaining a desired framerate, synchronous operations in the Update cycle must last less than the time between a frame and the next, minus the rendering time. Moreover, Unity's APIs are not thread safe, so operations based on those will always be executed synchronously. This means that we are able to parallelize some operations (loading) but not others (creating and combining meshes).

Several approaches have been attempted to mitigate the problem:

1. loading synchronously a few quadrants from the loading radius per frame;
2. loading asynchronously all quadrants at once;
3. having a single background thread loading all quadrants;
4. having a fixed number of tasks at all times dedicated to loading the quadrants;
5. Batching asynchronous loading operations.

Approach #1 stalls the update cycle for too long, causing massive lag. Even loading synchronously a single quadrant each frame would stall the application noticeably. Approach #2 performs better. The application is slow when it starts, because despite moving most of the work to other threads it still block the main thread with all the operations that are forced to be synchronous, but then performance improves. Approach #3 never stalls the application because it moves most of the work out of the main thread, but the loading takes much longer due to the limited resources dedicated to it. Approach #4 is satisfactory: the application has slight lag at the beginning but loading of all buildings happens in a few seconds. However, the slight lag is very noticeable because it causes an irregular framerate. This is due to the asynchronous tasks ending at random moments during runtime, distributing the load on the main thread unevenly.

In-depth analysis revealed that even if the concurrent threads are in a limited number, a significant amount of time is spent continuously starting new tasks and receiving results on the main thread. This causes additional overhead and an irregular framerate. Batching the asynchronous tasks (approach #5), waiting for each batch to complete execution before the next one yielded the best results. During the initial loading the application is slower for a few seconds, then the initial loading completes and the desired framerate is reached. This is the chosen approach and the one used in the final version of the application.

Conclusions

The objective of this thesis was to demonstrate that extended reality technology is mature to develop avant-garde decision support systems. We consider ourselves successful in tackling the main problems: integration of locally available data, performance, and ease of use. This chapter is dedicated to exploring future desirable features and flashing out what a complete decision support system featuring extended reality could look like.

Future developments

Virtual reality support

The current prototype was designed for the decision theatre and supports panoramic rendering. However, it could be easily converted to a virtual reality application with several resulting benefits. Using the app through a virtual reality headset would yield a greater feeling of presence and immersion and would be even better suited for multi-user experiences which we will describe later.

Online features

Decision support systems thrive on data. The current application lacks the rich data landscape offered by the energy center researchers, but it could easily be connected to become a full-fledged decision support system. As a digital twin of the city, it could be developed further to support advanced and customizable data visualizations and interactions.

Navigation

Navigation in the current prototype is completely manual, with no shortcuts or aid for searching interest points. The current dataset doesn't hold any street information, but a search feature allowing to jump directly to a chosen location is highly desired. A more comprehensive dataset could be formed by using Open Street Map [10], the most used open source data source for maps.

Visual improvements

The current prototype features single colors for buildings and an Open Street Map static texture for the terrain. Improvements to this aspect may involve allowing more colors to be selected for buildings based on data filtering and categorization, an making the map dynamic, increasing resolution for closer terrain and showing more details based on distance. Mapnik[11] in an OSM render which could be up to the task, but a system to implement this dynamically textured terrain has to be developed in unity. Moreover, real-time weather forecasts and APIs, could be used to alter the atmosphere of the environment and improve immersion.

Multi-user experiences

Thanks to VR compatibility, users could communicate and interact with data by annotating it, creating custom filtered views. For instance, a researcher could show the results of his analysis to stakeholders directly in the immersive environment, taking turns in navigating the data landscape. This way of interacting is expected to raise stakeholder's awareness and interest of the issues at hand, as the decision support system would become a virtual decision making space favouring a bird's eye view of the issues at hand.

Presentation mode

Filtered view of data could be stored, retrieved and view sequentially, for researchers to give overviews and presentations of data. Key points in the map could be saved and be integrated in a presentation, turning the application in a communication tool as well as a data analysis tool.

Conclusions

In the light of the work that was made, we judge extended reality technology mature to tackle the challenges of data visualization, user interaction, rendering and performance encountered during development of Decision Support Systems. Further exploration and research is possible and promises good results.

Bibliography

[1] ARPA Piemonte, Edifici 3D 2017, at https://www.geoportale.piemonte.it/geonetwork/srv/api/records/arlpa_to:EDIF_2017-12-21-11:00#gn-tab-datasetAndSeries (19/11/2024)

[2] Quintel Intelligence, Energy Transition Model, at https://energytransitionmodel.com/, (19/11/2024)

[3] D. Robison, R. A. Earnshaw and P. McClory, "Interactive and Augmented Information Spaces to Support Learning and Dynamic Decision-Making," 2009 International Conference on CyberWorlds, Bradford, UK, 2009, pp. 305-311, doi: 10.1109/CW.2009.11.

[4] P. Cassará, M. Di Summa, A. Gotta and M. Martelli, "E-Navigation: A Distributed Decision Support System With Extended Reality for Bridge and Ashore Seafarers," in IEEE Transactions on Intelligent Transportation Systems, vol. 24, no. 11, pp. 13384-13395, Nov. 2023, doi: 10.1109/TITS.2023.3311817.

[5] H. Arregui et al., "An Augmented Reality Framework for First Responders: the RESPOND-A project approach," 2022 Panhellenic Conference on Electronics & Telecommunications (PACET), Tripolis, Greece, 2022, pp. 1-6, doi: 10.1109/PACET56979.2022.9976376.

[6] J. Singh, G. Singh, R. Verma and C. Prabha, "Exploring the Evolving Landscape of Extended Reality (XR) Technology," 2023 3rd International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON), Bangalore, India, 2023, pp. 1-6, doi: 10.1109/SMARTGENCON60755.2023.10442251.

[7] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. 1992. The CAVE: audio visual experience automatic virtual environment. Commun. ACM 35, 6 (June 1992), 64–72. https://doi.org/10.1145/129888.129892

[8] Regione Piemonte - A1613B - Sistema informativo territoriale e ambientale, RIPRESA AEREA ICE 2009-2011 - DTM 5, at https://www.geoportale.piemonte.it/geonetwork/srv/api/records/r_piemon:224de2ac-023e-441c-9ae0-ea493b217a8e (19/11/2024)

[9] Yoshifumi Kawai, MessagePack for C#, https://github.com/MessagePack-CSharp/MessagePack-CSharp, (19/11/2024)

[10] Open Street Map Foundation, Open Street Map, at https://www.openstreetmap.org/ (27/11/2024)

[11] Artem Pavlenko, Mapnik, at https://mapnik.org/ (27/11/2024)