VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

GRAU EN ENGINYERIA TELEMÀTICA

TREBALL DE FI DE GRAU

APLICACIONS DE REPRESENTACIÓ DE TERRENYS OPTIMITZADES PER A LES GPU

AUTOR:
CÉSAR GONZÁLEZ SEGURA

TUTOR:
MARIANO PÉREZ MARTÍNEZ

TRIBUNAL:

PRESIDENT:

VOCAL 1:

VOCAL 2:

DATA DE DEFENSA:

QUALIFICACIÓ:

RESUMEN

Este proyecto ha tenido como objetivo el desarrollo de un sistema de representación de terrenos en tiempo real, utilizando técnicas basadas en la GPU, estudiando varios métodos de renderizado para obtener una comparación de rendimiento entre ellos.

Se ha desarrollado una aplicación de visualización de terrenos implementando renderizado mediante teselación en la GPU y diversos métodos de post-procesado, junto con una aplicación para preparar los datos a representar.

Se han extraído estadísticas de rendimiento y calidad de los diferentes métodos utilizados para realizar una comparación objetiva de los resultados obtenidos a nivel visual.

RESUM

Aquest projecte ha tingut com a objectiu el desenvolupament d'un sistema de representació de terrenys en temps real, utilitzant tècniques basades en la GPU, estudiant diversos mètodes de renderitzat per obtindre una comparació de rendiment entre aquests.

S'ha desenvolupat una aplicació de visualització de terrenys implementant renderitzat mitjançant tesselació a la GPU i diversos mètodes de post-procés, junt amb una aplicació per a preparar les dades a representar.

S'han extret estadístiques de rendiment i qualitat dels diferents mètodes utilitzats per a realitzar una comparació objectiva dels resultats obtinguts a nivell visual.

ABSTRACT

The objective of this project has been developing a real time terrain rendering system, using GPU based technologies, studying different rendering methods to obtain a performance comparison between them.

A terrain viewer application has been developed, implementing GPU tessellation rendering and diverse post-processing methods, together with an application to set-up the data to be represented.

Statistics of the quality and performance of the different methods have been extracted to do an objective comparison of the obtained results at the visual level.

AKNOWLEDGEMENTS

This project would not have been possible with the help and support of many people through all these months.

I would like to thank my advisor, Mariano Pérez, for letting me work in this project and for all the support and guidance he has given to me during the development. This has been an incredible opportunity to dive into the world of computer graphics, and the experience of completing the project is priceless.

I also would like to thank all the professors who have helped me on this project, Silvia Rueda for her help and corrections, Ignacio García, Rafa Martínez and Ángel Rodríguez for their support and input, and all the others who have helped me complete this project.

And finally I would like to thank all the people who have been with me all these months, making time pass like a breeze. Mar, for her unconditional support in every moment, my colleagues and friends, Juan, Adrià, Javier, Daniel and all the others for their suggestions and the shared time at the school, to my family for their trust in my work and to all the other people who I have not named here but have been with me all this time.

INDEX

FIGURE INDEX

TABLE INDEX

CHAPTER 1: INTRODUCTION

In this project, a terrain rendering system is developed to visualize real world terrains in real time, using state of the art GPU techniques. Several techniques will be studied, and using a selection of these technologies a terrain rendering application will be developed.

The different ways to store and generate the terrain data will also be studied, choosing the most appropriate data structures for the task.

This document relates all the steps taken during the development of the system, its design considerations and constraints, the techniques used to develop the project and the difficulties encountered through the process.

1.1: Motivations

Research on computer graphics technologies is one of the most prominent fields on computing, and has been a main point of interest since the advent of the first computer systems.

Many disciplines have been taking profit of the advances in computer graphics, including medical devices, scientific data representation and simulations, weather and geographical systems, industrial design applications, among many others. In recent times, the growth of the entertainment industry has also been a key point on the development of these techniques, pushed forward by the cinema and videogame industries.

These advances on computer graphics represented a tremendous leap forward for subjects related to geography and topography. With the introduction of computer graphics, geographic information systems (or GIS for short) constituted a revolution on its field.

Terrain visualization has always been a very important topic inside research on GIS systems since its beginning. Being able to represent a real world terrain as a three dimensional image has been a great advancement for many scientific fields as well as for many professional activities.

Creating GIS systems that can represent real world terrains with high visual fidelity is a never ending challenge. As computer graphics technologies advance, the techniques used to represent terrains take profit of these advances to build faster, more efficient systems that deliver more visual fidelity when compared to the real terrain.

On recent years there have been many advances on desktop graphics hardware, allowing developers to create powerful applications that can be run on consumer grade hardware.

1.2: Objectives

The main objective of this project is implementing a terrain visualization system using state of the art technologies, to compare the different rendering methods used to find the technology that best suits this task.

To decide which rendering technologies are more appropriate, a study of the state of the art on rendering technologies will be done. Using the results of the study, the technologies to implement will be decided.

To compare the performance between these methods, we must obtain samples of the representation of the terrain from different projections through a series of iterations and do statistical comparisons between them. To do this, the terrain visualization system will generate statistics of the performance of the system along with captures of the rendered terrain, to process them and compare the different methods.

In order to obtain relevant statistics, the properties of the terrain must not change on each iteration and remain constant on all representation methods. To achieve this, a terrain data set will be built using available real world geographic imaging data.

This terrain data set, and any other data set used to test the different methods must be easy to build using existing geographic imaging data. To achieve this, an intuitive, easy to use terrain design application will be built to make the data set construction process easier.

Along with the design application, a series of tools will be built to ease the collection of geographic imaging data, from existing data collections like the United State's Geography Survey department or from the Spanish National Geographical Institute.

In addition to the main objectives, an interesting side objective is building the project using as many open source elements as possible, and ultimately releasing the application as an open source project itself.

CHAPTER 2: STATE OF THE ART

Terrain rendering has been a key research field for many years. As computer technology evolves, new methods for representing real-world or fictitious terrains are constantly developed, offering visualizations that have a better fidelity with the real data, algorithms that are more efficient in processing and memory consumption and achieving better responsiveness.

In terrain rendering there are two main challenges that arise when compared with rendering small-sized objects: the physical size of the data in the real world and the logical size of the data in the computer.

Thanks to current technology, samples of real world terrain can be obtained with very high resolutions, for example the NASA's Shuttle Radar Mission satellite can sample terrain with a resolution of 30 meters per sample. With such resolutions a dataset containing considerable expanses of terrain can take up to hundreds or thousands of gigabytes to store. With current computer memory limitations it is clear that the whole dataset cannot be loaded at once to operate with it, making necessary to create techniques that allow data to be efficiently treated according to the rendering needs and computing limitations.

The physical size of the data raises more problems besides the storing techniques. On a typical view of a landscape the sections of terrain closer to the viewer could be at a distance of several tens of meters, while the sections farther from the viewer could be several kilometers away from the spectator.

When visualizing this landscape on a computer, both the data close to the spectator and the data far from the spectator are going to be rendered, but the smaller details on the farther parts of it will not be distinguishable. Therefore terrain rendering applications must manage level of detail efficiently to be able to display the appropriate detail for the sections of terrain at a given distance in an optimized fashion.

Terrain datasets are a direct representation of the real world features, not suited for representing them directly using 3D imaging techniques. Another point of research is studying how to convert these samples into a format that the graphics devices can understand and display.

2.1: Previous Efforts on Terrain Rendering

One of the first methods developed for the representation of 3D surfaces were ray tracing techniques. Ray tracing simulates the physical behavior of light and traces rays from the viewer to the elements of the virtual world to create a visual representation of the scene. Ray tracing techniques have been used for terrain rendering for years [Mu89] and there have been many developments to improve these methods. Visual fidelity of ray tracing methods has been increasing greatly since its first efforts [Wa01].

Although ray tracing techniques have achieved great visual fidelity and can create representations of real world terrains that are close to the real data, these methods suffer from very slow computation times. The fidelity and computing requirements of ray tracing methods have evolved along processing power, but even with actual technology traditional ray tracing methods are not appropriate for real-time rendering as achieving interactive speeds has very high computing requirements.

Along with ray tracing, the other main 3D imaging methods that have been researched intensively are polygon rasterization methods. These methods use vector operations to transform polygons, mainly triangles and quadrilaterals, into a representation of the virtual world. Many techniques let these methods apply lighting to the scenes to create very high visual fidelity representations.

Compared to ray tracing methods, polygon raster methods are notably faster and have been the main point of research for consumer 3D imaging systems, thanks to the development of hardware dedicated to rasterize and shade 3D polygonal surfaces.

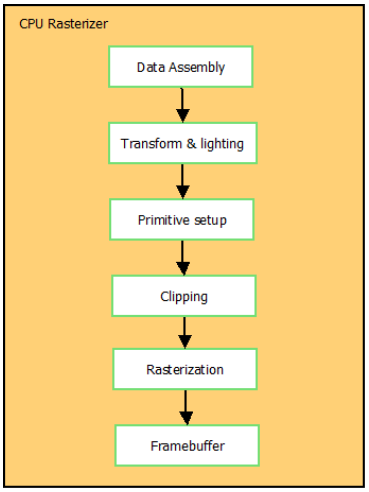


Figure 2.1.1: *Typical architecture of a software rasterizer. Green outlined blocks represent stages controlled by the application.*

Since the early nineties polygon rasterization hardware has seen an amazing increase in speed and visual fidelity. Modern hardware can draw scenes consisting of tens of millions of polygons, compared to the thousands of polygons of the hardware of the early nineties, at interactive speeds.

These raster techniques are used extensively in terrain rendering; particularly in applications where having interactive speeds are a requirement. The main methods for converting the sampled terrain data into geometry that can be understood by the hardware consists on converting the terrain samples into a mesh of triangles or quadrilaterals and transfer them to the graphics hardware.

A drawback of the early hardware rasterization hardware was the inability of programming the different stages of the rendering process. For this reason the capabilities of the renderer were fixed upon the capabilities of the graphics hardware, making the use of special lighting or post processing effects a hard task.

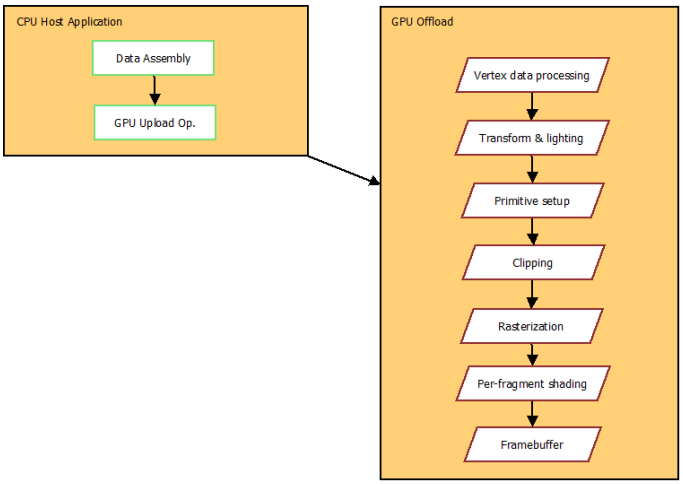


Figure 2.1.2: *"Fixed-function" rasterizer architecture. Red outlined blocks represent stages fixed by the hardware architecture.*

One of the main limitations of rasterization hardware has always been how much data can be transferred to the hardware at a time to render it in real time, whether from a time perspective (limitations on data transfer speeds) or from a computational perspective (lack of processing power or memory). Research has focused greatly on trying to create the most efficient data structures and polygon meshes to create representations as faithful to the original data set as possible and as fast as possible, using the minimum number of polygons needed.

Notable examples on techniques used to obtain the most efficient polygon structures are ROAM meshes [Ma97].

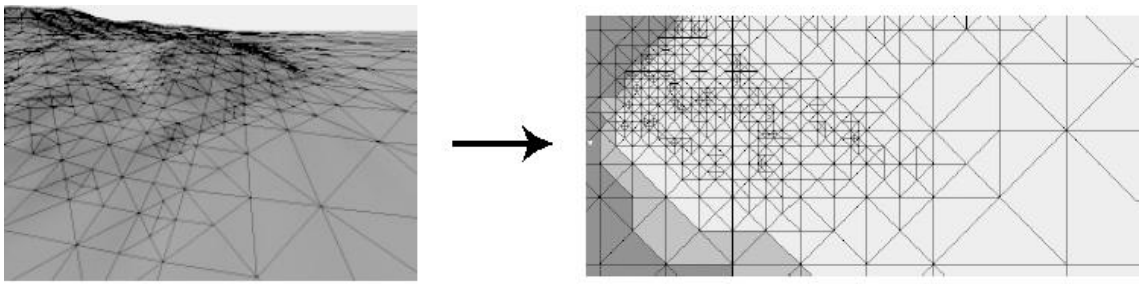


Figure 2.1.3: *An example surface triangulated using the ROAM method.*

ROAM meshes estimate an adaptive polygon structure, where the areas of the surface with a greater detail (i.e. more high frequency variations) will have a greater polygon density, whereas areas with few detail (low frequency variations) will have a much lower polygon density in comparison.

These techniques has the benefit of using just as much geometry as needed for the different areas of the terrain, and further reduces the polygon density as the distance between the viewer and the terrain increases.

Another example is geometry clipmap meshes [Fr04], one of the first algorithms to take advantage of GPU hardware. Meshes are built in concentric squares where the areas close to the camera have a greater polygon density, and it is reduced as the surface is further from the camera, introducing level of detail control dependent on the distance.

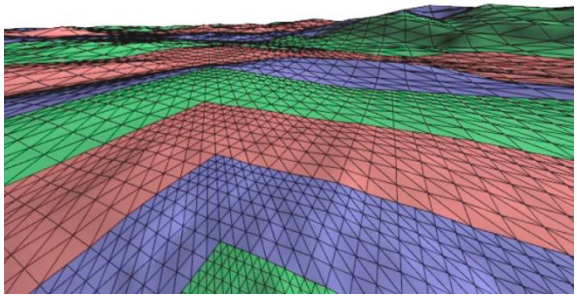


Figure 2.1.4: *Wireframe view of a terrain rendered using clipmaps. Color has been used to distinguish them from each other.*

Thanks to the fast evolution of rasterization hardware, many limitations of the past have disappeared or have been reduced. Nowadays many techniques used to circumvent the limitations of the hardware are not necessary anymore.

The decrease in the cost of memory has reduced by great means the limitations on data structures and the high increase in transfer speeds and computational power allow doing representations of large expanses of terrain.

2.2: Advanced Rasterization Techniques

The common approach in rasterization techniques was generating the polygonal meshes using the main processor and then uploading the generated data to the graphics hardware. With the advent of programmable graphics pipelines and general purpose GPU hardware, graphics hardware is not only responsible of polygon rasterization but now can be put in charge of doing any arbitrary task.

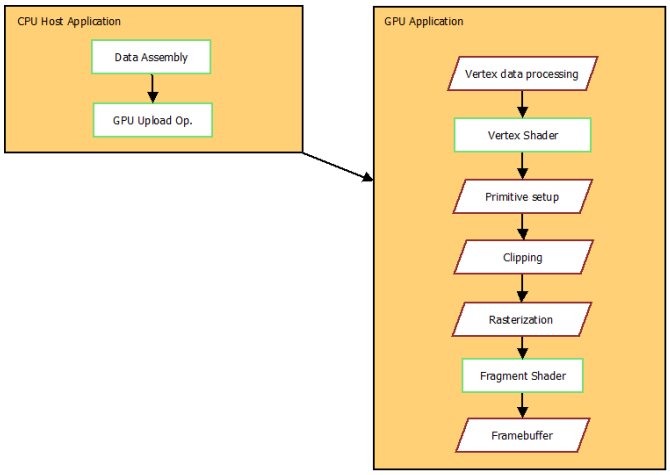


Figure 2.2.1: *Modern graphics rasterizer architecture. Now transform, lighting and shading operations can be controlled by the application, where algorithms like ray tracing can be implemented.*

Using these new features, researchers have tried to offload the data preparation work from the processor to the graphics hardware, taking advantage of the powerful parallel computing possibilities of general purpose GPU hardware. Many modern rendering techniques make a compromise between processing data in the CPU and in the GPU, trying to achieve the best possible performance and visual fidelity.

Thanks to the general purpose nature of modern GPU hardware, researchers have looked back at ray tracing rendering methods and made efforts on making them useful at interactive speeds, with good results.

One of the features of the new general purpose GPU systems featuring the Shader Model version 4, the geometry shader stage, allows creating geometry directly on the GPU. This allowed researchers to create methods that can increase the number of polygons without having to transfer all the data from the CPU, increasing the rendering speeds. However, this feature is limited and hard to use, making current research focus on other techniques.

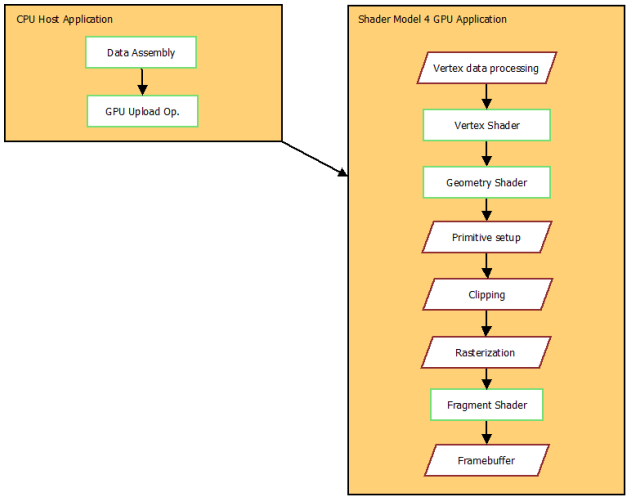


Figure 2.2.2: *The geometry shader stage allows the application to control behavior of the primitive generator, adding or removing geometry in a programmable fashion.*

In the latest general purpose GPU hardware featuring the Shader Model 5, the addition of a new feature, the tessellation shader stage, allows to create new geometry directly in the GPU without many of the difficulties of the previously added geometry shader stage. The latest research efforts have focused in using this feature combined with other techniques to achieve the best possible results.

Tessellation shaders add a new geometry domain called the *patch domain*. Patches are a collection of one or more control points, defining an arbitrary surface. The tessellation hardware triangulates these control points, creating a mesh with programmable polygon density.

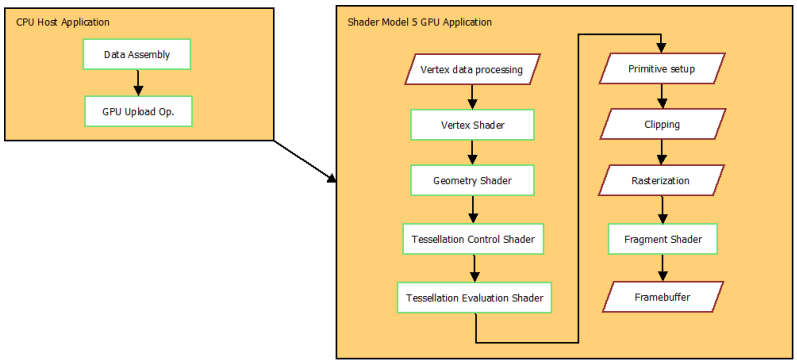


Figure 2.2.3: *The tessellation shader stage is especially useful when doing displacement mapping.*

One of the first efforts on terrain rendering using the tessellation shader was presented as a whitepaper on Direct3D 11 capabilities [Ca11], introducing techniques for rendering real world terrains achieving high visual fidelity and interactive speeds using GPU tessellation.

Currently, the tessellation shader stage is limited to tessellating the input geometry up to 64 times per patch. To overcome this limitation, the method proposed by Cantlay inputs the geometry directly to the graphics hardware as chunks with their level of detail set according to the wanted target tessellation level.

To set the tessellation level, this technique uses the distance between the patch and the viewer, reducing the polygon density of the terrain as the distance increases.

A posterior approach introduced a technique creating equally sized square patches, increasing the number of control points as the maximum tessellation level had to increase [Bon11]. This method also introduced a correction to the calculation of the maximum tessellation level, using the angle between the viewer and the patch plane, and the use of mipmaps on the heightmap textures.

The size of the heightmap textures on the video memory is a problem concerning all methods. Simultaneously to Bonaventura's method, a method was proposed to compress the heightmap textures using quadtrees [Yus11].

Posterior research efforts have mixed the tessellation techniques with previous ray-casting technologies, obtaining improved results. For example, doing an approach combining ray-casting and GPU tessellation and then selecting the fastest approach using heuristic methods [Di10].

2.3: Post-Processing Techniques

Post-processing techniques allows modifying the appearance of the rendered image, accessing directly to the output of the rasterization hardware and applying changes at the pixel level.

