POLITECNICO DI TORINO

Master of Science
in Computer Engineering

Master's Thesis

Development of a VR application for the visualization and navigation of astronomical data

Supervisor

prof. Andrea Giuseppe Bottino

Candidate

Davide Trenti

A.A. 2017/2018

Abstract

The purpose of this work is to follow the development of an application that allows realistic and precise visualization of astronomical catalogs in a virtual, navigable and interactive 3D environment. The software was built in the VR laboratory of ALTEC SPA – Aerospace Logistics Technology Engineering Company – with the game engine Unity3D that, despite its precision and performance limitations, offers greater simplicity, updatability and stability than a self-produced graphic engine.

This application could be useful for astronomers to visualize data obtained by roaming satellites - like Hipparcos or Gaia - in order to infer new information and characteristics of celestial bodies, just by analyzing their position and parameters. It can also be used as an educational virtual reality experience for everyone, by showing information about each astronomical object you can travel to. For this reasons, the application must be as much precise and realistic as simple and accessible.

Hundreds of thousands of astronomical objects are contained inside the Hipparcos catalogue alone. Being stars, these objects are relatively very small and distant from each other. The Unity3D engine is not suitable to handle navigation in a galactic scale environment in a precise and reliable way, due to the limitations of the API and data types it offers. It is therefore necessary to find a way around these limitations in order to meet the requirements.

This software's entire architecture is based on the concept of the scaled space technique, combined with the floating origin technique and a data type with arbitrary precision. Together, these techniques allowed to overcome the issues that arose with the use of Unity3D engine. These techniques will be described in depth in this work, along with how they were implemented within the visualization, navigation and interaction system of the application.

Contents

Chapter 1

Introduction

ALTEC – Aerospace Logistics Technology Engineering Company – is the italian center of excellence for providing engineering and logistics services, supporting the International Space Station operations and working on the development and implementation of space venture missions [1].

ALTEC recently set up a Virtual Reality Laboratory, with the purpose to support the ongoing activities and to start up new projects, researches and collaborations.

This project's aim is to develop an immersive, interactive 3D system, capable to manage and visualize a wide range of astronomical data, at the same time allowing to represent them according to some specific physical traits of the observed objects.

As a result, the 3D environment created allows the user to interactively explore the scene thus generated, and, counting on a wide astronomic catalog, to simulate or investigate data obtained from space venture missions.

The game engine Unity3D was chosen for the development of this application. While it is simple to develop, extend and update any kind of application with this tool, it is not ideal to simulate an astronomical scale environment.

From full size planets to a worm in an apple, Unity lets you work on any scale you like. However, if you try to do both the tiny and the huge at the same time, you might find yourself coming up against some serious difficulties [6].

Floating Point Accuracy

Unity allows you to place objects anywhere within the limitations of the float-based coordinate system. The limitation for the X, Y and Z position for the `Transform` component is roughly 7 significant decimal digits, with a decimal place anywhere within those 7 digits; in effect you could place an object at 12345.67 or 12.34567, for just two examples.

With this system, the further away from the origin you get, the more floating-point precision you lose. For example, an object at 1.234567 has a floating point accuracy to 6 decimal places, while an object at 76543.21 can only have two decimal places, and is thus less accurate.

The degradation of accuracy, as you get further away from the origin, becomes an obvious problem when you want to work on a small scale. If you wanted to move an object positioned at 123456.7 by 0.01, you wouldn't be able to as that level of accuracy doesn't exist that far away from the origin.

When the observer strays too far from the origin, things like **spatial jitter** and inaccurate physics will occur (Fig. 1.1).

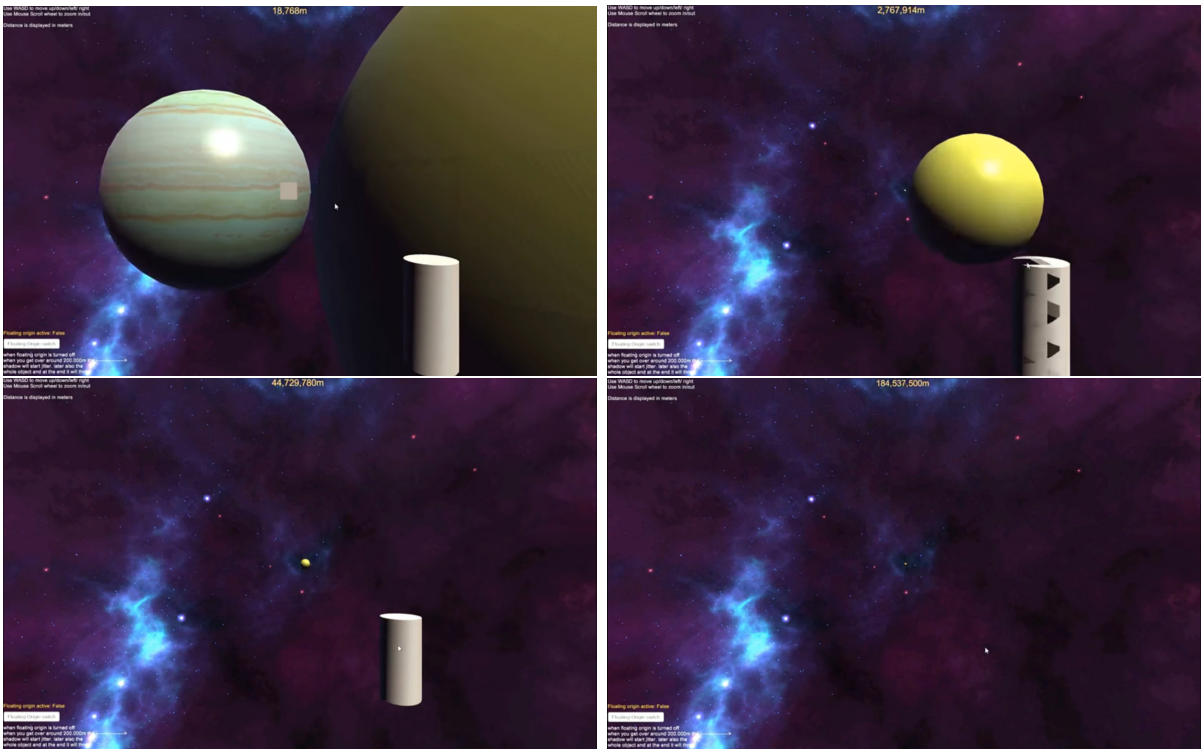

Figure 1.1: An example of object degeneration caused by SJ. The cylinder's position should be fixed to the camera, but while the camera moves further and further from the origin of the scene, it degenerates and moves in a jittery way, until it finally disappears.

Camera clipping

Unity, like any modern engine, operates on a **z-buffer**. The z-buffer describes how 'far away' things are from the camera. Every object has distance value stored as a float, which has a finite size and therefore a finite accuracy. A camera has a near and far clip plane, within which the z-buffer operates. The smaller the difference between the near and far clip planes, the more accurate a scene is going to be rendered. If the distance between the clip planes is too large, geometries will 'wiggle' because the step-increments for the z-buffer are too large [9] (Fig. 1.2).

All of those issues can be solved by implementing an architecture like the one described in this work.

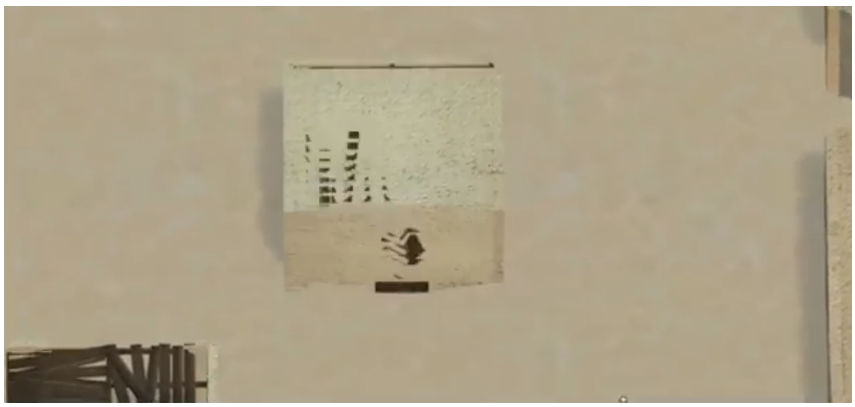

Figure 1.2: An example of aberration caused by z-buffer precision issues in Unity [11].

Chapter 2

State of the art

When dealing with large virtual worlds, the traditional approaches to avoid spatial jitter fall into three classes [8]:

On-the-fly shifting of coordinates

The on-the-fly approach to jitter shifts objects and the viewpoint close to the origin before calculations are performed that will noticeably affect how they are rendered.

