POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Development of a real-time solution for an interactive VR representation of large star catalogues

Supervisors

Prof. Andrea SANNA

Candidate

Paolo GALLO

April, 2021

Abstract

This thesis takes place in the context of Virtual Reality and Data Visualization techniques applied to large astronomical datasets. The goal of this work is to improve and extend the already existing *Astra Data Navigator* application, built with the Unity game engine, and make it capable of loading and displaying large star catalogues in a realistic and real-time 3D environment.

The software was built in the VR laboratory of ALTEC - Aerospace Logistics Technology Engineering Company - while interfacing with other European projects such as NEANIAS and ESA's Gaia mission, which is particularly relevant to this work. The catalogue of celestial objects observed by the *Gaia* astrometric satellite is the largest collection of stars available to date, counting over 1.8 billion entries; being able to navigate and interact with this data in 3D would be extremely useful for both scientific and educational purposes, but most VR tools are limited to a much smaller object count and cannot be extended further.

On the data management side, the application (which has an integrated star catalogue but also supports external data sources in the form of CSV files or SQL databases) offers the choice between two different modes: the user can choose to load all of the available data at startup and store it in the system memory, which requires more resources and increased loading time but then provides a seamless navigation of the 3D environment, or he can opt for a dynamic loading solution (better suited for large catalogues), that only selects relevant data based on the current observer position, saving a lot of resources but introducing additional loading times that interrupt the navigation experience.

Processing and rendering such large amounts of objects in real-time can be a difficult task, which was however solved by carefully managing the application's resources, choosing the right data structures, optimizing algorithms in order to take advantage of the multiple cores in modern processors and exploiting the massive parallelism provided by GPUs via programmable shaders. All the techniques employed to overcome these challenges are described in depth in this work, along with their implementation details and limitations, while also highlighting some of the features that are made possible by these systems.

The results of performance analyses and user testing of the final product are also presented in this study, showing how most of the project's goals have been successfully accomplished but also highlighting a few shortcomings of the implemented solution, which could be overcome by further development.

Table of Contents

List of Figures

List of Tables

Acronyms

2D bi-dimensional.

3D three-dimensional.

ADN Astra Data Navigator.

API Application Programming Interface.

AR Augmented Reality.

CAVE Cave Automatic Virtual Environment.

CPU Central Processing Unit.

CSV Comma-separated values.

DB Database.

DBMS Database Management System.

DEC Declination.

ECS Entity Component System.

ESA European Space Agency.

FPS Frames Per Second.

GB Gigabyte.

GC Garbage Collector.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

ID Identifier.

IPD Inter-pupillary distance.

IT Information Technology.

MB Megabyte.

RA Right Ascension.

RAM Random Access Memory.

SGT Space Graphics Toolkit.

SQL Structured Query Language.

UI User Interface.

UML Unified Modeling Language.

VR Virtual Reality.

XML Extensible Markup Language.

Glossary

asymptotic complexity The growth of the execution time of an algorithm as the input size (N) gets large.

billboard A 3D computer graphics sprite that is always facing the viewer.

database An organized collection of data, typically stored and accessed from a computer system.

effective temperature The temperature of a black body that would emit the same total amount of electromagnetic radiation, often used as an estimate of a body's surface temperature.

factory A creational pattern that deals with the problem of creating objects without having to specify the exact class of the object that will be created.

framerate The frequency (rate) at which consecutive images called frames appear on a display.

framework A universal, reusable software environment that provides particular functionality to facilitate the development of software applications.

garbage collection A form of automatic memory management, where the garbage collector attempts to reclaim memory occupied by objects that are no longer in use by the program.

headset A support framework with attached electronic devices that is worn on the head.

jitter High-frequency motion, which causes visual instability.

magnitude A measure of the brightness of an astronomical object as observed from Earth.

mesh A collection of vertices, edges and faces that defines the shape of a polyhedral object.

overhead Combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task.

parallax The apparent shift of position of any nearby star against the background of distant objects.

procedural generation A method of creating data algorithmically as opposed to manually, typically through a combination of human-generated assets and algorithms coupled with computer-generated randomness and processing power.

proper motion The astrometric measure of the observed changes in the apparent places of stars or other celestial objects in the sky, as seen from the center of mass of the Solar System, compared to the abstract background of the more distant stars.

query A command run on a database, to access or modify its content.

refresh rate The number of times per second that a raster-based display device displays a new image.

rendering The process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of a computer program.

shader A type of computer program, running on a GPU, used to calculate rendering effects in 3D scenes.

singleton A software design pattern that restricts the instantiation of a class to one single instance.

spacecraft A vehicle or machine designed to fly in outer space.

texture A bitmap image applied to a surface in computer graphics.

twinkling Variations in apparent brightness, colour, or position of a distant luminous object viewed through a medium.

virtual reality A simulated immersive experience that can be similar to or completely different from the real world.

Introduction

With large datasets comes the need for meaningful visual representations, that allow users to see beyond the rows of a database and grasp the actual information contained in such enormous volumes of data. This idea has been the starting point for all Data Visualization techniques, which have become more and more relevant since the advent of Big Data made it almost impossible for humans to extract knowledge from large datasets using "traditional" techniques. Astronomy and astrophysics, like all experimental sciences, are no strangers to this problem; in fact, the high rate of data acquisition in observational astrophysics is often a driving force in the development of innovative solutions for scientific data visualization [1]. Even beyond the research aspects, presenting this information to non-expert users in a way that they can easily understand (e.g. for dissemination purposes) is a difficult challenge in itself.

Virtual Reality (VR) technologies are, in this sense, extremely powerful tools that can be used to craft real-time interactive applications for data exploration, allowing astronomers to break down and understand large datasets. This is possible by creating realistic 3D environments where users can interact with visual representations of the data in otherwise impossible ways, offering different perspectives on the observed phenomenon and endless new possibilities, such as remote collaboration. At the same time, these immersive experiences (especially when combined with peripherals like headsets or motion tracking hardware) are a great opportunity to reach a large audience outside of the scientific community, by creating game-like scenarios (or sandboxes) that can be used for educational purposes and result in higher levels of engagement and often better retention.

Multiple research studies and real-world experiments conducted in recent years can validate these points, for example:

- A 2014 paper presented during the IEEE International Conference on Big Data [2] explored the possibilities offered by VR visualization of astronomical data using the *Unity 3D* engine along other platforms, and found out that scientists performed better in the immersive environment compared to their "traditional" bi-dimensional tools.

- A study [3] published in 2016, sponsored by HTC Vive, compared learning rates and test scores of two groups of high school students, one of which used the Universe Sandbox VR application and the other received a more

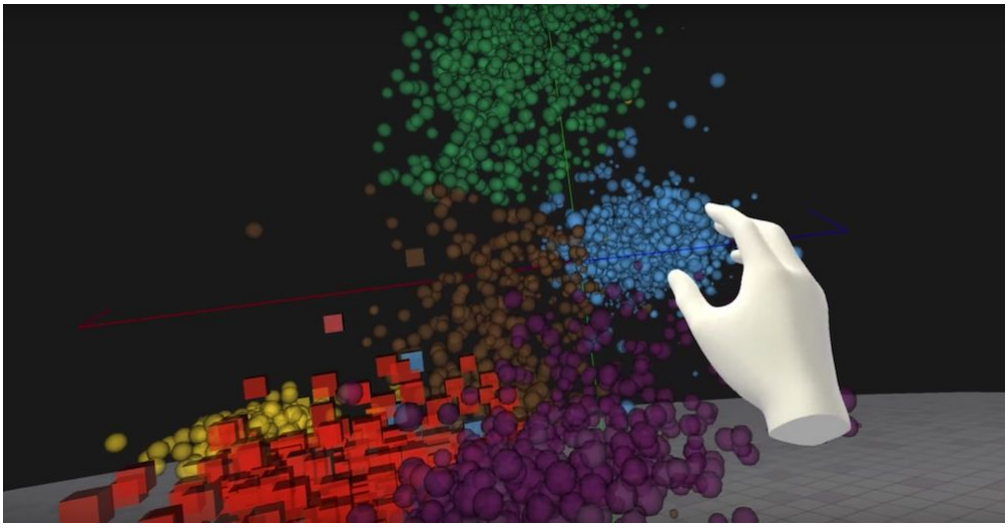


Figure 1: An example of data visualization in VR

"traditional" education. In the immediate post-test, the VR group scored an average of 93%, whereas the control group scored an average of 73%. These results persisted even after two weeks, when the average scores of the two groups were 90% and 68%, respectively. Especially meaningful is a comment collected from one of the student participants, who said:

"*I feel like I'm in the middle of the universe. It's so beautiful. I hope that VR can be available in my school as soon as possible – I will be extremely interested in the VR-based subjects.*"

Virtual Reality in aerospace and astronomy

Historically, the concept of a "virtual reality" has been around since the beginning of the 20th century [4]. However, it is only between 1970 and 1990 that, thanks to the rapid growth of 3D computer graphics, these technologies started to become real and actually usable, even though still in a rudimentary form; during these years, the VR industry mainly provided devices for medical, flight simulation and military training purposes. The aerospace field played a major role in the early development of these technologies, with NASA[1] developing the Virtual Interface Environment Workstation (VIEW) - a head-mounted stereoscopic display system equipped with motion-tracked gloves and suit, used for astronaut training - in the late '80s [5].

VR and AR technologies have therefore always been showing a lot of potential for transforming many fields of research and engineering, but since the '90s the commercial focus of these devices shifted heavily towards the entertainment industry.

[1]NASA: agency responsible for USA's space program and aerospace research

It is only in recent years that these tools began to be used in a more widespread way for scientific and engineering applications.

The intuitive, low-cost and low-risk immersive experiences that these technologies provide are very useful in the aerospace field, where they are mostly used to help with advanced personnel training (e.g. astronauts), spacecraft design, mission planning, scientific data analysis and simulation. NASA is still at the forefront of companies that research and employ these technologies for enhancing their operations. Since 2017, a team of experts at Goddard Space Flight Center, led by Thomas Grubb, has been exploring potential uses of VR and AR for the agency's own scientific and engineering needs; this research led to the development of several immersive applications currently in use at NASA, including a VR visualization of the Earth's magnetosphere (figure 2) and a tool to virtually assemble a spacecraft and validate some of its characteristics [6].

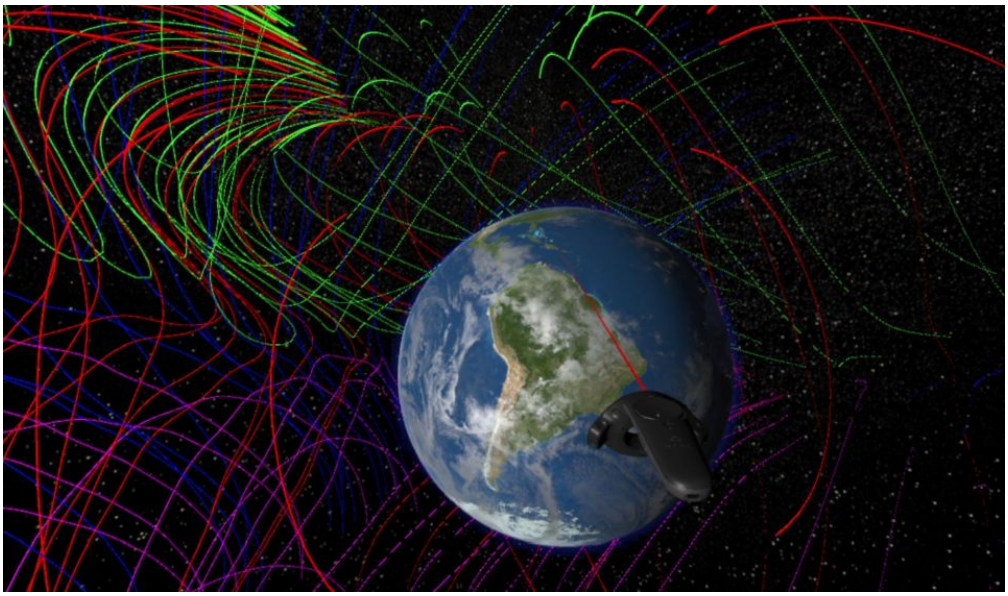


Figure 2: NASA's application to visualize the Earth's magnetosphere in VR

On the other hand, astronomers have been traditionally conducting their studies using paper graphs, databases and other legacy techniques, which make it difficult to have a comprehensive understanding of the observed phenomenon and spotting patterns or correlations inside the analyzed data. According to NASA, "[...] scientific discovery is one of the most compelling reason to develop an VR/AR capability" (P. Hughes) and Grubb's team has been dedicating to the creation of such tools, with very promising results (see paragraph 1.1).

The important new discoveries made possible by these technologies, along with the increasing amount of data available to astronomers, are pushing virtual reality forward as a first-class astronomical research and data visualization tool.

The thesis project

ALTEC - Aerospace Logistics Technology Engineering Company - is an italian excellence providing logistics and engineering support to the International Space Station and other space exploration projects, such as the *Gaia* mission. The company is part of the DPAC (Data Processing and Analysis Consortium), responsible for collecting and processing data to produce the Gaia star catalogue (see paragraph 2.3), which already counts over 1.5 billion observed objects.

In the context of the NEANIAS European project [7], which aims at developing digital services to satisfy the evolving needs of the scientific community, ALTEC intends to provide a realistic and interactive VR representation of astronomical datasets, that could be used to visualize the Gaia catalogue for both research and educational purposes.

This project's goal is therefore to extend the capabilities of the already existing *Astra Data Navigator* application, by developing a system able to load very large star catalogues (made up of millions or even billions of objects) and represent them in a 3D environment. The tool should also provide meaningful ways to explore and interact with the available data, all of it while running in real-time on a single workstation, using the Unity game engine.

The thesis is split into 5 chapters, whose contents are briefly described in the following paragraphs.

In the first chapter, several existing solutions for star catalogues visualization (including *Astra Data Navigator* in its original form) are analyzed and compared with each other, focusing on the number of stars that they are able to display, the fidelity of the representation, the types of interaction they provide (for both scientific and educational purposes) and the data sources they support.

The second chapter will contain an overview of the project requirements, along with a description of the major technologies and tools used throughout the development of this thesis, including the Unity game engine and the star catalogues themselves.

The third chapter of the thesis will focus entirely on the design process and the development of the application, providing details about how new features have been implemented and the optimization techniques chosen to overcome some technical limitations.

Results of this work will be presented in the fourth section, followed by an in-depth analysis of the major issues that were encountered during the development, some of which still remain open, and how they impact the application. Performance metrics will also be provided, highlighting both positive results and some of the aforementioned limitations. Finally, this chapter will contain an analysis of user feedback and the data that was collected through user tests.

The last chapter of the thesis will compare the final results of this work with the original design goals, and will explore potential improvements and new features that could be implemented in the future.

Chapter 1

State of the Art

As the amount of astronomical data available to the scientific community increases and larger catalogues are assembled, the need of data visualization tools has lead to the development of several solutions for representing and analyzing such information. Among the numerous systems that were proposed and developed for solving the visualization problem, the focus has been clearly shifting towards computer graphics and virtual reality solutions. The large amounts of data that these applications have to deal with, though, make the creation of such systems a difficult challenge mostly due to technological limitations (in terms of memory, storage, bandwidth and computing power requirements) that each application tried to overcome using a few different approaches.

1.1 Point clouds

One increasingly common way of representing very large datasets is the use of point clouds. A point cloud is a set of data points in space, each with its own set of X, Y and Z coordinates, which can be easily used to encode and display three-dimensional information; the point cloud can be 4D if color data for each point is also present. This technique is mainly adopted in the field of 3D scanning and photogrammetry, where it is used to encode very precise and fine-grained information about the shape of objects and their external surfaces (usually captured with LIDAR scanners), but it can also be applied to big data visualization and analysis thanks to its compactness (only 3 or 4 values per-point, with ongoing efforts for point clouds compression standards by the MPEG[1] [8]), efficient processing techniques (using machine learning or filtering algorithms) and the ease of rendering such points, even in large amounts, on modern GPUs.

[1]MPEG: Moving Picture Experts Group, an alliance of working groups that sets standards for media coding and transmission and file formats for various applications

The main drawbacks of this approach are the limited amount of information that can be encoded in each point and the poor level of interaction with the representation (often just restricted to scaling and rotating the dataset to observe its spatial distribution).

Despite these limitations, point clouds have recently been employed for visualizing large star catalogues: in 2020, using a VR simulation (powered by NASA's *PointCloudsVR* application) that animated the speed and direction of 4 million stars in the local Milky Way neighborhood, astronomers obtained a new perspective on the stars' motions, which helped them classify star groupings and gain a better understanding of how the local stellar neighborhood formed [9].

1.1.1 GAVS - Gaia Archive Visualization Service

The *Gaia Archive Visualization Service* [10], available on the official Gaia Archive website (see paragraph 2.3.1), is an interactive tool to visually explore and analyze the huge catalogue of stars observed by the Gaia satellite since 2014. Through the web interface, it is possible to navigate the dataset using 2D maps of the galaxy, plot user-configurable graphs of star attributes and view a 3D point cloud representation of (almost) the entire catalogue (figure 1.1).

This tool is of terrific value to any scientist interested in exploring this dataset, since it provides almost unrestricted access to all the information collected by the Gaia satellite in a visual and interactive way, but it is completely obscure to any non-expert user. It does not provide any immersion, since most of the data is visualized in a bi-dimensional form and the only 3D representation is a point cloud with very limited interaction.

1.1.2 Gaia 3D Starmap

This web application [11], created by developer Charlie Hoey using JavaScript and the WebGL API, provides a monochromatic point cloud representation (figure 1.2) of around 2 million stars taken from the first Gaia data release (DR1). Its peculiarity resides in the peer-to-peer system used to distribute the dataset to clients that are loading the application, which then share it to other users later on, making it extremely scalable and independent from a central server.

Being a WebGL app with low memory and bandwidth requirements, it can be run by any device connected to the internet, including smartphones, while also supporting VR headsets through the WebVR API. As already discussed, though, the interaction with the point cloud is very limited and only slightly improves what is offered by the *GAVS* (which on the other hand is able to load much more data points), heavily restricting its potential for both scientific and educational purposes.

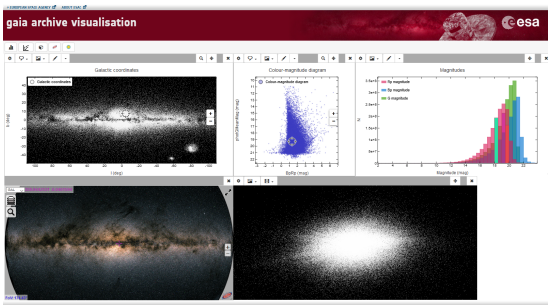


Figure 1.1: *GAVS* web interface

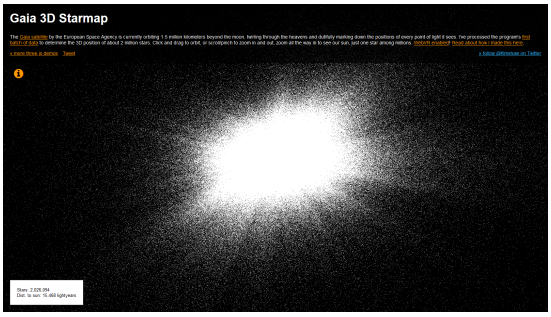


Figure 1.2: Screenshot of the *Gaia 3D Starmap* application

1.2 Planetarium software

Planetariums are theatres built primarily for presenting educational and entertaining shows about astronomy and the night sky, or for training in celestial navigation [12]. They feature a large dome-shaped projection screen where stars, planets and other celestial objects are shown and made to move realistically as they would appear from a point of view centered on the Earth (simulating their proper motion). Traditionally, these systems employed special projectors made by combining optical and electro-mechanical technologies (called "star balls") to create an image of the celestial scene on the planetarium dome. Over the last 20 years, an increasing number of planetariums have switched to using digital systems that rely on specific software to generate realistic images of the night sky and one or more digital projectors to display them.

Digital planetariums have several advantages over "traditional" ones beyond reduced costs: the data that is displayed can be changed dynamically at any point in time and additional interactivity is allowed by the system, such as getting more information about an object, zooming or changing the viewpoint in 3D space. These virtual reality capabilities provide important educational benefits, because they stimulate curiosity in the audience and convey an additional sense of depth that would otherwise not be present.