These techniques have been developed since the earliest graphics hardware was released. In the past applying post-process effects was an expensive task due to how was rasterization hardware designed, involving slow memory read and write operations.

On modern programmable graphics hardware, the shader pipeline allows programmers to define the appearance of the output picture, modifying the output of the rasterizer at the pixel level using the fragment shader at very low cost. This has boosted the use of post-processing techniques on most graphics applications.

There are many kinds of post-process effects, for example to make scenes look more realistic enhancing lighting, artistic effects like blurring or emulating the appearance of a black-and-white film… In this section post-process techniques used to enhance the visual fidelity of the rendered image are introduced.

2.3.1: Post-process displacement mapping

The intuitive way of increasing visual fidelity is increasing the number of the polygons of the surface to be rendered. This has a serious drawback, as increasing the number of polygons on the surface increases the time needed to render the scene as well, and will always be limited by the maximum number of polygons that the hardware can render at interactive speeds.

A technology that addresses this problem is displacement mapping. Its objective is modifying the appearance of the surface at the pixel level to make it resemble a surface with greater detail using as few polygons as possible.

One of the first examples of displacement mapping was introduced as early as in the seventies [Bli78]. The foundations introduced on this first article have been the base for posterior displacement mapping techniques.

The basic idea used on displacement mapping is defining surfaces as the combination of three surface components:

- ***Macrostructure*:** Low frequency variations which model the basic shape of the surface.

- ***Mesostructure*:** High frequency variations which represent small variations of the surface when compared with the low frequency values, but still distinguishable to the human eye.

- ***Microstructure*:** Very high frequency variations which are often cannot be distinguished by the human eye, reason why these variations will not be taken into consideration in this project.

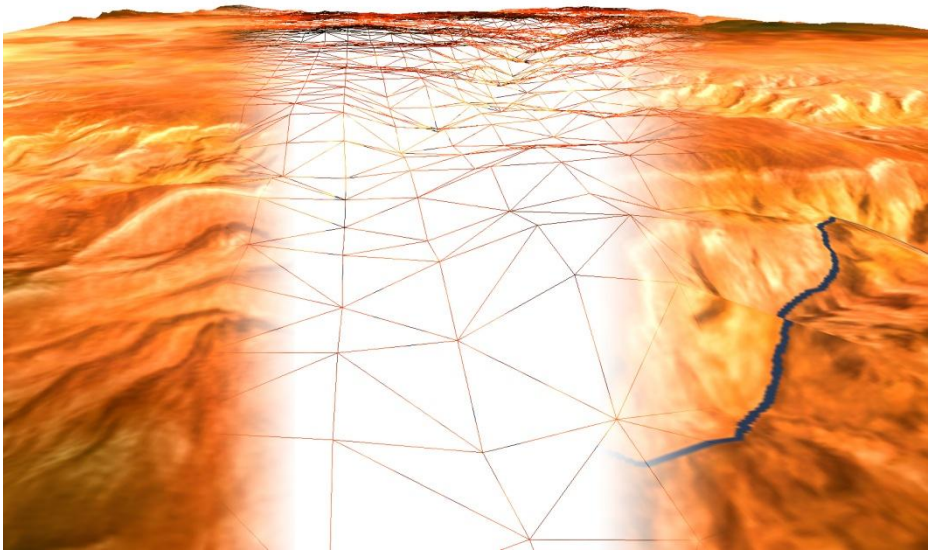


Figure 2.3.1: *Macrostructure of a terrain surface represented as a polygonal mesh.*

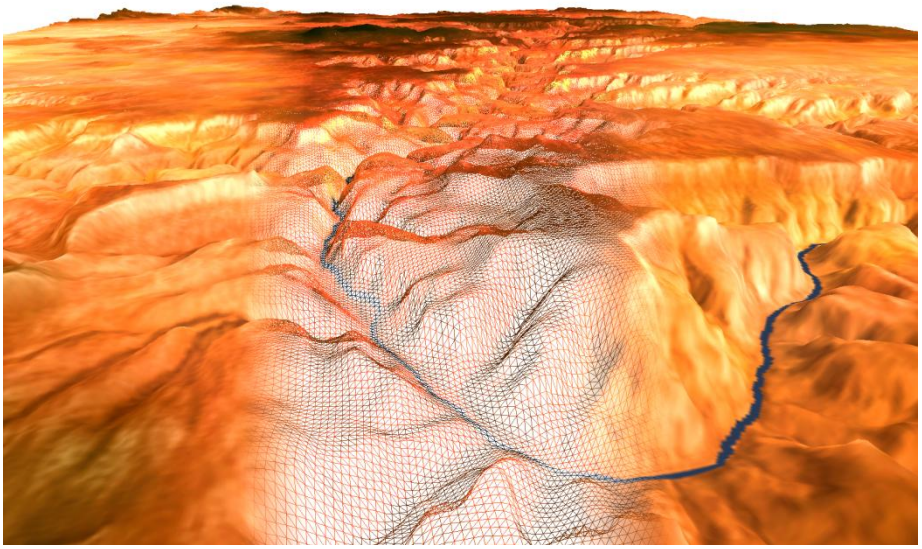


Figure 2.3.2: *Combination of the macrostructure and mesostructure of the same terrain represented as a polygonal mesh.*

When using displacement mapping only the macrostructure of the surface is defined as a polygonal surface. After these polygons have been rasterized, the displacement mapping algorithm estimates the positions of the pixels as if the surface would have been rendered defining all of its structure as polygons.

The mesostructure of the surface is defined as a height map, contained inside a texture where intensity values for each texel define the high frequency variations of the surface, and the position of the texel inside the texture define the position of the high frequency value on the low frequency structure.

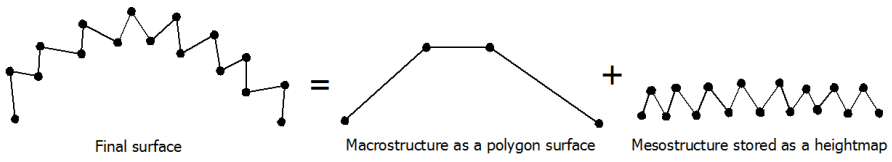


Figure 2.3.3: *Composition of a surface as a macrostructure and a mesostructure.*

Following several commonly used displacement mapping techniques are introduced along with a brief explanation on its operation.

2.3.2: Bump mapping

Bump mapping was introduced by *James Blinn* in the article mentioned earlier, uses the mesostructure to modify the incidence of light on the macrostructure.

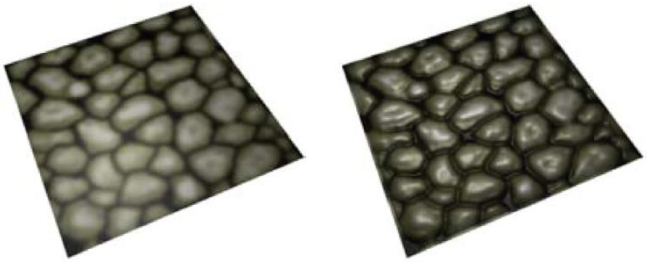


Figure 2.3.4: *Comparison of a texture mapped (left) and a bump mapped surface (right).*

Commonly this is achieved by defining the normal vector of each polygon on the macrostructure, which will be interpolated for each pixel at rasterization time, and then modifying the resulting normal vector with the variation of the mesostructure at the specified point.

This method is useful only for scenes where the effect of lighting is going to be rendered. On terrain rendering lighting is not simulated often, since the fractal behavior of terrain surfaces makes it complicated to obtain realistic results. Therefore bump-mapping is not a good candidate for terrain rendering.

2.3.3: Parallax mapping

Parallax mapping is an advance over bump mapping. Its objective is using the mesostructure of the surface to modify the texture coordinates of the color texture applied to the model instead of the normal vectors of the surface [Kan01].

Parallax mapping works by projecting a ray from the point of view of the camera to the point on the macrostructure to be rendered, evaluating the intersection between the ray and the mesostructure. The texture coordinates of the color texture on the point where the ray intersects with the mesostructure will be the correct coordinates for the texture.

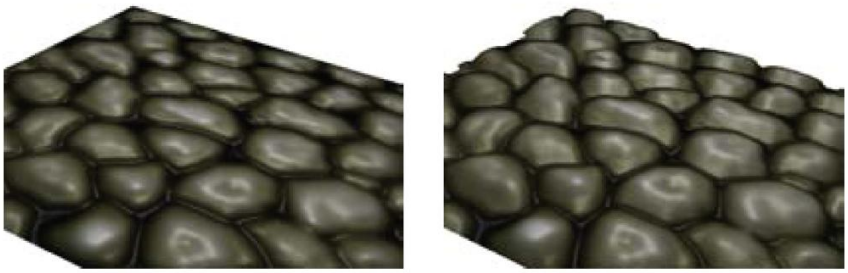


Figure 2.3.5: *Comparison between a bump mapped (left) and parallax mapped surface (right).*

In an intuitive way, parallax mapping corrects the coordinates of the color texture, approximating them to its coordinates as if the complete surface had been rendered as a polygonal surface.

One of the flaws of this first approximation of parallax mapping is caused when the angle between the view ray and the macrostructure is small. The corrected coordinates become too large, obtaining a wrong estimation and obtaining a result very far from the correct image.

Another flaw should be considered when applying this method to terrain rendering. Since terrains have a fractal structure, often the view ray will intersect with the mesostructure in other point than the intended. This can be solved using iterative parallax mapping approximations.

2.3.4: Improvements to parallax mapping

Parallax mapping has two major flaws on its first approximation; the quick amplification of the texture coordinates when the view ray angle with the macrostructure is reduced and the possibility of obtaining the wrong intersection between the ray and the mesostructure.

Efforts have been developed to try to reduce the effect of these problems as much as possible. Those methods can be classified in two main groups, non-iterative methods and iterative methods. Methods from both groups modify the algorithm used by parallax mapping, with the main difference between them being that iterative methods take an iterative approach to obtaining the intersection between the view ray and the mesostructure.

The following section introduces several improvements to the parallax mapping algorithm, featuring both iterative and non-iterative methods.

2.3.5: Non-iterative parallax mapping improvements

One of the first improvements is using parallax mapping limiting the offset of the texture coordinates [We04].

The main idea is limiting the displacement of the texture coordinates by scaling the estimated coordinates by the height of the eye ray with the intersection point. This reduces the amplification of the texture coordinates but causes a sliding effect of the texture when changing the point of view of the camera.

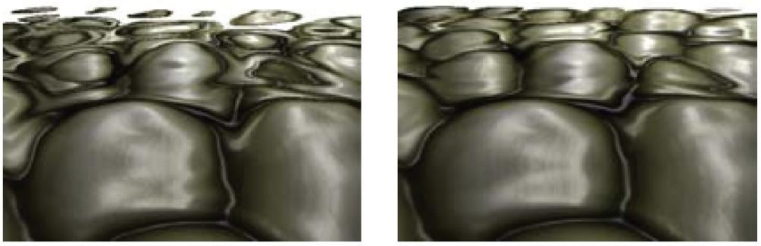


Figure 2.3.6: *Comparison between simple parallax mapped (left) and offset limited parallax mapped surface (right).*

Another improvement introduced later is taking into account the slope of the mesostructure [MM05].

Simple parallax mapping assumes that the normal vector of the macrostructure is always perpendicular to the surface plane. Steep parallax mapping assumes that the surface plane is still planar but that the normal vector can be arbitrary. This method yields much better results, particularly on surfaces with a fractal behavior.

2.3.6: Iterative parallax mapping improvements

One of the biggest issues when using parallax mapping is obtaining an unwanted intersection between the view ray and the mesostructure. In cases where the mesostructure has a fractal-like behavior, having more than one intersecting point is almost guaranteed. An approach to finding the right intersection point is evaluating the intersection in an iterative fashion.

A simple iterative approach consists on modifying the non-iterative method of parallax mapping taking into account the slope, evaluating the intersection at a height closer to the macrostructure for all the iterations [Pre06].

This method can significantly improve the results of parallax mapping but it does not ensure that the correct intersection point will be found, and on a worst case scenario convergence is not guaranteed.

Another iterative improvement is binary search [Pol05]. This method supposes that the intersection point will be confined between a maximum and minimum height, which on the initial state usually are set to the maximum and minimum height for the macrostructure.

The algorithm evaluates whether the intersection point is confined in the interval between the maximum and minimum points. If it is not, then it halves the interval each iteration until the intersection is confined between the maximum and minimum values.

Convergence using the binary search method is guaranteed, but the obtained point might not be the correct intersection point. This can cause step like artifacts on the surface, in part due to not taking into account the macrostructure of the surface.

Figure 2.3.7: *Step like artifacts caused by binary search parallax mapping.*

The secant method tries to avoid this by considering that the macrostructure between two intersecting points is planar. Then the algorithm evaluates the intersection between the plane formed by the two possible intersection points and the view ray for each iteration [Ye04].

Linear search is another iterative approach, where the intersection point between the view ray and the surface is evaluated inside a small interval close to the eye. For every iteration the ray is lengthened, until the ray intersects with the surface.

This method ensures finding the intersection point that is closer to the eye, but it can produce step-like artifacts as in the binary search method and convergence is not guaranteed when few iterations are used. Its first use was introduced in the same article as steep parallax mapping, by *Morgan* and *Max McGuire*.

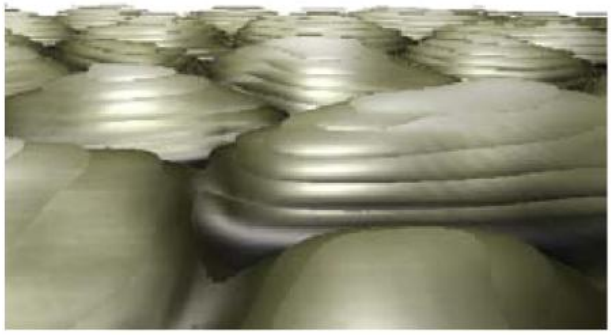


Figure 2.3.8: *Step like artifacts introduced by linear search parallax mapping.*

Further improvements to the linear search method have been introduced, like using a single secant step to reduce the step like artifacts [Tat06].

2.4: Implementation technologies

Once some of the available technologies have been introduced, the next step is choosing the technologies to be used on the implementation. This is a crucial step, since the technology used on the implementation will affect on the complexity of the solution, and therefore on how much time is it going to take to resolve.

To achieve the main objective of the project, testing the behavior of the different terrain rendering techniques, two applications have to be built: the terrain viewing application and the terrain design application.

This section introduces the technologies used for the implementation of both applications, the reason for their selection and a comparison with other available technologies.

2.4.1: Programming environment

Choosing the appropriate programming environment for the project is crucial, understanding the environment as the programming language and the development environment as a whole.

The requirements for the applications have influenced the environment that has been chosen for them. The terrain viewing application should have the smallest overhead possible, to obtain the best performance out of the terrain rendering methods and have a faithful measuring of their performance. On the other hand, the terrain design application should be easy to use, having a friendly user interface and supporting as many input formats as possible.

For the terrain viewing application, the first option to come up was using the *C++* programming language. It offers very high performance and the aids of modern programming languages, along with the support of many libraries such as *boost*, *OpenGL*, etc…

The main drawback of using the *C++* programming language is the complexity of the solution and the difficulty of debugging the errors that might arise, but its pros outweigh the cons enough as to choose this language over other languages as *C#* or *Java*.

As for the programming environment for the viewing application, using *Visual Studio* and programming directly using the *Windows API* was chosen, to further lower the impact of the overhead of the application.

On the other hand, for the terrain design application the chosen candidate was using the *C#* programming language. Along with the *Visual Studio* development environment, it offers a straightforward solution for the implementation of applications with a graphical user interface, plus the aid of the *.NET Framework* with all the available built-in libraries.

Other candidates for the terrain design application were *Java* and *C++* using an environment such as *Qt*, but *C#* was deemed to have the best compromise between performance and easiness of use, especially thanks to the intuitiveness of the *Visual Studio* interface designer.

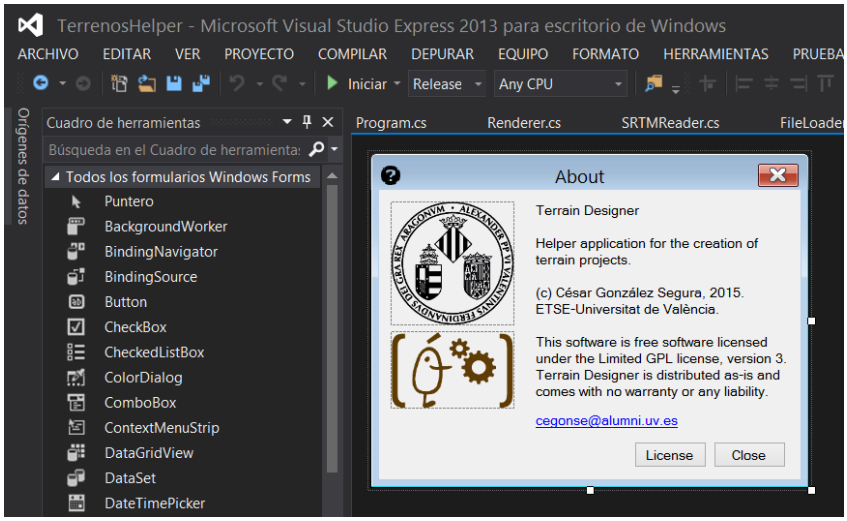


Figure 2.4.1: *A capture of the Visual Studio graphical user interface designer. The implementation is done through simple mouse actions, speeding up the development process.*

As for the development operating system and the target operative system for the system, there were two main candidates, *Linux* and *Windows*.

Linux offers a complete open-source environment, countless developer tools and an extensive support community. However, since this project requires using the latest 3D technology available using *Linux* might suppose a challenge. The support for 3D libraries and graphics hardware drivers is limited and often can be a source of problems. Plus the *Visual Studio* environment is not available under *Linux*, having to choose another platform. Therefore the chosen platform was finally *Windows*.

2.4.2: 3D Rendering Backend

The bulk of this project revolves around the rendering methods, their efficiency and performance. It is easy to see why the decision of the rendering backend is not a trivial one.

On the computer graphics industry there are many approaches to implementing 3D graphic applications. The two that were discussed for this project were using a middleware system and using a low-level backend.

Using a middleware rendering backend, such as *Unity 3D* could ease the implementation of the different methods, at the cost of a much larger overhead. Since one of the objectives of the project is measuring the performance of the different methods as accurately as possible, it is important to reduce the overhead to the maximum.

It was decided then to use a low-level library. There are two main low-level libraries available to implement 3D graphic applications: *OpenGL* and *Direct3D*.

OpenGL is the industry standard for graphics applications, available on many platforms and its development state is together with the state of the art. Its main advantages are its simplicity and the easiness of porting to different platforms with minimal changes to the application codebase.

However, debugging *OpenGL* lacks proper tools and detecting errors can turn into a cumbersome task, and the design of the library is far behind modern libraries, since it goes back to the first nineties and is not object-oriented.

On the other hand, *Direct3D* is a closed platform developed by *Microsoft*, used mainly on videogames for the *Windows* and *Xbox* platforms. *Direct3D* is known for being easier to debug than *OpenGL* and for having a more modern design compared to *OpenGL*, since it is object-oriented.

However, the fact it is a closed technology and that it is only available in a handful of platforms has made *OpenGL* the best candidate, since it would allow porting the applications to other platforms like *Linux* in case it was needed.

2.4.3: Data format technologies

Another point of interest of the project was choosing the appropriate data structures to store the terrain data, in order to let the viewing application access it as best performing as possible.

The first point where the attention should be set is on the formats used by the existing geographical data, which will be in most cases the input data of the terrain design application.

Since geographical data is composed of discrete samples of imagery taken by photographic cameras, satellites, among other imaging devices, the data formats used by the terrain applications usually are raster images, where the value of each element represents the physical height or color in that coordinates.

In this section, diverse data formats are introduced, explaining its pros and cons, and finally the most appropriate format is chosen for the viewing application input data format.

2.4.4: Lossless versus lossy compression

The size of raster images scales with the amount of information it contains. Taking into account that terrains might span tens of square kilometers with resolutions of less than 30 meters per sample, the size of the raw data would be considerable.

It is important to find a way to remove the redundancy of the data to achieve the smallest possible file size while keeping the information. To achieve this there are two main methods: using lossless compression and using lossy compression.

Lossless compression algorithms create an exact version of the original information, allowing the original data to be reconstructed from the compressed sequence. Usually, the algorithms look for patterns in the data to build a dictionary and encode the data using the words in the dictionary.

Examples of formats allowing the use of lossless compression are the *TIFF* (Tagged Image File Format) which can use LZW (Lempel-Ziv-Welch) or ZIP algorithms, or the *PNG* (Portable Network Graphics) format which uses the DEFLATE algorithm.