The viewpoint's position is temporarily set at the world origin and subtracts the true viewpoint position from that of all other objects, just before they are rendered to each frame. Thus everything used in rendering a frame now has small, accurate coordinates and the calculations are consequently higher fidelity, avoiding jitter effects.

Once the frame is rendered, viewpoints and object positions are restored to their previous values.

Multiple local coordinate systems

To minimize jitter, virtual worlds can also be divided into smaller regions. This segmentation requires additional structures and management overhead to handle transfers between regions.

When the observer crosses a certain region boundary, the local coordinate system changes to that of the region being entered. This ensures that coordinates do not get large enough to cause jitter.

Piece-wise shifting of coordinates in a continuous virtual world

A true continuous world has a single world coordinate system with no artificial segmentation of that space. To maintain the continuity, instead of diving the world into segments, there are special viewpoints that contain the origin of an area of interest and the position of a nearby viewpoint. Whenever the observer moves to a new area of interest, one of this viewpoint is used and the world is reverse transformed by subtracting the origin from the world coordinates of the nearby viewpoint and other objects resulting in small coordinate values, thus avoiding jitter. This way, the world is shifted in a piece-wise fashion as the user goes from viewpoint to viewpoint.

Then, there is the **Floating Origin** approach, that will be described in this work.
Another issue for large virtual worlds is camera clipping. To avoid using a very far clipping plane, that could result in z-buffer precision issues, a very common technique is to have

multiple cameras render different parts of environment. In many applications there is usually one camera to render local objects and one camera to render very far objects (Fig. 2.1).

This concept is extended with the use of the **Scale Space Technique**. In Kerbal Space Program [4], a game where you can build spacecrafts and explore planets, the scale space technique allowed to render very large scale environments, from the surface of a planet to a stellar system. Combined with the floating origin, it allowed the player to travel far distances without encountering spatial jitter [3].

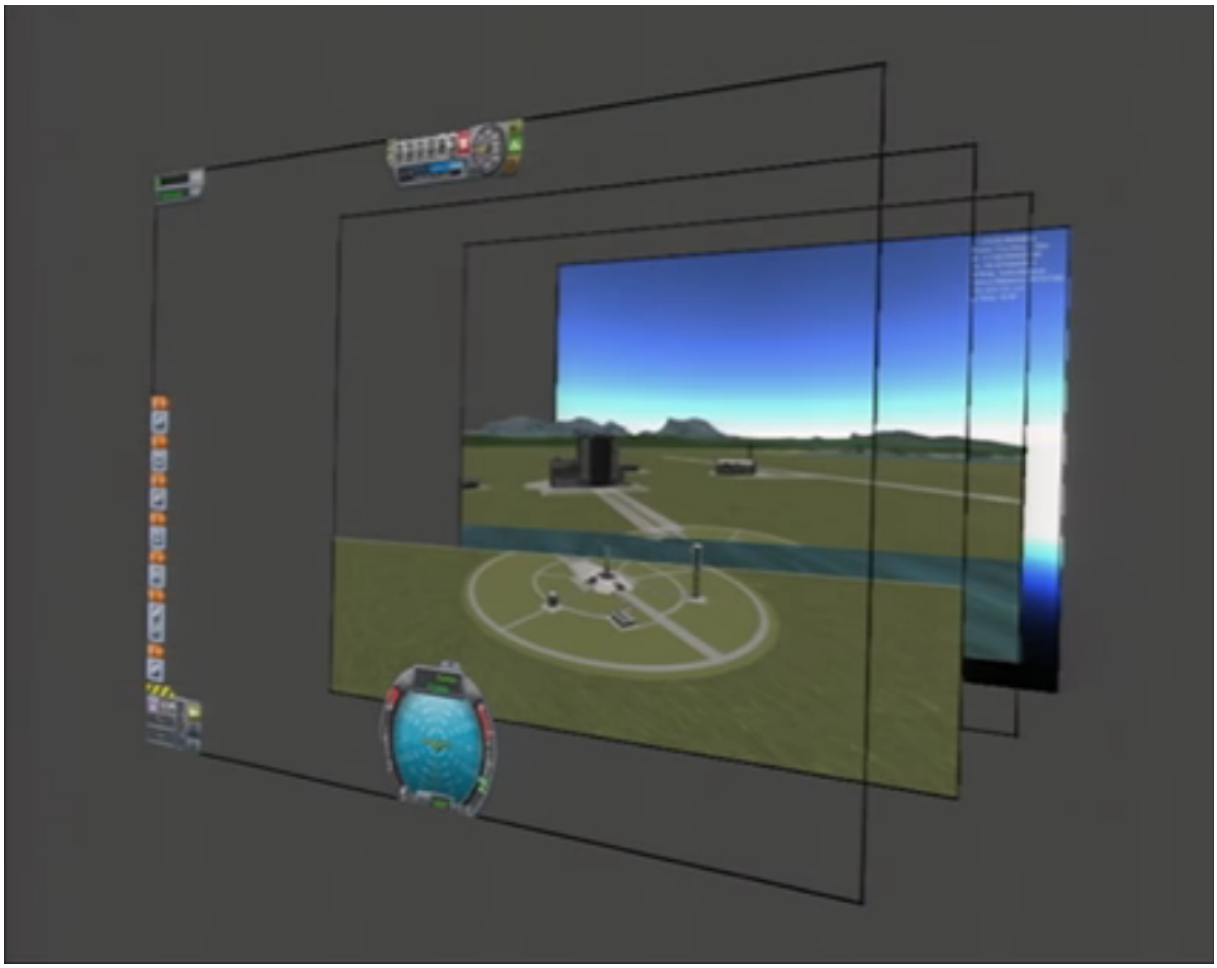

Figure 2.1: Multiple cameras rendering on top of each other in Kerbal Space Program.

Chapter 3

Architecture

The core of this software is based on two techniques, Floating Origin and Scale Space, working with an arbitrary precision data type. In this architecture, both the objects and the observer are restrained inside certain boundaries of a scene, while keeping the illusion of depth, and the potential to move in an apparently infinite space.

In order to achieve that, first you need to define multiple **layers** of space, each representing a different scale, depending on the degree of precision you want for the entire scene. It is also necessary to set the **scale ratio** with which each layer of space is defined.

3.1 Arbitrary precision numerical data type

The first issue to solve was to find a way to store the actual position and scale of an object with a more precise data type than a floating point. Considering that this application has to be scalable, it is also necessary to have an arbitrary precision.

.NET C# Type	**Description**
byte	8 bit unsigned integer (0 to 255)
sbyte	8 bit signed integer (-128 to 127)
decimal	fixed point decimal number (approx 28 significant digits)
double	double precision (64-bit) floating point number (approx 14 significant digits)
float	single precision (32-bit) floating point number (approx 7 significant digits)
int	32 bit signed integer
uint	32 bit unsigned integer
long	64 bit signed integer
ulong	64 bit unsigned integer
short	16 bit signed integer (-32768 to 32767)
ushort	16 bit unsigned integer (0 to 65535)

Table 3.1: .NET C# data types and their characteristics [2]

The current standard data types for .NET C# offer some means to have great precision, but none of them is arbitrary. There are libraries that offer data types with

arbitrary precision, but they work by exploiting strings, and could be therefore inefficient for a real-time application.

A straightforward solution was to develop a data type that stores a value for each layer that has been defined. This solution fits well with the scale space approach, because it can simplify the conversion from the precise position and scale values of the objects into the floating point values actually used in the scene, for a particular layer.

In order to develop a custom numerical data type, it is also necessary to define all the basic arithmetic operations and comparisons, and other operators that could be useful for the system.

3.1.1 Scale Space Scalar

A **scale space scalar** (*SSS*) consists in a list of 32-bit signed integers. Each of those *parts* represents a value for a certain scale, which is the scale ratio that has been set for the system. When one of those parts exceeds the maximum value for that scale, a new part is appended to represent the upper scale.

In other words, a *SSS* can be seen as a number in which every part is a **digit**, and the scale ratio is the **base**.

It is suggested to have a multiple of 10 as a base, to make the *SSS* more readable when printed. Obviously, when choosing the base you cannot exceed the maximum value of an signed integer, or else overflow issues could occur.

Addition and multiplication

The algorithm behind the addition and multiplication operators simply follows the paradigm of *pen and paper arithmetic*. Two corresponding digits will be summed or multiplied together into a temporary 64-bit signed long integer – to avoid overflow – then a carry will be computed and added to the next pair of digits.

Subtraction

The subtraction is nothing but an addition where an addend is negated. This could cause the representation of the *SSS* to have negative value in some of its parts, while others are still positive. This could affect some operations, and could also cause *aliasing* on the representation. This problem can easily be solved using a method to fix the *SSS* in order to have the same sign on all the parts. Whenever the *SSS* is printed, or needs to be used for other operations – like division or square root –, this method will be called first.

Division

The division algorithm between a dividend with any digits and a divisor with one digit is trivial. But whenever the divisor presents multiple digits, it is far from trivial.