Of course, such systems also have their limitations: the immersive experience cannot be recreated outside of these large installations (e.g. by a single user with a VR headset), but most importantly the viewpoint is typically restricted to the Earth's surface, which subtracts from the three-dimensional nature of the universe and makes it inappropriate to use these software for scientific data visualization.

Figure 1.3: Screenshot of *Stellarium*, a free planetarium software

1.2.1 Stellarium

Stellarium [13] is an open-source planetarium software, which is able to render a realistic projection of the night sky in real time. It is developed using OpenGL and is available on Windows, Linux and macOS as well as all major mobile operating systems and any web browser (figure 1.3).

The application works by simulating what would be seen (e.g. using a telescope) from some point on Earth's surface at any moment in time, showing a realistic 3D representation of the sky projected on a curved dome-shaped surface. In the default configuration, the provided astronomical catalogues contain over 600,000 stars and 80,000 deep sky objects, but they can be easily expanded to more than 177 million stars and 1 million deep sky objects. Using a powerful zoom feature, the user can explore the celestial dome (which is updated in real time) and select any object to open a panel containing additional information about it. Sunrise and sunsets, eclipses, constellations, comets and shooting stars as well as other effects such as star twinkling are also simulated and visualized by this application.

Stellarium is mostly oriented towards educational and entertainment purposes, whether used in a planetarium installation or by any field astronomy enthusiast (the so called "backyard astronomers"). Even though it can display very large catalogues and is extendable via plug-ins and scripts, it is not well suited for scientific purposes such as astronomical research because of the viewpoint, which is restricted on the Earth's surface and prevents the user to observe the dataset from different perspectives.

1.3 Universe exploration in Virtual Reality

Another common way of doing astronomical data visualization is to build real-time interactive applications that display celestial objects in a fully three-dimensional space and allow the user to navigate in the universe using a first-person perspective. Instead of representing the data from a statistical point of view (such as with point clouds) or from a static point of view (like a planetarium), all of these applications focus heavily on the exploration side of the experience, placing the user in a realistic 3D environment where the look and - to some extent - the behaviour of celestial bodies is being simulated, therefore providing an immersive and highly interactive way of visualizing such datasets.

These applications are well suited for educational purposes as well as scientific research, because they allow both professionals and non-expert users to find useful information and extract some meaning from the presented data. Their main limitation resides in the size of supported astronomical catalogues, which is often lower compared to a point cloud representation, due to the higher computational cost of rendering and updating such virtual universe.

Following this paragraph is an overview of the most advanced and successful applications of this kind available to date, with special attention given to their support of large star catalogues and the types of interaction offered to the user.

- **Celestia** [14] is a free and open source 3D astronomy software available for all major desktop and mobile operating systems, created by Chris Laurel and whose development is now carried on by the community. It displays the full Hipparcos catalogue (118,000 stars, see paragraph 2.3) along with many planets, satellites (both natural and artificial), asteroids and comets. Orbits are computed using the VSOP87 planetary theory, without simulating gravity, but the positions of both stars and galaxies are fixed. There are several add-ons available for download, which can also expand the number of celestial objects handled by the application.

  Inside of this simulated universe (figure 1.4) the user is free to navigate at different speeds, orbit around celestial bodies, track moving objects, zoom the view in and out and move forward and backwards through time (up to 2 billion years).

- **Space Engine** [15] is a single-user virtual universe simulator, created by the russian developer and astronomer Vladimir Romanyuk and available for Windows PCs. It features over 130,000 real objects (including the Hipparcos star catalogue and several exoplanets with their host stars) but also employs scientifically-accurate procedural generation algorithms to create (according to the author) trillions of pseudo-realistic objects and place them in the simulated universe. Celestial bodies that form a planetary system move, stars rotate around their axes and orbit each other in star systems, but stellar proper motion is not simulated and galaxies do not rotate nor move.

Similarly to *Celestia*, the user can freely travel inside the large universe collecting information about the selected objects (figure 1.5) and can also alter the speed of time to observe different astronomical phenomena simulated by the application.

- **CosmoScout VR** is a virtual universe simulator developed at the German Aerospace Center (DLR); the project is open source [16] and can be built for both Linux and Windows platforms. *CosmoScout VR* uses physically based rendering and is capable of accurately compute the positions of celestial bodies and spacecrafts using SPICE, it can also reproduce planetary surfaces using terrain maps. It is exclusively based on real astronomical data and can display the Hipparcos, Tycho and Tycho2 star catalogues (see paragraph 2.3) all in real-time.

  The application is subdivided into several plugins which are completely optional and can be used to customize the feature set of the application to suit each user's needs. Some of these features include landing on planets and using several terrain measurement tools, simulating and showing trajectories of celestial bodies, interfacing with Web APIs, record high-quality videos and so on. The application's support for a wide variety of input and tracking hardware, VR headsets and stereoscopic systems makes the user interactive experience much more immersive.

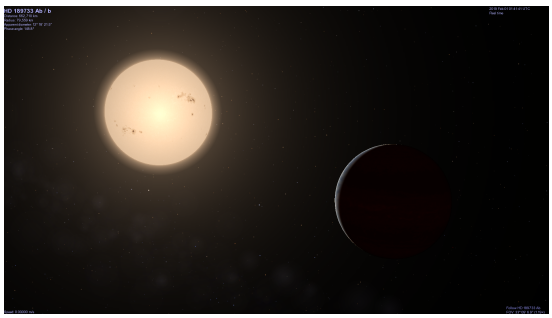


Figure 1.4: An example scene captured in *Celestia*

Figure 1.5: Screenshot of the *Space Engine* application

1.3.1 Gaia Sky

One of the most advanced real-time 3D astronomy visualization software is *Gaia Sky* [17], developed in the framework of ESA's Gaia mission by the Astronomisches Rechen-Institut (University of Heidelberg, Germany). It is free and open source, it runs on Windows, Linux and macOS and supports a wide variety of VR headsets and stereoscopic devices. The software is written in Java, using OpenGL as its graphics API and GLSL as the language for writing shaders.

The biggest technical achievement of this application is being able to load and display a great portion of the Gaia catalogue (up to more than 680 million stars) along with the Solar System and several other astronomical catalogues for asteroids, exoplanets, galaxies and nebulae. There are plenty pre-packaged datasets available for download from inside the application itself, which can be expanded by the user using standard formats such as VOTable, FITS and CSV.

The user is free to explore this large and complex virtual universe (figure 1.6) choosing between a "focus" mode (where the camera is always locked to the selected celestial object), a free camera mode and the option of following the point of view of the Gaia satellite. The navigation of the universe is completely seamless and accompanied by many features and tools for data visualization, such as displaying the orbit of all Solar System objects and motion vectors for stars, showing constellation names and their shape, highlighting star clusters, applying real-time filters to the data and visualizing phenomenons like relativistic aberration and gravitational wave effects [18]. As with the other applications of this kind, *Gaia Sky* also simulates the passing of time by moving planets, satellites and asteroids along their orbits and giving the user control over the speed of the simulation.

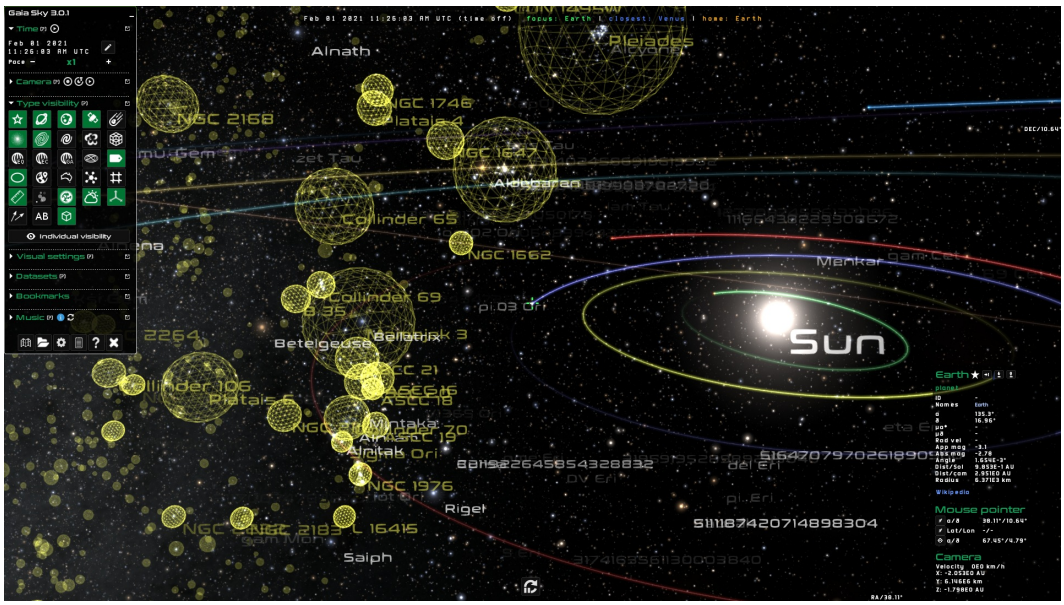


Figure 1.6: The virtual universe of *Gaia Sky*

Table 1.1 draws a comparison between star catalogues and VR support of *Gaia Sky* and all previously mentioned applications, which represent the state of the art of astronomical data visualization in Virtual Reality.

Application	Supported star catalogues	Expandable dataset	Navigation options	VR support
Gaia 3D Starmap	Gaia DR1 subset (2M objects)	No	Rotate point cloud, zoom in/out	WebVR
Stellarium	Up to 177M stars	Yes (with plug-ins)	Powerful zoom	Planetarium dome projection, spheric mirror projection
Celestia	Hipparcos	Yes (with add-ons)	Point-and-goto, exponential zoom, spaceflight	Specialized CAVE ports
Space Engine	Hipparcos and procedural generation	No	Click-and-go, exponential zoom, spaceflight	SteamVR, Windows Mixed Reality
CosmoScout VR	Hipparcos, Tycho and Tycho2	No	Free movement, Solar System map	VR headsets, stereo systems, tracking hardware, CAVEs
Gaia Sky	Gaia EDR3 (up to 680M stars)	Yes (VOTable, FITS, CSV and others)	Select-and-goto, free movement, spaceflight	VR headsets, 3DTV, stereoscopic and multi-projector systems

Table 1.1: Comparison between VR astronomical data visualization tools

1.4 Astra Data Navigator

Astra Data Navigator (ADN) is a stereoscopic and interactive 3D application developed by ALTEC. It is able to visualize many celestial objects in the Solar System (accurately positioned in space according to their real position at 00:00:00 of 20 June 2019) along with a large number of stars contained in the Hipparcos catalogue (around 113,000), which is loaded from a text file at application startup. All of these objects are static, but additional work is being carried on to introduce the concept of time in the simulation, updating planets' and satellites' positions over time and moving them along their orbits. Stars are rendered using a static particle system, except for those that are closest to the camera, which are instead represented using shaded 3D models.

ADN provides two main ways of navigating the virtual universe:

- **Warp:** the user can pick the celestial object of interest by either selecting it in the scene or using the search feature to find it by name; then, with the click of a button, a "warping" animation will move the camera near the position of the target object.

- **Free movement:** the camera, which by default is locked to the selected object, can be unlocked and controlled using W/A/S/D or the arrow keys. In this mode, the user is free to move at any speed (adjustable using the mouse scroll-wheel), albeit performance is sub-optimal when traveling too fast.

1.4.1 Large star catalogues in Astra Data Navigator

The goal with this new version of *Astra Data Navigator* was to provide a VR application capable of visualizing large astronomical datasets (in the order of several millions of stars) that are typically represented using point clouds, but with the interactivity and ease of navigation that only VR universe exploration tools can provide. The user should be able to look around and move freely between any entity in the scene, gathering additional information about the objects by interacting with them and observing each one as a realistic 3D model.

In the most recent version of *ADN* , the baseline dataset loaded by the application is a pre-computed subset (see paragraph 3.2.3) of the Tycho2 catalogue, containing around 2.5 million of the brightest stars in our galaxy. Instead of relying on text files, the star catalogue is now stored in the form of a SQLite database. The user also has the ability of extending the application dataset by connecting it to an external data source, such as another database or a formatted text file (CSV), from which more stars will be loaded.

Another new feature added to *Astra Data Navigator* is a configuration file, which can be edited by the user (according to the provided documentation) to customize the application behaviour and dataset loading. For example, star catalogues can be handled using two methods: a static loader, which reads all of the available data at startup (allowing for seamless movement in the universe, without further loading), or a more dynamic approach which loads new stars progressively as the observer moves through the scene. In any case, the maximum amount of stars that can be active at once is capped at a value of 5 million for performance reasons. Using the dynamic loading method it would technically be possible to use the full Gaia catalogue with *ADN*, but loading times would be extremely long and therefore impractical.

The data available for each star has also been updated to the latest data release of the Gaia catalogue (EDR3) and now includes the source identifier along with cross-matching IDs, X/Y/Z positions of the star, its radius and the effective temperature measured by the satellite. For a small subset of stars, the application now has additional information such as their proper name and the name of their constellations. All available data is shown to the user via an *InfoPanel* every time that an object is selected.

In order to improve the visual fidelity of the 3D environment, the rendering quality of both far-away and close-up stars has been improved using shaders and other effects based on some physical parameters (such as the temperature, which is used to determine the surface color of each star).

Astra Data Navigator is designed to work on desktop operating systems like Windows and macOS, as well as web platforms such as WebGL. It already supports active stereoscopic projectors, while a version offering integration with VR headsets is currently work-in-progress.

Chapter 2

Technologies and Tools

2.1 Project requirements

Virtual Reality applications are complex software systems whose developers often need to solve many different architectural or technological problems in order to achieve the final result and fulfill all project requirements; this is true for VR applications targeting the entertainment world as well as those developed with a scientific purpose, such as this project. Following is an overview of the main requirements and goals of this work on the *Astra Data Navigator* application.

- **Scalability to large datasets:** the most up-to-date star catalogues contain information for up to millions or even billions of celestial objects; being a data visualization software, *ADN* should be able to load and represent large datasets with reasonable performance, while fitting into the resource limits of the target hardware (system memory and processing power).

- **User configuration and extensibility:** the goal of *Astra Data Navigator* is to provide the user with a 3D representation of astronomical datasets and large star catalogues, where there often isn't a one-size-fits-all solution; for this reason, the user should be able to decide which data has to be loaded by the application, extending the baseline catalogues with additional data sources and configuring several parameters of the program's behaviour to better suit his own needs.

- **Display star catalogues information:** interactivity is a key advantage of virtual reality representations, because it allows users to not only see the data in a 3D environment but also gain additional information about it through actions performed in the virtual world. This means displaying the parameters used to represent the object itself (such as position and size) as well as additional information such as catalogue IDs, common names, constellations and anything else that can be read from astronomical catalogues.

- **Smooth and intuitive navigation:** with such a large universe at his disposal, the user should be provided with ways to conveniently explore all of the available data. In this sense, the application must support several navigation methods which may fit different usage scenarios but should all be equally effective, while the Graphical User Interface (GUI) should not get in the way of such exploration but instead facilitate it. An additional goal, considering the large amount of data handled by the application, would be to have the loading process interfere as little as possible with the navigation, in order to not ruin the user experience.

- **Graphical fidelity of the representation:** when creating a virtual representation of real-world data, the realism of the 3D environment and its level of visual detail are key ingredients for achieving a truly immersive experience for all kinds of users. In this scenario, the graphical representation of celestial objects can also be used to show relevant attributes of the underlying data, such as the size, temperature and luminosity of stars. The 3D models, effects and rendering techniques used by the application should therefore be as accurate and realistic as possible, without compromising its performance profile.

- **Stereoscopy support:** even though VR applications can still be used on traditional flat screens, the experience and the level of immersion are greatly enhanced by dedicated hardware. In this case, the application has been designed to work in ALTEC's VR laboratory, which is a single-screen CAVE equipped with an active stereoscopic projector, and therefore it must take advantage of this more immersive view mode.

Since some of these problems are common and not unique to *Astra Data Navigator*, there are many tools and technologies already available on the market (or in the open source community) that provide powerful frameworks and well-tested solutions to help overcome these challenges without having to create everything from scratch. For this reason, several existing software packages and tools came into play during the development of this application.

2.2 Unity

Unity [19] is a cross-platform game engine (figure 2.1) that can be used to create 2D, 3D, Virtual Reality and Augmented Reality applications such as games, simulations and other experiences. It integrates several tools and libraries to make the development of said applications easier, and provides support for deploying the final product to over 25 different platforms. The engine itself has been written using the C++ programming language, but the development of applications using Unity is done through C# scripting.

The Unity game engine has been chosen for the development of *Astra Data Navigator* because it provides a simple and flexible all-in-one framework for creating

complex VR interactive experiences. An additional advantage of using Unity is being able to access the Asset Store, where members of the community can publish their own packages (free or paid) which implement additional features or contain pre-made content that can be easily reused. Compared to creating everything from the ground up (e.g. using C++ with the OpenGL graphics API [20]) this approach greatly reduces the amount of complexity and time to get to the end result, as well as simplifying the maintenance of the application and its portability to multiple platforms; additionally, Unity provides out-of-the-box support for the stereoscopic 3D devices required for this project.

Of course, using a general purpose game engine like Unity also has some drawbacks, which are mainly found in some limitations of the API and the performance overhead caused by the lack of full control over the application runtime. An expert developer, given enough time, could probably get better results by building custom tools and technologies tailored to suit the application's needs, but more often than not this small sacrifice of performance and control is worth it compared to the large gains in project complexity and reduced development time.

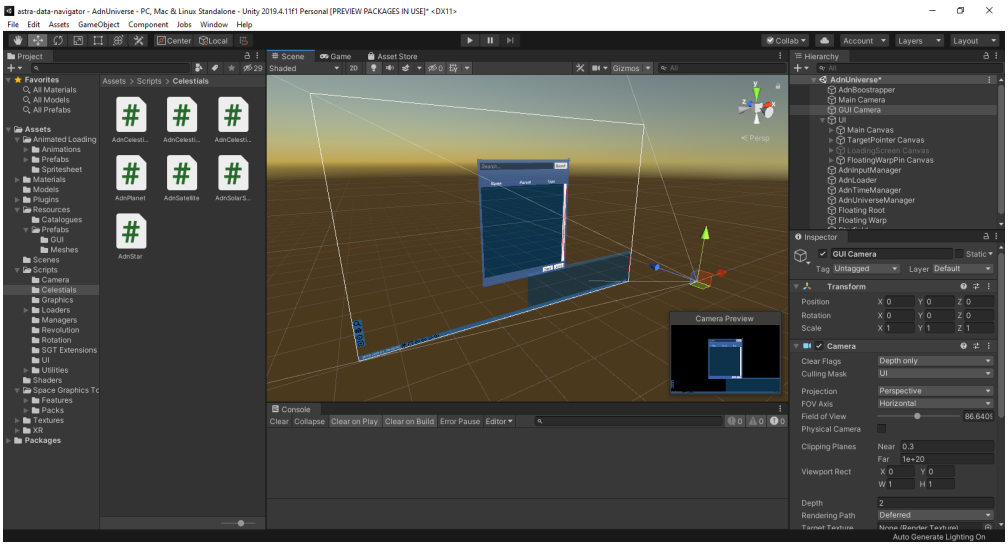


Figure 2.1: Screenshot of the Unity Editor

2.2.1 Floating point accuracy

One area where general-purpose game engines are usually lacking, when adopted to simulate large astronomical environments, is the precision of numerical data types used for rendering the scene. Unity, as many other game engines, uses single-precision (32 bit) `float` values to store the position, rotation and scale of each object; while this is fine for a typical application, large virtual worlds suffer from the accuracy degradation happening as objects get further away from the origin, causing visual artifacts (such as spatial jitter) and simulation inaccuracies.

Traditional approaches to overcome this issue fall into three main classes [21]:

- **On-the-fly shifting of coordinates:** using this approach requires that, before executing any meaningful calculation or rendering pass, the viewpoint and all objects are shifted to sit near the origin (to have small, accurate coordinates) and then restored to their previous positions.
- **Multiple local coordinate systems:** virtual worlds can also be subdivided into smaller regions, and objects' positions could be stored relative to the origin of their local coordinate system, limiting the accuracy degradation by keeping regions small enough. This technique requires additional data structures and management to handle the movement between different regions.
- **Floating Origin:** using this technique, instead of allowing the observer to move around the world, inverse transformations are applied to the entire scene while the camera is kept fixed at the origin (see figure 2.2). This way, objects closer to the observer (and therefore most relevant) will always have coordinates near the origin of the universe and small enough to not cause any visual artifact. This approach has an additional performance cost, especially for complex scenes, since transformations have to be applied to all of the objects instead of just the camera.