Alternatively, lossy compression algorithms create an inexact version of the original data, usually discarding information or taking advantage of the physical properties of the data, where the original data cannot be reconstructed.

Some examples are the *JPEG* (Joint Photographic Experts Group) which uses the transformation of the data from the image space to the frequency space to discard high-frequency, unappreciable information; other example is the *ECW* (Enhanced Compressed Wavelet) which uses the wavelet transform of the image and is thoroughly used on geographical applications.

When using compression algorithms the usual performance metric is the compression ratio, or the ratio between the size of the original data and the compressed data. A good compression algorithm should generate a representation several times smaller than the original data.

However, in some applications when using lossy compression the artifacts generated by the removal of information might cause artifacts which can degrade the quality of the reconstructed data, therefore in some cases a tradeoff between the compression ratio and the reconstructed quality.

The terrain rendering system uses two images to build the terrains: a color texture and a heightmap texture. In the color texture case, a lossy compression algorithm could be used. Color textures only give the appearance to the surface of the terrain, and as long as the result is visually appealing, the appearance of degradation is acceptable.

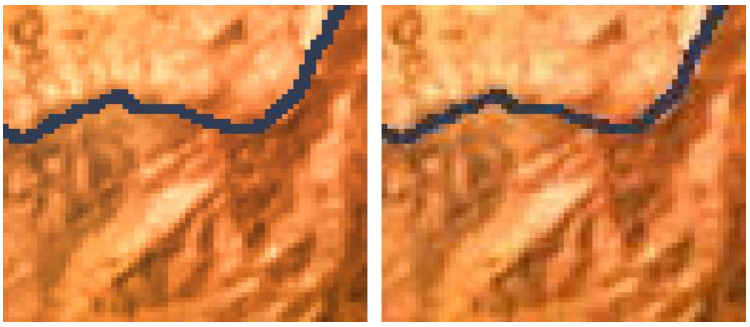


Figure 2.4.2: *Close-up from a section of a terrain color texture, on the left side the original data, on the right side the data compressed with a lossy compression algorithm.*

However, on height textures degradation caused by lossy compression would have a serious impact on the appearance of the terrain. In order to obtain the height of the terrain at each point, the value of the texture is fetched and translated into a height value.

Since the compression would introduce artifacts on the texture, the height values would be completely different on the areas affected by the degradation.

2.5: Summary

With the introduction of the state of the art for both terrain rendering technologies and implementation technologies, in this section a brief summary of the selected technologies for the project is introduced.

- **Terrain rasterization techniques:** The chosen technology is using a terrain tessellation method based on the techniques introduced in [Ca11] along some of the improvements introduced in [Bon11]. Terrain tessellation.

  Terrain tessellation takes advantage of the latest technologies in graphics hardware and makes a great basis for the further implementation of other techniques, in this case post-processing techniques.

- **Post-processing techniques:** From the proposed techniques, a selection has been done to test the performance of post-processing techniques when used on terrain rendering being, for non-iterative methods simple parallax mapping, parallax mapping taking into account the slope, and for iterative methods binary parallax mapping and secant parallax mapping with linear search.

  These four methods are an extensive enough selection to create a test bed and see the performance of adding post-processing to the previous tessellation method.

- **Programming environment:** From the proposed environments, the selection is using C++ with Visual Studio for the terrain rendering system and C# with Visual Studio for the terrain design system.

  C++ offers a fast, low level implementation to do an unbiased comparison between the different rendering methods, and C# offers an intuitive and easy to implement solution for applications featuring a complex graphical user interface.

- **3D rendering backend:** The technology that has been deemed best suited for this project is OpenGL, which offers an open, portable solution for 3D rendering, exposing all the power of the latest technology available on the graphics hardware.

- **Data format:** From the introduced formats, the selected format is a lossless format for both the terrain color and heightmap textures, exactly the TIFF format.

  The TIFF format offers a robust, tested and open source implementation in both C# and C++ languages, and lossless compression using the LZW algorithm, meaning a good enough compression ratio for the textures.

CHAPTER 3: SYSTEM SPECIFICATION

The objective of this chapter is defining the basic guidelines for the project and all its systems. Since this project is formed by two complete applications, the system definition and requirements and use cases will be defined for both subsystems separately, however those cannot be understood unless taken into account together.

The system requirements are a translation of the objectives of the project, showing the features that have to be implemented on the final system.

Also in this chapter is introduced the first cost estimation for the project, with the proposed task diagram and its related costs, both in time and economic terms. This estimated will be evaluated on the closure of the project to analyze if there have been deviations on the cost of the complete project.

The results of this analysis will stick to the objectives defined in the second chapter, and the success of the project will depend on how well the objectives translate to the final system behavior.

3.1: System Requirements

From the project objectives and the general definition of the system, a set of requirements was extracted to base the system design and the technology requirements on them.

The requirements have been divided in two subgroups: functional requirements, describing requirements that must be directly related to the system features, and non-functional requirements, which must be required to develop the project but are not related to the system features.

On the following sections functional and non-functional requirements are listed. The requirements have been divided depending on whether they apply to the viewing system, to the design system or to both.

3.1.1: Requirement List

The functional requirements of the terrain viewing system are:

- ***Func-View-1*:** The user must be able to take still pictures from the rendered view of the terrain and save them to let the user work with them later outside the application.

- ***Func-View-2*:** The application must let the user visualize terrains from different points of view, moving the point of view easily using the mouse. It is the main requirement of the system.

- ***Func-View-3*:** Following a set of waypoints created on the terrain designer, the application must let the user start an animated flight following the path created by said points, being able to pause and restart the animation at will.

- ***Func-View-4*:** Using the viewer interface, the user must be able to change the different render settings of the application.

- ***Func-View-5*:** The application must be able to load projects created with the terrain design subsystem.

The non-functional requirements of the terrain viewing system are:

- ***NonFunc-View-1*:** The application must respond in real-time speeds to user commands to ensure interactivity, achieving refresh rates of 60 FPS or more.

- ***NonFunc-View-2*:** Rendering methods must be customizable by the user, exposing the shader code to do so.

- ***NonFunc-View-3*:** The application overhead must be as small as possible, to let the user obtain faithful statistics about the rendering methods performance.

- ***NonFunc-View-4*:** The application must output a summary of the performance of the application with information about the mean elapsed times for the different operations, the memory consumption of the application, etc, to let the user work with them later outside the application.

The functional requirements of the terrain design system are:

- ***Func-Design-1*:** The application must let the user create terrains to be used on the view application, placing tiles with an associated position, heightmap texture and color texture. It is the main requirement of the system.

- ***Func-Design-2*:** The application should accept as many source formats as possible, to make the creation of terrains from the original GIS as easy as possible.

- ***Func-Design-3*:** The application must let the user place waypoints on the terrain, to use them as a path on the viewing application for the camera animation.

- ***Func-Design-4*:** The application should assist the user to obtain source data for the terrain creation from an existing data collection.

The non-functional requirements applicable to both systems are:

- ***NonFunc-Both-1*:** The user interface of both applications should be intuitive, letting the user work through the application with three or less interactions per command, in as many commands as possible.

- ***NonFunc-Both-2*:** Both systems should use as many available existing libraries as possible, to ease the implementation process, using open source libraries when available.

- ***NonFunc-Both-3*:** Both systems should be able to be licensed with an open source license, so the research and development community can benefit from the result of this work

3.2.2: Requirement Summary

Following is a summary of all system requirements, classified by their priority: mandatory, hidden and optional features.

Name	*Description*	*Priority*	*Type*
Func-View-1	*Take still pictures of the rendering.*	Mandatory	Visible
Func-View-2	*Visualize terrains from various points of view.*	Mandatory	Visible
Func-View-3	*Start camera flight animation over the terrain.*	Mandatory	Visible
Func-View-4	*Change rendering settings.*	Mandatory	Visible
Func-View-5	*Load terrain projects.*	Mandatory	Visible
Func-Design-1	*Create terrain projects.*	Mandatory	Visible
Func-Design-2	*Accept as many input formats as possible.*	Optional	Visible
Func-Design-3	*Place waypoints for the camera flight.*	Mandatory	Visible
Func-Design-4	*Assist user to obtain source data.*	Optional	Visible

Table 3.1: *Requirement Summary for both applications*

3.2: System Definition

The system to be developed is a terrain visualization system using state of the art GPU rendering technologies. The proposed solution is composed of two systems, a viewing system and a designing system.

The main purpose of the viewing system is verifying the competence of several modern GPU techniques for terrain rendering, as well as obtaining statistics to do a strict comparison between each method. The rendering of the terrains will be done using the selected rendering methods selected on the study of the state of the art.

Besides being used to verify the quality of said rendering methods, the terrain viewer system also can be used as a general purpose terrain viewer, letting the user view the terrain from different points of view and do a flight over the rendered terrain.

The design system is used to build the files used by the viewing system, using existing terrain datasets extracted from Geographic Information Services contributed by the user. If necessary, the system can assist the user in obtaining source data from an existing geographical dataset source.

Following is a detailed description of the solution, describing the blocks composing the systems and their behavior.

- ***Solution for the Terrain Designer***

The terrain designer is in charge of letting the user transform the available geographical data into a dataset to use on the terrain viewer application. It must let the user import height map and color data from a variety of sources and place those tiles intro an arbitrary position, rendering the terrain in a two-dimensional, orthogonal projection.

It also must let the user create a path to be used by the camera flight on the terrain viewer application. The points that create the path must be easy to place and to move on the space.

With these requirements at hand, a high level representation of the application architecture using descriptive components is created. These components act as the basis for the conceptual class model of the application, which are:

- ***Controller:*** In charge of coordinating the interaction between all the different components of the application.

- ***User Interface:*** In charge of interacting with the user to expose the functionality of application. It accepts the user info and shows the state of the application at every moment.

  The design of the user interface is composed of several dialogs, creating compartments for the different features of the application. The placement, design and flow of the different dialogs of the application must be intuitive as to let the user operate the application with the least assistance possible.

- ***Tile Renderer:*** The application displays the all the terrain times in real time, where the user can change the position and zoom of the viewport and change the position of the different tiles.

  Since the application must respond in real time to the user commands and the project might contain a considerable number of terrain tiles, the renderer must be able to handle all the data in an efficient way, with the minimum computational (memory consumption and processing power consumption) footprint possible.

- ***Project Manager:*** The main goal of the terrain designer application is converting the original geographical data into a dataset that can be understood by the terrain viewer application.

  The dataset is formed by a terrain project, containing terrain tiles with a height texture, defining the geometry of the tile, and a color texture, defining the appearance of the tile. The project manager is the system in charge of keeping track of the changes on the position and data of each one of these tiles.

  The manager is also in charge of saving and loading the state of the project to let the user interrupt and later continue working. Finally, the project manager is in charge of exporting the terrain project to the final dataset, converting the original data to the formats that the terrain viewer understands, and it is also in charge of importing an already converted existing terrain project, to let the user make changes if necessary.

- ***Data manager:*** It is in charge of converting the different data types used by the project and of the needed features to assist the user to obtain data from an existing dataset.

  In the system analysis it was stated that the application should let the user convert *SRTM* files and download a dataset from the *Iberpix* collection of the *IGN* (*Instituto Geográfico Nacional*, Spain). To be able to fulfill these features, the data manager can import *SRTM* files, export *TIFF* files and has all the necessary logic to download files from the *Iberpix* collection.

The following diagram illustrates how the different components interact, with each other and with the external systems.

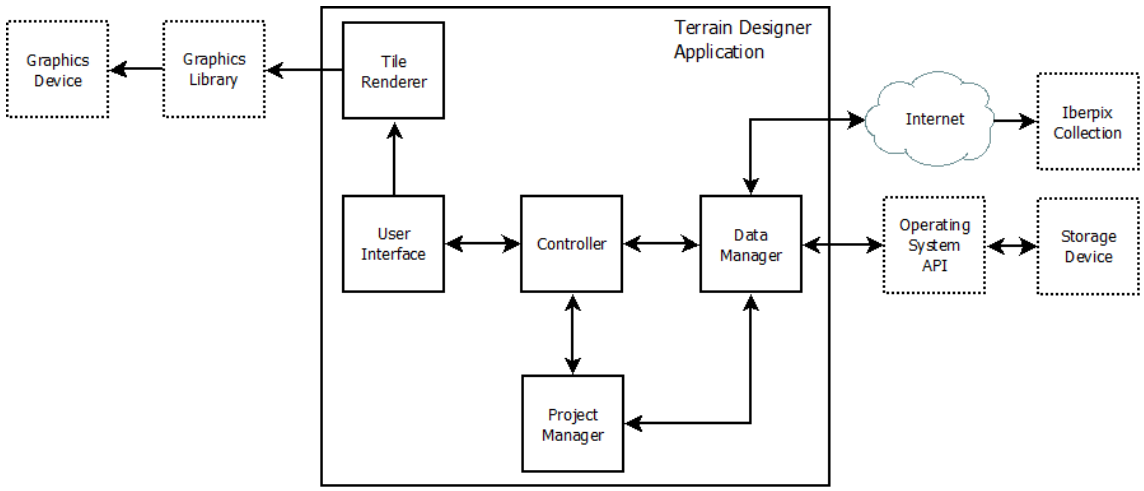


Figure 3.2.1: *Terrain Designer Block Diagram*

- ***Solution for the Terrain Renderer:***

The terrain viewer is the application in charge of displaying the terrain dataset that has been previously set-up using the terrain designer application. Terrains can be rendered using geometry tessellation alone or combining it with one of the proposed post-processing techniques.

The user can load a project and visualize the terrain from different perspectives, changing the point of view at will. If waypoints have been set on the terrain design application, the user can start a camera flight to visualize the terrain following the trajectory created by the waypoints.

To be able to take statistical data of the rendering methods, the application generates a file containing relevant statistics of the execution of the application. Besides this file, if the user runs the application and plays the camera flight, the application periodically saves screenshots of the viewport to let the user process them later.

Following the example of the terrain designer application, the first step consists on creating a high level model defining the different components of the application and how do they communicate with each other. The application components are:

- ***Controller:*** In charge of coordinating the interaction between the different components of the application.

- ***User Interface:*** The component in charge of exposing the features of the application to the user. It captures the user commands and shows the state of the application.

  On the second chapter it was stated that the technology used to build the user interface of the terrain viewer application was using the bare Windows API, which is not object oriented. The user interface component acts then as a wrapper between the object oriented application code and the non-object oriented Windows API code.

- ***Project Manager:*** It is in charge of loading the projects created with the terrain designer application, parsing the project configuration files to the structures that can be understood by the application.

- ***Terrain Renderer:*** It is the main component of the application, in charge of rendering the visualization of the active terrain project using the parameters set by the user.

  The renderer component acts as a controller for the shader, texture, tile and statistics managers, keeping track of all the data structures created by those components to create the visualization.

  To create the terrain visualization from its data, it communicates with the graphics hardware through the OpenGL libraries, as do all the following components except for the statistics manager.

- ***Shader Manager:*** It is used by the terrain renderer to set-up the programmable pipeline of the graphics card, loading and compiling the shader code using the OpenGL libraries.

  The terrain renderer component accesses the compiled shader program from the shader manager, to set-up the required variables and attributes.

- ***Texture Manager:*** The component in charge of loading the texture data from the storage device, converting it to the appropriate format and send it to the graphics device using the OpenGL libraries.

  It also performs other operations and transformations to the data which are necessary for the implementation of some of the rendering methods.

- ***Terrain Tile Manager:*** It is in charge of managing the different terrain tiles and of linking them with the project terrain tiles. When the project is being loaded, it creates the underlying geometry of each one of the tiles and sends it to the graphics device using the OpenGL libraries.

- ***Statistics Manager:*** It is the component used to collect different statistics about the performance of the application, including execution times, data loading times and memory consumption figures, to store when the application finishes running, to let the user analyze the performance of the selected rendering method.

  Besides collecting time and memory consumption statistics, it is also the component in charge of capturing the content of the rendering area when required and saving it to the storage device, to further analyze them afterwards.

The following block diagram illustrates the interaction of the different components of the system:

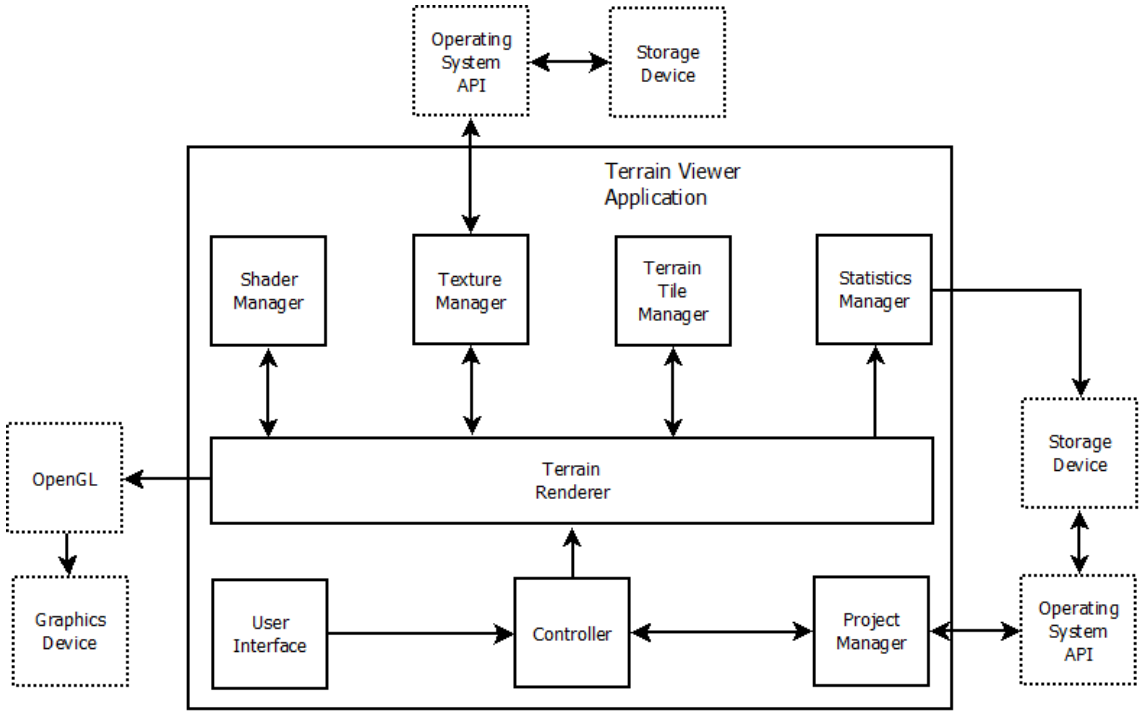


Figure 3.2.2: *Terrain Viewer Block Diagram*

3.3: Hardware and software requirements

This section describes the required hardware and software requirements needed to run the different systems, following the chosen technology on the previous chapter.

These requirements are an estimate taking into account the minimum requirements for the selected technology. After both applications are in the testing stage, the final system requirements will be disclosed.

To be able to use tessellation on the graphics card, the host system graphics processor must be able to run applications using OpenGL 4.0 or greater. Examples of hardware featuring OpenGL 4.3 support are the Nvidia Fermi platform and the AMD Evergreen platform.

Both applications are going to be developed using the 2013 version of Microsoft Visual Studio, and will target Windows for Intel 64 bit platforms. The terrain design application uses the .NET Framework version 4.5, which needs Windows Vista SP1 or greater to run, along the memory, disk and processor speed requirements noted below.

On the terrain design application, one of the features lets the user obtain terrain data from an existing online dataset. This feature be available only of the user has an active internet connection.

The summary of the preliminary system requirements of both the rendering and design systems is:

Component	***Minimum Required***
Processor	Intel (or compatible) 64-bit CPU at 1 GHz
RAM Memory	512 MB
Disk Space	2 GB
Graphics Hardware	OpenGL 4.3 / Shader Model 5 compatible graphics card (Nvidia GTX 400 series or greater, AMD Radeon 5000 series or greater)
Operating System	Windows Vista Service Pack 1
Internet Connection	Optional

Table 3.2: *Preliminary system requirements for both applications.*

3.4: Cost estimation

In this section, the objective is introducing the economical and time costs involved with the development of the project. To achieve this, the different steps of the project have been divided into tasks following the project requirements.

Using these tasks, a WBS (Work Breakdown Structure) has been developed. This WBS defines the base tasks for the development of the project and the time needed to complete the project will be the time needed to complete them.

The tasks on the WBS are further subdivided, creating a set of shorter tasks which will be the ones developed on the project. The time needed to complete each task will be specified by the expert committee.