Many division algorithms rely on logarithms, which are difficult to implement in an arbitrary precision, or guesses, which are computationally expensive. Fortunately, a simple iterative algorithm exists (Fig. 3.1).

This process uses error-correcting iterative steps to converge on the quotient and remainder. Each iterative quotient candidate is multiplied by the divisor, and the difference between the result and the numerator is halved. This results in a new quotient candidate.

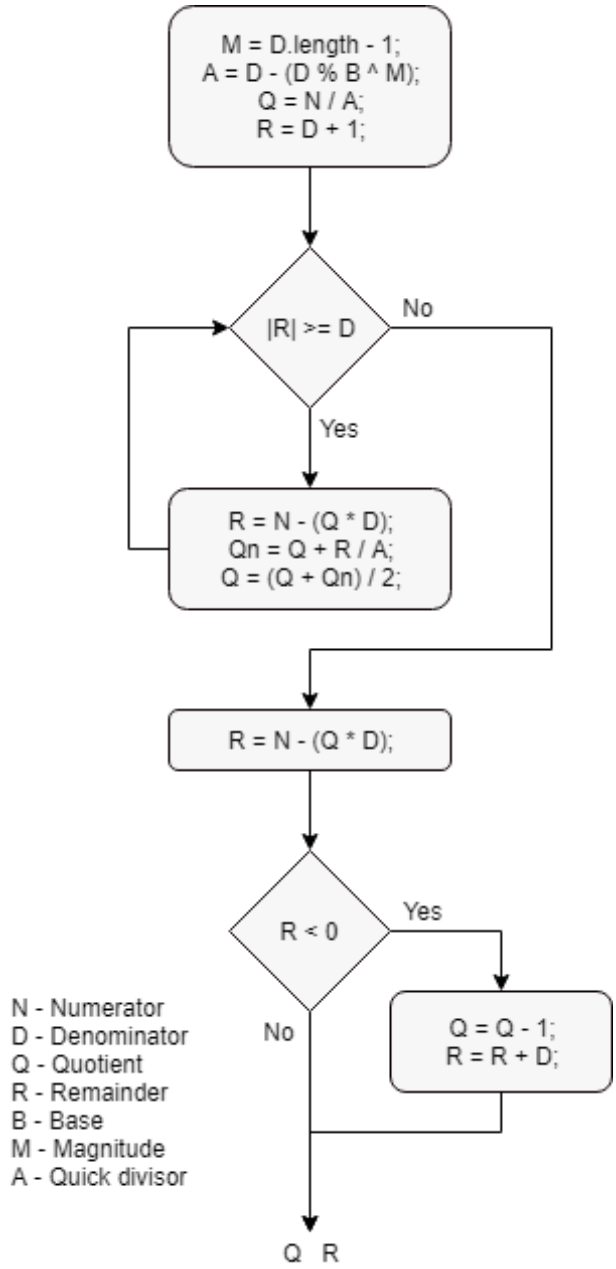

Figure 3.1: An iterative integer division algorithm

To achieve a quicker convergence, the previous two quotient candidates can be averaged. Once there is no further oscillation, and the absolute value of the remainder is less than the denominator, then the proper quotient has been determined [7].

Code reference in Appendix A.1.

Square Root

The square root of an *SSS* is implemented through the Babylonian method for finding the square root of an integer [10]. First, you make an initial guess x_0. Then, given the iterative formula

$$x_{k+1} = \frac{1}{2}(x_k + \frac{n}{x_k}), \quad k \geq 0, \quad x_0 > 0 \tag{3.1}$$

The sequence x_k converges quadratically to $\sqrt{n}$ as $k \to \infty$. One can stop as soon as

$$x_{k+1} < x_k \tag{3.2}$$

The complexity of this algorithm is $O(\log(n))$. As initial guess, you can right shift one digit of the number.

$$x_0 = n >> 1 \tag{3.3}$$

Code reference in Appendix A.2.

Conversion to floating point

The *SSS* is designed in a way that makes it easy for the whole system to get a re-scaled floating point value of a number. Given the index i of the layer you want, a simple algorithm will derive a floating point value that floats the point at the right digit.

Code reference in Appendix A.3.

3.1.2 Scale Space Vector

A **scale space vector** (*SSV*) is nothing but a container of three scale space scalars, one for each coordinate. Every operation for this data type consists in the same corresponding operation for a scale space scalar, for each coordinate, plus other vectorial operations, like the magnitude, squared magnitude and normalization.

Since the *SSV* is made out of integers, the only unit vectors you can obtain will point to the direction of an axis. It is impossible to obtain a normalized vector that points into a different direction.

However, it is still possible to do any operation that involves the normalization of a vector. For instance, the length of a vector can be resized.

Resize

Given the *SSV* to resize and a *SSS* representing the length you want for the vector, the algorithm will:

- Calculate the magnitude of the SSV;
- Left shift the SSV for the same number of digits as the magnitude;
- Divide the SSV by his magnitude, 'normalizing' it;
- Multiply the SSV by the SSS;
- Right shift back the SSV;

This way, the SSV is resized without losing any information about its direction. Code reference in Appendix A.4.

3.2 Floating origin

The further the observer goes from the origin, the more likely it for it is to end up against spatial jitter effects, which causes the observer to move in a jittery fashion. The shape, appearance and position of geometries close to it may also degenerate (Fig. 1.1). This happens when the observer's coordinates are stored as floating point.

The gap between successive representable numbers of a floating point increases by the size of the number itself; Therefore, the further the observer goes, the bigger the gaps will be, the more likely visible aberrations will appear.

The **floating origin** (*FO*) approach consists into floating the origin of the scene with the position of the observer. Instead of allowing the observer to move around the world, the world is reverse transformed to keep the observer at the origin (Fig. 3.2). In other words, when the observer translates, it actually stays still while the rest of the world moves around it. This way, the coordinates of the observer will never be great enough to cause SJ.

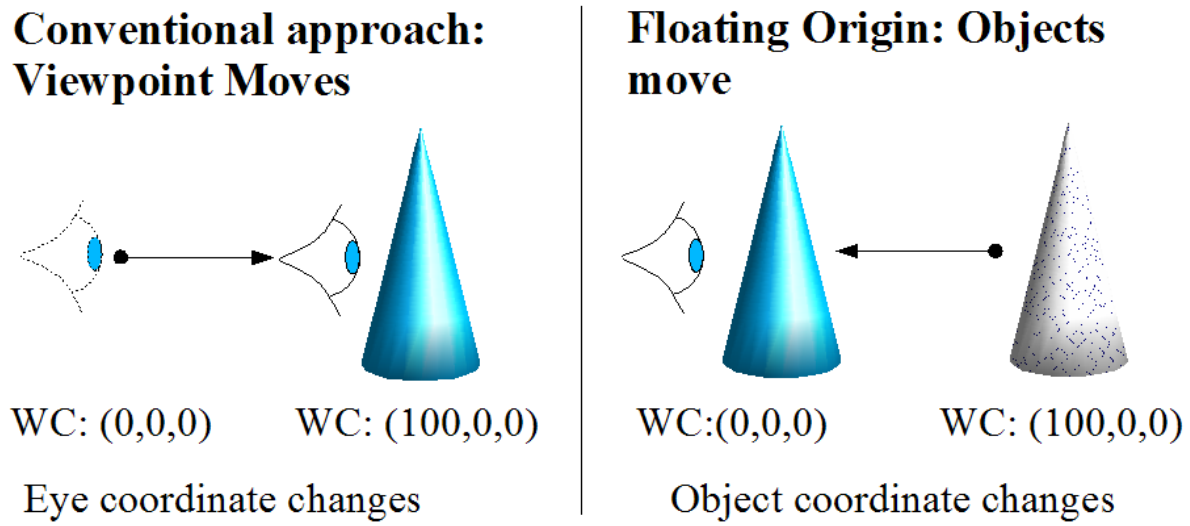

Figure 3.2: Comparing conventional and floating origin navigation.

Actually, it is more efficient to implement the FO in a way that the observer is not always at the origin, but it is shifted when it exceeds a certain **shift threshold**. This way, you can avoid updating the position of every object in the scene every time the observer 'moves', reducing the overall amount of computation. To work with the Scale Space approach – where multiple cameras move in different spaces at different scales – you also need to store a **shift** vector for each camera. While the position of the camera will be reset to the origin, every object related to the camera will be translated by subtracting the current position of the camera, and the Sf will be updated by adding the current position to it (Fig. 3.3).