In *Astra Data Navigator* , this problem has been solved by integrating the *Space Graphics Toolkit* asset, described in more detail in section 2.2.2.

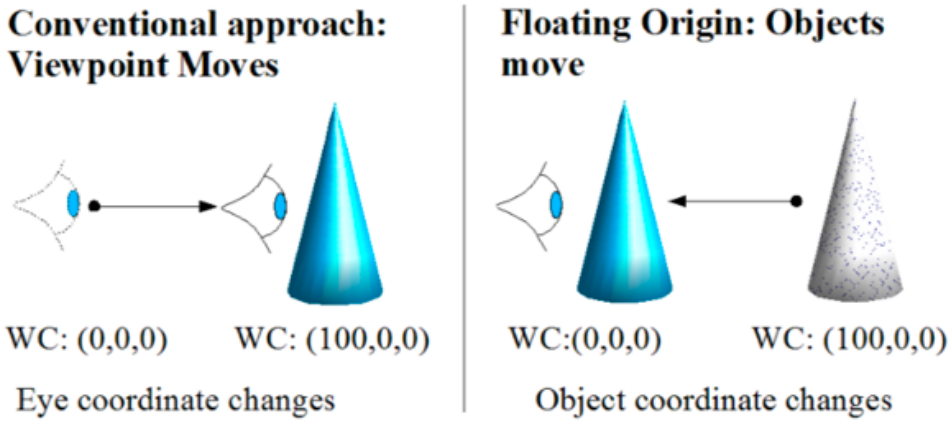


Figure 2.2: Comparison between traditional navigation and the floating origin technique

2.2.2 Space Graphics Toolkit

The *Space Graphics Toolkit (SGT)* is an asset developed by Carlos Wilkes, available for purchase on the Unity Asset Store [22]. It provides a vast collection of pre-made

(but customizable) scripts and graphical effects which are designed to simplify the development of space games and achieve realistic visuals.

The solution to the floating point precision problem adopted by *SGT* is an hybrid between the "traditional" floating origin technique and multiple local coordinate systems. Every *GameObject* in the scene with a `SgtFloatingObject` component attached to it will be moved as part of the floating world, while the camera is always kept near the origin (as an optimization, the shifting happens only when the observer has moved more than 100 units away from the origin) but, in addition to that, the universe is virtually split into cubic cells (each one being 5×10^7 units large). The position of each object is stored in a `SgtPosition` component using three 64 bit integers to identify the cell and three floating point values to position the object inside of it. According to the author, combining these two methods together allows to represent a Unity scene as large as the observable universe, without incurring in any noticeable precision issue.

2.3 Star catalogues

There are several astronomical catalogues containing information about stars, which have been produced for different purposes over the years [23] and are all publicly available to consult and download. Since listing every single star in the sky is an almost impossible goal, the so called "full-sky" catalogues attempt to record every star brighter than a given magnitude.

The star catalogues which are more relevant to this work are described in the following list.

- **Hipparcos (HIP) [24]:** compiled with the data collected by European Space Agency (ESA)'s astrometric satellite *Hipparcos*, this catalogue was published in 1997 and contains high-precision measurements of brightness, proper motions, parallaxes and spectral information of 118,218 stars.
- **Tycho-2 [25]:** an astronomical catalogue containing positions, proper motions and photometric data, derived from the observations collected by the *Hipparcos* satellite, for more than 2.5 million of the brightest stars in our galaxy. The reason why this catalogue is much larger than HIP is because the former only contains data that was measured with an accuracy greater than 2 milliarcseconds[1], whereas the overall error for all stars in Tycho-2 is 60 milliarcseconds.
- **Gaia:** the *Gaia* satellite (figure 2.3) was launched in 2013 as a follow-up to the *Hipparcos* mission and aims to construct the largest and most precise 3D space catalogue ever made; the full dataset will be published in 2022, but available data is being released in stages that contain increasing amounts

[1] 1 arcsecond = $\frac{1}{3600}$ of a degree

of information. Currently, the latest Gaia early data release (EDR3 [26], published December 2020) contains positions and brightnesses for 1.8 billion stars, including distances and proper motions for more than 1.3 billion stars.

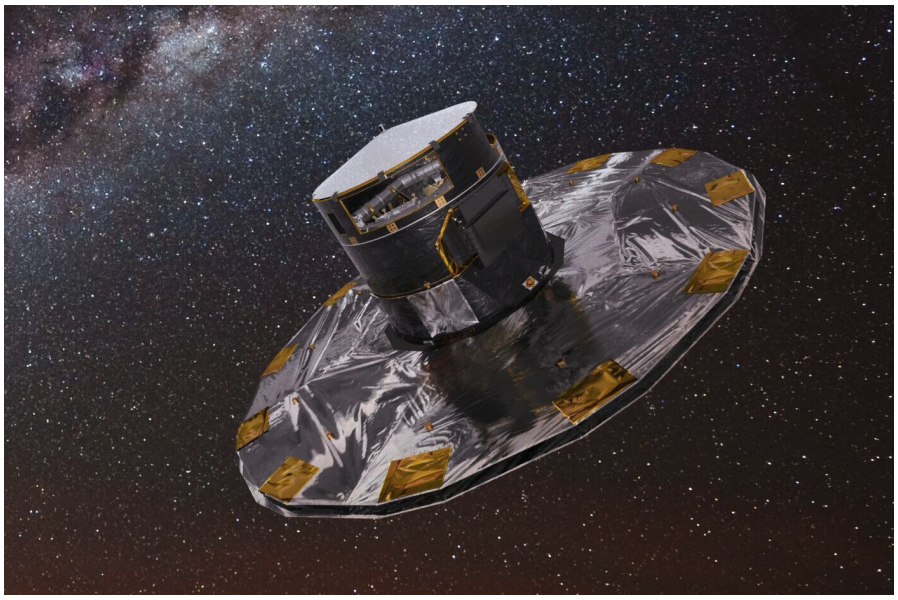


Figure 2.3: Image of the *Gaia* astrometric satellite

2.3.1 Gaia Archive

The Gaia Archive [27] is the official ESA web portal to the entire Gaia catalogue. It contains up-to-date information about the *Gaia* mission and provides ways to download the data files, run basic searches or advanced queries on the database (using the Astronomical Data Query Language, ADQL) as well as having a detailed documentation about every aspect of the dataset and its meaning. In addition to the various Gaia data releases, the archive also contains several external catalogues such as HIP, Tycho-2 and many others, while providing cross-matching tables which can be used to correlate entries of multiple catalogues that refer to the same star.

Since the amount of data in the Gaia Archive is huge, manual queries might not always be the best way of accessing the star catalogue information. For this reason, ESA provides a Python *Astroquery* package, called `astroquery.gaia`, to allow programmatic access to the Gaia Archive. *Astroquery* is a set of tools for querying astronomical web forms and databases, developed as part of the *Astropy* project [28], which aims at creating an ecosystem of interoperable astronomy packages for Python. In the context of this work, the `astroquery.gaia` package was used to query and download chunks of the Gaia catalogue and insert them in the local database used by *Astra Data Navigator* , after applying some pre-processing (for more details, see paragraph 3.2.3).

2.4 Database systems

Relational databases have multiple advantages over the naïve approach of storing a large dataset in a text file that has to be parsed at application startup, because they allow for a much more structured and efficient access to the data (or just portions of it) using Structured Query Language (SQL). Additionally, writing information that is mostly numeric in the form of text often results in larger file size and slower parsing performance (see paragraph 3.2.1).

In order to store and access large star catalogues in *Astra Data Navigator* , more than one Database Management System (DBMS) has been used throughout the development of the application.

SQLite3

SQLite [29] is a publicly available software library, written using the C programming language, that implements a full-featured SQL DBMS. What makes it unique, compared to other DBMS software, is the fact that it's not a standalone process but a self-contained library which has to be built into the application that uses it, making it extremely lightweight and easy to run on any platform (it is, in fact, the most widely deployed and used database engine in the world).

SQLite3 has been integrated into *ADN* through the `System.Data.SQLite` library [30] for C# and is used to store and access the baseline star catalogue, which is included with any release of the application.

PostgreSQL

PostgreSQL [31] is a powerful open source and cross-platform object-relational database system. As most DBMS software, it is run as a separate process (typically on a dedicated server) that applications can connect to for accessing its data. Compared to SQLite, it is much better suited to handle very large amounts of data since it doesn't rely on a single file.

In the context of this project, PostgreSQL was used to set up an external data source (larger than the baseline catalogue) from which *Astra Data Navigator* could load additional star information.

2.5 Hardware

The application is designed to run on a *Dell Precision Tower 7810* machine, which has two *Intel Xeon E5-2650 v3* processors (each with 10 cores and a frequency of 2.30 GHz), 32GB of RAM and a *NVIDIA Quadro K5200* graphics card.

2.5.1 Stereoscopic system

To achieve the stereoscopic 3D effect (also called "3D view") each of the viewer's eyes must receive a slightly different image accounting for the distance between the pupils (IPD); the two views are then merged together by the human brain, creating the illusion of a three-dimensional image. There are many technologies that can make this happen, but they can all be grouped into two categories [32]:

- **Passive** systems rely on the ability of the screen (or projector) to display two images at once in such way that they can be easily separated, for example using different light polarization or chromatically opposite colors (typically red and cyan). Through the use of passive filters such as polarized glasses or colored lenses, viewers can send the appropriate input to each eye and perceive the stereoscopic effect.
- **Active** stereoscopy, instead, makes use of glasses with electronics that interact with the display. In this scenario, the screen rapidly alternates between presenting an image for the left eye and one for the right eye, while the viewer's glasses (which need to be synchronized with the display) employ a shutter system to let each eye only see the appropriate images; this is usually achieved by using liquid crystal shutters on the lenses, that become dark when some voltage is applied and can be made transparent again when the next frame is displayed on the screen.

ALTEC's VR laboratory (figure 2.4) is equipped with a *Barco RLM-W14* active stereoscopic projector (figure 2.5), which operates at a resolution of 1920x1200 pixels and a refresh rate of 120Hz; the stereoscopic glasses in use are synchronized with the projector through an infrared signal. Due to the nature of active stereoscopy, the overall perceived (per-eye) image refresh is halved with respect to the screen's refresh rate, and therefore in this case will be 60Hz.

Figure 2.4: ALTEC VR room

Figure 2.5: ALTEC's stereoscopic projector

Chapter 3

Design and Development

3.1 Architecture

The main goal of this work, as already mentioned, was to extend the capabilities of an existing VR application to support the visualization of large star catalogues. A choice that was made since the beginning of development has been to keep the overall system architecture as close as possible to the existing one and only make the changes required to add new features or improve performance. By doing so, any user that was familiar with the previous version of the application would still be able to use it without too much effort, because of the similar user experience and GUI, while also taking advantage of the new features and improved functionality.

The Unity project of *Astra Data Navigator* consists of a single scene, which contains a few elements that take care of initializing the application state at startup and populate the scene with instances of the celestial objects loaded from catalogues. Some of these components are also responsible for updating the virtual universe at runtime, as a consequence of actions performed by the user inside the simulation. The diagram in figure 3.1 represents a simplified overview of the application's flow of execution, from startup to handling user interaction.

3.2 Star catalogues

Before the development of any new or improved feature began, an initial analysis was conducted in order to plan how *ADN* could support large datasets and satisfy the requirements described in paragraph 2.1. The main points addressed by this analysis were the following:

- **Choice of a data source format:** the encoding of the catalogue, as well as its storage format, could have a major impact on the resource requirements of the application (e.g. storage space) and its cross-platform compatibility.

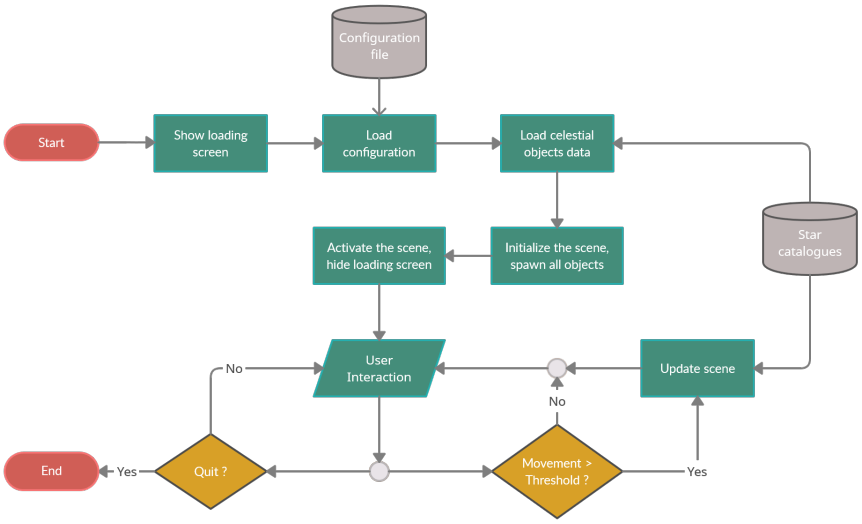


Figure 3.1: Simplified overview of *Astra Data Navigator* 's execution flow

- **Selection of star parameters:** increasing the set of catalogue parameters used by *ADN* means that users will have more data at their disposal, but will also impact the amount of resources used by the application.

- **Planning support for user-defined catalogues:** one of the project requirements is to make the astronomical dataset easily expandable by the user. In order for this to be possible, *Astra Data Navigator* must provide a standard way of creating compatible star catalogues and connect them to the application so that they can be used.

3.2.1 Data format

The previous version of *ADN* used regular text files to store the astronomical catalogues loaded by the application, which could then be easily serialized as a Unity *TextAsset* resource and used from the code without having to access any external file. This solution worked fine for the Hipparcos catalogue previously in use, but had to be reconsidered when aiming at loading datasets with several million of entries.

Text files are not ideal for storing large amounts of numeric data, because the ASCII encoding does not provide an efficient representation of numbers with many digits, thus increasing the overall file size. Additionally, these values require an extra conversion from their literal representation (C# `string`) to the appropriate numeric data type (`int`, `float` or `double`), making the loading process slower.

A database, on the other hand, offers a more standardized and flexible way of storing large amounts of information and should be much more capable of efficiently handling different data types (text, numeric, binary, etc...). Another fundamental

feature of database systems is the ability of randomly accessing information using SQL queries, a powerful tool for extracting a subset of the available data that matches some conditions, instead of having to sequentially read the entire file looking for the required entries.

The original solution was compared to a SQLite3 DB using a Python script that generated catalogues with a size between 10.000 and 1,5 million of randomized entries, in both `*.txt` and `*.db` format, which were then loaded by *Astra Data Navigator* while measuring the startup time (from code, using C# `Stopwatch` class). Results of this test (figure 3.2) show a 33% reduction in file size and roughly double the speed when loading the same dataset from a database rather than using a text file. Following this analysis, SQLite3 has been adopted as the main star catalogue format.

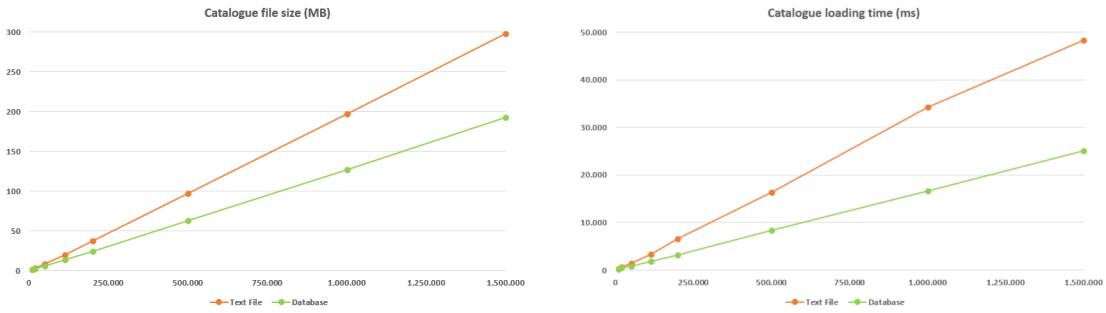


Figure 3.2: Comparison between text file catalogues and SQLite3 databases

3.2.2 Parameters

Typical star catalogues contain tens or even hundreds of astrometric and photometric parameters measured by the satellite for each observed celestial object. Many of these values are useful for astronomers or contain metadata about the catalogue itself (e.g. measurement errors), but do not translate to any meaningful property for a 3D representation of stars. The attributes chosen to represent a star in the application dataset must be carefully selected because, even though each one can be stored as a `double` (8 bytes), adding a couple of parameters may increase the storage and memory footprint of the application of a few hundred MBs (considering millions of stars in the catalogues).

For *Astra Data Navigator* , the set of required parameters has been reduced to those which are essential for a proper 3D visualization, while other fields can provide additional information to the user but are completely optional and not strictly required by the application (see table 3.1).

Parameter	Required	Type	Description
SOURCE_ID	Yes	`String`	Alphanumeric identifier of the star, must be unique inside the catalogue
X/Y/Z	Yes	`double`	Star position according to the J2000 reference frame, in parsec (pc)
TEMP_EFF	Yes	`float`	Effective temperature, in Kelvin (K)
RADIUS	Yes	`float`	Radius, in Solar Radius units (RSUN)
PROPER_NAME	No	`String`	Common name of the star (or NULL)
CONSTELLATION	No	`String`	Name of the constellation which this star is part of (or NULL)
HIPPARCOS_ID	No	`int`	Unique identifier of the star in the HIP catalogue (or NULL)
TYCHO_ID	No	`String`	Unique identifier of the star in the Tycho-2 catalogue (or NULL)

Table 3.1: *Astra Data Navigator* star catalogue parameters

3.2.3 Internal catalogues

While offering to users the option of loading custom datasets, *ADN* also includes a baseline star catalogue which is always loaded by the application and provides a starting point for populating the virtual universe with stars.

This database contains the vast majority of Tycho-2's entries, excluding stars with negative parallax measurements (removed because it wasn't clear how it should be handled when generating X/Y/Z coordinates), for a total of 2.441.300 objects.

Instead of taking parameters from the Tycho-2 catalogue, which is over 20 years old, all the information for these stars has been extracted from the latest Gaia data releases (DR2 and EDR3) using cross-matching tables and identifiers. This whole process has been automated through a Python script which uses the `astroquery` package to run a query on the Gaia Archive, executing a `JOIN` on the *source_id* attribute between the *gaiadr2.gaia_source*, *gaiaedr3.gaia_source* and *gaiaedr3.tycho2tdsc_merge_neighbour* tables, selecting all stars with positive parallax and then downloading the results. Additional pre-processing steps were required to prepare the data for the application catalogue:

1. Stellar positions are represented using equatorial coordinates (figure 3.3): Right Ascension (RA), Declination (DEC) and distance (which can be obtained from the parallax measurement). These are transformed to cartesian

coordinates using `astropy`'s *SkyCoord* function to convert those values into the galactocentric reference frame and then applying a simple offset to the result.

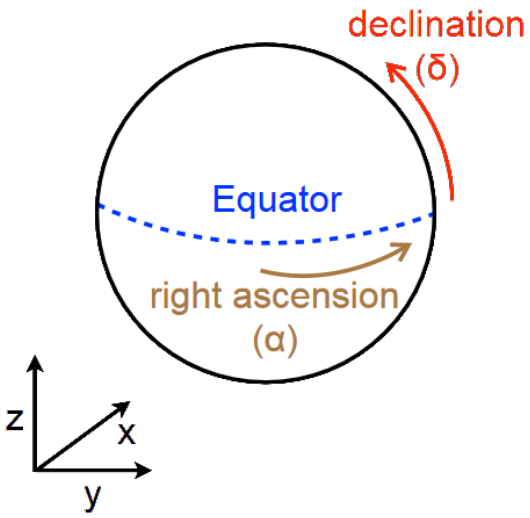


Figure 3.3: Example of equatorial coordinates

2. Some of the stars in the catalogue might have missing values for some of the parameters that are required by the application. In that case, the script provides a default value for those fields, which is taken from the corresponding parameter of the Sun.

3. Using a separate text-file catalogue which contains proper names and constellation data for over 300 stars in HIP, the script is also able to compile those fields for the most well-known objects in the dataset.

After performing all these operations, the script proceeds to insert the data into a SQLite3 database file called `adn_base.db`, which is loaded at startup by the *AdnStarDefaultLoader* script.