- ***Time Estimation:***

In order to estimate the time required to complete the project, each one of the tasks deduced from the WBS need an estimated completion time. To obtain these times, an expert committee consisting of three experts on the matter is asked to make an estimation of the time needed to complete these tasks, taking into account the scope of the project and its requirements.

To create the expert committee, a survey was made looking for experts who may want to participate on the committee. All the possible experts have considerable experience in the field, and their opinion is considered highly valuable.

The following experts have agreed to participate in the project task judgment:

- **Mariano Pérez Martínez:** Ph. D. on computer science and researcher professor at the University of Valencia.
- **Ignacio García Fernández:** Ph. D. on computer science by the University of Valencia and researcher professor at the University of Valencia.
- **Angel Rodríguez Cerro:** Member of the *LSyM* at the *IRTIC* research institute of the University of Valencia.

The times indicated by the expert committee are grouped in three categories: optimistic times, pessimistic times and most probable times. Optimistic times represent the shorter time for a given task, pessimistic times represent the longer time for a given task and most probable times represent the most occurring time.

Once each one of the three experts have decided an optimistic, pessimistic and most probable time for each task, the next step is calculating the estimated time for each one of the tasks.

To do this, the chosen option is using a beta distribution, where the most probable time has more weight than the pessimistic and optimistic times, implying that there is a strong confidence on the most probable time given by the experts.

$$t_{expert} = \frac{t_{optimistic} + t_{pessimistic} + 4 \cdot t_{most\ prob.}}{6}$$

Formula 3.3.1: *Task time for each one of the committee experts.*

Finally, from the estimated task times for each one of the experts, the final task times for the project is estimated. To do this, the chosen option is using a triangular distribution, which means calculating the mean from all the times given by the experts.

Using a triangular distribution implies that there is equal confidence on the values given by each one of the experts, since the times are weighted equally.

$$t_{task} = \frac{t_{optimistic} + t_{pessimistic} + t_{most\ prob.}}{3}$$

Formula 3.3.2: *Final expected task times.*

With these final task times, the final structure of the project tasks will be developed, showing the expected time for each of them using a Gantt diagram, and the expenses needed to develop the project will be calculated using these times.

The project starting time is the January 10th, 2015. The project should be finished at approximately the end of the Q2 of 2015.

The development cycle selected for the project is a traditional development cycle, where the tasks are resolved in series and one at a time. This development cycle is easier to manage but a delay in one of the tasks will delay the completion of the whole project.

Following are the times that each one of the experts has estimated for the project tasks.

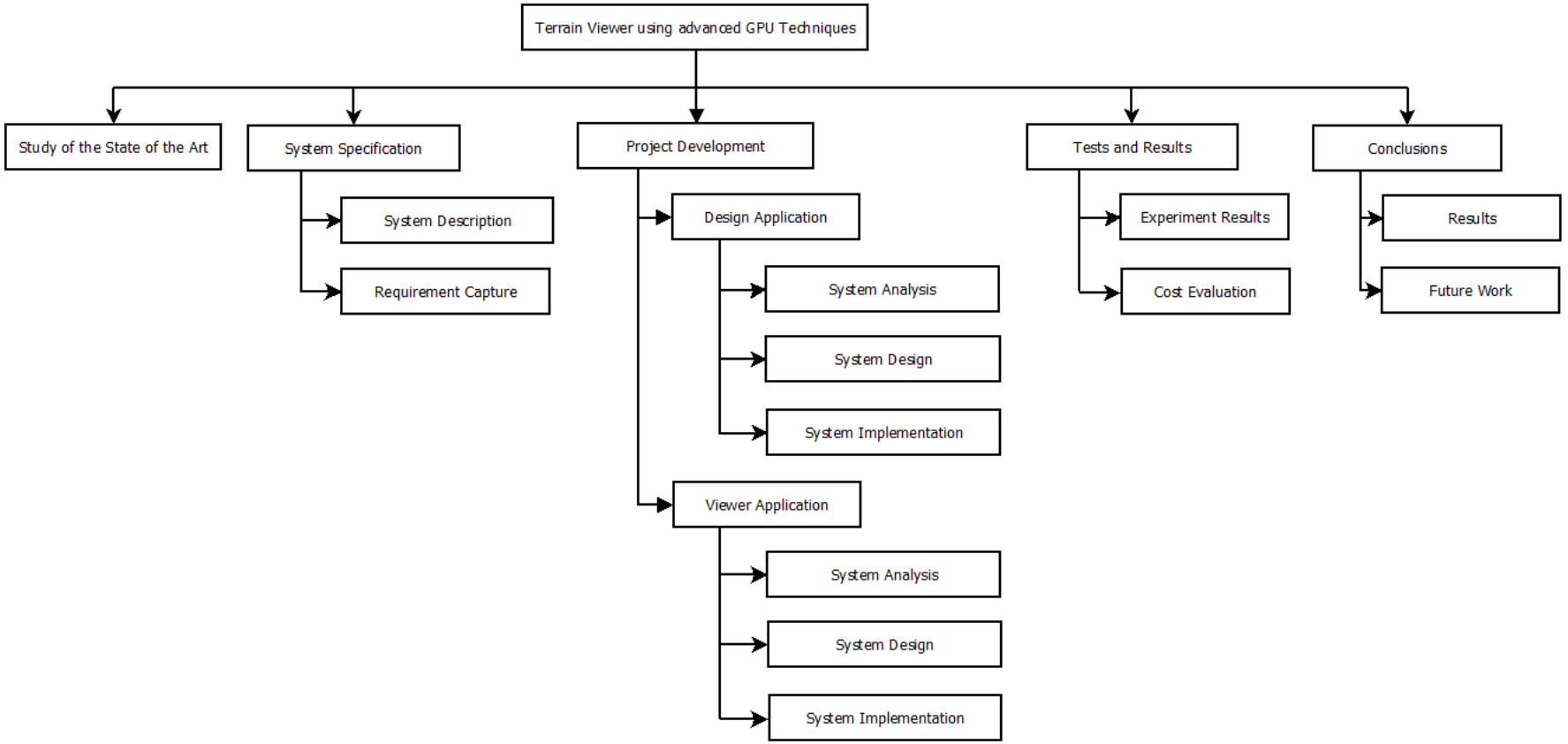


Figure 3.4.1: *Work Breakdown Structure of the project.*

- ***Expert #1, Mariano Pérez Martínez:***

The expected task times estimated by the first expert, Mariano Pérez Martínez are:

Task Name	***Optimistic Time***	***Most Probable Time***	***Pessimistic Time***	***Task time (β distribution)***
1. Study of the State of the Art	7 days	10 days	12 days	11 days
2. System Definition	2 days	4 days	5 days	4 days
2.1. Requirement Capture	1 day	1 day	2 days	2 days
2.2. Cost Estimation	1 day	1 day	2 days	2 days
3. Analysis of the Designer Application	5 days	7 days	9 days	8 days
3.1. Analysis of the Viewer Application	8 days	12 days	15 days	13 days
4. Design of the Designer Application	5 days	7 days	9 days	8 days
4.1. Design of the Viewer Application	8 days	12 days	15 days	13 days
5. Implementation of the Designer Application	7 days	10 days	12 days	11 days
5.1. Implementation of the Data Structures	4 days	5 days	6 days	5 days
5.2. Implementation of the Renderer (Viewer Application)	5 days	7 days	9 days	8 days
5.3. Implementation of the Tessellation Rendering Method (Viewer Application)	4 days	6 days	8 days	7 days
5.4. Implementation of the Post-processing Effects (Viewer Application)	5 days	7 days	9 days	8 days
6. Result Processing	5 days	7 days	9 days	8 days
7. Writing of the documentation	15 days	20 days	25 days	22 days

Table 3.3: *Estimated task times by the expert #1.*

- ***Expert #2, Ignacio García Fernández:***

The expected task times estimated by the second expert, Ignacio García Fernández are:

Task Name	*Optimistic Time*	*Most Probable Time*	*Pessimistic Time*	*Task time (β distribution)*
8. Study of the State of the Art	10 days	15 days	20 days	15 days
9. System Definition	4 days	6 days	8 days	6 days
9.1. Requirement Capture	3 day	5 day	7 days	5 day
9.2. Cost Estimation	1 day	1 day	1 days	1 day
10. Analysis of the Designer Application	6 days	7 days	8 days	7 days
10.1. Analysis of the Viewer Application	4 days	5 days	6 days	5 days
11. Design of the Designer Application	2 days	3 days	4 days	3 days
11.1. Design of the Viewer Application	2 days	3 days	4 days	3 days
12. Implementation of the Designer Application	20 days	25 days	30 days	25 days
12.1. Implementation of the Data Structures	4 days	6 days	8 days	6 days
12.2. Implementation of the Renderer (Viewer Application)	15 days	20 days	25 days	20 days
12.3. Implementation of the Tessellation Rendering Method (Viewer Application)	10 days	12 days	14 days	12 days
12.4. Implementation of the Post-processing Effects (Viewer Application)	4 days	6 days	8 days	6 days
13. Result Processing	10 days	15 days	20 days	15 days
14. Writing of the documentation	20 days	25 days	30 days	25 days

Table 3.4: *Estimated task times by the expert #2.*

- ***Expert #3, Angel Rodríguez Cerro:***

The expected task times estimated by the third expert, Angel Rodríguez Cerro are:

Task Name	*Optimistic Time*	*Most Probable Time*	*Pessimistic Time*	*Task time (β distribution)*
15. Study of the State of the Art	20 days	40 days	60 days	40 days
16. System Definition	5 days	10 days	15 days	10 days
16.1. Requirement Capture	7 day	14 day	21 days	14 day
16.2. Cost Estimation	3 day	7 day	10 days	7 day
17. Analysis of the Designer Application	5 days	8 days	14 days	8 days
17.1. Analysis of the Viewer Application	8 days	12 days	18 days	12 days
18. Design of the Designer Application	15 days	22 days	30 days	22 days
18.1. Design of the Viewer Application	15 days	22 days	30 days	22 days
19. Implementation of the Designer Application	20 days	30 days	40 days	30 days
19.1. Implementation of the Data Structures	4 days	7 days	14 days	8 days
19.2. Implementation of the Renderer (Viewer Application)	15 days	20 days	30 days	20 days
19.3. Implementation of the Tessellation Rendering Method (Viewer Application)	7 days	10 days	14 days	10 days
19.4. Implementation of the Post-processing Effects (Viewer Application)	10 days	17 days	25 days	17 days
20. Result Processing	10 days	20 days	30 days	20 days
21. Writing of the documentation	20 days	24 days	30 days	24 days

Table 3.5: *Estimated task times by the expert #3.*

With the time estimation for the three experts, the final task times are calculated using a triangular estimation. These times will be used to build the final Gantt diagram, representing the evolution of the project through the time.

The following table shows the final task times of the project and next is the Gantt diagram of the project. As can be observed on the diagram, the estimated project finish date is the September 23rd, 2015.

Task Name	***Task time (Triangular distribution)***
1. Study of the State of the Art	22 days
2. System Definition	7 days
2.1. Requirement Capture	7 day
2.2. Cost Estimation	3 day
3. Analysis of the Designer Application	8 days
3.1. Analysis of the Viewer Application	10 days
4. Design of the Designer Application	11 days
4.1. Design of the Viewer Application	13 days
5. Implementation of the Designer Application	22 days
5.1. Implementation of the Data Structures	6 days
5.2. Implementation of the Renderer (Viewer Application)	16 days
5.3. Implementation of the Tessellation Rendering Method (Viewer Application)	10 days
5.4. Implementation of the Post-processing Effects (Viewer Application)	10 days
6. Result Processing	14 days
7. Writing of the documentation	24 days

Table 3.6: *Estimated task duration.*

Id	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		Study of the State of the Art	22 días	lun 12/01/15	mar 10/02/15	
2		**System Specification**	**17 días**	**mié 11/02/15**	**jue 05/03/15**	1
3		System Definition	7 días	mié 11/02/15	jue 19/02/15	
4		Requirement Captu	7 días	vie 20/02/15	lun 02/03/15	3
5		Cost Estimation	3 días	mar 03/03/15	jue 05/03/15	4
6		**System Analysis**	**18 días**	**vie 06/03/15**	**mar 31/03/15**	2
7		Analysis of the Designer	8 días	vie 06/03/15	mar 17/03/15	
8		Analysis of the Viewer Application	10 días	mié 18/03/15	mar 31/03/15	7
9		**System Design**	**24 días**	**mié 01/04/15**	**lun 04/05/15**	6
10		Design of the Designer	11 días	mié 01/04/15	mié 15/04/15	
11		Design of the Viewer Application	13 días	jue 16/04/15	lun 04/05/15	10
12		**System Implementati**	**64 días**	**mar 05/05/15**	**vie 31/07/15**	9
13		Implementation of the Designer	22 días	mar 05/05/15	mié 03/06/15	
14		Implementation of the Data	6 días	jue 04/06/15	jue 11/06/15	13
15		Implementation of the Renderer (Viewer	16 días	vie 12/06/15	vie 03/07/15	14
16		Implementation of the Tessellation Rendering Method (Viewer	10 días	lun 06/07/15	vie 17/07/15	15
17		Implementation of the Post-processing	10 días	lun 20/07/15	vie 31/07/15	16
18		Result Processing	14 días	lun 03/08/15	jue 20/08/15	12
19		Writing of the Documentation	24 días	vie 21/08/15	mié 23/09/15	18

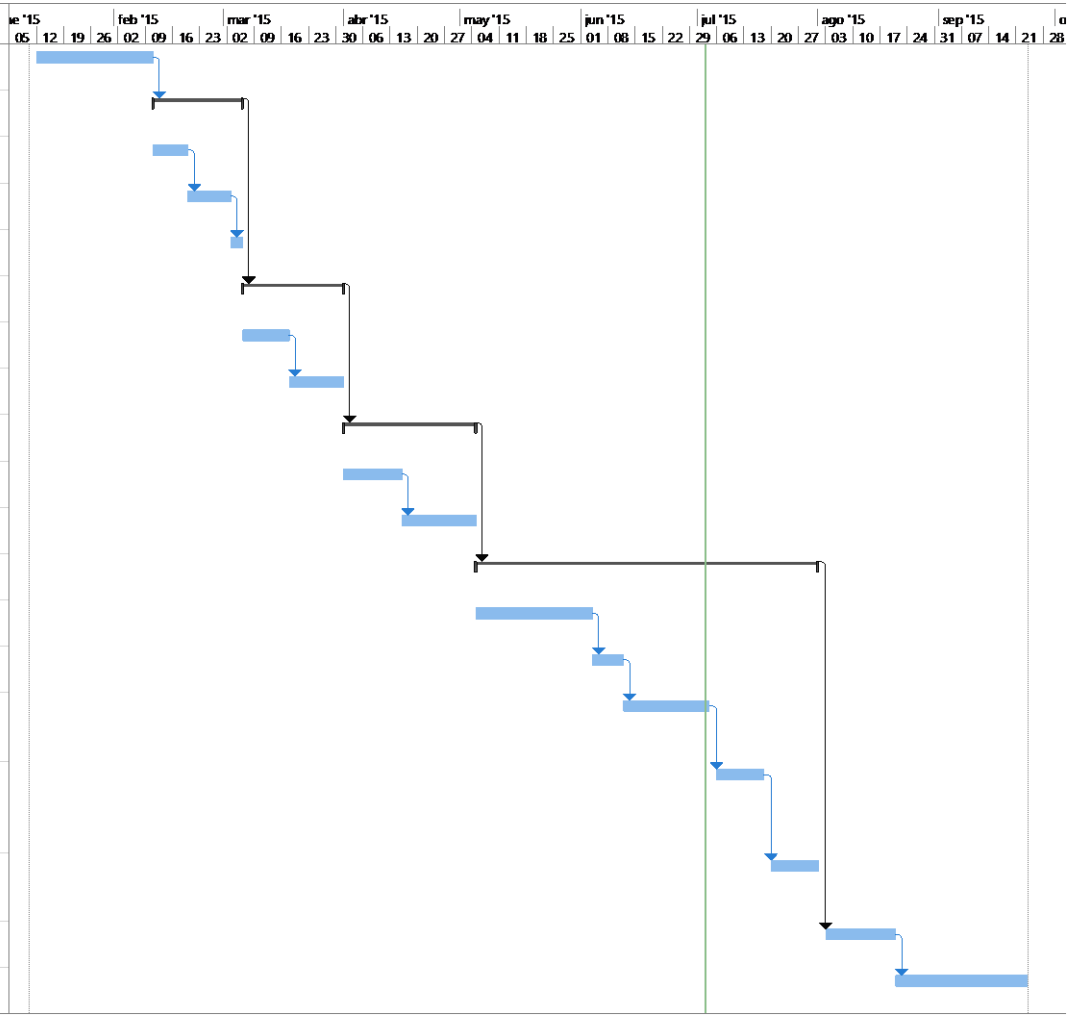


Figure 3.4.2: *Gantt diagram of the project.*

- ***Economic Cost Estimation***

Once the time cost estimation is complete, the next step is the economic cost estimation. This estimation takes into account the personnel and material resources used in the different tasks to obtain an estimation of the economic cost of the project.

  - ***Personnel Resources***

Personnel resources are the people and systems that intervene on the completion of the tasks. In this project, the single personnel resource is the developer in charge of completing the tasks of the project.

In the year 2011, the Official College of Computer Science Engineers of Valencia published a report on salary and working trends of Spanish CS engineers, where it was stated that the most occurring early salary before taxes was between 20000€ and 30000€. Using this data, the selected cost for the resource was the mean salary on the most occurring range, **25000€ / year**.

Taking into account that each year has 12 months, with 4 weeks and 40 working hours per week, the cost of the personnel resource is **13€ / hour**. The project spans a total of 183 days, which translate into **1464 hours**, using 8 working hour days. The cost of the developer then is **19032€**.

  - ***Material Resources***

On the other hand, material resources are elements used by the personnel to complete the project. These resources can be divided into software resources and hardware resources.

Many of the software resources do not represent a direct cost on the project, since the project can take profit of licenses available for academic environments at no cost.

To calculate the amortization amount of the material resources, the hardware resources are considered to have an amortization time of 4 years and the software resources have an amortization time of 3 years.

The following table shows the different material resources used in the project, along with their economic cost.

Hardware Materials		
Name	***Cost***	***Amortization***
Laptop Computer (*MSI GP60)*	719€	12%
Software Materials		
Name	***Cost***	***Amortization***
Operating System (*Microsoft Windows 8.1 Complete Version)*	119€	17%
Microsoft Visual Studio 2010 Professional (*University of Valencia Student License*)	0€	17%
Visual Paradigm 11.2 (*University of Valencia Team License*)	0€	17%
Microsoft Visual Studio 2013 Express	0€	17%
Microsoft Office 365 Student Version	79€	17%
Microsoft Project 2013 Standard Version	769€	17%

Table 3.7: *Material resource costs of the project.*

- ***Total Economic Cost***

Once all the resource costs have been estimated, the total economic cost of the project can be obtained. The total cost is the sum of the personnel resource cost, the material resources costs and the associated taxes. The project has a total cost of **25068,78€.**

The following table shows the total cost of the project, before and after the application of taxes.

Element	***Cost***
Personnel Resources	19032€
Material Resources	1686€
Total before taxes	20718€
Total after taxes (21%)	**25068,78€**

Table 3.8: *Total cost estimation of the project.*

3.5: Summary

In this chapter, the objectives of the project were analyzed to create a requirement list, which will be used to base the design of the system. The economic and time cost of the project was estimated, obtaining a time cost of 183 days, spanning from January 10th to September 23rd of 2015, and an economic cost of 25068,78€ after taxes.

In the next chapter, the system requirements will be analyzed to create a high level model of the application through the use of use cases. This will be the basis for the design of the system.

CHAPTER 4: SYSTEM ANALYSIS

This chapter describes the system analysis stage, where the system requirements are analyzed to obtain a high level representation of the application using use case models, based on the behavior of the system towards the user input.

Taking the system requirements as the reference, the use cases are modeled for both subsystems. Use cases will help to do a thorough analysis taking the interaction between the systems and the user as the point of view.

For the development of the use case models, the same principle as on the development of the system requirements is followed: both subsystems, the terrain viewer and the terrain designer, have its own use case models.

For each one of the system operations, an operation contract is defined. The contract describes relevant information about the behavior of the operation, specifying its responsibilities, inputs, outputs, exceptions, pre-conditions and post-conditions.

4.1: Viewer System Use Case Model

Following is the use case model for the terrain viewer subsystem, with the description of each one of the use cases. The model has been designed to fit each one of the requirements as an independent use case, since all were considered mandatory.