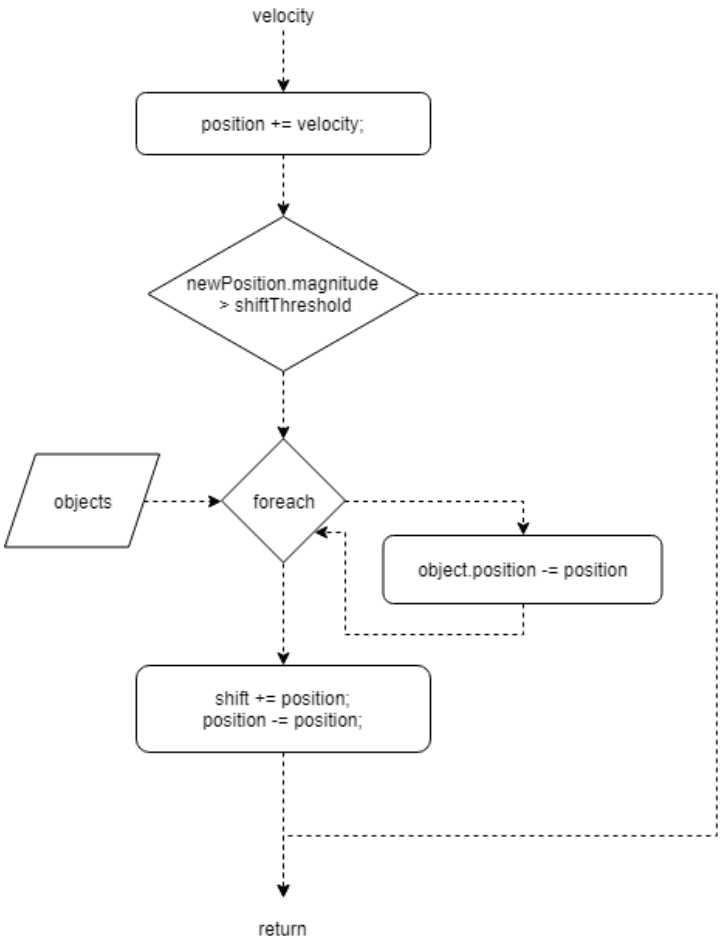

Figure 3.3: Flow chart diagram for the FO algorithm.

3.3 Scale Space Technique

Whenever you need to render very large scale environments, you must have a great distance between the camera's near and far clipping planes (Fig. 3.4), otherwise part of the environment will be cut out from the view. If the distance is too great though, the z-buffer will run into precision issues, caused by the spatial jitter of the floating point value it uses to store z values, and there will be visible aberrations while rendering (Fig. 1.2). Also, objects moving too far from the origin will, like the observer, start to move in a jittery fashion, and will eventually lose precision on the position value they store. To overcome these issues, you can take advantage of the Scale Space Technique.

A **scaled space** represents the world in a certain scale and influences the size, position and velocity of the objects that belong to it. There can be multiple layers of space, each representing a different scale, living inside the same scene. They are ordered from the smallest to the largest scale. When the distance from the origin of an object exceeds a certain **transition threshold**, it will be transferred into another scaled space, and it will be accordingly rescaled and repositioned in the scene. Therefore, when an object moves very close to the origin of a space, it will be put into a smaller scale space, hence shrunk and repositioned further from the origin in the real space; when it moves very far from the origin of a space, it will be put into a larger scale space, hence enlarged and repositioned closer to the origin in the real space (Fig. 3.5).

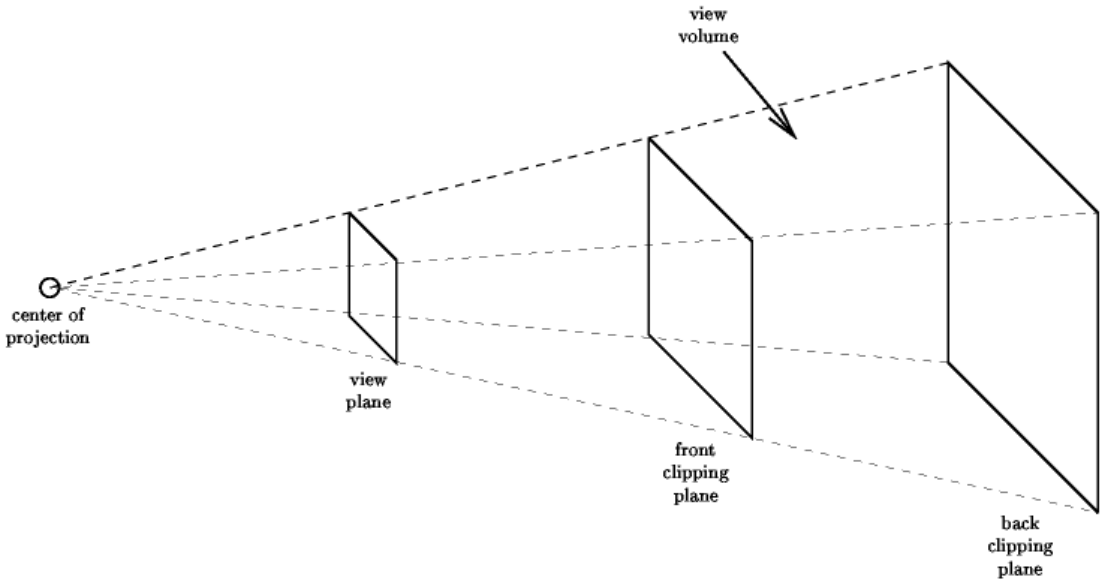

Figure 3.4: A camera near and far clipping planes. Objects outside the view frustum will not be visible.

Each space has its own camera. When the observer moves, each camera moves simultaneously – in the scene – with a velocity rescaled accordingly to the scale of the space it belongs to. Each camera will also render nothing but the objects belonging to its corresponding space, except for the biggest scale camera, which will also render the skybox.

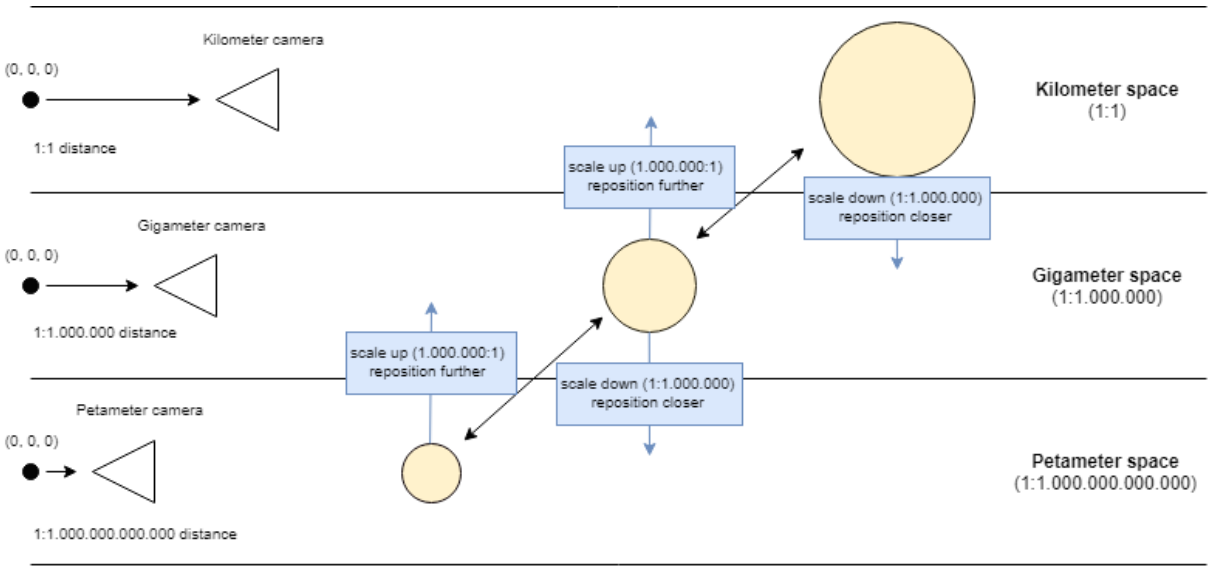

Figure 3.5: Scale space visualized.

In order to implement this technique, you need three entities: Scale Space Object (SSO), Scale Space Camera (SSC) and Scale Space Manager (SSM).

3.3.1 Scale Space Object

A Scale Space Object represents an object in the scene. It features:

- a precise position (SSV);
- a precise scale (SSS);
- a reference to the camera belonging to the same layer of space (SSC);
- a method to find the correct space it belongs to;

This entity acts as a support for the related object to host its precise coordinates, which can be easily converted to the floating point coordinates for the object in the scene.

It also has an update method that checks if its rescaled position reached the space threshold. If so, it finds the new space it belongs to and finally transfers to that space, updating the camera reference and adding itself to the list of objects related to that camera.

When the transfer occurs, the FO shift from the previous camera is unapplied, and the FO shift of the current camera is applied.

Code reference in Appendix A.5.