A secondary catalogue, containing similar information (excluding the cross-matching IDs) but only for HIP stars, is also built using this method but written to a text file instead. This is done to provide a fallback for those scenarios where, due to some platform restrictions (e.g. on WebGL) or because of some error, the `adn_base.db` file might not be accessible. In those cases *ADN* will run with a reduced dataset loaded by the *AdnStarHipparcosLoader* script, because text files can be serialized as a *TextAsset* and therefore are always available to the application thanks to Unity's resource system.

3.2.4 External catalogues

Additional dataset support in *Astra Data Navigator* is implemented through two loader scripts and the configuration file (paragraph 3.3). By editing `config.xml`, the user can specify the external data source type and provide the parameters for connecting to it at application startup.

The provided catalogue must contain all the required star attributes and can be in one of the following formats:

- **SQL database:** using Microsoft's ADO.NET framework [33] and the *AdnDatabaseAdapter* factory class, the application is compatible with multiple database providers (currently SQLite, PostgreSQL and Oracle). This is completely transparent to the *AdnStarDatabaseLoader* script, which is in charge of validating and loading data from the provided source using either the static or dynamic method (see paragraph 3.4).

- **Text file (CSV):** support for text files is provided because, while not being optimal for loading large amounts of data, it is a much easier format to produce (even by hand) and could still perform well for small or medium sized catalogues. The script which takes care of validating and loading the provided file is *AdnStarFileLoader*, but it only supports the static loading option.

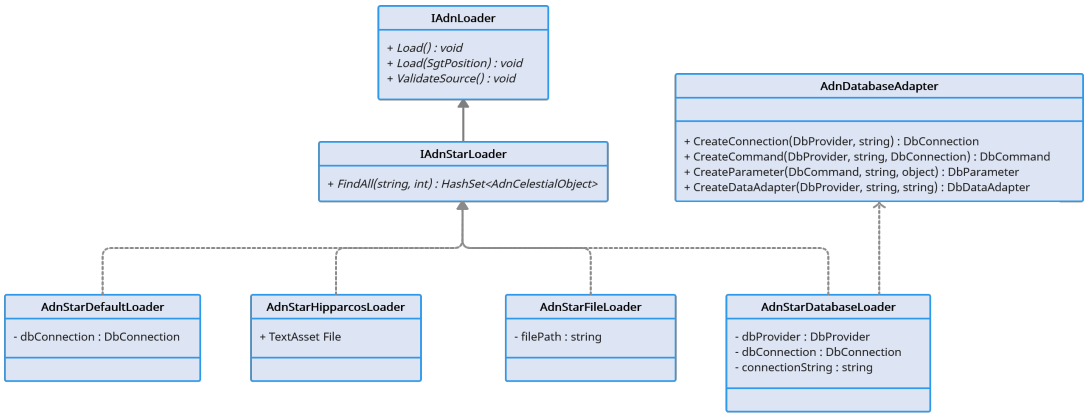


Figure 3.4: Architecture diagram of all star loader scripts, inheriting from the *IAdnStarLoader* interface, each handling a different data source type

3.3 Configuration file

As already mentioned, in *Astra Data Navigator* the user has control over a few aspects of the application's behaviour. This customization system is implemented through the use of a configuration file (`config.xml`, located in the the same directory as the executable) and the *AdnSettingsManager* script.

The choice of XML for the *config* file is mainly due to the human-friendliness nature of this format, which makes it easily readable and editable even by non-expert users, and its native support by the C# language (through the `System.Xml` library). File size and parsing performance are not a concern in this case, since `config.xml` will always contain a very limited amount of data.

The file is subdivided in a few top-level sections (called "nodes" by the XML standard), each one containing some user-configurable settings:

- **Execution:** used to specify the application's execution mode, which can be one of [PC, WebGL, HeadsetVR]
- **Revolution:** contains a few settings related to revolution motion simulation
- **StarLoader:** allows to customize the way *Astra Data Navigator* loads the provided star catalogues
  - **Method:** specifies which loading method will be used, available options are [Static, Dynamic] (see paragraph 3.4 for more details)
  - **DataSource:** type of the additional dataset that will be loaded by the application, supported values are [None, File, SQLite, PostgreSQL, Oracle]
  - **Params:** required for connecting to the external data source, it may be a file path or a database connection string
  - **StarPrefix:** this string will be combined with the SOURCE_ID to create the name of each star loaded from the user catalogue
  - **ValueToKm:** conversion factor from the dataset's length unit (default: parsec) to Km
- **Simulation:** general options regarding the universe simulation (such as the start date/time)
- **Logger:** this section contains a few settings to customize the event logging behaviour of the application

The *AdnSettingsManager* script is responsible for parsing the `config.xml` file at startup and loading settings that will be used by other scripts to perform their operations. This is ensured by adding the `[DefaultExecutionOrder(-1000)]` directive at the top of the file, which tells Unity to run this code before anything else in the project.

Inside its *Awake* function, the script opens the configuration file by creating an `XmlDocument` object with its path and then reads all the XML nodes using the *SelectSingleNode* method. All settings are initialized using values found in `config.xml` and then stored as `public static` variables, accessible from any part of the code. In case an error happens while parsing the options file, *ADN* will fall back to a set of default "safe" parameters after writing a warning message to the application log stream.

3.4 Dataset loading

In the application code, each star is represented by an *AdnStar* instance which holds all the available data about the object. This entity is part of a hierarchy of classes which derive from the common *AdnCelestialObject* base (figure 3.5).

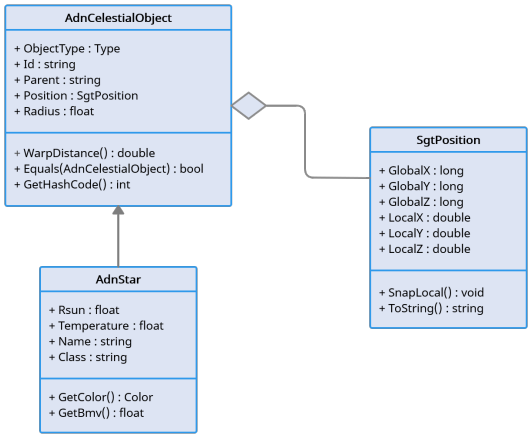


Figure 3.5: UML class diagram of *AdnStar* and related entities

Taking into account the size of all member fields, as well as the additional memory overhead introduced by the C# runtime for all reference types, each *AdnStar* object is estimated to be atleast 140 bytes large. This consideration, along with the fact that the execution time of a few algorithms scales linearly with the number of *AdnStar* entities, led to the introduction of a hard limit to the maximum amount of active stars in the application, set at 5 million. Loader scripts check how many objects are present in the virtual universe and stop loading new stars if the cap has been reached, writing a log message to warn the user about the event.

As it was briefly mentioned in section 3.3, the user is given control over the technique used by the application to load data from the available catalogues. Two methods are offered - **Static** and **Dynamic** - providing different trade-offs between loading times, navigation speed and system resource usage.

3.4.1 Static loading

The static method represents the most "traditional" and simple way of loading data from star catalogues. Using this option, loader scripts will only call the *Load* method once when the application is started and will proceed to read all the available entries from the provided data source (e.g. using a `SELECT *` statement without any `WHERE` clause), constructing a new *AdnStar* object for each one until the end of the catalogue or the limit on the number of stars is reached.

Using this solution means that, after a longer loading time at application startup, the navigation inside the virtual universe is a completely seamless experience. The main limitations of this mode are found in the increased resource usage (mostly RAM) and the inability of fully representing star catalogues of a (combined) size larger than 5 million objects. In all those cases where the dataset size does not exceed the maximum supported value, this method should be preferred over the alternative as long as the target system has the required resources (1.5 GBs of available memory).

3.4.2 Dynamic loading

When handling very large star catalogues, the static loading option is not a feasible solution for the reasons already discussed. This alternative method approaches the problem from a different perspective: it is based on the observation that, from a given position in the universe, only a minor portion of stars can actually be seen "by naked eye" and therefore not all data needs to be in memory at once.

When the application operates in dynamic mode, it periodically searches in the available catalogue to determine which objects are actually relevant for that viewpoint and loads them, resulting in a much smaller working set.

The most naïve way of determining star visibility would be to check the distance of each object from the observer and load all those that fall inside a pre-determined range. Using proximity as the only metric, though, results in much of the loaded data not being visible and the universe looking extremely empty.

The correct approach would be to consider the magnitude of each object but, since that information is not present in the application's internal catalogues, the visibility of stars is estimated based on their radius and their distance from the observer. The following query is used to select all stars in the catalogue which are considered "visible" from the current observer position:

```
SELECT source_id, x, y, z, temp_eff, radius
FROM stars
WHERE radius > 0.032 * SQRT( (@posX - x)*(@posX - x)
                           + (@posY - y)*(@posY - y)
                           + (@posZ - z)*(@posZ - z) )
```

where `@posX`, `@posY` and `@posZ` are placeholders that get substituted with the observer position values (in parsec) by the *DbCommand.Parameters.Add* function.

To avoid accumulating too many objects in memory, those that are no longer relevant should also be removed from the working set. For each entry selected by the dynamic loading query, the script looks up the object ID to check if it's already active in the scene and constructs a new *AdnStar* only when needed, while marking existing objects as "keep-alive". A cleanup pass is run after the loading is completed, to remove all stars that are no longer visible and free precious system resources.

The dynamic loading approach is still limited to a maximum of 5 million active stars, but it should rarely (if ever) hit it. The main advantages of this mode reside in faster startup time, reduced memory usage and scalability to much larger catalogues beyond 5M objects. The downside is that the application will freeze (after showing a message in the UI) to load new stars at runtime, which happens frequently when travelling at faster-than-light speed or if the user is warping between locations that are far away from each other.

3.4.3 AdnLoader

The script responsible for managing the whole loading process is *AdnLoader*. It's set up with a `[DefaultExecutionOrder(-80)]` directive to run it before the majority of other application scripts; inside the *Start* function (figure 3.6) it takes care of executing specialized loader scripts in a fixed order, handling any error that may happen (e.g. switching to *AdnStarHipparcosLoader* if the `adn_base.db` database cannot be accessed) and finally calls *AdnUniverseManager* to create the scene (see paragraph 3.5.2).

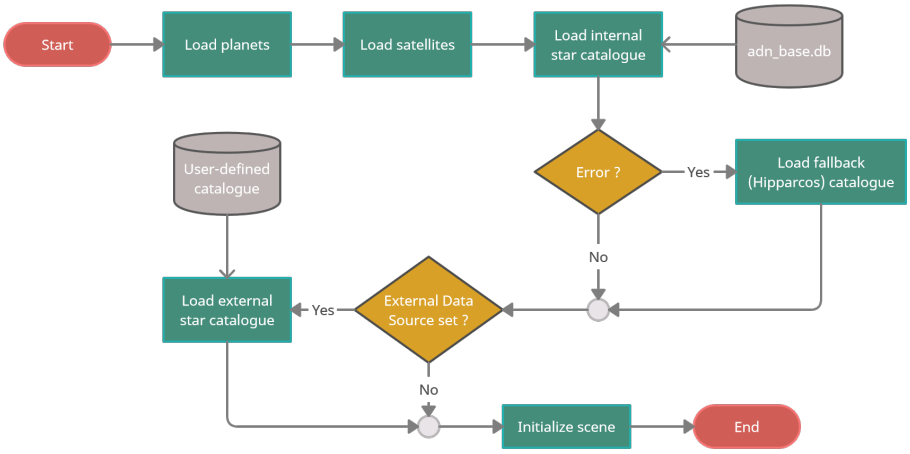


Figure 3.6: Flow chart of the *AdnLoader.Start* method

A special event, called `OnLoadingTrigger`, is defined and used by the camera script to signal every time that the observer has moved more than 10 parsec away from the previous loading point, with an additional parameter that specifies the

current position. When such event is raised and the application is operating in dynamic loading mode, all loader scripts (including those for external data sources) run their *Load* function using the given position; after completing this task, the *RemoveUnused* method takes care of cleaning up old objects and the scene is updated with the new stars.

The *AdnLoader* script also manages access to the underlying dataset as needed by the *SearchPanel* (described in paragraph 3.9). Using C# `Task` objects, a search command is launched in parallel on all catalogues; each loader will return an `HashSet` with the objects that matched the provided input, which are then merged together using the *UnionWith* function and finally returned to the caller.

3.5 Celestial objects management

Once they are loaded, all *AdnCelestialObject* instances (including stars, planets and satellites) are stored in the *AdnCelestialMap* data structure, while the task of initializing and updating of the scene is delegated to the *AdnUniverseManager* script. These components will be described in the following sections.

3.5.1 AdnCelestialMap

This class implements the application's core data structure, therefore its efficiency is key for many algorithms to perform well on large datasets. Their computational cost scales linearly with the number of active objects, thus it's critical that operations on *AdnCelestialMap* introduce the smallest possible overhead.

The original implementation of this data structure was a simple C# `Dictionary` of `<String, AdnCelestialObject>` pairs, which used object identifiers as the key and had a $\mathcal{O}(1)$ complexity when accessing items by their ID.

This was improved by a new design (figure 3.7), which combines a dictionary and a many C# `Lists` (which are just dynamic arrays), one for each celestial object type. Items are stored in the appropriate `List` in an unordered fashion, while the `Dictionary` maps each ID to a position inside of the corresponding `List`. The resulting data structure has the following characteristics:

- $\mathcal{O}(1)$ **insertion** of new items, in both the `Dictionary` and at the end of the appropriate `List`.
- $\mathcal{O}(1)$ **access by ID**, first looking up the identifier in the `Dictionary` and then using the result to access the object in the actual `List`.
- $\mathcal{O}(1)$ **existence check**, which is useful for dynamic loading as well as handling potential conflicts if two objects (coming from different catalogues) have the same ID. In that case, only the first one is kept while the other is ignored, but the user is notified with a warning message.

- $\mathcal{O}(1)$ **removal**, which would normally be $\mathcal{O}(n)$ using `List`, is possible thanks to the lack of item ordering. The object that has to be removed can be swapped with the last element of the collection, and therefore its deletion will not cause any shifting of following items.
- **Access to all objects of a single type** without having to iterate through the entire map and perform type casts, because *AdnCelestialMap* can return a read-only pointer to the collection of any type of entity (e.g. *AdnStar*). This also allows for $\mathcal{O}(1)$ **indexed access** among objects of a single type.
- **Convenient iteration across all entities** is achieved by providing a custom `IEnumerator` implementation, which is used by `foreach` statements, that moves through the available `Lists` in a sequential way.
- Only **finding objects by partial name is** $\mathcal{O}(n)$, but that's a non-critical operation which doesn't require further optimization.

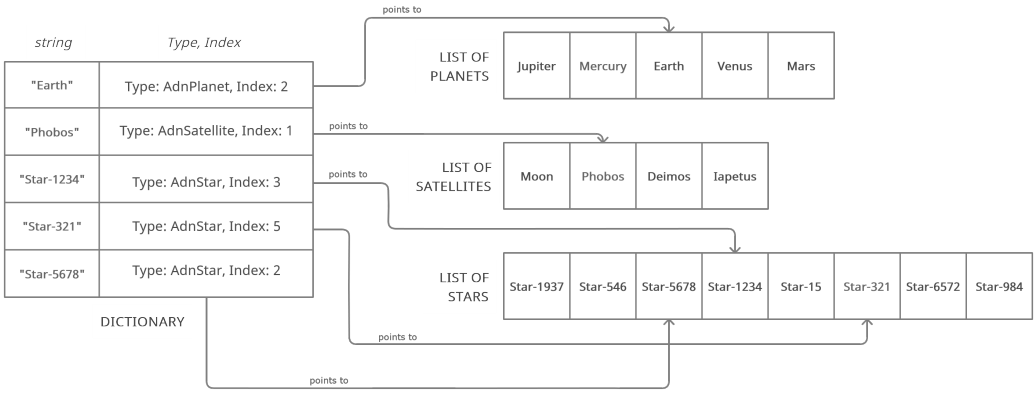


Figure 3.7: Example of the *AdnCelestialMap* data structure

3.5.2 AdnUniverseManager

The scene and all active celestial objects are managed by the *AdnUniverseManager* script, which is a singleton class and contains the *AdnCelestialMap* data structure as a `public static` member.

Scene initialization

Once all loader scripts have finished reading data from the available catalogues, *AdnLoader* calls the *Initialize* function that takes care of spawning objects in the scene. In particular, the *SpawnAll* function iterates over all objects in the celestial map, calls Unity's *Instantiate* function to spawn their `Prefab` in the scene and sets a few parameters such as the position, size and name of the new entity.

While *AdnPlanet* and *AdnSatellite* entities are always created at the beginning, instantiating a `GameObject` for every star in the catalogue is not feasible due to resource and performance limitations. Therefore, stars are rendered in a different way (see paragraph 3.6) and only those that will be interactable (which is a small subset of the total) are spawned as actual objects in the scene. The criterion used to determine if a star should be interactable or not is implemented in the *IsReachable* function, which checks if the ratio between the object size and its distance from the camera is greater than a fixed threshold. Since calculating the euclidean distance between two points requires computing a square root, which is often a slow operation even on modern processors, one optimization exploited by *IsReachable* is using the squared distance to save some CPU time without compromising the results (the radius and threshold values are squared as well).

Scene update

The subset of reachable stars has to be periodically updated in order to provide a useful interaction with the virtual universe. Similarly to how dynamic loading works, this process is also based on a `OnStarsUpdateTrigger` event raised by the camera script to signal every time that the observer has moved more than 1.5 parsec away from the previous position. *AdnUniverseManager* provides an handler for this event, called *UpdateNearbyStars*, which iterate on every star in the working set to determine which ones are reachable from the current camera position and takes care of instantiating their `GameObjects` in the scene.

Since this operation may become slow when there are millions of items in *AdnCelestialMap*, the algorithm takes advantage of object pooling (see section 3.7) and multi-threading (see section 3.8) to reduce the runtime cost of this operation.

3.5.3 Selectable objects

`GameObjects` instantiated according to the previous logic must provide a way for the user to conveniently move between celestial objects in the scene and interact with them. This is done by using the *SgtFloatingTarget* component, that allows to select a star, planet or satellite by clicking on it and activates the *SgtFloatingWarpPin* UI element, which can be used to see the object's name and warp to it using a button.

All interactable entities have a *Selector* attached to them, that can be globally toggled on or off by the user and helps to distinguish which objects are reachable from the current position. A selector is implemented using a `SpriteRenderer` component and a *SgtFastBillboard* script that rotates it to always be facing towards the camera; selector objects are also scaled according to their distance from the observer so that they all appear of the same size (figure 3.8).

Figure 3.8: Screenshot of star selectors and *AdnFloatingWarpPin*

3.6 Star visuals

A lot of effort has been put towards making both far-away and close-up stars to look as realistic as possible, while keeping an eye out for performance.

3.6.1 Rendering stars

In typical 3D applications or videogames, a starry sky would be rendered using a "skybox", which is a simple 6-sided textured cube drawn behind all other graphics. That approach works well when the sky is just a background to the main environment, but in this case it makes up the entire scene where the user can navigate and therefore it must be rendered in full 3D.

Stars are very large objects, but they're also placed in the scene at astronomical distances and therefore will almost never be visible if rendered just as they are. In *ADN* , the actual radius of a star is multiplied by a factor of 450000, altering the apparent size of the object so that it can be visible from further away. With this approach, the radius should be scaled down progressively as the camera gets closer to the object, or otherwise it would be inaccurate. The formula used to determine the size of a star is the following:

```
float radius = 450000f * star.Radius;
if (distance < STARFIELD_RESIZE_DISTANCE)
    radius *= (distance / STARFIELD_RESIZE_DISTANCE);
```

where `STARFIELD_RESIZE_DISTANCE` is set to 50 parsecs.

Several techniques were tried for rendering a large number of stars in the virtual universe, trying to get to the required level of performance. The following list describes the steps which led to the currently adopted solution:

1. The application originally used Unity's **Particle System** to draw stars in the universe. This solution worked for a smaller catalogue (Hipparcos) but had noticeable performance issues when scaling to larger amounts of entities; additionally, particles cannot be individually resized from code without rebuilding the entire *Particle System*, which slowed down the navigation.

2. Creating a `GameObject` instance for each star is not really possible (as discussed in paragraph 3.5.2), but in recent years Unity has introduced a new way of dealing with large amounts objects in a scene: the **Entity Component System (ECS)**. This system is based on the principles of Data-Oriented Design [34] and aims at taking advantage of modern multi-core CPUs and their caches to deliver greater performance when dealing with lots of objects with similar behaviour. An attempt was made at rendering all stars with full 3D models using this system: performance was much better compared to the `GameObject` approach, but it still wasn't good enough because with the Hipparcos catalogue the framerate was only around 40 FPS.