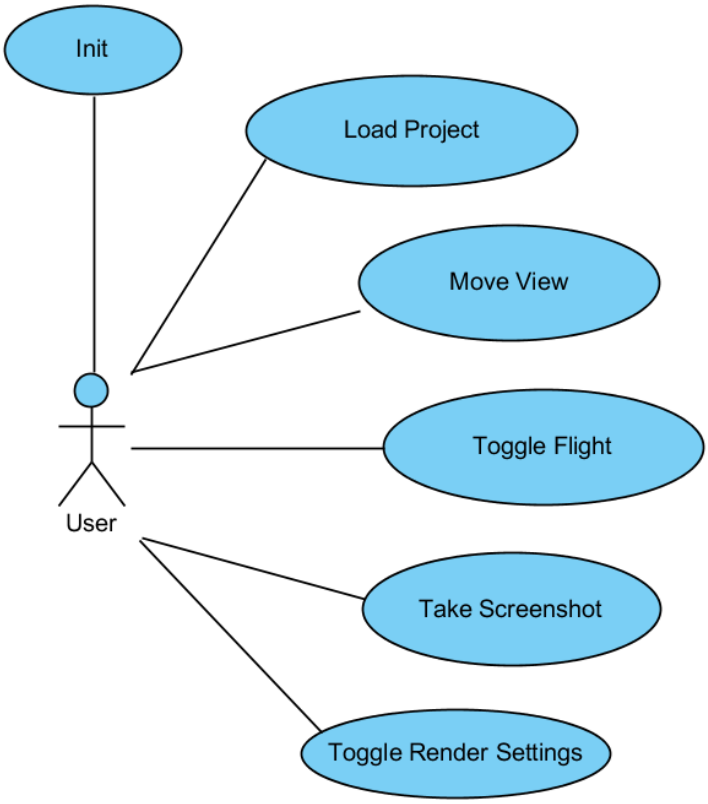


Figure 4.1.1: *Use case diagram of the terrain viewing system.*

The behavior of each use case is described next. To explain their behavior, a sequence diagram illustrating the flow of events of the use case is included. At the end of this section is included a table including the table numbers for every use case.

- ***Init use case:***

This use case describes the behavior of the system when the user first loads the application. This use case only happens the first time the application is launched.

The system must load all the required subsystems and allocate all the needed resources for the correct working of the application.

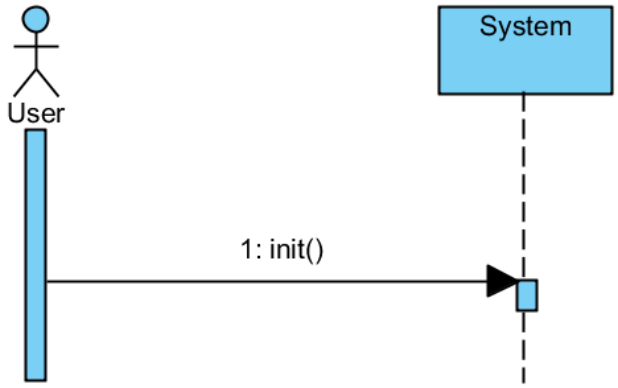


Figure 4.1.2: *Sequence diagram for the "Init" use case.*

Name	**Init**		
Description	This use case describes the behavior of the system when the user first opens the application.		
Preconditions	The application must have not been opened yet.		
Post-conditions	The application subsystems will be loaded.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user opens the application.	
	2		The system starts up all the required subsystems.

Table 4.1.1: *Use case details for "Init".*

Responsibilities	The operation is in charge of loading the required subsystems for the application, namely the user interface and the terrain renderer.
Type	System.
References	Use case: Init.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have not been opened yet.
Post-Conditions	The application will be loaded.

Table 4.1.2: *Contract of the terrain viewer "init()" system operation.*

- ***Load Project use case:***

This use case describes the behavior of the system when the user commands the application to load a project. This use case can happen in any moment after the application has loaded, since the user must be able to load a new project in any moment.

The system must let the user choose a project file from the disk, and handle any errors that might appear during the loading. If there is an error when loading the terrain project, the application will return to its initial state. This use case is directly related to the requirement *Func-View-5*.

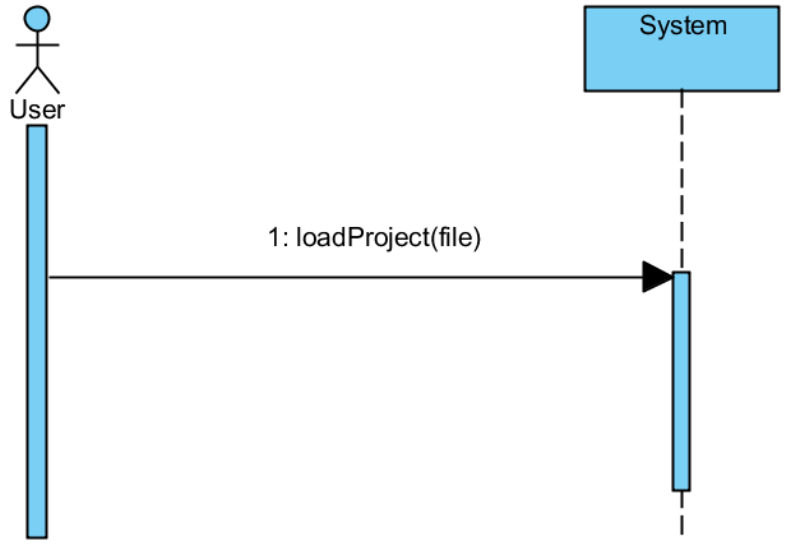


Figure 4.1.3: *Sequence diagram for the "Load Project" use case.*

<table>
<tr><td>Name</td><td colspan="3">Load Project</td></tr>
<tr><td>Description</td><td colspan="3">This use case describes the system behavior when the user commands the application to load a project.</td></tr>
<tr><td>Preconditions</td><td colspan="3">The application must have finished loading.</td></tr>
<tr><td>Post-conditions</td><td colspan="3">If the loading succeeds, the project will be set as the current active project and the terrain will be rendered on the screen.</td></tr>
<tr><td>Flow of Events</td><td></td><td>User Input</td><td>System Response</td></tr>
<tr><td></td><td>1</td><td>This use case starts when the user commands the application to load a project.</td><td></td></tr>
<tr><td></td><td>2</td><td></td><td>The application starts the project load process for the specified project file.</td></tr>
</table>

Table 4.1.3: *Use case details for "Load Project".*

Responsibilities	The operation is in charge of creating the project instance, loading all the required data from the project file and creating the texture instances of each terrain tile.
Type	System.
References	Use case: Load Project.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The terrain project will be loaded.

Table 4.1.4: *Contract of the terrain viewer "loadProject()" system operation.*

- ***Move View use case:***

This use case describes the behavior of the system when the user commands the application to move the point of view of the camera. This use case can happen in any moment after the application has finished loading.

The application must register when the user clicks the mouse button, track the changes on its position and modify the camera angle consequently. This use case is directly related to the requirement *Func-View-2*.

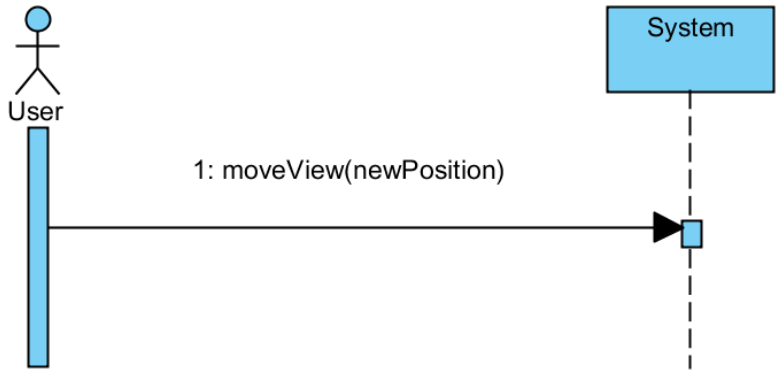


Figure 4.1.4: *Sequence diagram for the "Move View" use case.*

<table>
<tr><td>Name</td><td colspan="3">Move View</td></tr>
<tr><td>Description</td><td colspan="3">This use case describes the behavior of the system when the user changes the point of view of the camera to render the terrain from a different projection.</td></tr>
<tr><td>Preconditions</td><td colspan="3">The application must not be in flight mode.</td></tr>
<tr><td>Post-conditions</td><td colspan="3">The camera's point of view will be changed.</td></tr>
<tr><td>Flow of Events</td><td></td><td>User Input</td><td>System Response</td></tr>
<tr><td></td><td>1</td><td>This use case starts when the user commands the application to change the point of view of the camera.</td><td></td></tr>
<tr><td></td><td>2</td><td></td><td>The application calculates the new rotation of the camera using the mouse coordinates and changes the point of view of the camera.</td></tr>
</table>

Table 4.1.5: *Use case details for "Move View".*

Responsibilities	The operation is in charge of changing the position of the viewport camera on the renderer.
Type	System.
References	Use case: Move View.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The position and target of the camera will be changed.

Table 4.1.6: *Contract of the terrain viewer "moveView()" system operation.*

- ***Toggle Flight use case:***

This use case describes the behavior of the system when the user commands the application to start or stop the camera flight. This can only happen when a project is loaded, and when it has enough waypoints to do a camera flight. This use case is directly related to the requirement *Func-View-3*.

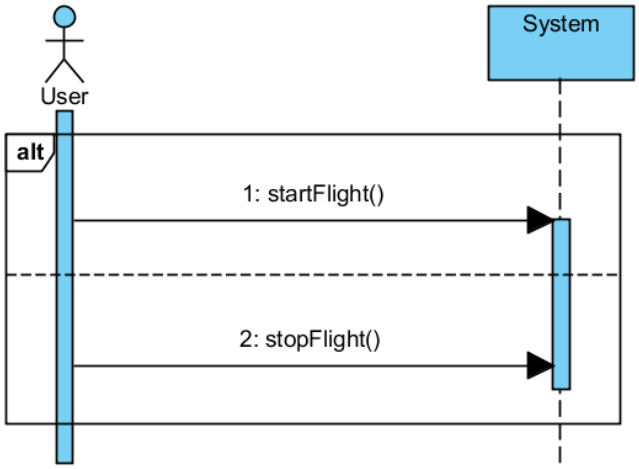


Figure 4.1.5: *Sequence diagram for the "Toggle Flight" use case.*

Name	**Move View (Start Branch)**		
Description	This use case describes the behavior of the system when the user commands the application to toggle the camera flight from inactive to active.		
Preconditions	A project must have been loaded. The project must have a flyable set of way-points. The application must not be in flight mode.		
Post-conditions	If the process succeeds, the camera will start following the project's way-points.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to start the camera flight.	
	2		The camera starts following the way-points.

Table 4.1.7: *Use case details for "Toggle Flight" start branch.*

Responsibilities	The operation is in charge of starting the camera flight animation.
Type	System.
References	Use case: Toggle Flight.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active project with enough camera flight animation waypoints.
Post-Conditions	The camera will start following the waypoints.

Table 4.1.8: *Contract of the terrain viewer "startFlight()" system operation.*

Name	**Move View (Stop branch)**		
Description	This branch describes the system behavior when the user commands the application to toggle the camera flight from active to inactive.		
Preconditions	The application must be in flight mode.		
Post-conditions	If the process succeeds, the camera will stop following the project's way-points.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to stop the camera flight.	
	2		The camera stops following the way-points.

Table 4.1.9: *Use case details for "Toggle Flight" stop branch.*

Responsibilities	The operation is in charge of stopping the camera flight animation.
Type	System.
References	Use case: Toggle Flight.
Exceptions	None.
Output	None.
Pre-Conditions	The camera flight animation must be active.
Post-Conditions	The camera will stop following the waypoints.

Table 4.1.10: *Contract of the terrain viewer "stopFlight()" system operation.*

- ***Take Screenshot use case:***

This use case describes the behavior of the system when the user commands the application to take a screenshot of the renderer. This can happen in any moment after the application has finished loading.

The system must inform the user if there is any kind of error that prevents the application from saving the image. This use case is directly related to the requirement *Func-View-1*.

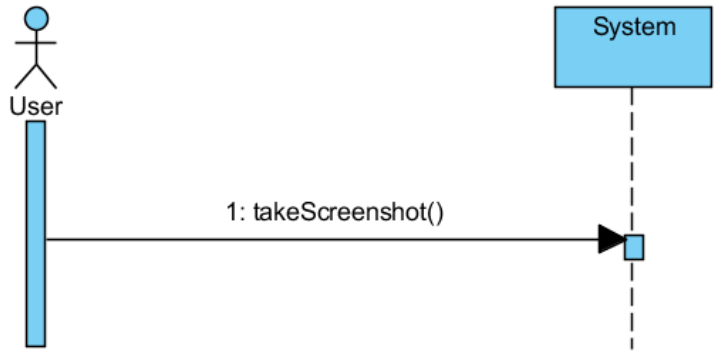


Figure 4.1.6: *Sequence diagram for the "Take Screenshot" use case.*

<table>
<tr><td>Name</td><td colspan="2">Take Screenshot</td></tr>
<tr><td>Description</td><td colspan="2">This use case describes the behavior of the system when the user commands the application to take a screenshot.</td></tr>
<tr><td>Preconditions</td><td colspan="2">The application must have finished loading.</td></tr>
<tr><td>Post-conditions</td><td colspan="2">If the process succeeds, a screenshot of the rendered scene will be saved.</td></tr>
<tr><td>Flow of Events</td><td colspan="2"><table>
<tr><th></th><th>User Input</th><th>System Response</th></tr>
<tr><td>1</td><td>This use case starts when the user commands the application to take a screenshot.</td><td></td></tr>
<tr><td>2</td><td></td><td>The application saves a copy of the contents of the renderer into memory into a file on the project root folder with a unique name.</td></tr>
</table></td></tr>
</table>

Table 4.1.11: *Use case details for "Take Screenshot"*

.

Responsibilities	The operation is in charge of taking a screenshot of the renderer and saving it to the storage device.
Type	System.
References	Use case: Take Screenshot.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The contents of the renderer viewport will be saved to the storage device.

Table 4.1.12: *Contract of the terrain viewer "takeScreenshot()" system operation.*

- ***Toggle Render Settings use case:***

This use case describes the behavior of the system when the user commands the application to change the render settings. This can happen any time after the application has finished loading.

The user may change the triangle render mode (wireframe or fill), the management of the level-of-detail, the maximum level-of-detail and the minimum level-of-detail distance. The application must be able to change each one of these values appropriately. This use case directly relates to the requirement *Func-View-4*.

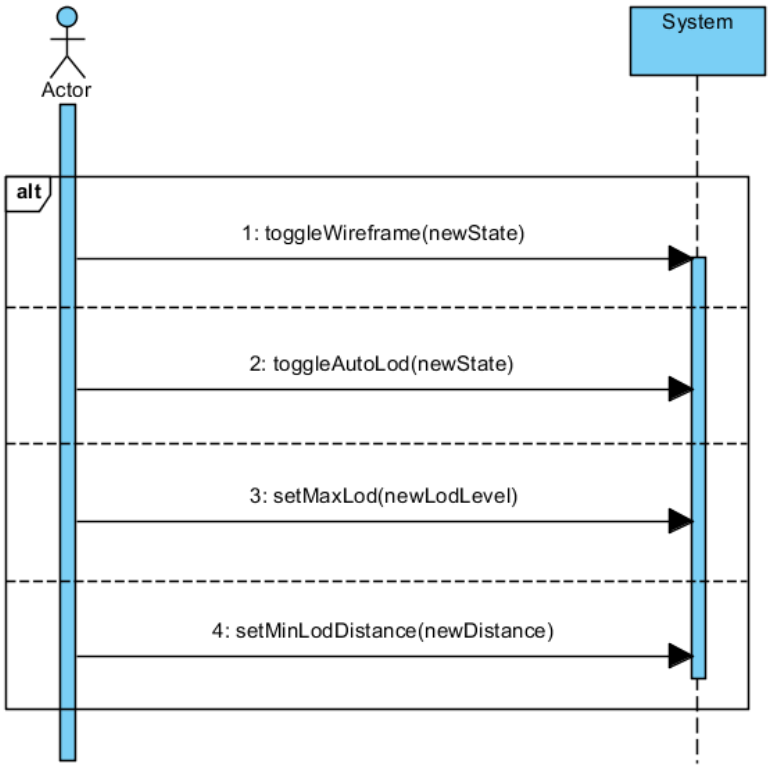


Figure 4.1.7: *Sequence diagram for the "Toggle Render Settings" use case.*

Name	**Toggle Render Settings (Wireframe branch)**		
Description	This use case describes the behavior of the system when the user commands the application to toggle one of the render settings. In this branch the wireframe mode state is changed.		
Preconditions	The application must have finished loading.		
Post-conditions	The status of the wireframe mode will be changed.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to change the wireframe mode state.	
	2		The system changes the wireframe mode to the specified new state.

Table 4.1.13: *Use case details for "Toggle Render Settings" wireframe branch.*

Responsibilities	The operation is in charge of changing the wireframe state of the renderer.
Type	System.
References	Use case: Toggle Render Settings.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The wireframe state will be changed.

Table 4.1.14: *Contract of the terrain viewer "toggleWireframe()" system operation.*

<table>
<tr><td>Name</td><td colspan="3">Toggle Render Settings (Automatic LOD branch)</td></tr>
<tr><td>Description</td><td colspan="3">This branch describes the behavior of the system when the user commands the application to change the status of the automatic level of detail mode.</td></tr>
<tr><td>Preconditions</td><td colspan="3">The application must have finished loading.</td></tr>
<tr><td>Post-conditions</td><td colspan="3">The status of the automatic level of detail mode will be changed.</td></tr>
<tr><td>Flow of Events</td><td></td><td>User Input</td><td>System Response</td></tr>
<tr><td></td><td>1</td><td>This use case starts when the user commands the application to change the automatic LOD status.</td><td></td></tr>
<tr><td></td><td>2</td><td></td><td>The system changes the automatic LOD mode status to the new specified status.</td></tr>
</table>

Table 4.1.15: *Use case details for "Toggle Render Settings" automatic LOD branch.*

Responsibilities	The operation is in charge of changing the automatic management of the LOD of the renderer.
Type	System.
References	Use case: Toggle Render Settings.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The automatic LOD state will be changed.

Table 4.1.16: *Contract of the terrain viewer "toggleAutoLod()" system operation.*

<table>
<tr><td>Name</td><td colspan="3">Toggle Render Settings (Maximum LOD change branch)</td></tr>
<tr><td>Description</td><td colspan="3">This branch describes the behavior of the system when the user commands the application to change the maximum level of detail.</td></tr>
<tr><td>Preconditions</td><td colspan="3">The application must have finished loading.</td></tr>
<tr><td>Post-conditions</td><td colspan="3">If the process succeeds, the maximum level of detail will be changed.</td></tr>
<tr><td>Flow of Events</td><td></td><td>User Input</td><td>System Response</td></tr>
<tr><td></td><td>1</td><td>This use case starts when the user commands the application to change the maximum level of detail.</td><td></td></tr>
<tr><td></td><td>2</td><td></td><td>The system checks if the new level of detail is inside the allowed range. If it is, the new level of detail is set.</td></tr>
</table>

Table 4.1.17: *Use case details for "Toggle Render Settings maximum LOD level branch.*

Responsibilities	The operation is in charge of setting the maximum level of detail of the renderer to the specified value.
Type	System.
References	Use case: Toggle Render Settings.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The maximum LOD will be changed.

Table 4.1.18: *Contract of the terrain viewer "setMaxLod()" system operation.*

Name	**Toggle Render Settings (Minimum LOD distance change branch)**		
Description	This branch describes the behavior of the system when the user commands the application to change the minimum distance automatic level of detail should start working.		
Preconditions	The application must have finished loading.		
Post-conditions	If the process succeeds, the minimum level of detail distance will be changed.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to change the minimum LOD distance.	
	2		The system checks if the new minimum level of detail is inside the available range. If it is, the new minimum LOD distance is set.

Table 4.1.19: *Use case details for "Toggle Render Settings minimum LOD distance branch.*

Responsibilities	The operation is in charge of setting the minimum level of detail distance for automatic LOD of the renderer to the specified value.
Type	System.
References	Use case: Toggle Render Settings.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The minimum LOD distance will be changed.

Table 4.1.20: *Contract of the terrain viewer "setMinLodDistance()" system operation.*

4.2: Designer System Use Case Model

Next is the use case model for the terrain design subsystem, with the descriptions of all use cases. Since in this case there are both mandatory and optional requirements, translating them into use cases is not as simple.

All mandatory requirements have been translated into use cases, as in the previous system, optional requirements however have been translated taking a compromise between the proposed requirement and the available resources, trying to obtain the best possible scenario.