3.3.2 Scale Space Camera

A Scale Space Camera manages a camera in a specific layer of space. It features:

- a precise position (SSV);
- a precise shift (SSV);
- an index, representing its position in the space layers;
- a list of every object (SSO) belonging to the same space;
- a movement method that handles the FO shift;

The camera related to this entity must render only the objects belonging to the same layer of space of the camera itself, by setting their culling mask.

Every camera renders on top of each other, so the depth of the camera must be set in order to keep the objects belonging to the spaces with the smaller scales render on top of the ones with a bigger scale. This way, even if in the scene an object from a bigger scale space is actually positioned in front of an object from a smaller scale space, it will still be rendered behind it, keeping the illusion of depth.

When translated, the camera must check if its distance from the origin exceeded the shift threshold. If so, the camera must be reset to the origin of the scene, while every object belonging to the same layer must accordingly shift by the same amount. This amount will also be added to the shift property of the camera.

Code reference in Appendix A.6.

3.3.3 Scale Space Manager

The Scale Space Manager handles the entire scale space system. It features:

- an array of scale space cameras (SSC);
- the scale ratio for the entire system;
- the shift threshold;
- the transition threshold;
- a method to move the cameras;
- a method to update the whole system;

This entity has the purpose of setting all the global parameters for the other entities to follow, and it is the only one who communicates with the rest of the system. When the observer need to move, he can do it through this entity, that will properly move the cameras. From this entity you can also make the whole system update.

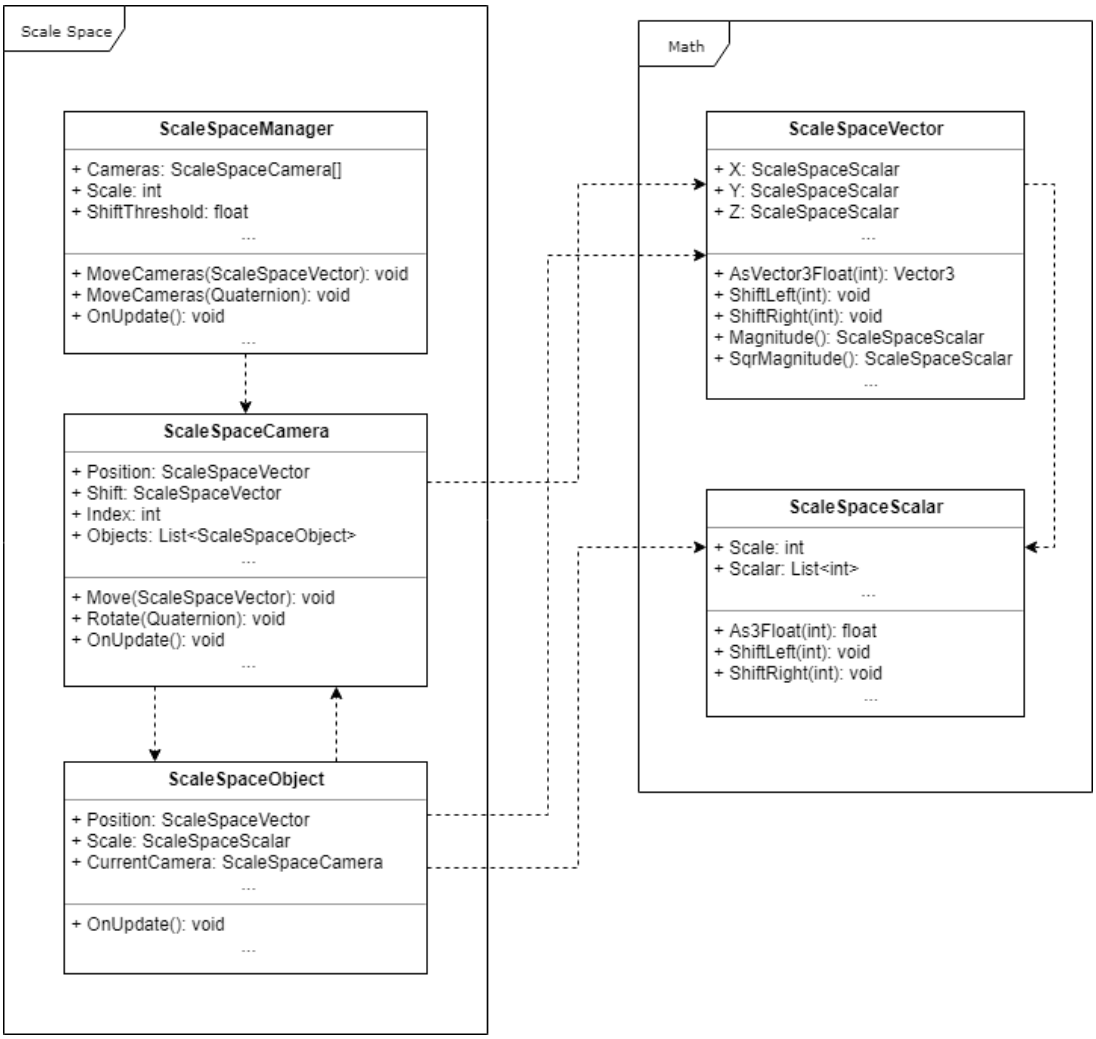

Figure 3.6: UML Class Diagram of the scale space implementation.

Chapter 4

Development

The application was developed with the Unity3D engine. The `Monobehaviour` derived classes in Unity have an update function called at each frame, but, to avoid performance issues, it will not be used, since there is a great amount of objects to be updated in the scene. The update cycle is defined separately from the original one for the majority of objects, and it will be launched only after an actual change occurs in the scene.

Since this application is still a prototype, it doesn't interface with a database yet, but loads a formatted file instead.

Hipparcos catalogue

The file contains data from the Hipparcos catalogue. The first line represents the total number of astronomical objects, then every tuple has the following data:

Name	**Description**
HIPNO	Progressive number of the star in the catalogue – 0 is the sun –
(X, Y, Z)	Coordinates of the star (in parsec), – (0, 0, 0) is the position of the sun –
RA(D), DEC(D)	Polar coordinates of the star (from the earth)
MAGH	Brightness magnitude (apparent)
BMV(J)	Blue-Violet color rate
SP, CL	Spectral classification
RSUN	Radius with respect to the sun
DIS(PS)	Distance from the sun (in parsec)
D(RSUN)	Distance from the sun (in solar radius)

4.1 Implementing the scale space

The `ScaleSpaceScalar` and `ScaleSpaceVector` have been implemented as utility scripts, like a math library.

The `ScaleSpaceManager` script is attached to a `GameObject` in the scene and it is possible to set its scale ratio and threshold parameters from the inspector, for those are public variables. Since there is no need to differentiate them, the *shift threshold* acts as a *transition threshold* too.

The *scale ratio* is set to 10^6, while the *shift-transition threshold* is set to 10^5. This way, objects and cameras will never go too far from the origin and be affected by spatial jitter. The base of the `ScaleSpaceScalar` and `ScaleSpaceVector` is set during the startup with the same scale ratio defined for the `ScaleSpaceManager`.

Three scale spaces are defined, following the $1 : 10^6$ ratio between spaces:

- Kilometer space, $1 : 1$;
- Gigameter space, $1 : 10^6$;
- Petameter space, $1 : 10^{12}$;

Each of them is implemented as an empty `Gameobject`s with the `ScaleSpaceManager` as parent, and has one `Gameobject` child with a `Camera` component and a `ScaleSpaceCamera` attached to it (Fig. 4.1). There is a Unity Layer related to each space, and every child a space contains will be set with the corresponding layer. Each camera have a Culling Mask parameter set to render only the Layer related to the space their parent represents, and the Clipping Planes set from 0.1 to 100000 (Fig. 4.2). Finally, each camera is added to the list of cameras of the `ScaleSpaceManager` through the inspector.

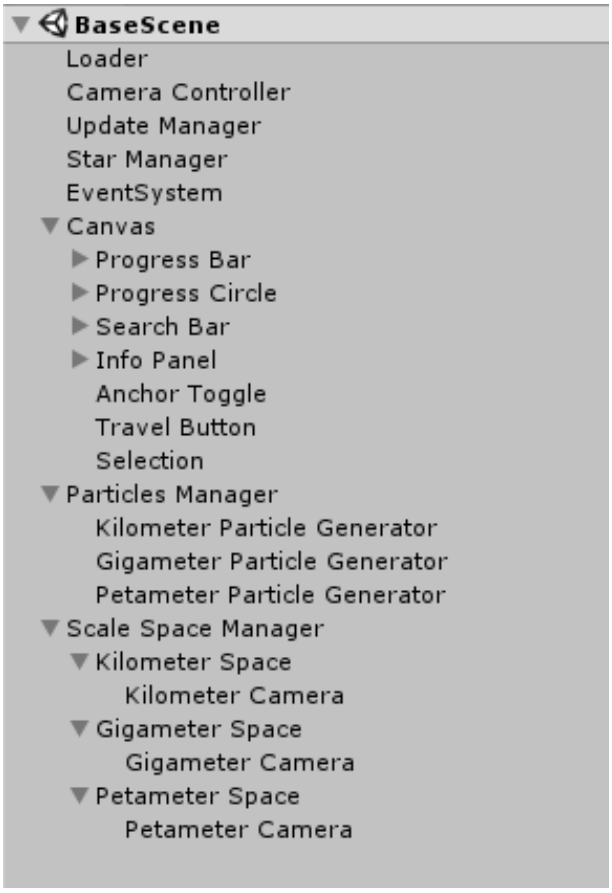

Figure 4.1: Overall organization of the scene.