3. The *Space Graphics Toolkit* asset provides a few components to render groups of stars in 3D space, using camera-facing billboards. Most of them use procedural generation algorithms to create starfields based on user-defined parameters, but the ***SgtStarfieldCustom*** component operates by rendering a set of user-provided *SgtStarfieldStar* objects which define the position, radius and color of each star. This component is very efficient at rendering large star catalogues, but it has 2 main limitations:

   - It only works with *SgtStarfieldStar* objects, which means that the entire collection of stars must be duplicated, wasting a lot of memory, especially because the class has several fields not used by *ADN* .
   - Internally, this components builds a mesh with a lot of *Quads* (one for each star) on which it maps the billboard texture; this mesh is then submitted every frame to the GPU for rendering. Whenever a single piece of the *Starfield* is changed, though, this mesh has to be rebuilt and the process may take up to a few seconds when dealing with millions of stars.

4. The current solution is implemented by the ***AdnStarfield*** **script and a custom shader**, inspired by the *SGT* solution but heavily improved for the scope of this application. More details are described in the following sections.

AdnStarfield

AdnStarfield is a customized (and simplified) version of the *SgtStarfieldCustom* component, from which it removes all of the unnecessary features. It overcomes the main limitations of the *SGT* solution by operating directly on the collection of *AdnStar* objects contained in *AdnCelestialMap* (see paragraph 3.5.1) and by employing a different solution for rendering the starfield.

The technique in use is called **procedural draw call** and consists in generating vertices inside the shader instead of receiving a pre-determined mesh object. In this case, the *AdnStarfield* script fills a *ComputeBuffer* with `structs` that hold the position, radius and color of each star, and then calls the *Graphics.DrawProcedural* function to submit a procedural draw call to the GPU. Inside the vertex stage of the graphics pipeline, the shader reads data from the *ComputeBuffer* and outputs 6 vertices (2 triangles that make up a quad) for each star (figure 3.9). This method has two advantages:

- The CPU only needs to pack data in the *ComputeBuffer* instead of building the whole mesh, which reduces the amount of data transferred to the GPU by a factor of 6 and makes draw calls faster. This task is executed only at startup or when the active stars change (e.g. with dynamic loading), but can be easily accelerated using multi-threading (see paragraph 3.8).
- The size of each star is computed on the GPU, by changing the offsets of the 6 generated vertices from the provided (center) position. This means that the resizing algorithm can run every frame with little to no additional cost, based on the camera position which the shader can easily access.

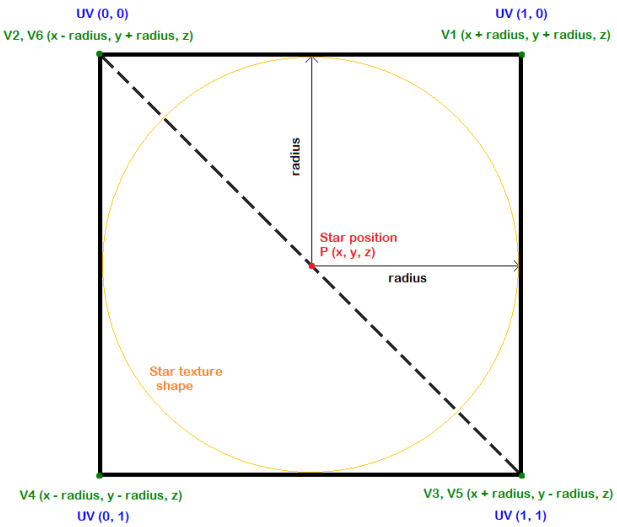


Figure 3.9: Illustration of the quad generation process operated by *AdnStarfield*'s vertex shader

Rendering precision

The approach of generating vertices on the GPU with a procedural draw call had one issue: numeric precision. Stars are not part of the Unity scene in the form of *GameObjects* and therefore do not benefit from the Floating Origin technique provided by the *SGT* asset, instead their position always refers to the center of the reference system, which in our case is the Sun; this results in a loss of rendering precision for stars that are further away from the center of the scene. This issue was accentuated by the fact that shader programs can only operate using single-precision (32 bit) `float` values, which caused flickering and jittering to star billboards when the camera was getting closer to them.

A good solution to this problem was found in a blog post [35] by graphics developer Philip Rideout, who suggests a way to emulate double-precision math using pairs of 32 bit `float` values and doing a few extra calculations to minimize errors introduced by arithmetic operations (see vertex shader code in Appendix A.2). While this isn't exactly on par with 64 bit math, it is good enough to drastically reduce the severity of the issue and make it just barely noticeable for extremely distant stars.

3.6.2 3D star models

When the camera gets close enough to a star, a 3D model will be spawned in the scene to represent that object (figure 3.10). The base model of a star is just a Unity *Sphere*, but with additional components provided by the *SGT* asset that allow to create a few visual effects (figure 3.11). In particular, the surface is rendered using the *StarSurface* animated material, whose color is set in code based on the actual star temperature (through scientific calculations and the use of a look-up table). The stellar corona and prominences are recreated using the *SgtCorona* and *SgtProminence* components, placed on children `GameObjects`, colored with a variant of the surface color whose saturation has been increased by a factor of 3. Additionally, the star model has a *Light* component attached to a children `GameObject`, set to be an omni-directional point light and marked as a *SgtFloatingLight* to work properly with *SGT* 's floating origin system.

The script attached to the star `GameObject`, in charge of handling the materials and setting up visual effects with the correct parameters is called *AdnStarRenderer*. Since multiple 3D stars cannot be visible at the same time (due to how far they are from each other), there is almost always just a single model in the scene, which gets reused through object pooling (see section 3.7) by updating the parameters of *AdnStarRenderer* to match the characteristics of the star that is being represented.

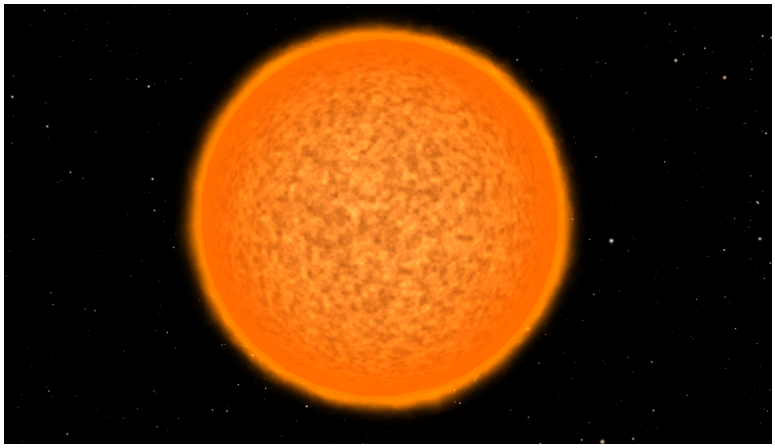


Figure 3.10: The 3D model of a star in *ADN*

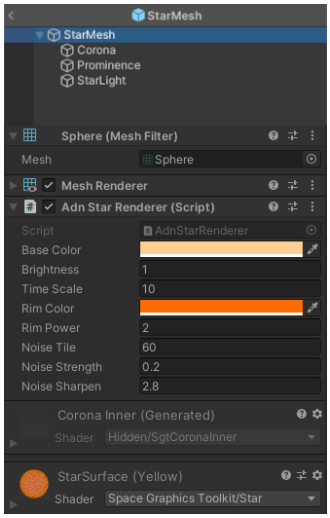


Figure 3.11: Editor settings of the *StarMesh* object

3.7 Object pooling

As discussed in a few of the previous points, creating a `GameObject` for every entity in the scene would not be feasible when dealing with such large amounts of objects, due to memory limitations and performance reasons. The easiest way to overcome this would be to instantiate only a subset of the data which is "relevant" and then updating the scene over time by creating new objects and destroying older ones as they become unused. The issue with this approach is that, in a managed programming language like C#, creating and destroying objects means that a lot of heap allocations and garbage collection will happen. These are slow operations that may heavily impact the application's framerate and responsiveness, and should therefore be avoided as much as possible.

A solution to this problem is given by the **object pooling** pattern and consists in keeping unused objects in a separate "pool" instead of destroying them, to be reused later on. In *ADN* this is implemented by keeping a `Stack` of the desired type and then, when an entity has to be removed, it is simply deactivated and added to the collection. The next time a new object of that type has to be instantiated, the item on top of the stack will be popped, re-purposed with new parameters and finally enabled using `SetActive(true)`.

Through this approach it is possible to avoid the majority of memory allocations and garbage collection when instantiating objects. The first set of entities will still need to be created from scratch (because the pool is empty at the beginning) but following allocations will be avoided by re-using objects in the pool. Since activating an object is much faster than calling *Instantiate*, this provides a noticeable performance improvement for operations that frequently create and remove `GameObjects`, such as *UpdateNearbyStars* (figure 3.12).

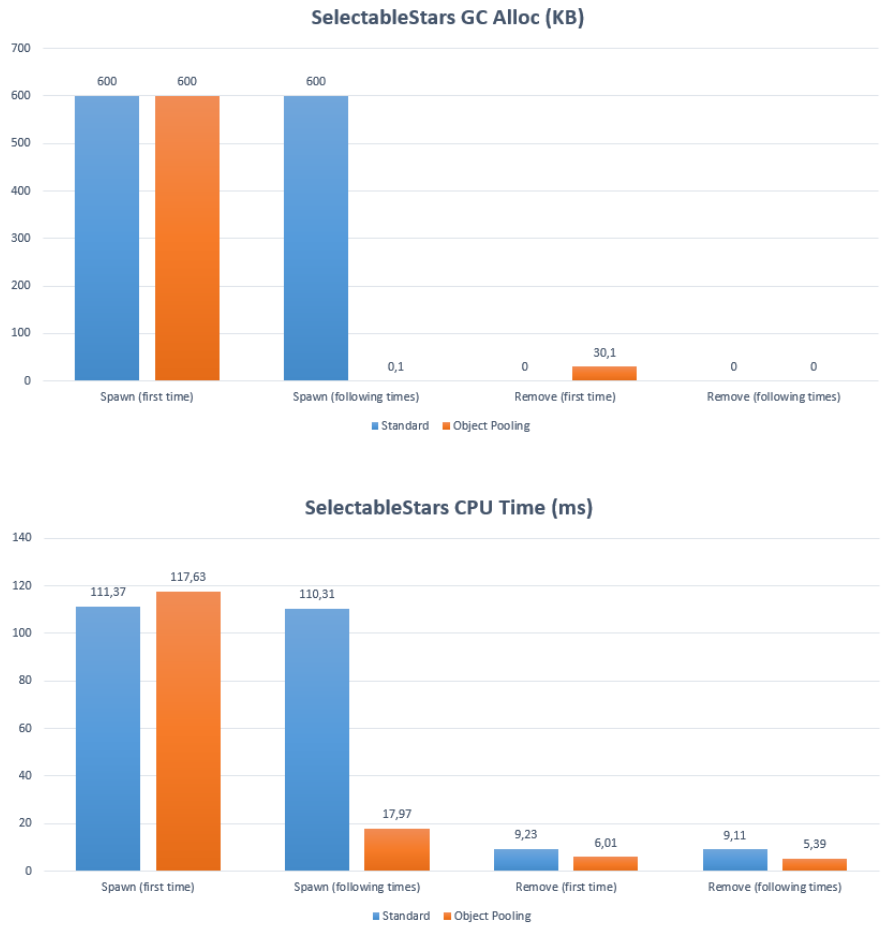


Figure 3.12: Performance and GC allocations related to spawning and removing stars, with and without object pooling

3.8 Multi-threading

Modern CPUs are equipped with many physical and logical processors (cores and threads, respectively) which can execute multiple operations in parallel. In *Astra Data Navigator* , this is exploited by *UpdateNearbyStars* and other functions that need to process large amounts of data (all active stars). Instead of having a single `for` or `foreach` loop to iterate on all objects of interest and performing some operations on them, C#'s `Parallel.ForEach` is used to spread the execution of the algorithm across all available threads. This is achieved by creating a *RangePartitioner* object to subdivide the data array in a specified number of chunks (in this case, equals to `Environment.ProcessorCount`) and then passing it to the `Parallel.ForEach` construct, that will take care of processing each partition of the data on a separate thread (see Appendix A.1).

One limitation of Unity is that all of the core engine API calls are not thread-safe, therefore operations like spawning and destroying a `GameObject` can only be executed from the main thread. In the case of *UpdateNearbyStars*, which needs to remove unused objects and spawn new selectors in the scene, this problem has been solved by accumulating all *AdnStar* objects classified as "reachable" into a thread-safe `ConcurrentQueue` collection and then processing these items on the main thread, after the `Parallel.ForEach` loop has finished, to spawn the required objects in the scene.

While not changing the asymptotic complexity of these algorithms (which is always $\mathcal{O}(n)$), spreading the workload across multiple threads speeds up execution times by a factor of 10 (figure 3.13), thanks to the parallelism provided by the 20-core CPU on the target machine.

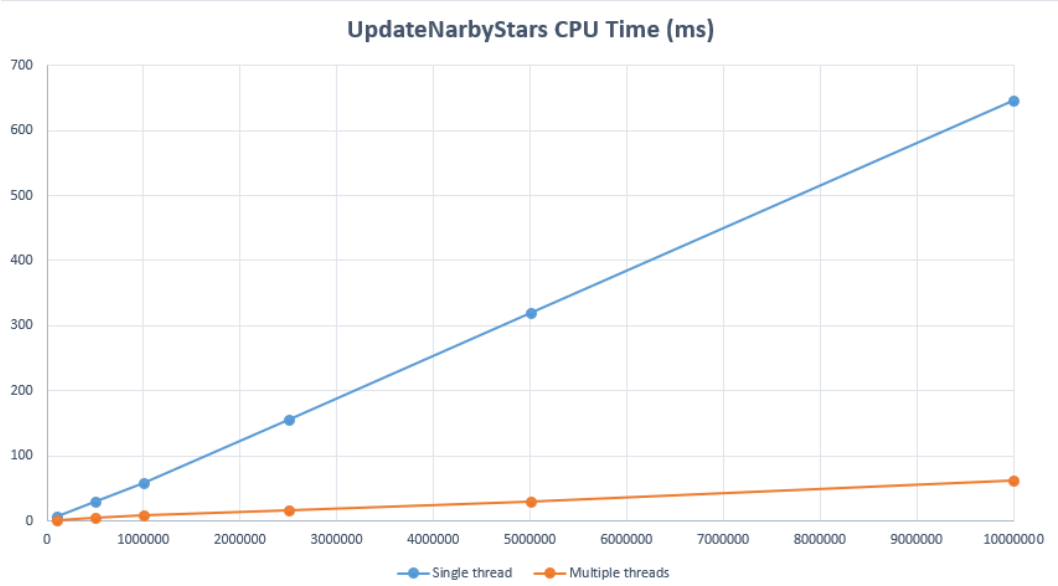


Figure 3.13: Comparison graph between single and multi-threaded performance

3.9 User Interface

The Graphical User Interface (GUI) of *Astra Data Navigator* (figure 3.14) has been kept similar to the older version of the application, in order to maintain consistency. Nevertheless, several changes and additions were made to support the new features and improve the experience:

- Because the application startup could take up to 30 seconds, a **loading screen** is shown on top of the UI while the application loads data from catalogues using a background `Task`. It is made up of an *Image* component, an animated (rotating) loading icon, a *Text* object used to display the current status (provided by *AdnLoader*) and a *LoadingScreen* script which fades out the image when the scene is fully loaded, using a Unity *Coroutine*.

- A **dynamic loading screen** is shown to the user when *ADN* is populating the virtual universe with new stars as the observer moves in the scene, which may take a few seconds. This element is a semi-transparent *Panel*, with a static loading icon and *Text* in the bottom left, that is enabled and disabled by *AdnLoader* when needed. Because Unity updates the UI at the end of the current frame, the dynamic loading function must be delayed by 1 frame or it would freeze the application before the *Panel* is shown: this is achieved using the `yield return null` instruction inside of a *Coroutine*.

- The existing ***SearchPanel*** and the search function itself have been modified to work with much larger datasets. Instead of updating results every time the user types a new character in the *InputField*, the search is now launched with a *Button* and is executed as a background `Task` on all available catalogues in parallel. Since this operation may take a few seconds, an animated (rotating) icon is shown in the search panel while the tasks are running, and disabled when results are available.

- A new ***InfoPanel*** is shown every time a celestial object in the scene is selected, to show all the available data about that object. To collect the required information, this script executes a query on the internal catalogue to load additional fields that are not contained in the *AdnStar* class (to save memory), such as the available cross-matching IDs. Other values are not directly stored anywhere but can be derived from existing parameters (e.g. stellar classification and B-V Index can be extrapolated from the temperature). This panel is highly modular because it spawns each info entry as a separate *Text* object, therefore it will be easy to modify and extend in the future.

- The application has a custom logging system implemented by the *AdnLogger* class, used to record errors and events to a `logs.txt` file. At runtime, the ***LogPanel*** UI element (which can be enabled and disabled using a *Button* on the sidebar) shows these warning and log messages to the user. The panel has

a *ScrollRect* component which contains the log entries and has a *Scrollbar* to navigate between them. Since each log entry is a separate `GameObject`, having too many of them might impact the application's performance and therefore only the most recent ones are kept in the panel (history size can be set in the configuration file, using the *UILogMaxLines* field).

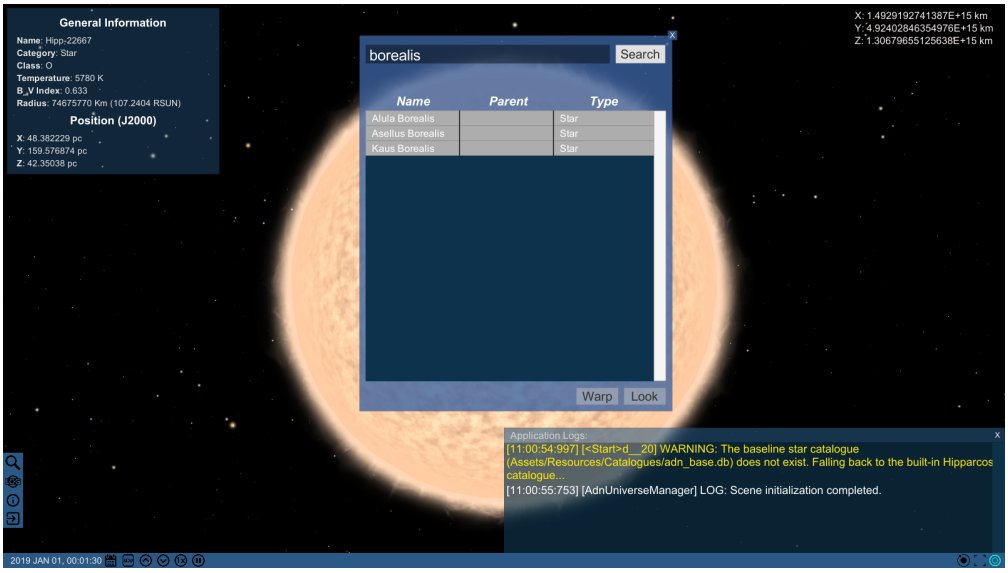


Figure 3.14: Screenshot of *Astra Data Navigator* GUI, showing the *InfoPanel*, *LogPanel* and *SearchPanel* elements

3.10 Stereoscopy

Support for stereo 3D view in *Astra Data Navigator* has been implemented using Unity's XR Management plugin, which can be installed directly from the editor, and setting it up to work with the Windows Mixed Reality platform (see figure 3.15). When using this plugin, no additional configuration is required inside Unity and the application immediately runs in stereoscopic mode when a compatible display is detected.

The stereo separation, which determines how different the left and right-eye images are, can be tuned from the graphics driver (in this case, *NVIDIA Control Panel*). This value has been lowered to 10%, because the default setting of 15% was found to cause too much strain on the viewer's eyes and also because it would cut off part of the UI on the edges of the screen.