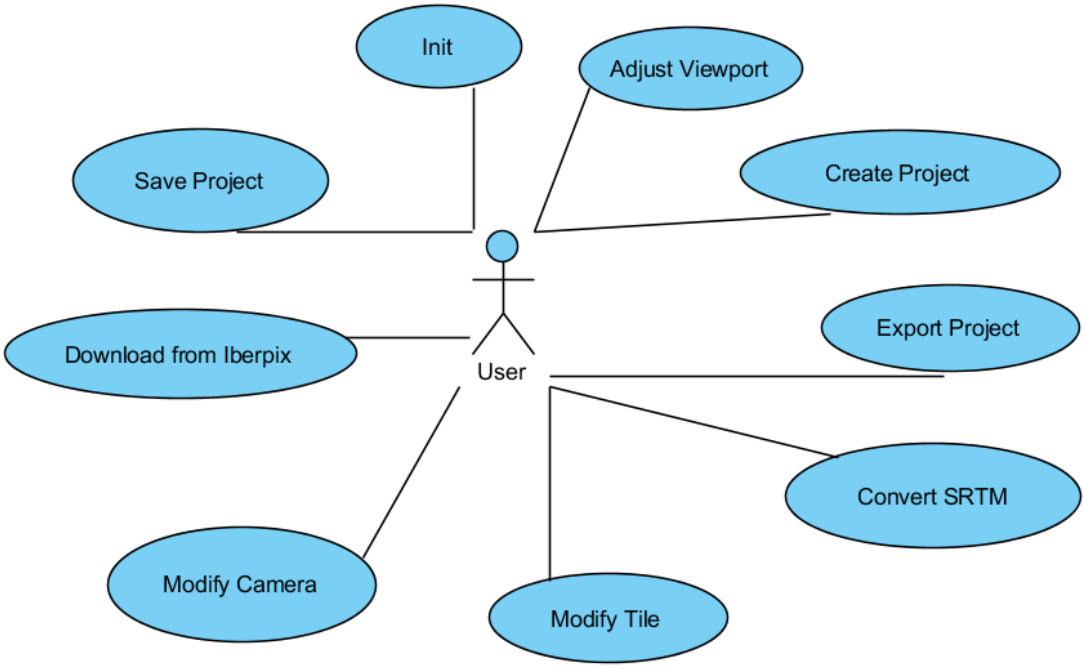


Figure 4.2.1: *Use case diagram of the terrain design system.*

In the same fashion as in the terrain viewing system, each case use is described following, including a sequence diagram to illustrate its flow of events, and includes a table with the relations with their complete descriptions on the annex.

- ***Download from Iberpix use case:***

This use case describes the behavior of the system when the user commands the application to download a tile set from the "*Instituto Geográfico Nacional"* Iberpix data collection.

The Iberpix dataset has been chosen to fulfill the requirement *Func-Design-4*, since it represents a good starting point to obtain GIS data to build a terrain project.

This use case can happen at any moment after the application has started, but it is mandatory to have an active Internet connection. The application must take all necessary parameters from the user (initial latitude and longitude coordinates, final latitude and longitude coordinates and the requested layer), and show the progress of the download for every file, and inform the user if there is an error.

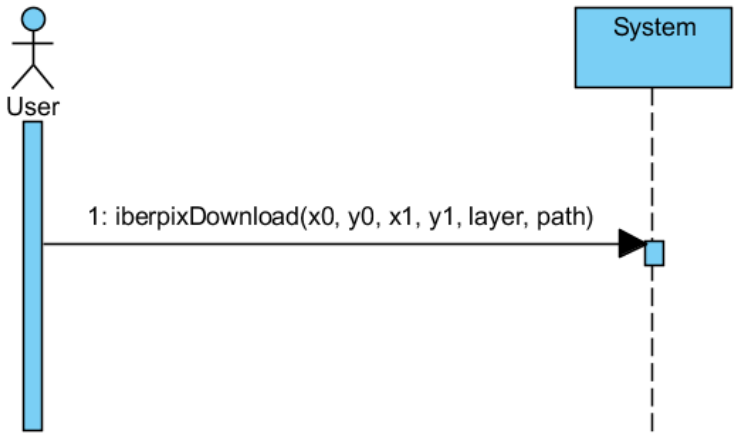


Figure 4.2.2: *Sequence diagram for the "Download from Iberpix" use case.*

Responsibilities	The operation is in charge of downloading the selected range of data from the Iberpix dataset and saving it to the storage device.
Type	System.
References	Use case: Download from Iberpix.
Exceptions	None.
Output	None.
Pre-Conditions	There application must have finished loading.
Post-Conditions	If the data is available for download, it will be saved on the target directory.

Table 4.2.1: *Contract of the terrain designer "iberpixDownload()" system operation.*

Name	**Download from Iberpix**		
Description	This use case describes the behavior of the system when the user commands the application to download a tile set from the Iberpix dataset.		
Preconditions	The application must have finished loading. There must be an active Internet connection.		
Post-conditions	If the process succeeds, the dataset will be downloaded on the directory specified by the user.		
Flow of Events		***User Input***	***System Response***
	1	This user starts when the user commands the application to download an Iberpix set with initial coordinates, final coordinates, requested layer and destination folder.	
	2		The system downloads each one of the files from the server and stores them on the destination folder.

Table 4.2.2: *Use case details for "Download from Iberpix".*

- ***Save Project use case:***

This use case describes the behavior of the system when the user commands the application to save the current state of the project to a file. This can happen at any moment after the application has started and there is an active project.

The system will prompt the user to input the destination of the project file. If the save process succeeds, the application will save a copy of the state of the project into the disk, and if there is any kind of error the user will be notified. This use case is relevant to fulfill the requirement *Func-Design-1*.

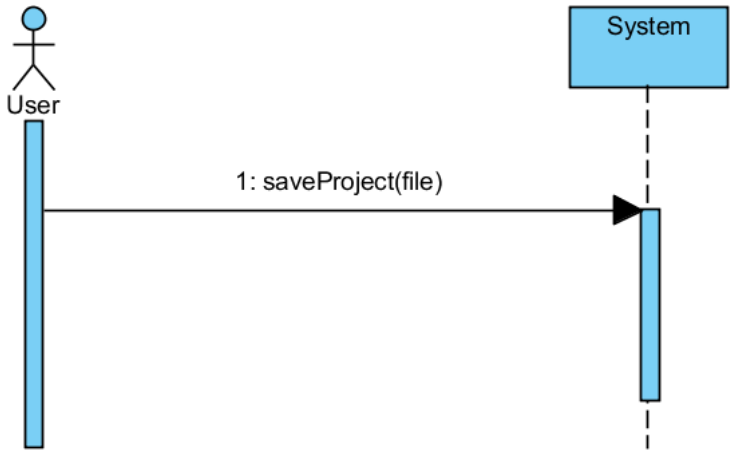


Figure 4.2.3: *Sequence diagram for the "Save Project" use case.*

<table>
<tr><td>Name</td><td colspan="3">Save Project</td></tr>
<tr><td>Description</td><td colspan="3">This use case describes the behavior of the system when the user commands the application to save the current state of the project to a file.</td></tr>
<tr><td>Preconditions</td><td colspan="3">The application must have finished loading and there must be an active project.</td></tr>
<tr><td>Post-conditions</td><td colspan="3">If the process succeeds, the project will be saved.</td></tr>
<tr><td>Flow of Events</td><td></td><td>User Input</td><td>System Response</td></tr>
<tr><td></td><td>1</td><td>This use case starts when the user commands the application to save the active project to a given path.</td><td></td></tr>
<tr><td></td><td>2</td><td></td><td>The system tries to save the project. If the process succeeds, the project is now saved on the file.</td></tr>
</table>

Table 4.2.3: *Use case details for "Save Project".*

Responsibilities	The operation is in charge of saving the state of the project to a storage device.
Type	System.
References	Use case: Save Project.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active project.
Post-Conditions	The state of the project will be saved to the storage device.

Table 4.2.4: *Contract of the terrain designer "saveProject()" system operation.*

- ***Init use case:***

This use case describes the behavior of the system when the user first loads the application. This use case only happens the first time the application is launched.

The system must load all the required subsystems and allocate all the needed resources for the correct working of the application.

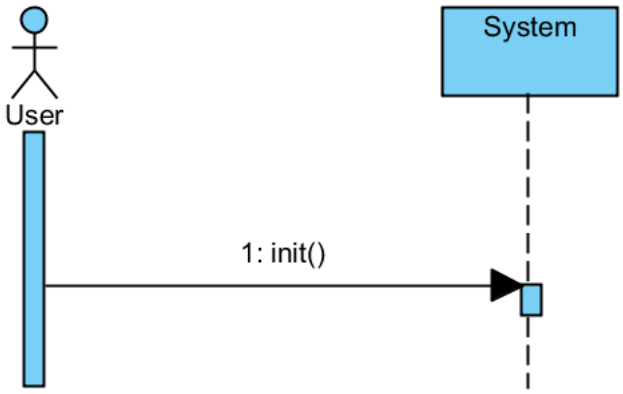


Figure 4.2.4: *Sequence diagram for the "Init" use case.*

Name	**Init**		
Description	This use case describes the behavior of the system when the user first opens the application.		
Preconditions	The application must have not been opened yet.		
Post-conditions	The application subsystems will be loaded.		
Flow of Events			
	User Input	***System Response***	
1	This use case starts when the user opens the application.		
2		The system starts up all the required subsystems.	

Table 4.2.5: *Use case details for "Init".*

Responsibilities	The operation is in charge of creating an instance of the main form, which will in turn instance its child forms: the viewer form, which will instance the tile renderer, and the explorer form.
Type	System.
References	Use case: Init.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have not been opened yet.
Post-Conditions	If the process succeeds, all the child forms and the tile renderer will be instanced.

Table 4.2.6: *Contract of the terrain designer "init()" system operation.*

- ***Adjust Viewport use case:***

This use case describes the behavior of the system when the user commands the application to change the viewport, be its zoom level or position. This use case may happen at any moment after the application has finished loading.

After the user is in move or in zoom mode, the application must register the mouse coordinates after the user clicks the viewport, and then adjust the viewport using these values. This use case assists to fulfill the requirement *Func-Design-1*.

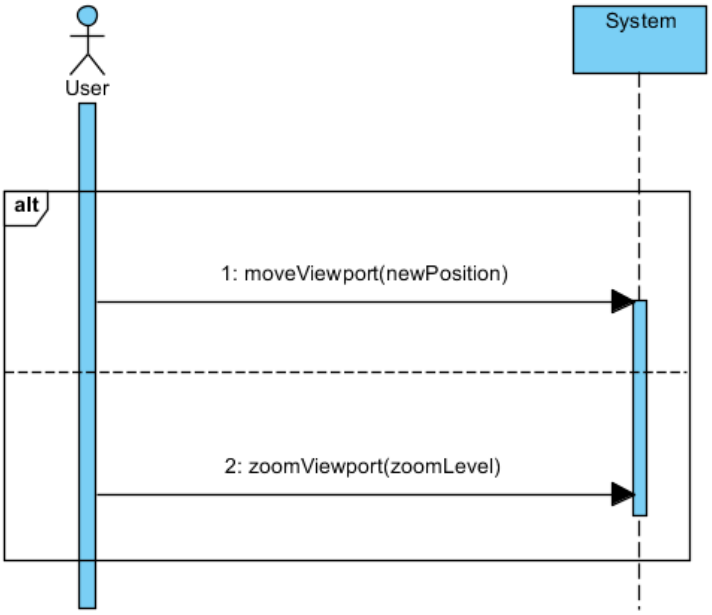


Figure 4.2.5: *Sequence diagram for the "Adjust Viewport" use case.*

Name	**Adjust Viewport (Move viewport branch)**		
Description	This use case describes the behavior of the system when the user commands the application to change the viewport of the designer. This branch describes the behavior when the system is commanded to change the position of the viewport.		
Preconditions	The application must have finished loading.		
Post-conditions	The viewport position will be changed.		
Flow of Events		**User Input**	**System Response**
	1	This use case starts when the user commands the application to set the new viewport position.	
	2		The system calculates the new position of the viewport based on the specified position and sets it.

Table 4.2.7: *Use case details for "Adjust Viewport" move branch.*

Responsibilities	The operation is in charge of changing the position of the camera of the viewport of the tile renderer, setting the viewport position to the specified new position.
Type	System.
References	Use case: Adjust Viewport.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The position of the viewport will be changed.

Table 4.2.8: *Contract of the terrain designer "moveViewport()" system operation.*

<table>
<tr><td>Name</td><td colspan="3">Adjust Viewport (Zoom viewport branch)</td></tr>
<tr><td>Description</td><td colspan="3">This branch describes the behavior of the system when the user commands the application to change the zoom of the viewport.</td></tr>
<tr><td>Preconditions</td><td colspan="3">The application must have finished loading.</td></tr>
<tr><td>Post-conditions</td><td colspan="3">The zoom of the viewport will be changed.</td></tr>
<tr><td>Flow of Events</td><td></td><td>User Input</td><td>System Response</td></tr>
<tr><td></td><td>1</td><td>This use case starts when the user commands the application to set the new viewport zoom.</td><td></td></tr>
<tr><td></td><td>2</td><td></td><td>The system calculates the new zoom of the viewport based on the specified value and sets it.</td></tr>
</table>

Table 4.2.9: *Use case details for "Adjust Viewport" set zoom branch.*

Responsibilities	The operation is in charge of changing the zoom level of the camera of the viewport of the tile renderer, setting the viewport zoom level to the specified new level.
Type	System.
References	Use case: Adjust Viewport.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The position of the viewport will be changed.

Table 4.2.10: *Contract of the terrain designer "zoomViewport()" system operation.*

- ***Create Project use case:***

This use case defines the behavior of the system when the user commands the application to create a project. This can be achieved by three different actions, defined by branch behaviors: opening an existing project, importing an existing compiled project or creating a new project.

This may happen at any time after the application has finished loading. If the process succeeds, the created project will be set as the default project. This use case helps fulfill the requirement *Func-Design-1*.

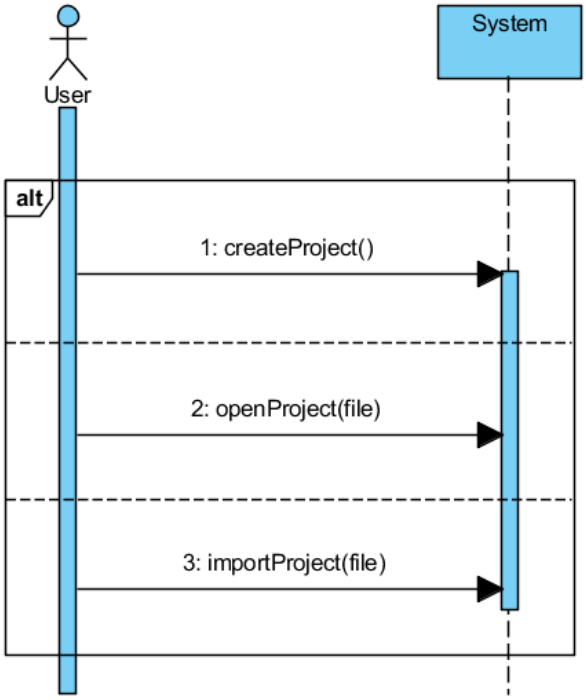


Figure 4.2.6: *Sequence diagram for the "Create Project" use case.*

Name	**Create Project (New Project branch)**		
Description	This use case defines the behavior of the system when the user commands the application to create a project. This branch describes the behavior of the system when the system is commanded to create a new project.		
Preconditions	The application must have finished loading.		
Post-conditions	A new project will be set as the active project.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to create a new project.	
	2		The system creates a new project and sets it as the active project.

Table 4.2.11: *Use case details for "Create Project" new project branch.*

Responsibilities	The operation is in charge of creating a new project instance and setting it as the active project, discarding any active project in case there was one.
Type	System.
References	Use case: Create Project.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The new project will be set as the active project

Table 4.2.12: *Contract of the terrain designer "createProject()" system operation.*

Name	**Create Project (Load Project branch)**		
Description	This branch describes the behavior of the system when the system is commanded to load an existing project.		
Preconditions	The application must have finished loading.		
Post-conditions	The existing project will be set as the active project.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to load an existing project from a given path.	
	2		The system loads the project and sets it as the active project.

Table 4.2.13: *Use case details for "Create Project" load project branch.*

Responsibilities	The operation is in charge of creating a new project instance, setting its data to the data contained in an existing project which will be loaded from a storage device.
Type	System.
References	Use case: Create Project.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The loaded project will be set as the active project.

Table 4.2.14: *Contract of the terrain designer "openProject()" system operation.*

Name	**Create Project (Import Project branch)**		
Description	This branch describes the behavior of the system when the system is commanded to import an existing compiled project.		
Preconditions	The application must have finished loading.		
Post-conditions	The existing project will be set as the active project.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to import an existing compiled project from a given path.	
	2		The system loads the project and sets it as the active project.

Table 4.2.15: *Use case details for "Create Project" import project branch.*

Responsibilities	The operation is in charge of creating a new project instance, setting its data to the data contained in an existing compiled project which will be loaded from a storage device.
Type	System.
References	Use case: Create Project.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The loaded project will be set as the active project.

Table 4.2.16: *Contract of the terrain designer "importProject()" system operation.*

- ***Export Project use case:***

This use case describes the behavior of the system when the user commands the application to export the active project into a compiled project. The application will convert the source GIS files into the format expected by the viewer application, to be able to visualize them.

This may happen at any time if there is an active project. When the export process finishes, the active project will be compiled into the selected directory. This use case helps to fulfill the requirement *Func-Design-1*.

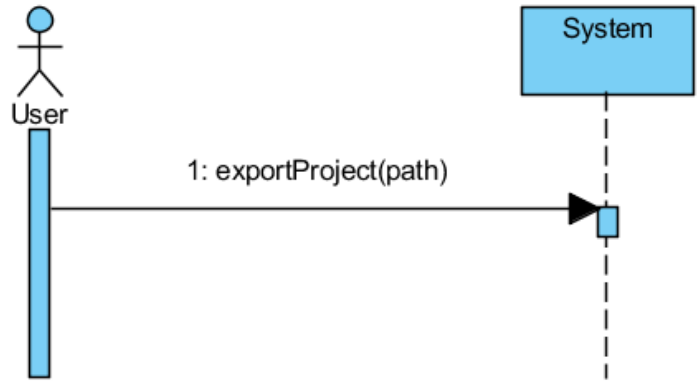


Figure 4.2.7: *Sequence diagram for the "Export Project" use case*

<table>
<tr><td>Name</td><td colspan="3">Export Project</td></tr>
<tr><td>Description</td><td colspan="3">This use case describes the behavior of the system when the user commands the application to export the active project into a compiled project. The application will convert the source GIS files into the format expected by the viewer application, to be able to visualize them.</td></tr>
<tr><td>Preconditions</td><td colspan="3">The application must have finished loaded and there must be an active project.</td></tr>
<tr><td>Post-conditions</td><td colspan="3">The active project will be compiled into the selected directory.</td></tr>
<tr><td>Flow of Events</td><td></td><td>User Input</td><td>System Response</td></tr>
<tr><td></td><td>1</td><td>The use case starts when the user commands the application to export the active project to the specified directory.</td><td></td></tr>
<tr><td></td><td>2</td><td></td><td>The application converts all the files to target formats, and saves them to the destination directory, showing the compilation progress in a dialog.</td></tr>
</table>

Table 4.2.17: *Use case details for "Export Project".*

Responsibilities	The operation is in charge of exporting the active project to a storage device, serializing the project data and converting the height and color textures to the appropriate format.
Type	System.
References	Use case: Export Project.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The loaded project will be set as the active project.

Table 4.2.18: *Contract of the terrain designer "exportProject()" system operation.*

- ***Convert SRTM use case:***

This use case describes the behavior of the system when the user commands the application to convert a GIS file with SRTM-HGT format to TIFF image format. This aids to fulfill the requirement *Func-Design-2*, since the SRTM format is very popular for real world relief imagery.

The application should prompt the user with a dialog asking for the target file, and store the converted file on the same directory but with HGT format and extension.