The `ScaleSpaceObject` script just acts as a container to store precise coordinates. One `ScaleSpaceObject` is created for every object during the startup, together with a `Star` entity that has a reference to it.

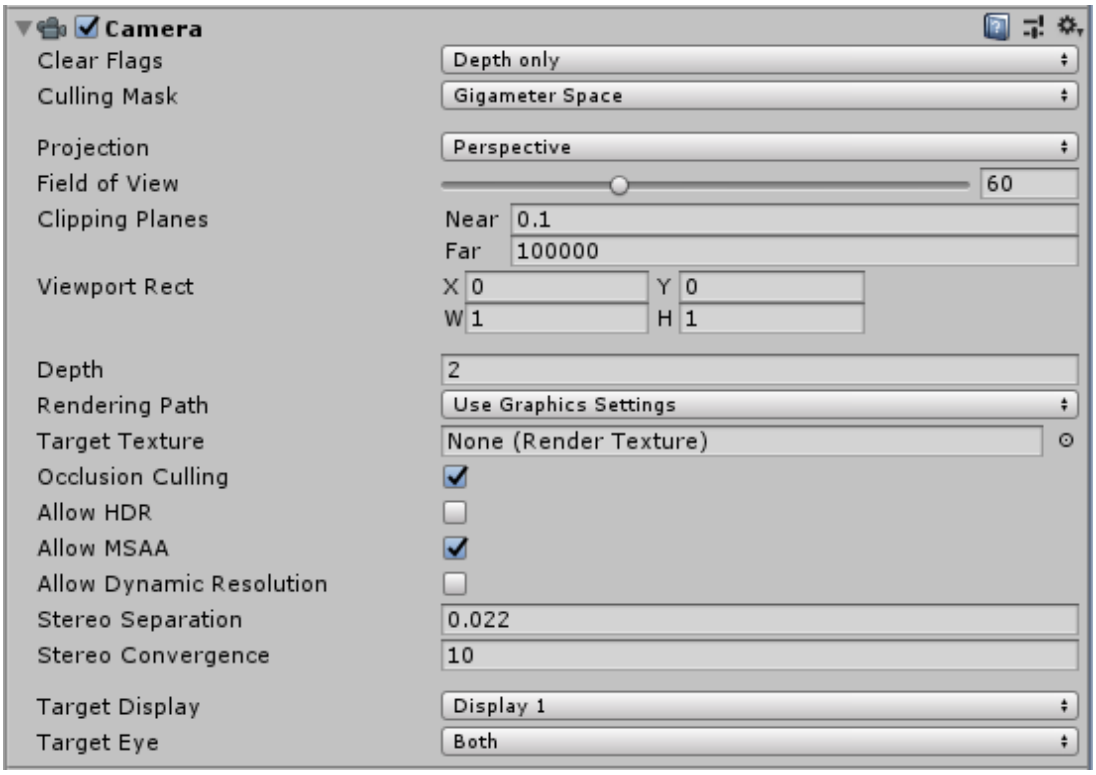

Figure 4.2: Settings for a scale space camera.

4.2 Generating the environment

When the application launches, the `Loader` starts reading the Hipparcus catalogue file, showing a loading screen with a progress bar. For each tuple, it creates a `Star` object storing all the significant data related to a star, including a reference to a new `Scale Space Object` storing the precise position and scale of said star. Each `Star` will be then added to the list of the `StarManager`.

Once the file has been fully read, the `UpdateManager` starts its first update. When doing so, it will create a model or a particle for each star – depending on its distance from the observer –, finally populating the scene (Fig. 4.3).

Figure 4.3: An example of populated spaces.

4.3 Visualization

Each time it is called by the `UpdateManager`, the update method of a `Star` calculates the precise distance between its `ScaleSpaceObject` and the observer and then determines if it is necessary to instantiate a model or a particle, after comparing it to a given threshold. It is also possible to make a particle *selectable*, by adding a `Collider` component to it in order to be a target for raycasting.

It is not efficient nor useful to instance a `GameObject` with a model for each star, because the majority of them will be very far from the observer, therefore impossible to see. The same goes for the selectable particles: instancing too many `GameObject`s with a `Collider` component would be too heavy for the CPU to handle, and it would also be useless for the further stars, because they would be covered by closer ones, hence unreachable by the raycast. That is why there is also a distance threshold for the particles to be selectable.

4.3.1 Model and shader

Since this application is a prototype, the mesh used to represent a star is a simple Unity base sphere. To add more realism, it was applied a shader that, given some parameters, procedurally generates and animates a realistic material for the star. There is also corona mesh for the star, which material is generated by another shader (Fig. 4.4).

The script `StarCreation`, attached to the star object, handles the generation of both shaders. It has a function to set the BV color rate, which will be converted to a `Color` after some scientific calculations and then applied to the material. The same function is used to color the particles too.

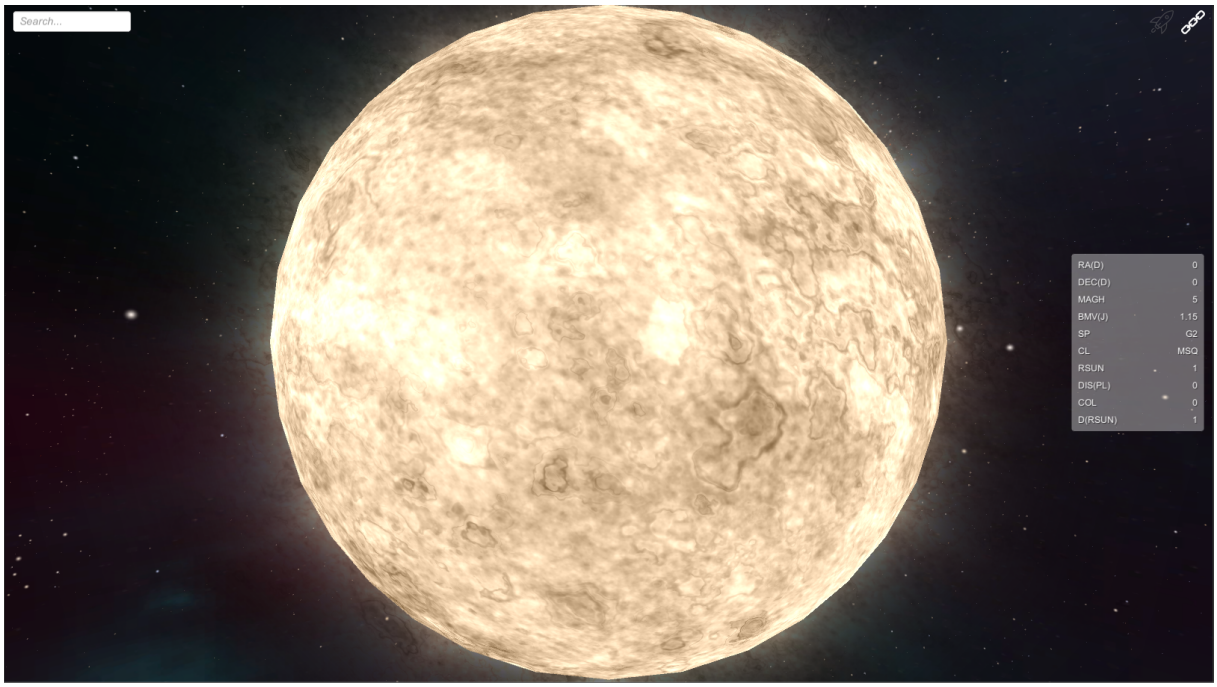

Figure 4.4: A simulated sun.

The corona is flat and visible only when facing it frontally. In order to alway keep the corona oriented towards the observer, a `Billboard` script has been attached to it.

The whole *Star Creation Tools* assets, with scripts and shaders, is available for free in the Asset Store [5].

4.3.2 Particle effects

Given the great amount of objects in the scene – the Hipparcos catalogue alone has hundreds of thousands of elements – it is greatly inefficient to instantiate everything as a `GameObject`, because at each frame Unity iterates the list of all of those objects, even if there is no script attached to them.