With the default configuration, the stereoscopic effect increases the depth of the scene and makes objects appear further than they actually are, with the GUI "floating" on top. The opposite effect, with objects appearing closer to the viewer,

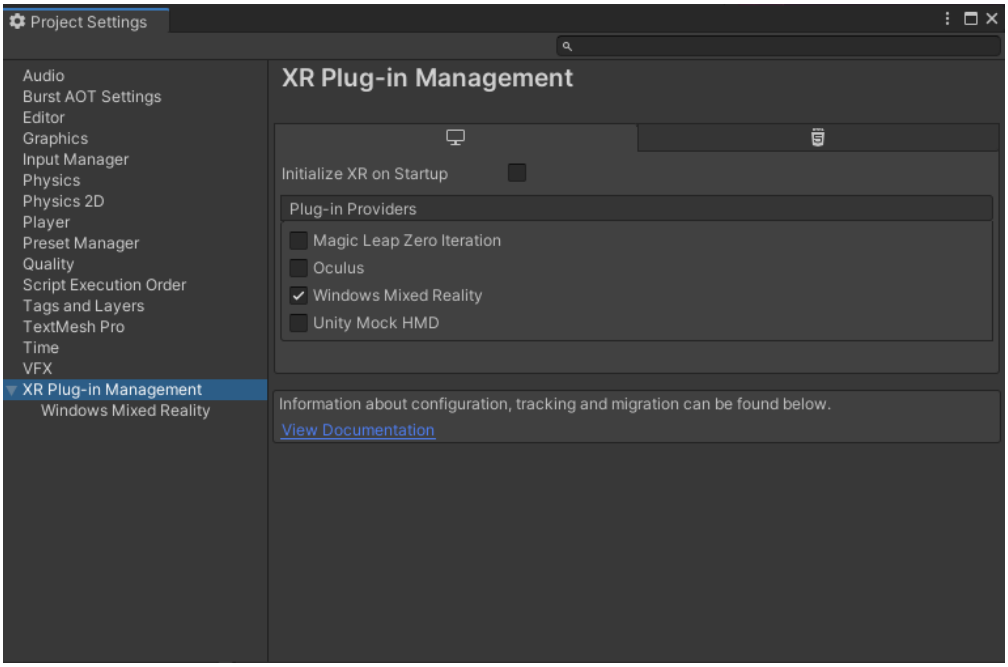


Figure 3.15: XR Plug-in Management settings inside Unity

can be achieved by enabling the *Stereo - Swap eyes* option in the graphics driver settings. The first configuration was chosen because, after trying both versions, the latter was causing too much fatigue to the user and the nature of the content (which doesn't offer many depth cues) meant that the 3D effect wasn't very pronounced.

Chapter 4

Results and Analysis

4.1 Performance analysis

Performance characteristics of the developed solution play a very important role in determining how close the application has come to the original project goals. Low framerates, excessive memory usage and slow loading times should be avoided in order to fulfill the requirements of scalability to very large datasets and smooth navigation inside the virtual universe.

Since this work was a modification over an existing solution, section 4.1.1 compares the final results with the starting point of the thesis to quantify the performance improvements. Then, in section 4.1.2, the static and dynamic loading methods are tested to see how well *ADN* can scale up to larger catalogues.

4.1.1 Comparison with the previous version

When analyzing the performance of real-time VR applications, **framerate** is an important metric to consider, because it indicates how much time is taken by the CPU and GPU to complete the work of a single frame, and defines how smooth the application will feel for the user. For a good VR experience that feels responsive and doesn't cause any sickness, the framerate should be at least 60 FPS.

The use of a particle system to render stars in the old version of the application came with a big performance hit, which became worse when more stars were added to the simulation. The new shader-based solution (see paragraph 3.6.1), instead, is extremely lightweight and its cost barely increases with the number of stars.
Figure 4.1 compares the framerate of the original application running with the Hipparcos catalogue (113,000 objects) and the new version of *Astra Data Navigator*, operating with 5 million stars. It can be seen how the render thread on the CPU now does very little work, because the majority of the calculations for star rendering are done on the GPU with a much greater parallelism and speed.

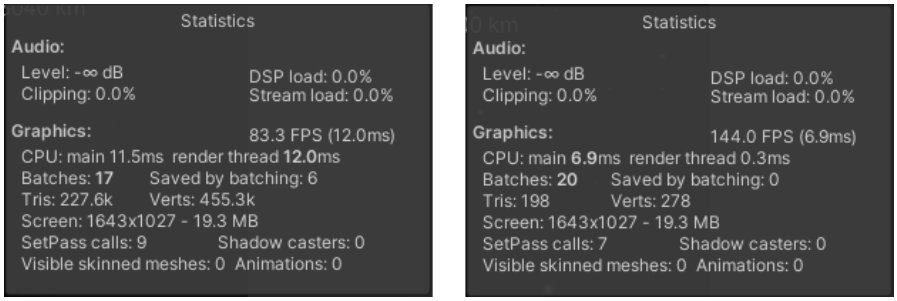


Figure 4.1: Performance comparison between the original application (left) and the new version of *ADN* (right)

As mentioned in paragraph 1.4, the free movement navigation mode was previously afflicted by poor performance and noticeable slowdowns, which made it almost unusable. These issues are solved in the new version of *ADN* thanks to the combination of object pooling, multi-threaded update algorithms and the new starfield shader.

When measuring the application's capabilities of handling large star catalogues, **memory usage and loading times** are also very important to consider. Both of these metrics have been significantly improved thanks to data structures optimization, careful selection of the star catalogue parameters, improvements made to the loader scripts and, most importantly, the use of a database as the main application data source. The graphs in figure 4.2 show the difference in loading times and memory footprint between the old and the new version of *ADN* (using the static loader), when provided with a catalogue of 1 million stars.

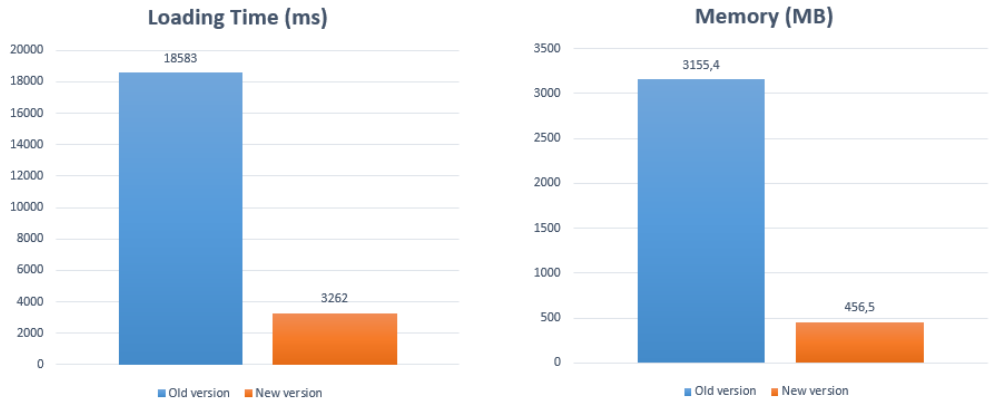


Figure 4.2: Startup time (left) and memory usage (right) of *Astra Data Navigator* when loading 1 million stars

4.1.2 Scalability of static and dynamic loaders

The improvements in loading performance compared to the previous version of *Astra Data Navigator* are clear and already allow for a noticeable increase in the amount of stars that can be handled by the application (24x just with the baseline catalogue), but it would also be useful to compare the two loading methods - Static and Dynamic - against each other to see how they behave when scaling up to very large datasets.

Figure 4.3 plots the loading time and RAM usage of the application running in both modes, with catalogues that go from 100,000 entries to a maximum of 10 million stars. The static loader graph plateaus after the 5 million mark because it has reached the maximum amount of objects that can be loaded at runtime, whereas the dynamic loader can scale up to catalogues of any size but the time required by the DBMS to execute the selection query increases linearly with the number of entries in the dataset (and it should reach that of the static loader at around 20 million stars).

Figure 4.3: Time and memory usage analysis for Static and Dynamic loading methods

The memory usage graph highlights the advantage of the dynamic loading method, which is able to select only a small subset of relevant stars out of all those contained in the catalogue, resulting in a much smaller memory footprint even if the actual dataset is larger. In any case, the 5 million limit to the number of active stars guarantees that the application will never use more than 1.5 GBs of RAM.

It must be noted, though, that the loading time reported for the Dynamic method will not only affect the application startup, but will also happen every time that the user moves by over 10 parsecs in the scene, causing a much more fragmented and discontinuous experience, filled with loading screens.

Therefore, even if this method would technically allow to connect the entire Gaia catalogue to *Astra Data Navigator* without saturating the system resources, loading times experienced by the user when starting the application and navigating through the virtual universe would be extremely long (estimated over 30 minutes) and totally impractical.

4.2 Issues

Several issues were encountered during the development of *Astra Data Navigator*, some of which have not been resolved yet. The following list contains an analysis of the most relevant problems still present in the current version of the application:

- **Dynamic loading times:** the main goal of this project was to introduce support for large star catalogues id *ADN* , and eventually being able to navigate the entire 1.8 billion Gaia stars in real-time 3D. As demonstrated by paragraphs 4.1.1 and 4.1.2, this work achieved substantial improvements compared to the original version of *Astra Data Navigator* , but not quite enough to actually support the full Gaia catalogue. The dynamic loader supports large datasets but has to periodically update the scene with new objects read from the catalogue, which means rebuilding the entire *AdnStarfield* even if just a single star is added or removed from the working set, causing continuous interruptions to the user experience and preventing a seamless navigation of the virtual universe.

  Solving this issue would require a new architecture (e.g. splitting the universe in an octree-like structure and having an independent starfield instance for each cell) and possibly even moving away from relational databases in favor of some custom compressed binary format, which arranges the data in such a way that is faster to access at runtime.

  Since the implementation these techniques is a complex task that would take a lot of time, for the current version of *ADN* the choice has been to tolerate this limitation and simply show a loading screen every time that the scene has to be dynamically updated, to inform the user about what is happening instead of just freezing the application.

- **Star representation:** the components used to create the 3D model of a star and its visual effects (see paragraph 3.6.2) offer many tweakable parameters to define the object's appearance. Since *ADN* is aiming to achieve realistic visuals, aspects like the number and length of star prominences, the brightness of the stellar corona and the animation parameters of the surface shader should be tied to physical properties of the celestial object. This is not implemented in the current version of the application, for two reasons:

  1. It would require the consulting of an astrophysics expert, in order to find formulas that correctly map star parameters to realistic visual effects of the 3D objects.
  2. Storing all these additional values (for each entry) in the *AdnStar* class and in the catalogue would drastically increase the memory and storage requirements of the application.

  Currently, the only scientifically-accurate aspects of a star's 3D representation are its position, size and color (which is derived from the effective temperature). Every other parameter has been hand-picked to achieve a convincing appearance but is fixed for any star in the virtual universe, which causes many objects to look almost identical and may take away from the realism of the representation.

- **Stereoscopic effect:** the type of scene rendered by *ADN* doesn't have the optimal characteristics to achieve a good stereo 3D effect. This is due to a couple of reasons:

  1. There is no actual background other than deep space, which is only populated by very small and far-away stars that do not provide meaningful depth cues to create the perception of 3D.
  2. There is no real foreground, because even objects that occupy the entire screen are always many thousands of kilometers away from the camera. This means that Unity considers them as part of the background, which will make these objects appear further away from the observer instead of "coming out of the display", as one would expect from a stereoscopic application.

  These issues are tied to the nature of the content and cannot be easily resolved; the only component that actually benefits from the stereoscopic rendering is the UI, which is drawn in the foreground and therefore appears to be "floating" in front of the display. Some elements of the User Interface that are close to the left and right edges of the screen, though, are partially cut off by the camera and might cause some trouble to the viewers.

  Solving this issue would require to rework the entire UI layout, moving buttons and panels away from the edges of the window and maybe allowing the user to position each panel manually according to his/her preferences.

4.3 Subjective evaluation tests

After the development had concluded, a group of 12 people was asked to test this new version of *Astra Data Navigator* in order to obtain a subjective quality assessment of the application and receive suggestions for future improvements.

The test group was made up of 7 male and 5 female subjects, with an age between 24 and 59 years, selected to have a wide variety of skills and different professional backgrounds. Among these people there were a wearers of glasses, which is an important factor because it may cause additional fatigue when viewing stereoscopic content; 11 out of 12 people declared to have already had at least one Virtual Reality experience.

The subjects were presented with a survey (see Appendix A.3) that guided the testing of the application and was designed to measure the overall system usability, intuitiveness of the UI, realism of the representation and perceived performance characteristics.

The questionnaire consists of three parts: the first one has to be completed prior to starting the application and collects general information about the test subject, such as age, sex and previous VR experiences; the second part contains 2 tasks which have to be performed by the user, followed by a few questions taken from the NASA TLX tool [36] for assessing the perceived workload of the requested actions and check if the application is causing any nausea or sickness to the user. The first task consists in launching *Astra Data Navigator* with the default configuration and navigating between stars in the virtual universe using the three available modes (click-and-go with selectors, search panel and free camera movement); the second task requires the user to edit the `config.xml` file by changing the loading method from Static to Dynamic, then restart the application and repeat similar actions to those of task 1. These actions are designed to evaluate the ease of exploring a large star catalogue in 3D with the provided navigation tools, but also to see if the XML-based configuration system is intuitive enough and, most importantly, to have all users try both loading so that they can directly compare their performance characteristics. During these tests, the application was set up to use the internal Tycho-2 catalogue plus an external PostgreSQL database with an additional 10 million stars.

After completing each task, the user has to answer 5 questions (mental workload, physical workload, time to complete the task, success in performing the actions, sickness or nausea felt) with a value between 1 and 10. Figure 4.4 reports the average result for each of these questions, showing good and consistent values for both tasks. A very positive result is the little to no sickness experienced by test subjects when using *ADN* (probably due to the weak stereoscopic effect, but also thanks to the choice of not swapping the two images in the NVIDIA driver), while the increase in mental workload and the lower overall self-evaluation score of task 2 can be attributed to the configuration system not being extremely intuitive for non-expert users.

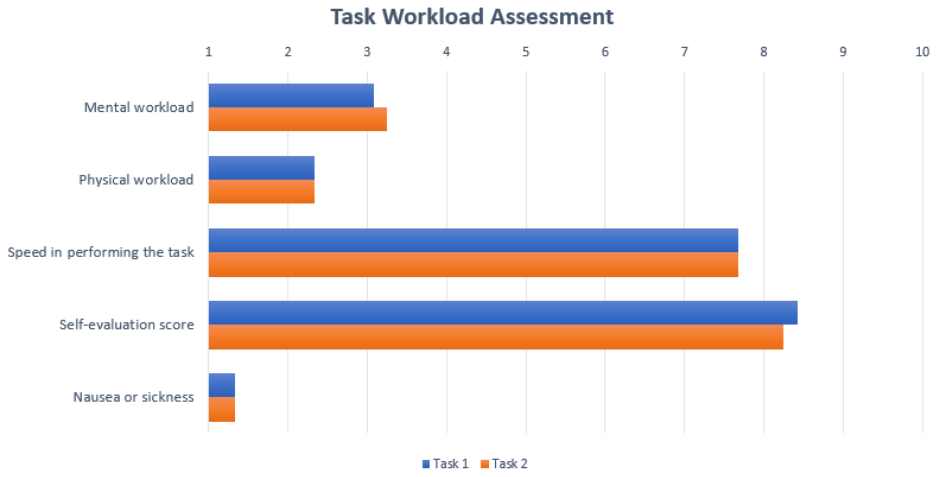


Figure 4.4: Average results for perceived workload, self-evaluation and sickness in both tasks

The third and last section of the survey, which must be compiled after using the application, is designed to assess the intuitiveness, usability, realism and performance (using both loading methods) of the tool when navigating large star catalogues. Each question can be answered with a score on a scale that goes from 1 to 5 (worst to best) to evaluate different aspects of the user experience.

At the end, the user also has the chance to write any improvement that he or she may want to suggest for future development of *Astra Data Navigator* .

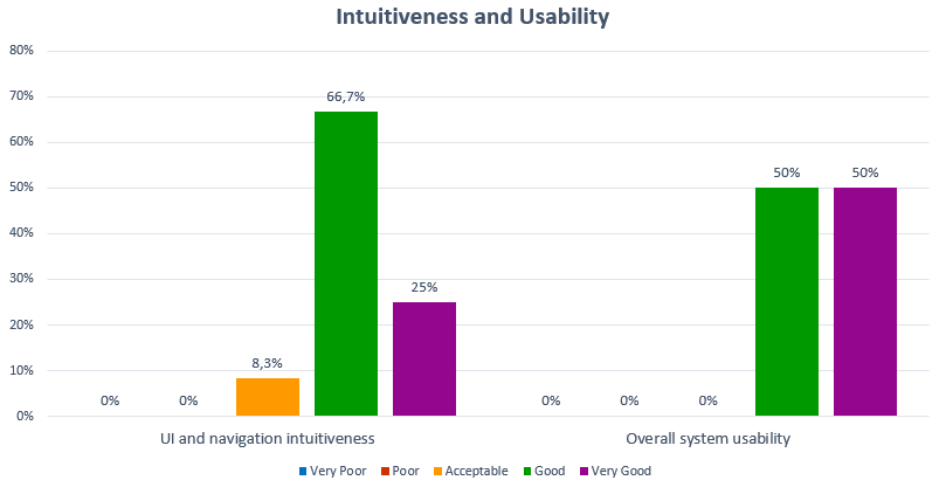


Figure 4.5: User evaluation of interface intuitiveness and system usability

The graph in figure 4.5 illustrates the scores assigned by users to the intuitiveness of the User Interface and the navigation in the virtual universe, as well as a usability score which refers to the application as a whole. Results are mostly positive, with the majority of users (66.7%) considering the UI "Good" and a system usability score that contains only "Good" and "Very Good" answers, equally split. This doesn't mean that the User Interface of *Astra Data Navigator* cannot be improved, as most of the user suggestions focus on aspects of the UI, asking to make the controls bigger and more easily reachable (2 suggestions), to simplify the configuration system (1 suggestion) and add a new panel containing the navigation history in order to go back to previously visited celestial objects (1 suggestion).

When questioned about the perceived performance (framerate, responsiveness) of the application and the realism of 3D stars, users also answered rather positively. As it can be seen in figure 4.6, test subjects were very satisfied with the application's performance (66.7% "Very Good" and 33.3% "Good" answers), while judgements regarding the realism of stars were also mostly positive (over 83% of the users felt that it was "Good" or better) but also included a 16.7% of lower "Acceptable" scores, possibly coming from users with a good understanding of these subjects.

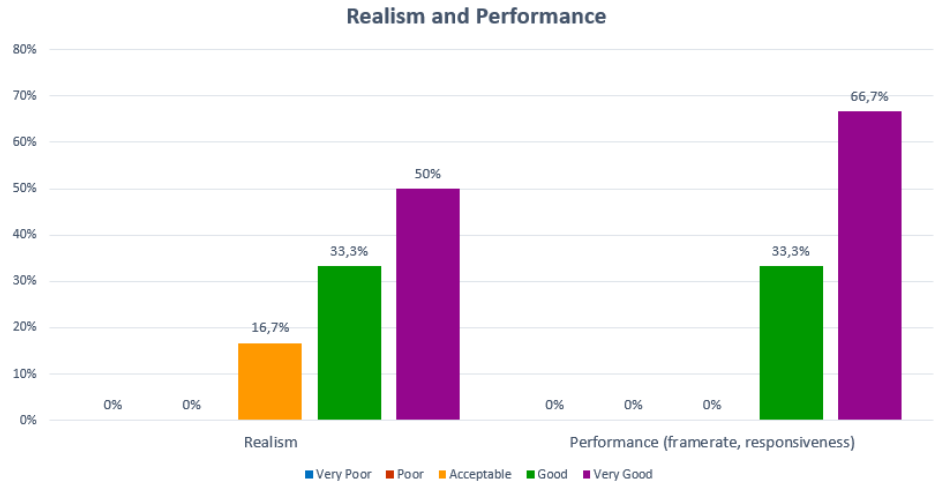


Figure 4.6: Evaluation of the perceived performance and realism of stars in *ADN*

Finally, a few questions required a direct comparison between the Static and Dynamic loading methods, more specifically regarding the application startup time, the amount of visualized data and the experience of navigating the 3D environment. Figure 4.7 highlights a very positive result: the vast majority of the users (91.7%) didn't notice any reduction in the amount of visible stars when switching to the dynamic loading method, which means that the visibility formula in use is very effective at selecting stars which can be seen from the current observer position.