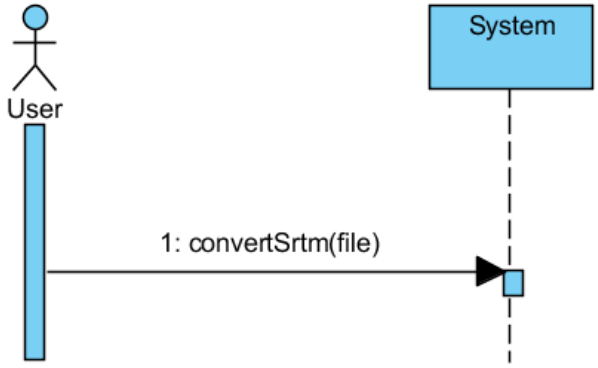


Figure 4.2.8: *Sequence diagram for the "Convert SRTM" use case.*

Name	**Convert SRTM**		
Description	This use case describes the behavior of the system when the user commands the application to convert a SRTM HGT file to a TIFF file.		
Preconditions	The application must have finished loading.		
Post-conditions	The file will be converted.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to convert a specified HGT file.	
	2		The system converts the file, saving it on the same directory, with TIFF format.

Table 4.2.19: *Use case details for "Convert SRTM".*

Responsibilities	The operation is in charge of converting a specified SRTM file into a TIFF file, saving the result at the same directory of the storage device.
Type	System.
References	Use case: Convert SRTM.
Exceptions	None.
Output	None.
Pre-Conditions	The application must have finished loading.
Post-Conditions	The file will be converted to the SRTM format.

Table 4.2.20: *Contract of the terrain designer "convertSrtm()" system operation.*

- ***Modify tile use case:***

This use case describes the behavior of the system when the user commands the application to modify a tile. Tiles are the elements creating the terrain, and have latitude, longitude, heightmap and color textures.

This use case contains four branches: adding tiles, removing tiles, changing the tile position and changing one of the tile textures. This use case is directly related to the requirement *Func-Design-1*.

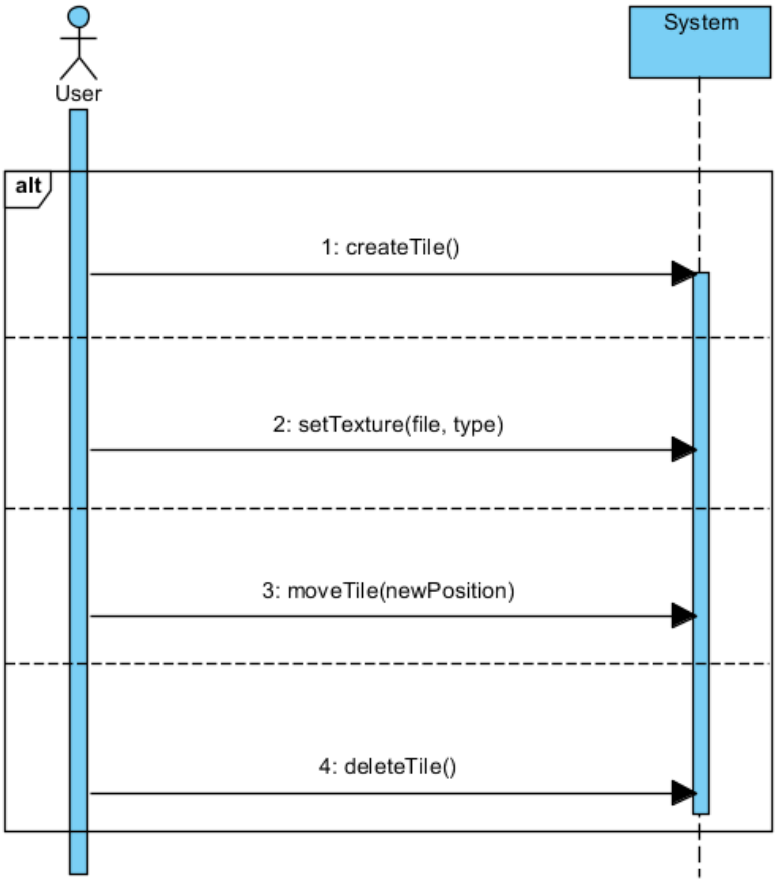


Figure 4.2.9: *Sequence diagram for the "Modify Tile" use case.*

Name	**Modify Tile (New tile branch)**		
Description	This use case describes the system behavior when the user commands the application to modify a tile. This branch defines the behavior when the application is commanded to add a new tile.		
Preconditions	The application must have finished loading and there must be an active project.		
Post-conditions	The new tile will be created.		
Flow of Events		**User Input**	**System Response**
	1	This use case starts when the user commands the application to create a new tile.	
	2		The system creates a new tile centered on the terrain's coordinate system and with no associated texture or height map.

Table 4.2.21: *Use case details for "Modify Tile" new tile branch.*

Responsibilities	The operation is in charge of creating a new tile instance and adding it to the active project.
Type	System.
References	Use case: Modify Tile.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active project.
Post-Conditions	A new tile instance will be added to the project.

Table 4.2.22: *Contract of the terrain designer "createTile()" system operation.*

Name	**Modify Tile (Move tile branch)**		
Description	This branch describes the behavior when the application is commanded to move an existing tile		
Preconditions	A tile must be active.		
Post-conditions	The selected tile will be moved.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to change the position of a tile to a specified new position.	
	2		The system places the tile in the new position.

Table 4.2.23: *Use case details for "Modify Tile" move tile branch.*

Responsibilities	The operation is in charge of changing the position of the active tile to the specified value.
Type	System.
References	Use case: Modify Tile.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active tile.
Post-Conditions	The position of the tile will be changed.

Table 4.2.24: *Contract of the terrain designer "moveTile()" system operation.*

Name	**Modify Tile (Remove tile branch)**		
Description	This branch describes the behavior of the system when the application is commanded to delete an existing tile.		
Preconditions	A tile must be active.		
Post-conditions	The selected tile will be removed.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to remove a tile.	
	2		The system removes the selected tile from the system.

Table 4.2.25: *Use case details for "Modify Tile" remove tile branch.*

Responsibilities	The operation is in charge of removing the active tile from the project.
Type	System.
References	Use case: Modify Tile.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active tile.
Post-Conditions	The tile will be deleted and there will not be any active tile.

Table 4.2.26: *Contract of the terrain designer "deleteTile()" system operation.*

Name	**Modify Tile (Set texture branch)**		
Description	This branch describes the behavior of the system when the application is commanded to set the color texture or the height map texture of the tile.		
Preconditions	A tile must be selected.		
Post-conditions	The target texture will be set on the selected tile.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to set a specified texture as the tile height or color texture.	
	2		The system loads the target texture and sets it to the selected tile.

Table 4.2.27: *Use case details for "Modify Tile" set texture branch.*

Responsibilities	The operation is in charge of loading a height or color texture and setting it as the texture for the active tile.
Type	System.
References	Use case: Modify Tile.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active tile.
Post-Conditions	The texture will be set as the tile texture.

Table 4.2.28: *Contract of the terrain designer "setTexture()" system operation.*

- ***Modify camera use case:***

This use case describes the behavior of the system when the user commands the application to modify a camera way-point. The use case consists of three branches: add way-point, remove way-point and move way-point.

These way-points will be used on the terrain viewer application to create the camera flight. This use case is directly related to the requirement *Func-View-3*.

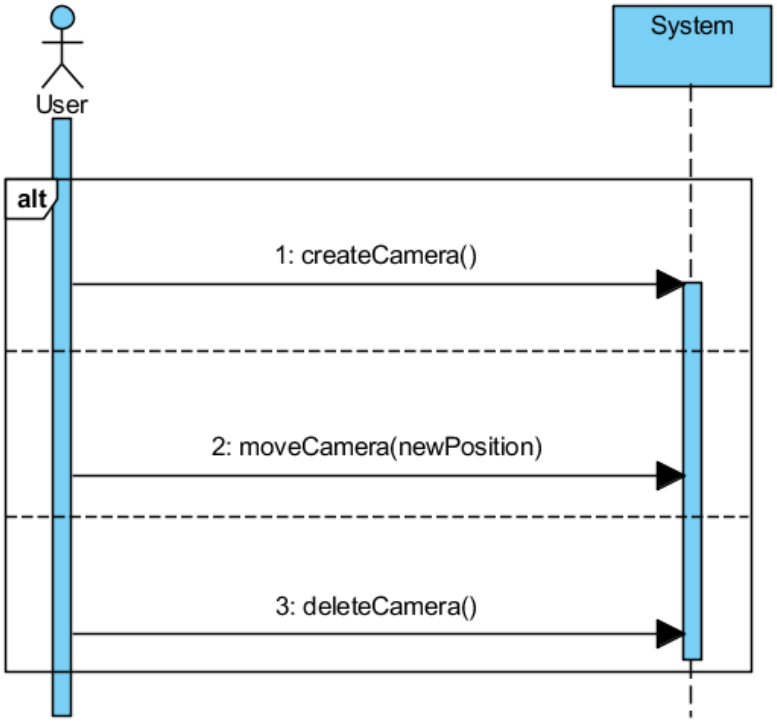


Figure 4.2.10: *Sequence diagram for the "Modify camera" use case.*

Name	**Modify Camera (Create camera branch)**		
Description	This use case describes the behavior of the system when the user commands the application to modify a camera way-point. This branch defines the behavior when the application is commanded to create a new camera.		
Preconditions	The application must have finished loading and there must be an active project.		
Post-conditions	A camera way-point will be created.		
Flow of Events		**User Input**	**System Response**
	1	This use case starts when the user commands the application to create a new camera waypoint.	
	2		The system adds a new camera way-point on the center of the terrain coordinates.

Table 4.2.29: *Use case details for "Modify Camera" create camera branch.*

Responsibilities	The operation is in charge of creating a new camera waypoint and adding it to the project.
Type	System.
References	Use case: Modify Camera.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active project.
Post-Conditions	The new waypoint will be created.

Table 4.2.30: *Contract of the terrain designer "createCamera()" system operation.*

Name	**Modify Camera (Remove camera branch)**		
Description	This branch describes the behavior when the system is commanded to remove a camera way-point.		
Preconditions	A camera way-point must be active.		
Post-conditions	The selected way-point will be removed.		
Flow of Events		**User Input**	**System Response**
	1	This use case starts when the user commands the application to remove the active camera waypoint.	
	2		The system removes the selected way-point.

Table 4.2.31: *Use case details for "Modify Camera" remove camera branch.*

Responsibilities	The operation is in charge of setting the position of the active camera waypoint to the specified value.
Type	System.
References	Use case: Modify Camera.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active camera waypoint.
Post-Conditions	The waypoint position will be changed.

Table 4.2.32: *Contract of the terrain designer "moveCamera()" system operation.*

Name	**Modify Camera (Move camera branch)**		
Description	This branch describes the behavior when the system is commanded to move a camera way-point.		
Preconditions	A camera way-point must be active.		
Post-conditions	The way-point will be moved.		
Flow of Events		***User Input***	***System Response***
	1	This use case starts when the user commands the application to change the position of a camera waypoint to a new specified position.	
	2		The system calculates the target position of the tile using the specified new position and sets it

Table 4.2.33: *Use case details for "Modify Camera" move camera branch.*

Responsibilities	The operation is in charge of removing the active camera waypoint.
Type	System.
References	Use case: Modify Camera.
Exceptions	None.
Output	None.
Pre-Conditions	There must be an active camera waypoint.
Post-Conditions	The waypoint will be removed and there will not be an active waypoint.

Table 4.2.34: *Contract of the terrain designer "deleteCamera()" system operation.*

4.3: Summary

In this chapter, the system was analyzed from the system requirements and the proposed solution of the previous chapter, in order to create a use case model focused on the operations between the user and the system.

The following chapter focuses on the high level design of both the terrain viewer and terrain designer system, where the systems will be modeled as a series of blocks along with the systems class structure.

CHAPTER 5: SYSTEM DESIGN

In this chapter the high level design of the system is introduced. The high level design takes an insight into the conceptual model of the terrain designer and viewer applications at the class level, the way their classes must communicate to achieve the required features and how do the applications behave to fulfill the requirements set by the use cases.

Since the project uses the graphics hardware through OpenGL, this implies the mixing of object oriented code on the CPU (host code) and sequential code on the GPU (graphics card program). This chapter will introduce the communication between the host and graphics programs and how do they cooperate.

5.1: Conceptual Design of the Terrain Designer

The following diagram illustrates how the different components interact, with each other and with the external systems.

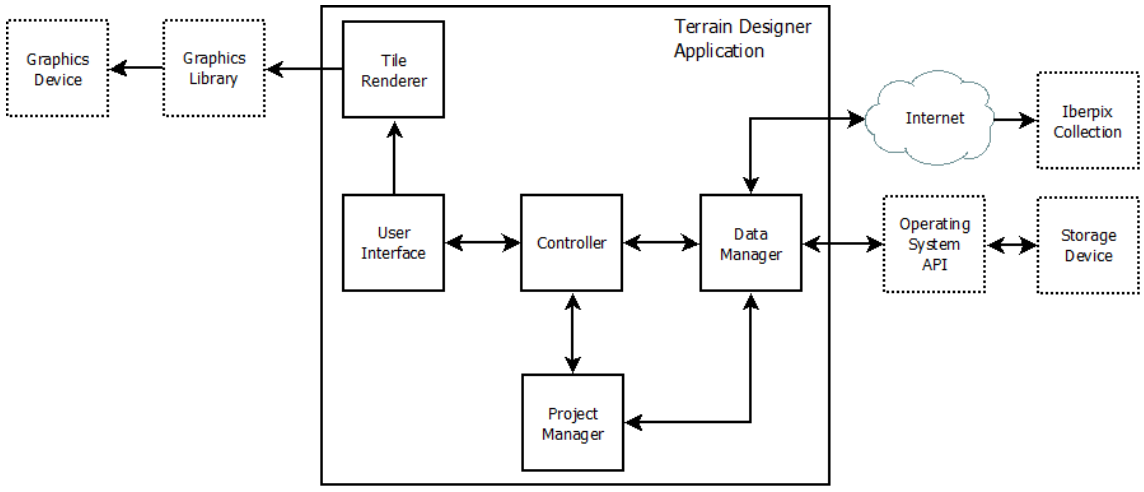


Figure 5.1.1: *Terrain Designer Block Diagram*

The application class diagram will be based upon this block diagram. The detailed design of each one of the blocks is detailed next.

5.1.1: User interface design

The user interface is the component in charge of exposing the features of the application to the user. It must accept the commands by the user, and through a series of dialogs, let the user operate the application.

The different use cases of the application are the starting point for the definition of the user interface. All use cases must be represented as a sequence of commands on the user interface, as it is the way the user has to interact with the system.

Besides its behavior, the design, flow and behavior of the user interface are a key point on the final result of the application. The application must be as easy to use as possible, and the layout of the different elements of the user interface will be designed around this principle.

- ***User Interface Structure:***

There are many possible user interface structures, like for example using individual dialog windows to implement each one of the features, or using a single dialog window where all features are accessible.

In this case, the chosen structure was using a single window structure, where the main focus is set on the data being edited (the terrain tiles), and the different features can be accessed through toolboxes and menus.

This interface structure centers the focus of the user on the content, and forces the user to only change the point of attention when the tool used to modify the content has to be switched.

The interface main dialog can be divided on the following parts: content area, toolboxes, and menu bars. The following figure shows an example of an application using the proposed structure.

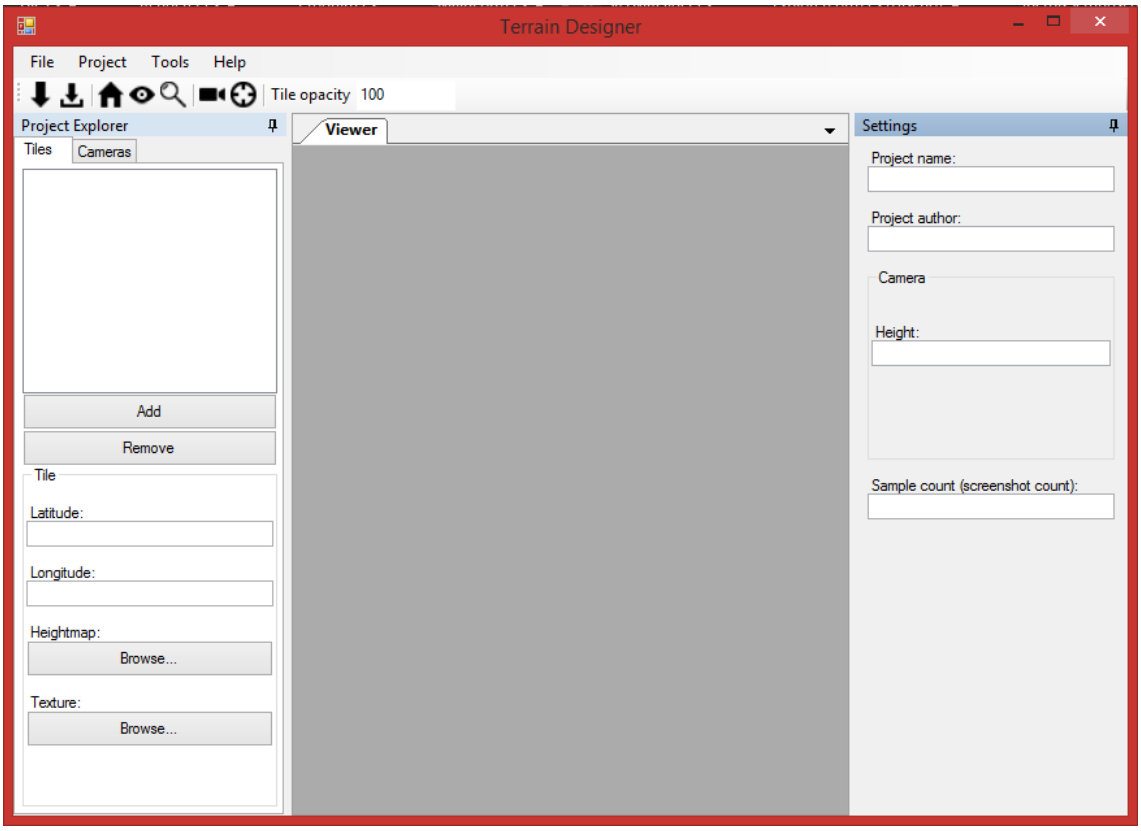


Figure 5.1.2: *Interface of the terrain designer application.*

- ***Use Case Analysis Oriented to the User Interface:***

In this section of the user interface design, the different use cases of the designer application are studied to define how the flow of events of each one of the use cases is going to translate into one or more window menu options, tool box options or dialogs.

Dialogs and toolboxes translate into a class on the conceptual class diagram. Menu bars translate into attributes on the main window dialog class. Controls or menu options inside them will translate into attributes inside the classes containing them.

The use cases are going to be analyzed in the order of appearance on the analysis annex for the terrain designer use case descriptions.

  - ***Download from Iberpix Use Case:***

On this use case, the first step on the flow of events involves the user using a menu option to start the use case. This option will be integrated on the main window menu bar option "*Tools -> Download Iberpix Dataset*".

The next two steps on the flow of events involve the user introducing the required data inside a dialog, and then commanding the application to start the download. This will be integrated inside an independent dialog window, "*IberpixDownloadForm*", containing the required controls to introduce the needed set-up data.

The final step involves presenting a dialog to the user displaying the progress of the download. This will be integrated inside an independent dialog window, "*IberpixDownloadProgressForm*".

The use case error branch involves displaying a dialog in case of error. This dialog will be implemented as a standard message dialog, offered by the system environment.

- ***Save Project Use Case:***

On the first step of the use case, the user commands the application through a menu option. This will be integrated as the main window menu bar option "*File -> Save Project*". Additionally, the command can be accessed through the secondary menu bar "*Save*" button.

The next step involves the user selecting the destination directory on a dialog. This dialog will be implemented using a standard file save dialog, as offered by the system environment.

On the error branch, the user is notified using a message dialog. This will be implemented using a standard message dialog from the system environment.

- ***Create Project Use Case:***

This use case presents three branches, corresponding to the three ways a project can be set as the active project: through creating a new project, through loading an existing project or through importing an already exported project.

The first step of all three branches involves the user commanding the application through a menu. This will be translate into three menu options on the main window menu bar "File -> New Project", "File -> Load Project" and "Project -> Import" respectively.

In the second and third steps of the load and import branches, the user selects the target project on a dialog. This dialog will be implemented using a standard file load dialog as offered by the system environment.

- ***Convert SRTM Use Case:***

On this use case there are only two interactions with the user interface. On the first step, the user commands the application through a menu, which will be represented by the main window menu option "*Tools -> Convert SRTM to TIFF*".

On the second and third steps the user selects the target SRTM file on a dialog, which will be implemented using a standard file open dialog as offered by the system environment.

- ***Adjust Viewport Use Case:***

This use case defines the operations that let the user change the viewport of the content area, changing the position of the viewport or changing the zoom of the viewport.

On the first branch, the user sets the application in the move viewport mode. This is implemented through a button on the secondary menu bar. On the second branch, the same happens with the viewport zoom mode, also implemented through a button on the secondary menu bar. The difference in position or zoom is obtained dragging the mouse on the content area.