Generally, the observer will only see a few objects that actually need a model, and the majority of other objects can be reduced to simple fading circles. This is where the `ParticleManager` kicks in.

A `ParticleManager` features an array of `ParticleGenerator`s, one for every layer of space in the scene, and manages them (Fig. 4.1). Each generator has a `ParticleSystem` attached to it, and will generate particles only for his related camera to see.

In Unity, for a `ParticleSystem` component it is possible to define and set the parameters of every single particle that you want to generate. You can disable the emission and every pseudo-random option, and simply set and visualize the particles by script.

Every time an update occurs, all the current particles will be cleared, and every `Star` that has to be shown as a particle will call the `ParticleManager` to set a new particle to the corresponding layer of `ParticleGenerator`. Then, each `ParticleGenerator` will generate the particles and populate the scene (Fig. 4.5).

Figure 4.5: Stars as particles. One star has been selected with the RMB.

4.4 Navigation

The navigation part for this application is far from trivial from a design perspective. How to navigate a scene populated by relatively small objects only, with a really great distance between them?

A simple approach is to make the user navigate close to the objects of the scene, effectively *anchoring* the observer to them.

When anchored, the user can spherically move around the surface of the object, to observe it from every angle. The user can also disable the anchor and rotate the view while staying still, to look around for other objects of the scene. The closest objects can also be selected, and the user can decide to travel and anchor to them. This can also be done by searching and selecting the name of an object from a GUI textbox.

Right now, travelling between objects is implemented in a *teleport* fashion: whenever a travel occurs, the observer's position is instantly set to the destination. While still not implemented, there are available functions to smoothly traslate between two precise points using a linear interpolation.

4.4.1 Camera controller

The `CameraController` handles all the input from the user, and manages the whole observer's movement and rotation.

When anchored to an object, the user can hold the LMB and use the mouse to navigate spherically around the object while looking at it, like a third person camera whose focus is the anchored object. To be able to see the entire object, the observer is positioned at a certain distance, set as the scale - the diameter - of the object itself.

When not anchored, the user can hold the LMB and move the mouse to rotate the view while staying still, being able to look for other objects on the scene.

Whether anchored or not, the user can also select one of the closest objects of the scene, by pointing that object and clicking it with the RMB (Fig. 4.5). Once selected, the user can choose to travel towards it by pressing the GUI travel button.

Whenever a translation occurs, the `CameraController` also sets the `UpdateManager` to launch an update cycle for all the objects in the scene.

Raycast

The closest stars shown as particles can be selected in order to navigate towards them. Each of them is related to an empty `GameObject` instanced inside the proper space with a `Collider` component attached to it.

When the user clicks RMB, a ray will be cast forward from the mouse position in the screen space. Actually, many rays will be cast, one for each camera, starting from the one with the smallest scale. Each of those rays will ignore the objects that does not belong to the same space of the related camera, to avoid the ray to be interrupted from an object belonging to a different space.

Once an object is found, it will be considered as selected and a GUI pointer will follow the position of the star.

4.4.2 Threading

The `UpdateManager` serves the purpose of launching an update for the `StarManager` and the `ScaleSpaceManager` together. The `ScaleSpaceManager`'s task is to correctly position and scale the objects both in the scale spaces and in the scene, while the `StarManager` calculates the distances between the objects and the observer and chooses how to show the objects. Both this operations are executed by the CPU and, since they iterate every object of the scene, they are computationally expensive, hence cannot be executed at every frame. Therefore, the best approach to handle those operations is to have another thread to deal with them.

When enabled, the `UpdateManager` starts a `Coroutine` that checks, at each frame, if an update is needed. If so, it launches a thread that handles those updates.

Unity, though, does not allow to run its API operations - like `GameObject` instancing - outside of the main thread, because they are not thread safe. Fortunately, there is a small amount of API operations to be done at each update, since they are limited to `GameObject` instancing. It is therefore possible to leave those operations to the `Coroutine`, which runs on the main thread, through the use of events.

The API operations are isolated from the others, confined in a delegate function which will then subscribe itself to an `UpdateManager`'s event. When the thread ends, the `Coroutine` fires the event to finally execute those operations.

During the whole process, a loading icon will show on screen, and the changes on the scene will not be visible until this process ends. This way, the user will still be able to move smoothly through the scene during this process, without encountering any frame rate drops.

4.5 GUI

The GUI, while still primitive, offers some means of data visualization and interaction. It features:

- A progress bar: to show during the the file loading;
- A progress icon: to show whenever an update occurs;
- A search bar: to search for a star by name or index;
- An information panel: to show data about the star in which the observer is anchored;
- The travel button: to be able to travel to a selected star;
- The anchor button: to toggle the navigation method;
- The selection pointer: to track a far star that has been selected;

Search bar

Through the search bar it is possible to find the star you're looking for in order to travel to it. When text is entered, the `StarManager` will be iterated in order to suggest some results. Those results will be shown under the search bar as buttons, and the user will be able to click on them to travel to the related star (Fig. 4.6).

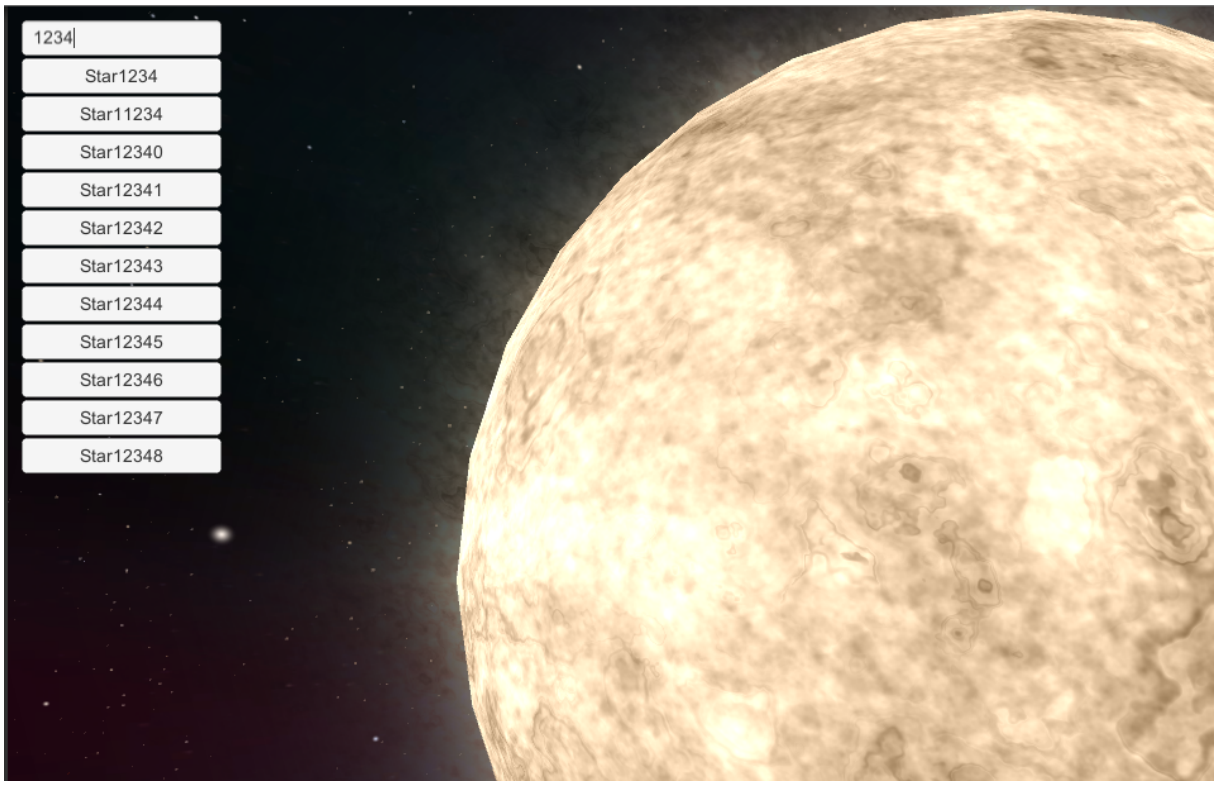

Figure 4.6: Searching for a star with the search bar.

Chapter 5

Conclusion

The prototype for this application is capable to populate a scene with stars from the Hipparcos catalogue and offers the main means of navigation and realistic visualization of the environment, maintaining the real distances and scales of the objects, in a very precise way.

The *SSS* and *SSV* offer a way to store both precise and scalable numbers, to never lose precision even when stretched the most. I wrote all the test cases to cover all possibilities and limits of this representation, and the SSS passed all of them.

The floating origin makes the observer move without the risk of having jittery trajectories, while the scale space approach makes sure that both very close and very far objects are visibile and do not degenerate. Though, right now there usually is only one object close to the observer, while the others are very far. When solar systems and other astronomical objects will eventually be added, the full potential of the scale space approach will unlock, because there will be many objects with different scales relatively close to the observer.