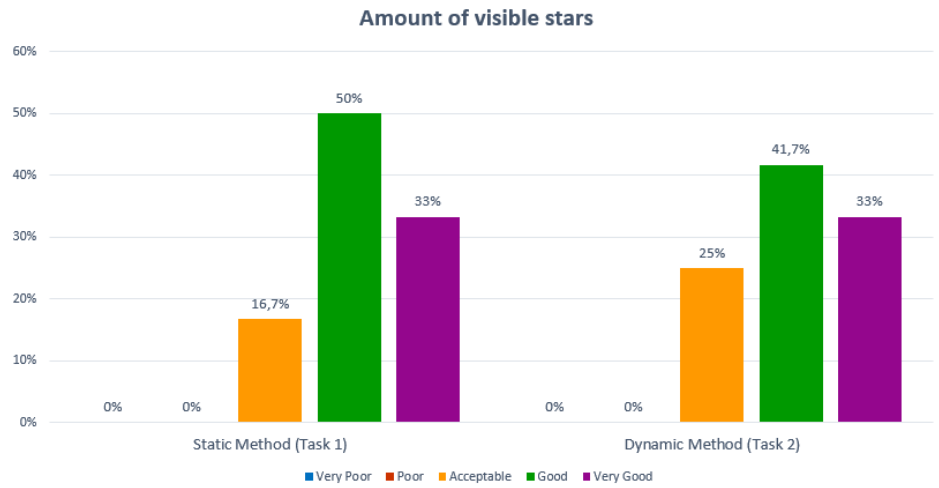


Figure 4.7: Assessment of the number of visible stars in the universe, in both Static and Dynamic mode

When it comes to comparing the loading times of the two modes (figure 4.8), the static loader is considered "Slow" by 16.7% of testers but 50% of them still find it "Fast", while the dynamic loader is "Very Fast" and "Fast" according to 33.3% and 41.7% of answers, respectively.

After the initial loading is done, though, the situation is reversed (figure 4.9): the majority of users (66.7%) find the Static method to allow a "Very Fast" navigation inside the virtual universe, while the Dynamic method is considered "Acceptable" by 41.7% of the subjects, but 16.7% of users find it to be "Slow" and 8.3% of the testers even mark it as "Very Slow".

These results are expected and confirm what has already been discussed in sections 3.4 and 4.1.2: the dynamic loader allows the application to operate with larger datasets requiring less system resources, but the additional loading times interrupt the experience and are not well accepted by users.

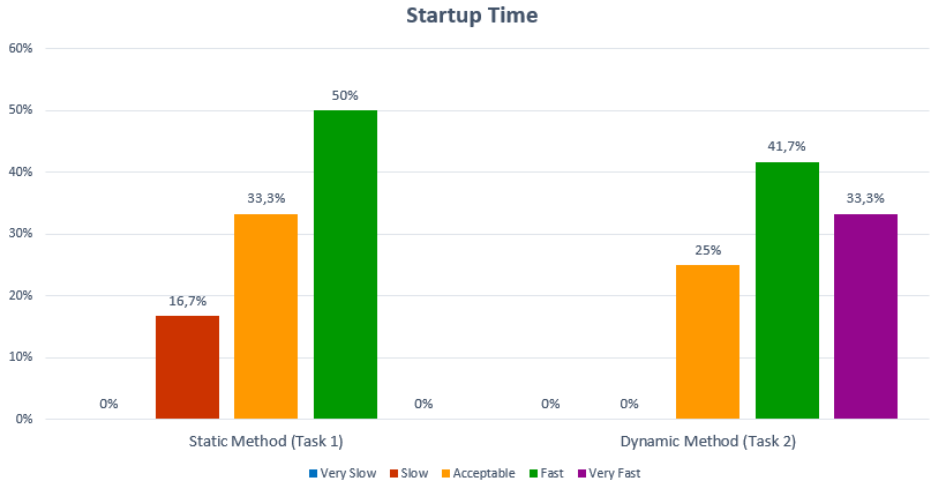


Figure 4.8: User rating of *Astra Data Navigator*'s startup time, with both loading methods

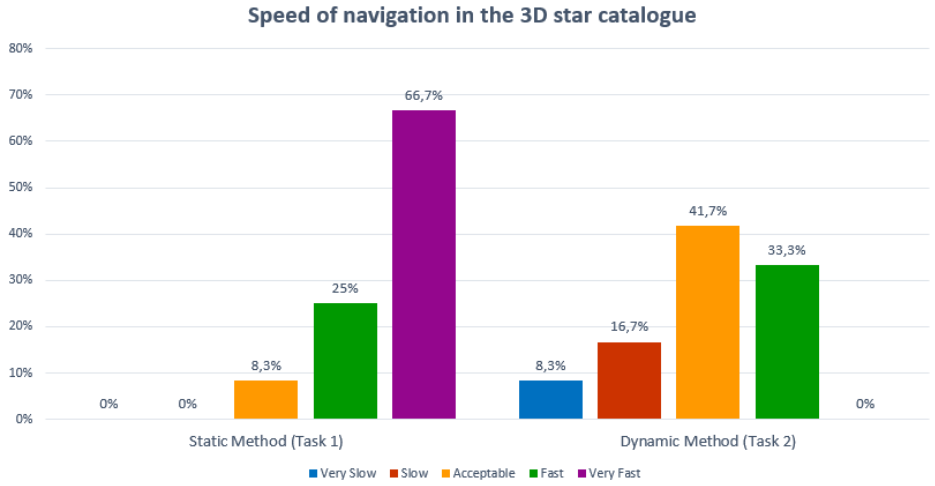


Figure 4.9: Subjective scores evaluating the speed of navigation in the 3D star catalogue in each loading mode

Chapter 5

Conclusions and Future Work

The presented work has successfully led to the development of a real-time solution for loading and visualizing large star catalogues in an interactive VR application. The capabilities of *Astra Data Navigator* have been considerably improved with respect to the original version, delivering a 24 times increase in baseline star catalogue size (with better performance) and introducing support for user-provided data sources in both file and database formats. In its latest version, *ADN* can display larger astronomical datasets than most state-of-the-art VR universe exploration tools, up to several million entries, even though the capabilities of *Gaia Sky* remain currently unmatched.

The analysis and subjective tests conducted on the final version of the application highlighted several positive results with regards to the intuitiveness and performance of this tool as well as the realism of the represented 3D environment, but also uncovered a few shortcomings of the current solution, which could be improved upon by further development.

Most user suggestions focus on the UI, which could benefit from a few usability improvements as well as a general rework to overcome some of the issues related to stereoscopic rendering. In this sense, the choice of keeping the User Interface as consistent as possible with the original version of the application did not pay off as expected. Some areas of the UI could be enhanced in the future, for example:

- Implementing a graphical interface for the configuration system, instead of relying on the user manually editing the `config.xml` file, would make this feature more accessible and less prone to human errors even for people with less IT skills.

- Introducing a new panel, which displays the user's navigation history and allows to quickly go back to previously visited object by double-clicking on their name in the list.

- Panels could be made repositionable, so that the user can drag them anywhere according to personal preference and they would keep that position until the application is closed.
- Considering compatibility with VR headsets and other devices, the User Interface could be moved to exist in world space instead of being drawn as a 2D overlay on the screen.
- Additional input systems, such as 3D mouse or motion tracking, could provide a more natural way to interact with the application and navigate the virtual universe.

Another very useful feature would be a "visual query" system that allows users to select and filter which stars have to be loaded from the catalogue and rendered in the scene, based on some parameters or conditions (e.g. all objects within a certain distance or all stars with a temperature inside the specified range), and then offers the option to save the query results to a FITS or VOTable file.

One major issue, pointed out by performance analyses and the user survey, concerns the additional loading times introduced by the application when operating with the Dynamic method. In order to provide a better user experience and a seamless navigation, the dynamic loading system could be improved by making it work in the background, reading data from the catalogues as the observer moves across the virtual universe and updating the scene in small chunks without freezing the application or requiring any kind of loading screen. To achieve this level of performance, a standard database might not be sufficient but a more complex binary file format could work better, even though it would require an additional pre-processing step to convert user catalogues in the desired form.

The realism and graphical fidelity of celestial objects could also be improved, by adding more parameters to the supported star catalogues and using them to drive the visual representation of 3D objects in a scientifically accurate way. Finally, the introduction of new astronomical objects to the scene, such as galaxies and nebulae, would drastically increase the realism of the virtual environment for both astronomers and non-expert users.

This study showed the potential of using Virtual Reality technologies to support scientific research in the astronomy field, while also providing a valuable tool for educational purposes, thanks to the visualization of large star catalogues in a real-time interactive 3D environment. It would require additional time and resources to perfect the user experience and improve the aspects found lacking in the current solution, but results achieved so far have been very convincing and encourage further development of this application.

Appendix

A.1 UpdateNearbyStars function

```
public void UpdateNearbyStars(SgtPosition cameraPosition)
{
    // Loop through all active stars to see if some can be removed
    AdnFloatingStar[] activeStars = SgtFloatingRoot.FirstInstance.
        GetComponentsInChildren<AdnFloatingStar>(false);

    for (int i = 0; i < activeStars.Length; i++)
    {
        if (!IsReachable(ref cameraPosition, activeStars[i].Star) ||
            !CelestialMap.Contains(activeStars[i].Star))
        {
            DestroyStar(activeStars[i]);
        }
    }

    ReadOnlyCollection<AdnStar> stars = CelestialMap.GetStars();
    ConcurrentQueue<AdnStar> toBeSpawned = new ConcurrentQueue<AdnStar>();

    // Partition the array in chunks based on the number of logical processors
    int partitionCount = Environment.ProcessorCount;
    int partitionSize = (stars.Count / partitionCount) + 1;
    var rangePartitioner = Partitioner.Create(0, stars.Count, partitionSize);

    Parallel.ForEach(rangePartitioner, (range, loopState) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
            if (IsReachable(ref cameraPosition, stars[i]) &&
                !_spawnedStars.Contains(stars[i]))
            {
                toBeSpawned.Enqueue(stars[i]);
            }
        }
    });

    // Spawn all stars that have been found to be reachable
    foreach (AdnStar star in toBeSpawned)
    {
        SpawnStar(star);
    }
}
```

A.2 AdnStarfield component

StarInfo struct

```
[StructLayout(LayoutKind.Sequential)]
readonly struct StarInfo
{
    readonly Vector3 pos_highpart;
    readonly Vector3 pos_lowpart;
    readonly float radius;
    readonly Color color;

    public StarInfo(in (Vector3 highpart, Vector3 lowpart) pos,
                    in float radius, in Color color)
    {
        this.pos_highpart = pos.highpart;
        this.pos_lowpart = pos.lowpart;
        this.radius = radius;
        this.color = color;
    }
}
```

Starfield generation

```
protected void BuildStarfield()
{
    if (stars.Count == 0)
        return;

    var data = new NativeArray<StarInfo>(stars.Count, Allocator.Temp,
        NativeArrayOptions.UninitializedMemory);

    // Spread the workload of compute buffer generation across all processors
    int partitionSize = (stars.Count / Environment.ProcessorCount) + 1;
    var rangePartitioner = Partitioner.Create(0, stars.Count, partitionSize);

    Parallel.ForEach(rangePartitioner, (range, loopState) =>
    {
        for (int i = range.Item1; i < range.Item2; ++i)
        {
            data[i] = new StarInfo(
                AdnUtils.SgtPositionToHighPrecision(ref stars[i].Position),
                450000f * stars[i].Radius, stars[i].Color);
        }
    });

    starsBuffer?.Dispose();
    starsBuffer = new ComputeBuffer(stars.Count, (3 + 3 + 1 + 4) * sizeof(float),
        ComputeBufferType.Default);
    starsBuffer.SetData(data, 0, 0, data.Length);

    // Bind the compute buffer to the starfield shader
    material.SetBuffer("_Stars", starsBuffer);

    data.Dispose();
}
```

Draw call

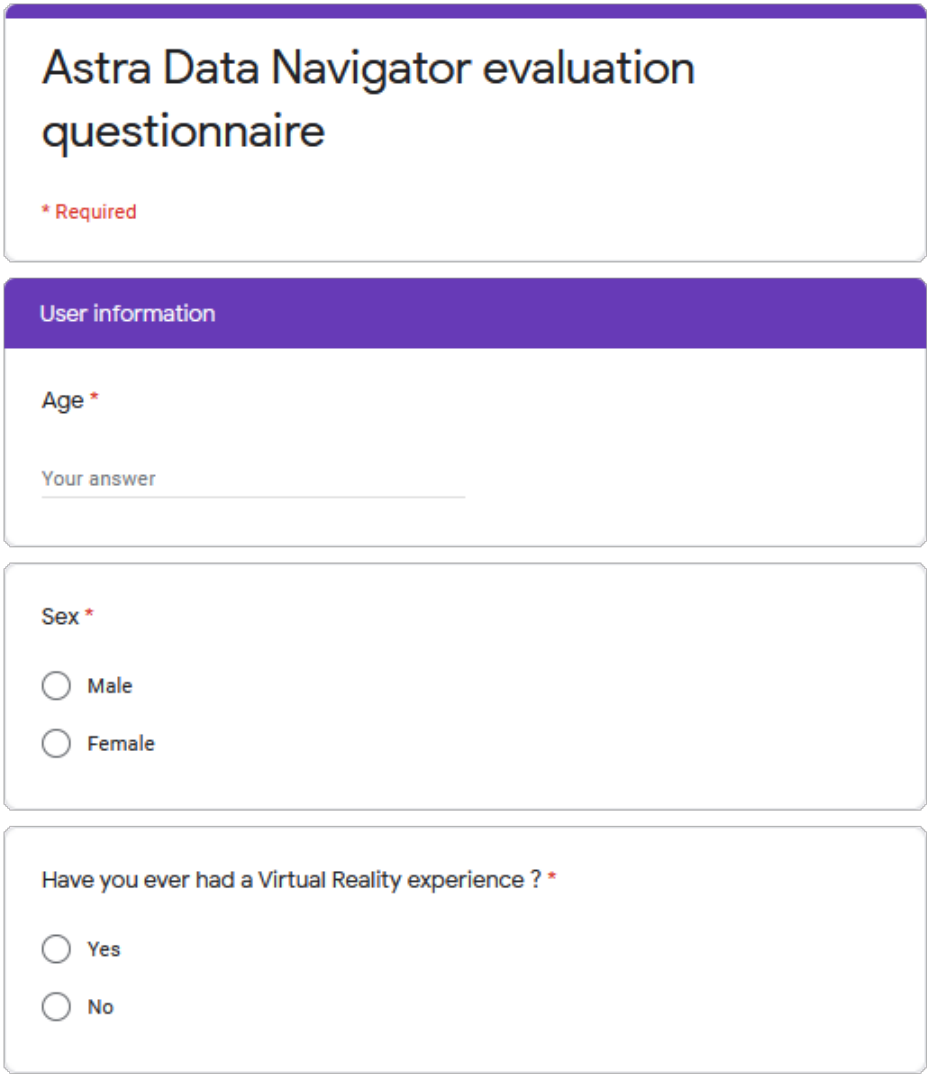
```
protected void HandleCameraDraw(Camera camera)
{
    if (SgtHelper.CanDraw(gameObject, camera) == false)
        return;

    var sgtCamera = default(SgtCamera);
    if (SgtCamera.TryFind(camera, ref sgtCamera) == true)
    {
        properties.SetFloat(SgtShader._CameraRollAngle,
            sgtCamera.RollAngle * Mathf.Deg2Rad);

        // Decompose camera position in 2 floats, for double precision emulation
        SgtPosition position = AdnCamera.Instance.SgtCamera.SnappedPoint;
        (Vector3 highpart, Vector3 lowpart) =
            AdnUtils.SgtPositionToHighPrecision(ref position);

        properties.SetVector("_CameraPositionHighpart", highpart);
        properties.SetVector("_CameraPositionLowpart", lowpart);
    }
    else
    {
        properties.SetFloat(SgtShader._CameraRollAngle, 0.0f);

        properties.SetVector("_CameraPositionHighpart", Vector3.zero);
        properties.SetVector("_CameraPositionLowpart", Vector3.zero);
    }

    // "Infinite" bounds so that the starfield never gets culled
    Bounds bounds = new Bounds(this.transform.position,
        new Vector3(float.MaxValue, float.MaxValue, float.MaxValue));

    // Submit a procedural draw call to the GPU for rendering the star quads
    Graphics.DrawProcedural(material, bounds, MeshTopology.Triangles,
        vertexCount: stars.Count * 6, instanceCount: 1, camera, properties,
        ShadowCastingMode.Off, receiveShadows: false, gameObject.layer);
}
```

Vertex shader

```
#define STARFIELD_RESIZE_DISTANCE 1.5e15f

struct star
{
  float3 position_highpart;
  float3 position_lowpart;
  float radius;
  float4 color;
};

static const float3 _OFFSETS[6] =
{
  float3(-1.0f, 1.0f, 0.0f),
  float3(1.0f, 1.0f, 0.0f),
  float3(-1.0f, -1.0f, 0.0f),
  float3(1.0f, -1.0f, 0.0f),
  float3(-1.0f, -1.0f, 0.0f),
  float3(1.0f, 1.0f, 0.0f)
};
```

```
float3    _CameraPositionHighpart;
float3    _CameraPositionLowpart;

uniform StructuredBuffer<star> _Stars : register(t1);

struct v2f      // Vertex shader outputs
{
  float4 vertex    : SV_POSITION;
  float4 color     : COLOR;
  float2 texcoord0 : TEXCOORD0;
  float3 texcoord1 : TEXCOORD1;

  UNITY_VERTEX_OUTPUT_STEREO
};

void Vert(in uint id: SV_VertexID, out v2f o)
{
  UNITY_INITIALIZE_OUTPUT(v2f, o);
  UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);

  // Calculate quad and vertex numbers from the current index
  uint quadId = id / 6;
  uint vertexId = id % 6;

  // Retrieve the correct star data from the compute buffer
  float3 pos_highpart = _Stars[quadId].position_highpart;
  float3 pos_lowpart = _Stars[quadId].position_lowpart;
  float radius = _Stars[quadId].radius;
  float4 color = _Stars[quadId].color;
  float3 offset = _OFFSETS[vertexId];

  // Vertex positions have to be calculated using double-precision emulation
  float3 t1 = pos_lowpart - _CameraPositionLowpart;
  float3 e = t1 - pos_lowpart;
  float3 t2 = ((-_CameraPositionLowpart - e) + (pos_lowpart - (t1 - e))) +
      pos_highpart - _CameraPositionHighpart;
  float3 high_delta = t1 + t2;
  float3 low_delta = t2 - (high_delta - t1);
  float3 vertex = high_delta + low_delta;

  // Scale the actual radius based on distance
  float dist = length(vertex.xyz - _WorldSpaceCameraPos);
  if (dist < STARFIELD_RESIZE_DISTANCE)
    radius *= (dist / STARFIELD_RESIZE_DISTANCE);

  // Generate the vertex coordinate
  float3 vertexMV = UnityObjectToViewPos(vertex);
  float4 cornerMV = float4(vertexMV, 1.0f);
  // The offset vector is rotated to always match the camera roll angle
  offset.xy = Rotate(offset.xy, _CameraRollAngle);
  cornerMV.xy += offset.xy * radius;

  // Project the vertex in view space, but also pass the actual
  // vertex position to the fragment shader using the UV1 channel
  o.vertex    = mul(UNITY_MATRIX_P, cornerMV);
  o.texcoord1 = cornerMV.xyz;

  // Forward color information and texture mapping to the fragment shader
  o.color     = color;
  o.texcoord0 = _UVS[vertexId];
}
```