Every feature of this prototype was built in a way that leaves the opportunity to extend the application for future additions, thanks to the scalability offered by the overall architecture, and the ease of use of Unity editor. Every parameter of this application – especially the scale space part – can be modified inside the Unity editor.

Since there is a low number of geometries to render, there isn't much computational load on the GPU. Even though there is a great amount of operations loading the CPU at each frame, the multithreading approach to updates makes it possible to dissipate the heavy load. The application runs smoothly at 60+ FPS on most middle-end systems.

5.1 Future work

The next steps we are going to take to improve the current application include:

- More information to show about the selected object, like conventional names other than their index in the catalogue;
- Smooth traveling between objects by means of linear interpolation, to overcome the *teleport* approach;
- *Free* movement throughout the scene, with GUI options to choose different scales of velocity;

- Integrate the solar system and other celestial bodies (planets, asteroid, galaxies, and so on);
- Replace the Hipparcos catalogue with the Gaia one;
- A better GUI design for managing the selection of objects and the travel between them;
- A better GUI look and feel;
- Integration with active stereoscopy;

In the future, it is also foreseen to implement:

- Integration with interaction devices, other than mouse and keyboard;
- Interfacing with astronomical databases;
- Time;
- Integration with Oculus Rift, HTC Vive, Hololens;

References

[1] *Altec SpA website*. URL: https://www.altecspace.it/.

[2] Microsoft corporation. *Data Types*. 2012. URL: https://msdn.microsoft.com/en-us/library/bb483010.aspx.

[3] Felipe Falaghe and Michael Geelan. *Unite 2013 - How hard can be rocket science anyway? – Building a new universe in Kerbal Space Program*. 2013. URL: https://www.youtube.com/watch?v=mXTxQko-JH0.

[4] *Kerbal Space Program website*. URL: https://www.kerbalspaceprogram.com/en/.

[5] Jacob Lane. *Star Creation Tools*. URL: https://assetstore.unity.com/packages/2d/textures-materials/star-creation-tools-80595.

[6] Dave Newson. *Unity: coordinates and scales – Creating huge games in Unity's coordinate system*. 2013. URL: http://davenewson.com/posts/2013/unity-coordinates-and-scales.html.

[7] Justin A. Parr. *An Algorithm for Arbitrary Precision Integer Division*. 2015. URL: http://justinparrtech.com/JustinParr-Tech/an-algorithm-for-arbitrary-precision-integer-division/.

[8] Chris Thorne. "Using a floating origin to improve fidelity and performance of large, distributed virtual worlds". In: *2005 International Conference on Cyberworlds (CW'05)* (2005).

[9] *Unity answers, Question "Make Object be rendered far away", initiated by "Zitoox", answer by "SergeantBiscuits"*. 2016. URL: https://answers.unity.com/questions/1255982/make-object-be-rendered-far-away.html.

[10] Rick Wicklin. *The Babylonian method for finding square roots by hand*. 2016. URL: https://blogs.sas.com/content/iml/2016/05/16/babylonian-square-roots.html.

[11] Zarkov. *Issues: FOV, Floating Point Precision*. 2016. URL: https://www.youtube.com/watch?v=1lSR7KHg3D8.

Appendix A

Appendix

A.1 ScaleSpaceScalar division

```
public static ScaleSpaceScalar operator /(ScaleSpaceScalar a,
    ScaleSpaceScalar b) {
     if (b.IsZero())
         throw new Exception("Division by 0");
     if (a.IsZero())
         return a;
     ScaleSpaceScalar aAbs = Abs(a);
     ScaleSpaceScalar bAbs = Abs(b);
     ScaleSpaceScalar quickDividend = aAbs.Clone();

     // Get the most significant digit, the quick divisor
     int m = b.Scalar.Count - 1;
     int quickDivisor = bAbs.Scalar[m];
     // Shift divisor for M digits to obtain the quick dividend
     quickDividend.ShiftRight(m);
     ScaleSpaceScalar quotient = quickDividend / quickDivisor;
     // To start the while loop
     ScaleSpaceScalar remainder = bAbs + 1;

     while (Abs(remainder) >= bAbs) {
         // R = N - (Q * D)
         remainder = aAbs - (quotient * bAbs);
         // To correct the sign
         remainder.ExtendSign();
         // Qn = Q + R / A;
         ScaleSpaceScalar shiftedRemainder = remainder.Clone();
         shiftedRemainder.ShiftRight(m);
         ScaleSpaceScalar guess = quotient + shiftedRemainder / quickDivisor;
         guess.ExtendSign();
         // Q = (Q + Qn) / 2
         quotient = (quotient + guess) / 2;
     }

     remainder = aAbs - quotient * bAbs;
     if (remainder.Sign() < 0)
```

```
        quotient = quotient - 1;
    // Return with the proper sign
    return quotient * a.Sign() * b.Sign();
}
```

A.2 ScaleSpaceScalar square root

```
public static ScaleSpaceScalar Sqrt(ScaleSpaceScalar n) {
    if (n.Sign() < 0)
        throw new Exception("Negative radicand");
    if (n.IsZero())
        return new ScaleSpaceScalar();

    ScaleSpaceScalar n1 = new ScaleSpaceScalar(n);
    // First guess
    n1.ShiftRight(1);
    n1++;
    ScaleSpaceScalar n2 = (n1 + (n / n1)) / 2;
    // Iterative formula
    while (n2 < n1) {
        n1 = n2.Clone();
        n2 = (n1 + n / n1) / 2;
    }
    return n1;
}
```

A.3 ScaleSpaceScalar conversion to floating point

```
public float AsFloat(int index) {
    float sum = 0;
    int i = 0;
    for (; i < index && i < Scalar.Count; i++) {
        sum += (float)(Scalar[i] / Math.Pow(Scale, index - i));
    }
    if (i < Scalar.Count) {
        sum += Scalar[i++];
        for (; i < Scalar.Count; i++) {
            sum += (float)(Scalar[i] * Math.Pow(Scale, i - index));
        }
    }
    return sum;
}
```

A.4 ScaleSpaceVector resize

```
public static ScaleSpaceVector Resize(ScaleSpaceVector vector,
   ScaleSpaceScalar scalar) {
    ScaleSpaceVector temp = new ScaleSpaceVector(vector);
    ScaleSpaceScalar magnitude = vector.Magnitude();
    temp.ShiftLeft(magnitude.Scalar.Count);
    ScaleSpaceVector result = temp * scalar / magnitude;
```

```
    result.ShiftRight(magnitude.Scalar.Count);
    return result;
}
```

A.5 ScaleSpaceObject update method

```
public void OnUpdate() {
    while ((CurrentCamera.Vector3Position - Vector3Position).sqrMagnitude <
        1 && CurrentCamera.Index > 0) {
        // Scale down
        Position += CurrentCamera.Shift;
        CurrentCamera.RemoveObject(this);
        CurrentCamera =
            ScaleSpaceManager.Instance.Cameras[CurrentCamera.Index - 1];
        CurrentCamera.AddObject(this);
        Position -= CurrentCamera.Shift;
    }
    while ((CurrentCamera.Vector3Position - Vector3Position).sqrMagnitude >
        ScaleSpaceManager.Instance.ShiftThreshold *
        ScaleSpaceManager.Instance.ShiftThreshold && CurrentCamera.Index <
        ScaleSpaceManager.Instance.MaxLayerIndex) {
        // Scale up
        Position += CurrentCamera.Shift;
        CurrentCamera.RemoveObject(this);
        CurrentCamera =
            ScaleSpaceManager.Instance.Cameras[CurrentCamera.Index + 1];
        CurrentCamera.AddObject(this);
        Position -= CurrentCamera.Shift;
    }
}
```

A.6 ScaleSpaceCamera move method

```
public void Move(ScaleSpaceVector velocity) {
    Position += velocity;
    Vector3 newPosition = Position.AsVector3Float(Index);
    if (newPosition.magnitude > ScaleSpaceManager.Instance.ShiftThreshold) {
        foreach (ScaleSpaceObject obj in _objects) {
            obj.Position -= Position;
        }
        Shift += Position;
        Position = new ScaleSpaceVector();
    }
        transform.position = Vector3.zero;
    } else {
        transform.position = newPosition;
    }
}
```

Acknowledgements

I would like to acknowledge and thank all the people who supported me during this project and throughout my Master's degree.

First, I would like to express my gratitude to my supervisor Andrea Giuseppe Bottino, for his support and availability throughout the course of this project.

I would also like to thank all the people of ALTEC SpA, especially Eugenio Topa, who helped me a lot and supervised my work inside the company, and all of my fellow colleagues.

Finally, I would like to thank all my close friends and family, who encouraged and believed in me during my studies, filling me with determination. A special thank to my dear friend Anna, who helped me with translations, typesetting and drafting of the thesis.