A.3 User survey



Astra Data Navigator evaluation questionnaire

* Required

User information

Age *

Your answer

Sex *

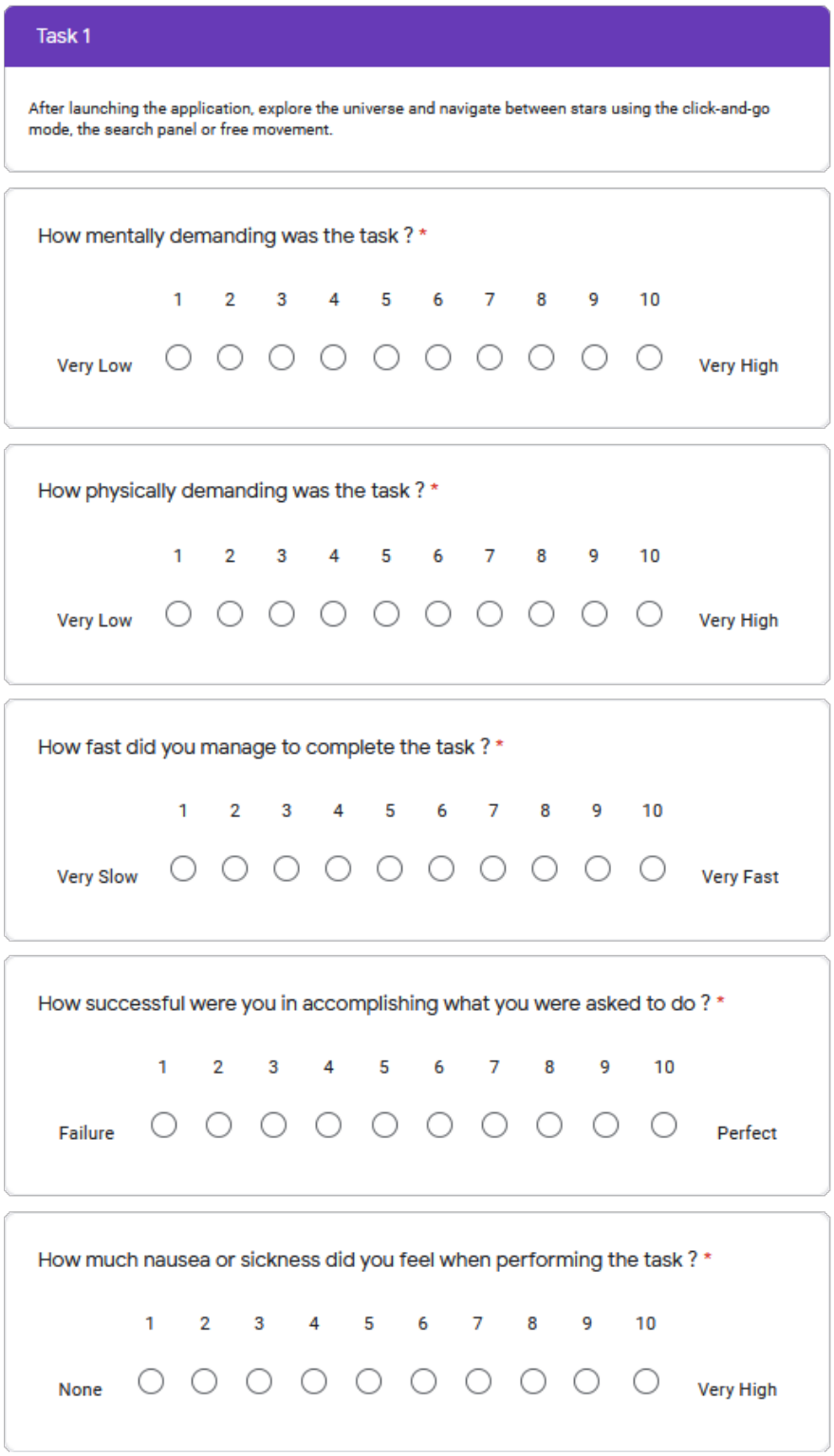
Male

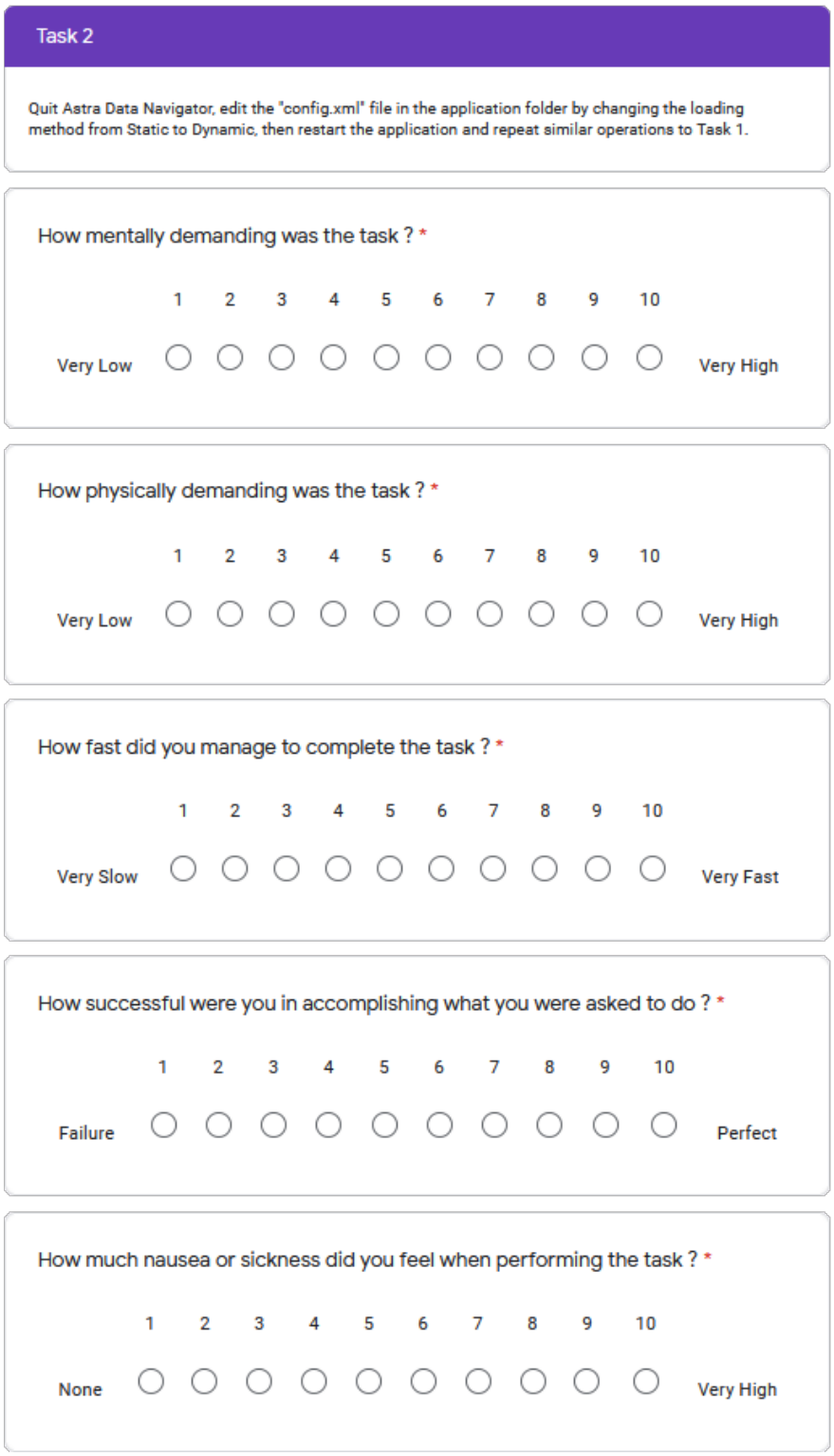
Female

Have you ever had a Virtual Reality experience ? *

Yes

No


Task 1
After launching the application, explore the universe and navigate between stars using the click-and-go mode, the search panel or free movement.
How mentally demanding was the task ? *
1 2 3 4 5 6 7 8 9 10
Very Low
Very High
How physically demanding was the task ? *
1 2 3 4 5 6 7 8 9 10
Very Low
Very High
How fast did you manage to complete the task ? *
1 2 3 4 5 6 7 8 9 10
Very Slow
Very Fast
How successful were you in accomplishing what you were asked to do ? *
1 2 3 4 5 6 7 8 9 10
Failure
Perfect
How much nausea or sickness did you feel when performing the task ? *
1 2 3 4 5 6 7 8 9 10
None
Very High



Task 2

Quit Astra Data Navigator, edit the "config.xml" file in the application folder by changing the loading method from Static to Dynamic, then restart the application and repeat similar operations to Task 1.

How mentally demanding was the task ? *

	1	2	3	4	5	6	7	8	9	10	
Very Low	○	○	○	○	○	○	○	○	○	○	Very High

How physically demanding was the task ? *

	1	2	3	4	5	6	7	8	9	10	
Very Low	○	○	○	○	○	○	○	○	○	○	Very High

How fast did you manage to complete the task ? *

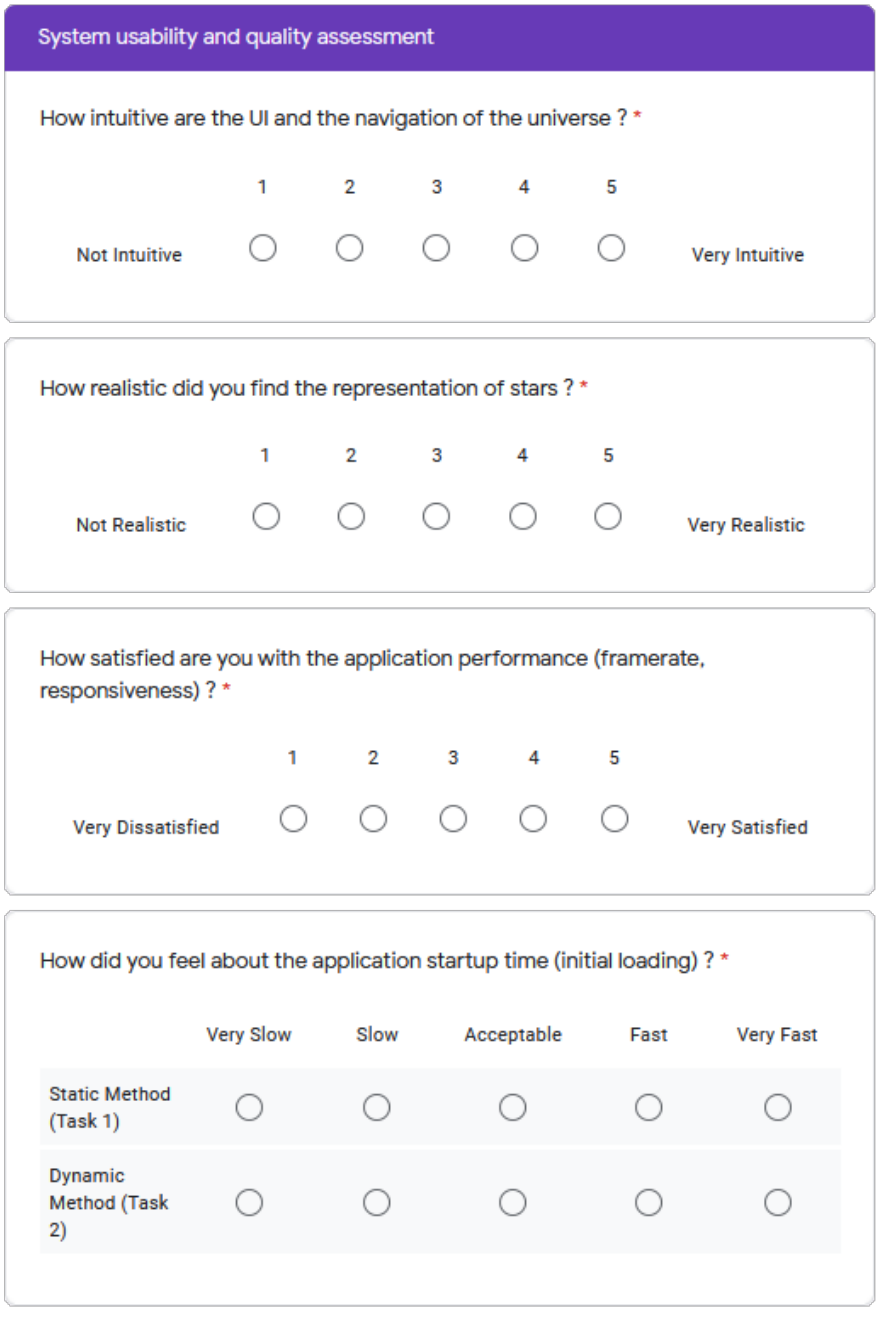
	1	2	3	4	5	6	7	8	9	10	
Very Slow	○	○	○	○	○	○	○	○	○	○	Very Fast

How successful were you in accomplishing what you were asked to do ? *

	1	2	3	4	5	6	7	8	9	10	
Failure	○	○	○	○	○	○	○	○	○	○	Perfect

How much nausea or sickness did you feel when performing the task ? *

	1	2	3	4	5	6	7	8	9	10	
None	○	○	○	○	○	○	○	○	○	○	Very High



System usability and quality assessment

How intuitive are the UI and the navigation of the universe ? *

	1	2	3	4	5	
Not Intuitive	○	○	○	○	○	Very Intuitive

How realistic did you find the representation of stars ? *

	1	2	3	4	5	
Not Realistic	○	○	○	○	○	Very Realistic

How satisfied are you with the application performance (framerate, responsiveness) ? *

	1	2	3	4	5	
Very Dissatisfied	○	○	○	○	○	Very Satisfied

How did you feel about the application startup time (initial loading) ? *

	Very Slow	Slow	Acceptable	Fast	Very Fast
Static Method (Task 1)	○	○	○	○	○
Dynamic Method (Task 2)	○	○	○	○	○

How would you rate the density of stars in the virtual universe ? *

	Very Poor	Poor	Acceptable	Good	Excellent
Static Method (Task 1)	◯	◯	◯	◯	◯
Dynamic Method (Task 2)	◯	◯	◯	◯	◯

How fast is the navigation inside of the represented 3D star catalogue ? *

	Very Slow	Slow	Acceptable	Fast	Very Fast
Static Method (Task 1)	◯	◯	◯	◯	◯
Dynamic Method (Task 2)	◯	◯	◯	◯	◯

How frequently would you like to use this tool ? *

	1	2	3	4	5	
Never	◯	◯	◯	◯	◯	Very Often

Overall system usability score *

	1	2	3	4	5	
Poor	◯	◯	◯	◯	◯	Excellent

Suggestions

Your answer

Bibliography

[1] Brian R. Kent. *Techniques and Methods for Astrophysical Data Visualization*. National Radio Astronomy Observatory. 2017. URL: https://iopscience.iop.org/journal/1538-3873/page/Techniques-and-Methods-for-Astrophysical-Data-Visualization (cit. on p. 1).

[2] C. Donalek et al. «Immersive and Collaborative Data Visualization Using Virtual Reality Platforms». In: *IEEE International Conference on Big Data*. 2014. URL: https://arxiv.org/ftp/arxiv/papers/1410/1410.7670.pdf (cit. on p. 1).

[3] *New Research Suggests VR Offers Exciting New Ways to Unlock Student Potential*. HTC Vive. Dec. 2016. URL: https://www.prnewswire.com/news-releases/new-research-suggests-vr-offers-exciting-new-ways-to-unlock-student-potential-300375212.html (cit. on p. 1).

[4] *History of VR - Timeline of Events and Tech Development*. URL: https://virtualspeech.com/blog/history-of-vr (cit. on p. 2).

[5] *The Virtual Interface Environment Workstation (VIEW)*. NASA. URL: https://www.nasa.gov/ames/spinoff/new_continent_of_ideas/ (cit. on p. 2).

[6] *NASA Explores Potential of Altered Realities for Space Engineering and Science*. NASA. Aug. 2017. URL: https://www.nasa.gov/feature/goddard/2017/nasa-explores-potential-of-altered-realities-for-space-engineering-and-science (cit. on p. 3).

[7] *NEANIAS Project website*. URL: https://www.neanias.eu (cit. on p. 4).

[8] D. Graziosi, O. Nakagami, S. Kuma, A. Zaghetto, T. Suzuki, and A. Tabatabai. «An overview of ongoing point cloud compression standardization activities: video-based (V-PCC) and geometry-based (G-PCC)». In: *APSIPA Transactions on Signal and Information Processing* 9 (2020), e13. DOI: 10.1017/ATSIP.2020.12 (cit. on p. 5).

[9] *NASA Scientists Tap Virtual Reality to Make a Scientific Discovery*. NASA. Jan. 2020. URL: https://www.nasa.gov/feature/goddard/2020/scientists-tap-virtual-reality-for-discovery (cit. on p. 6).

[10] *Gaia Archive Visualization Service*. URL: https://gea.esac.esa.int/archive/visualization (cit. on p. 6).

[11] *Gaia 3D Starmap*. URL: https://charliehoey.com/threejs-demos/gaia_dr1.html (cit. on p. 6).

[12] *Planetarium (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Planetarium (cit. on p. 7).

[13] *Stellarium website*. URL: https://stellarium.org/ (cit. on p. 8).

[14] *Celestia website*. URL: https://celestia.space/ (cit. on p. 9).

[15] *Space Engine website*. URL: http://spaceengine.org/ (cit. on p. 9).

[16] *CosmoScout VR repository*. URL: https://github.com/cosmoscout/cosmoscout-vr (cit. on p. 10).

[17] *GaiaSky website*. URL: https://www.zah.uni-heidelberg.de/gaia/outreach/gaiasky (cit. on p. 10).

[18] A. Sagristà, S. Jordan, T. Müller, and F. Sadlo. «Gaia Sky: Navigating the Gaia Catalog». In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 1070–1079. DOI: 10.1109/TVCG.2018.2864508 (cit. on p. 11).

[19] *Unity game engine website*. URL: https://unity.com (cit. on p. 15).

[20] *OpenGL website*. URL: https://www.opengl.org (cit. on p. 16).

[21] C. Thome. «Using a floating origin to improve fidelity and performance of large, distributed virtual worlds». In: *2005 International Conference on Cyberworlds (CW'05)*. 2005, 8 pp.–270. DOI: 10.1109/CW.2005.94 (cit. on p. 17).

[22] *Space Graphics Toolkit in the Unity Asset Store*. URL: https://assetstore.unity.com/packages/tools/level-design/space-graphics-toolkit-4160 (cit. on p. 17).

[23] *Star catalogues (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Star_catalogue (cit. on p. 18).

[24] M. A. C. Perryman et al. «The Hipparcos Catalogue.» In: *Astronomy and Astrophysics* 500 (July 1997), pp. 501–504 (cit. on p. 18).

[25] E. Høg, C. Fabricius, V. V. Makarov, S. Urban, T. Corbin, G. Wycoff, U. Bastian, P. Schwekendiek, and A. Wicenec. «The Tycho-2 catalogue of the 2.5 million brightest stars». In: *Astronomy and Astrophysics* 355 (Mar. 2000), pp. L27–L30 (cit. on p. 18).

[26] Gaia Collaboration, A. G. A. Brown, A. Vallenari, T. Prusti, J.H.J. de Bruijne, et al. «Gaia Early Data Release 3. Summary of the contents and survey properties». In: (Oct. 2020). DOI: 10.1051/0004-6361/202039657 (cit. on p. 19).

[27] *Gaia Archive*. URL: https://gea.esac.esa.int/archive (cit. on p. 19).

[28] *Astropy project website*. URL: https://www.astropy.org (cit. on p. 19).

[29] *SQLite website*. URL: https://www.sqlite.org/index.html (cit. on p. 20).

[30] *System.Data.SQLite library*. URL: https://system.data.sqlite.org/index.html/doc/trunk/www/index.wiki (cit. on p. 20).

[31] *PostgreSQL website*. URL: https://www.postgresql.org (cit. on p. 20).

[32] *Stereoscopy (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Stereoscopy (cit. on p. 21).

[33] *Microsoft ADO.NET documentation*. URL: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet (cit. on p. 27).

[34] *What is Data-Oriented game engine design ?* URL: https://gamedevelopment.tutsplus.com/articles/what-is-data-oriented-game-engine-design--cms-21052 (cit. on p. 36).

[35] Philip Rideout. *Emulating Double Precision*. URL: https://prideout.net/emulating-double-precision (cit. on p. 38).

[36] *NASA Task Load Index (TLX) Website*. NASA. URL: https://humansystems.arc.nasa.gov/groups/TLX/ (cit. on p. 50).

Acknowledgements

Having reached the end of my studies, I would like to acknowledge and thank everyone who supported me so far and allowed me to get to this point.

First, I would like to express my gratitude to my supervisor, professor Andrea Sanna, for his continuous availability and the valuable advice that guided me in the writing of this thesis.

I would also like to thank ALTEC for giving me the opportunity of working on this project and the additional support provided to overcome the technical difficulties caused by the current pandemic. I am very grateful to Eugenio Topa, along with the other members of the team, for the supervision of my work and the warm welcome within the company.

I also want to acknowledge my fellow colleague Simone Terzuolo, whom which I shared the majority of my time in ALTEC and collaborated with for the development of *Astra Data Navigator* .

Finally, I would like to thank my close friends and family for encouraging and supporting me during the entirety of my studies; a special "thank you" goes to Teresa, who has filled the last 2 years of my life and hopefully many more to come, for bearing with me during the most stressful moments and never letting a single day pass without showing genuine interest about the progress of my thesis work.