- ***Modify Tile Use Case:***

This use case introduces all the possible operations that can be done on the project terrain tiles: adding new tiles, changing their position, changing their color or height textures and removing tiles.

The system exposes these features through two parts of the main window dialog: the commands to do the operations are exposed on the main toolbox, implemented as the "*ExplorerForm*" class. The controls will be in their own tab, called "Tiles".

On the other hand, the changes done by these operations are represented by the terrain tile rendering component, which is exposed to the used through the content area on the "*ViewerForm*" class.

In order to do any of the operations on the use case except for creating new tiles, the user must have an active tile selected. The selection is performed through a list box on the main toolbox, where the user can pick one of the exiting tiles to set as the active tile.

On the new tile branch, the user commands the application to create a new tile through a menu option. This will be implemented as a control on the main toolbox called "Add".

On the move tile branch, the user can change the longitude and latitude of the tile on the menu. This will be implemented as two edit fields called “Latitude” and “Longitude” on the main toolbox.

On the set texture branch, the user can change the height and color textures of the tile. This will be implemented as two buttons on the main toolbox to change the color and height textures respectively.

And finally, on the remove branch the user commands the application to remove the active tile. This will be implemented as a button on the main toolbox, called “Remove”.

- ***Modify Camera Use Case:***

This use case introduces all the possible operations that can be done over the camera flight waypoints: adding waypoints, removing waypoints and moving waypoints.

These features are exposed, as in the case of the modify tile use case, through two parts of the main window dialog: the commands are exposed through the main toolbox and the result of those commands is represented by the content area.

The controls on the main toolbox will be on a different tab than the controls of the modify tile use case, on a tab called “Cameras”. Additionally, some of the commands are available on the secondary menu bar for quick access.

To do any of the operations on the use case except for adding waypoints, there must be an active waypoint. The user can select a waypoint through a list box on the main toolbox listing all the existing waypoints.

On the create camera branch, the user can command the application to create a new waypoint. This is implemented as a button on the main toolbox, called “Add” and through a button on the secondary menu bar.

On the remove camera branch, the user can command the application to remove the selected waypoint. This is implemented as a button on the main toolbox, called “Remove”.

And finally on the move camera branch, the user can command the application to change the latitude and longitude of a waypoint. The user can change the waypoint position by pressing the move waypoint button on the secondary menu bar, and then dragging the waypoint to its final position on the content area. Additionally, the user can directly change the latitude and longitude of the waypoint.

- ***Summary:***

The user interface of the designer application will translate into the following classes after analyzing its use cases: “*MainForm*”, “*ExplorerForm*”, “*ViewerForm”, “IberpixDownloadForm”* and *“IberpixDownloadProgressForm”*.

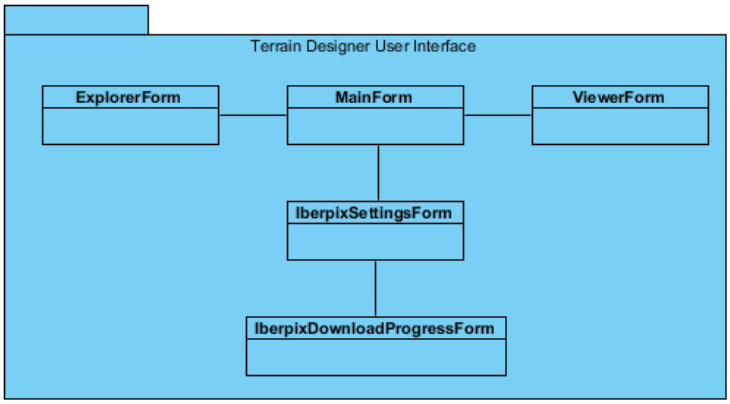


Figure 5.1.3: *Class sub-diagram of the user interface component.*

5.1.2: Tile Renderer Design

The tile renderer is the component in charge of displaying the state of the terrain project to the user, showing the existing tiles, waypoints and a preview of the final flight path. To achieve this, the tile renderer must be able to access the content of the project at any given moment.

The renderer is implemented on a class implementing methods to display a tile, to display a decal indication the position of a waypoint and to display a curve showing the trajectory to be followed by the camera flight on the terrain viewing application.

It also must implement methods to change the position and the zoom of the viewing area, to let the user adjust the viewport. The user interface is in charge of changing the zoom and position of the renderer when commanded by the user, as explained on the “adjust viewport” user interface section.

Since terrain projects may contain a large number of tiles, the renderer must display them in the most efficient way possible. It must be implemented with this principle in mind.

To display the curve of the camera flight, classes implementing the behavior of cubic curves are implemented, specialized on classes to evaluate uniform B-Splines and Bézier curves.

- ***Summary:***

The tile renderer component will translate into the following classes: "*Renderer"* and "*CubicCurve*".

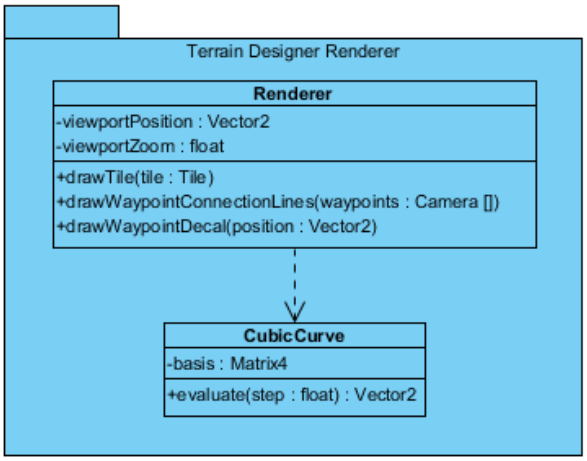


Figure 5.1.4: *Class sub-diagram of the renderer component.*

5.1.3: Project Manager Design

The project manager is in charge of keeping track of the stage of the project at every moment, including saving the state and loading it back from a storage location.

The project is composed of tiles: each one of these tiles has an associated height and color texture and a position defined by its latitude and longitude.

This will be implemented as a project class, with methods to add, remove and get tiles, and having an array of tile classes, which contain references to the height and color textures and their associated latitude and longitude.

Besides the tile data, the project also contains the waypoints for the camera flight of the viewing application. These points will be implemented as a camera class, containing their latitude and longitude.

To be able to save and load the state of the project, a helper project serializing class will contain methods to serialize and marshal the content of the project.

- ***Summary:***

The project manager component will translate into the following classes: "*Project*", *"Tile", "Camera"* and "*ProjectSerializer*".

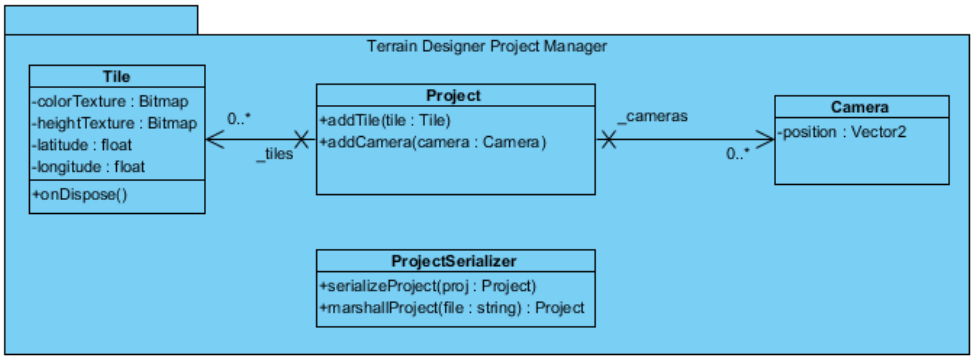


Figure 5.1.5: *Class sub-diagram of the project manager component.*

5.1.4: Data Manager Design

The data manager is the component in charge of transforming the different data in the project from the original format to the format accepted by the terrain viewer with the appropriate content and of communicating with the Iberpix server.

The TIFF writer class is in charge of converting the height and color textures of each one of the tiles in the project to the format accepted by the terrain viewer. Besides writing the files with the appropriate format and color depth, it also calculates the normal vectors of the height surface and stores it along the height data.

The SRTM reader class is in charge of reading the height data stored in HGT files and converting it into a bitmap raster image. It must be able to handle the different types of SRTM files correctly.

The Iberpix downloader class is in charge of connecting to the Iberpix server, sending the data required by the server and finally downloading and storing the height and color textures generated by the server. It must handle the possible errors that might arise during the request or the download.

- ***Summary:***

The data manager component is translated into the following classes: *"TIFFWriter"*, "*SRTMReader"* and *"IberpixDownloader"*.

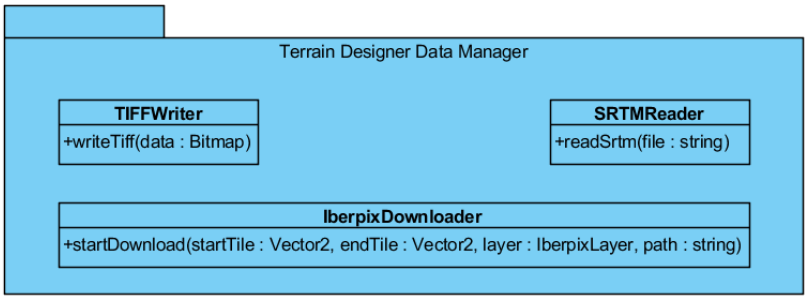


Figure 5.1.6: *Class sub-diagram of the data manager component.*

5.1.5: Controller Design

The controller is the component in charge of coordinating the communication between all the other components, implemented as a controller master class, also containing the entry point of the application.

- ***Summary:***

The controller component translates into the *"Controller"* class.

5.1.6: Design Class Diagram

From the high level description of the different building blocks of the application and the classes obtained through their analysis, a high level class diagram is built, showing the different classes of the application and the interaction between each other.

This class diagram will be the basis for the implementation of the final application, where the methods and attributes of each one of the classes will be completely describes in a low level, detailed fashion.

However, the classes, methods and attributes in this conceptual diagram might change on implementation time, depending on the peculiarities of the implementation platform.

Using the objects in this class diagram and its methods, the operation of the use case operations are described, showing how the different classes communicate to solve the different use cases. In this diagram the access operators of the class attributes are not represented, as these methods are not relevant on the system design.

The following figure shows the design class diagram for the terrain designer system. The next section introduces the sequence diagram where the system operations are described.

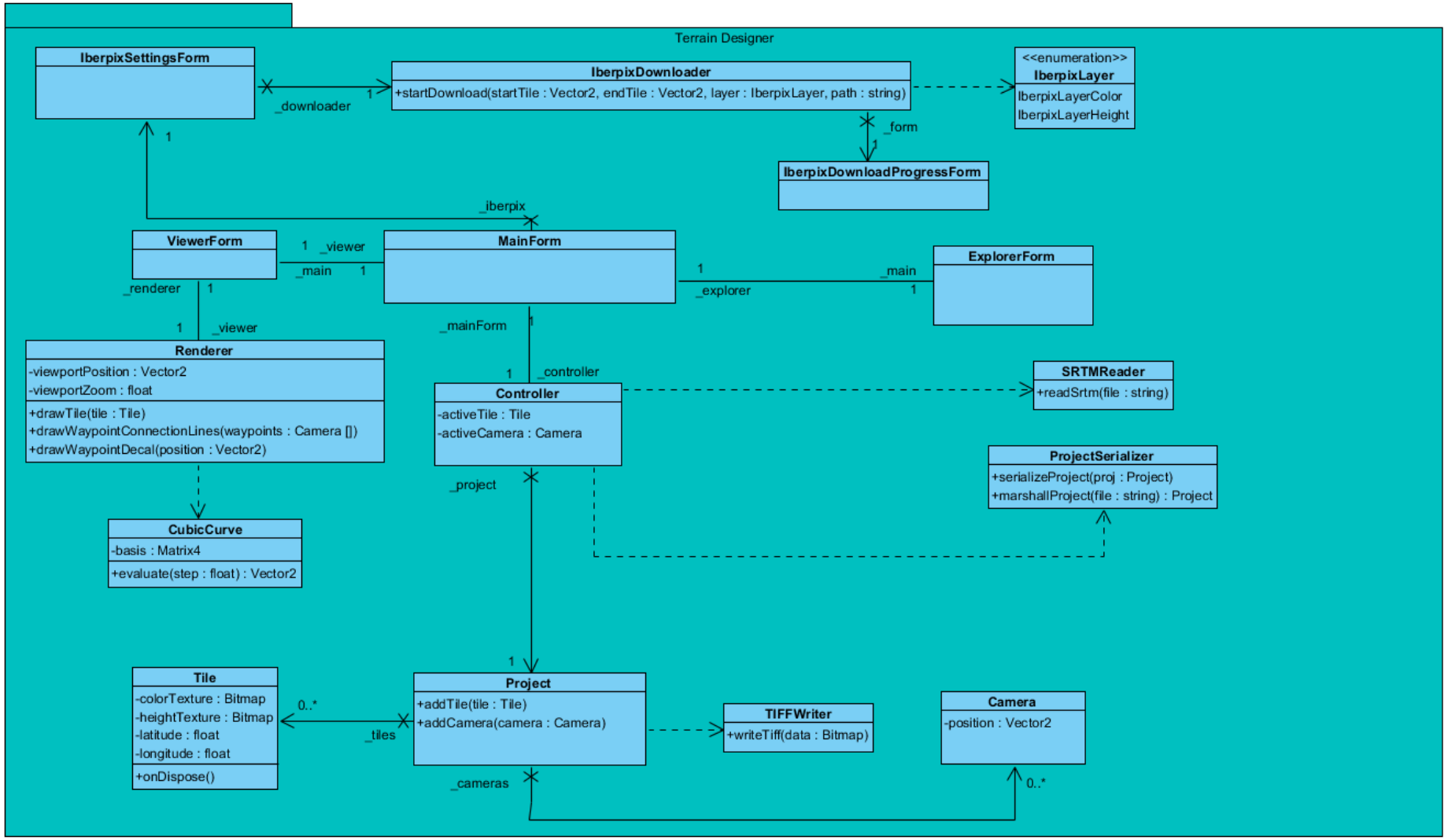


Figure 5.1.7: *High-level conceptual class diagram of the terrain designer application.*

5.1.7: System Operation Interaction Diagrams

In this section, the different operations modeled in the use case sequence diagrams are analyzed to build the operation interaction diagrams. Operation diagrams represent how the different classes communicate through their methods to obtain the solution for a particular operation.

- ***"init()" system operation:***

The *"init()"* system operation is executed when the user first starts the application, loading all the required subsystems and instancing all the required objects for the application to work correctly.

The initialization of the different subsystems: main form, explorer form, viewer form and tile renderer are encapsulated on their class constructors.

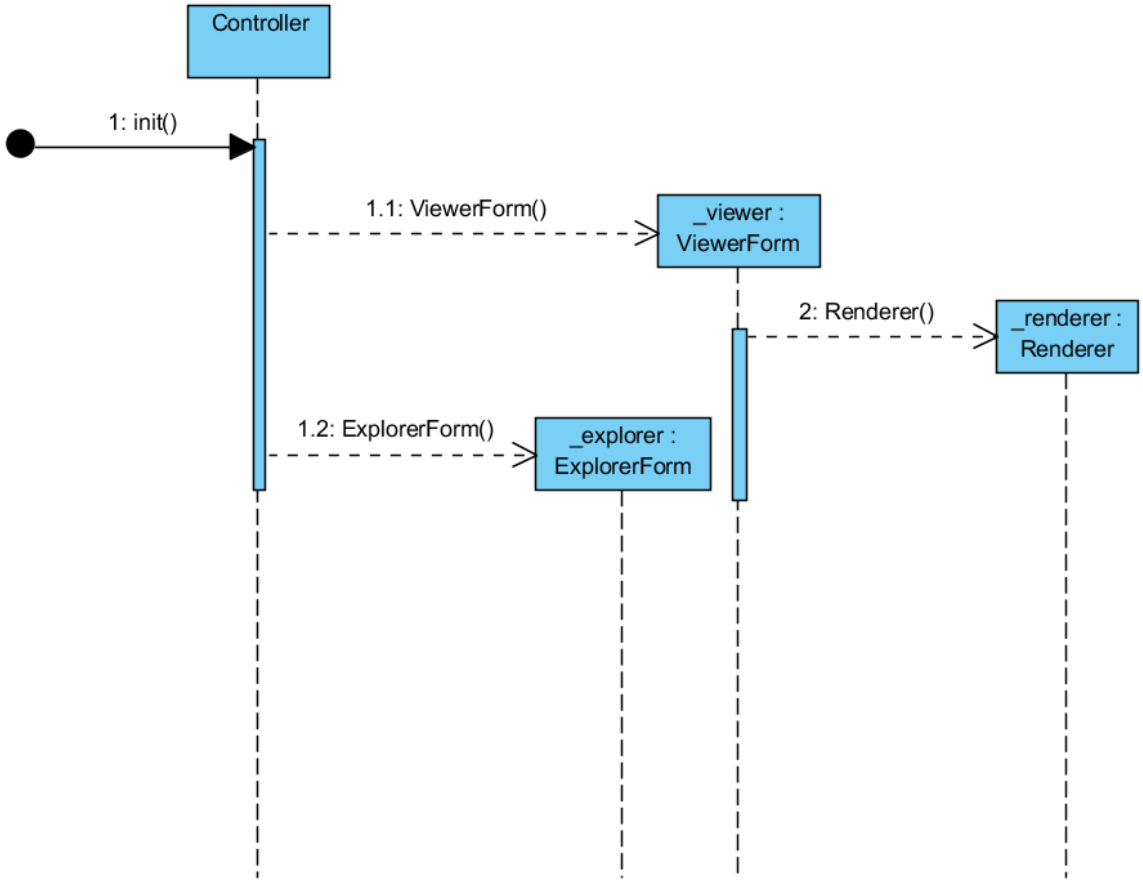


Figure 5.1.8: *Interaction diagram of the terrain designer "init()" system operation.*

- ***"moveViewport(newPosition)" system operation:***

The *"moveViewport()"* operation is executed when the user is going to change the position of the camera on the tile renderer viewport. The operation receives the target position to set as the new viewport position.

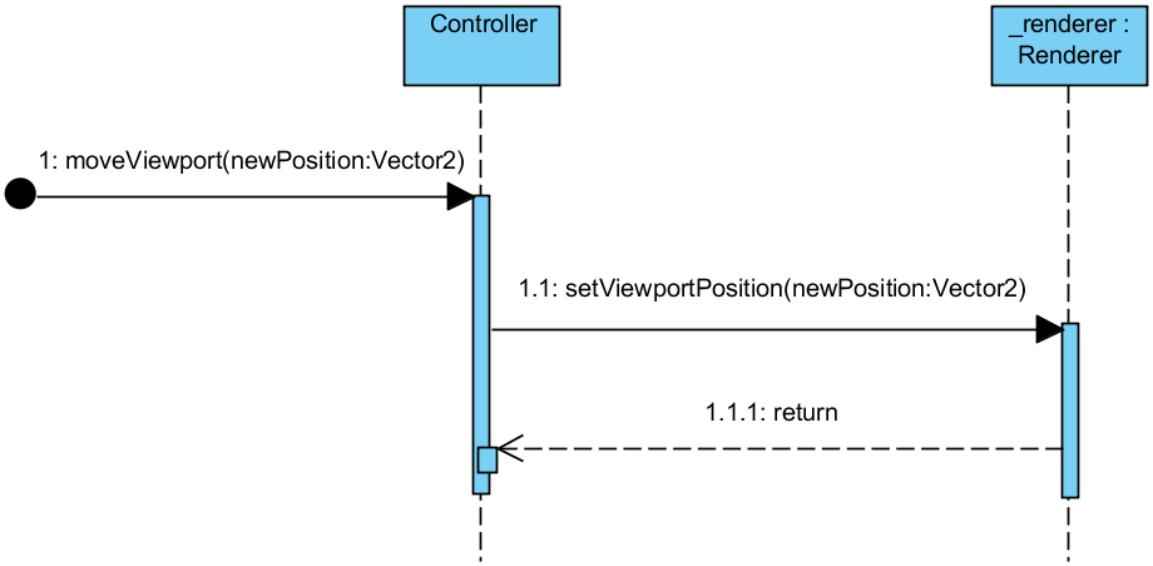


Figure 5.1.9: *Sequence diagram of the terrain designer "moveViewport()" system operation.*

- ***"zoomViewport(zoomLevel)" system operation:***

The *"zoomViewport()"* operation is executed when the user is going to change the zoom level of the camera on the tile renderer viewport. The operation receives the target zoom level to set as the new viewport zoom level.

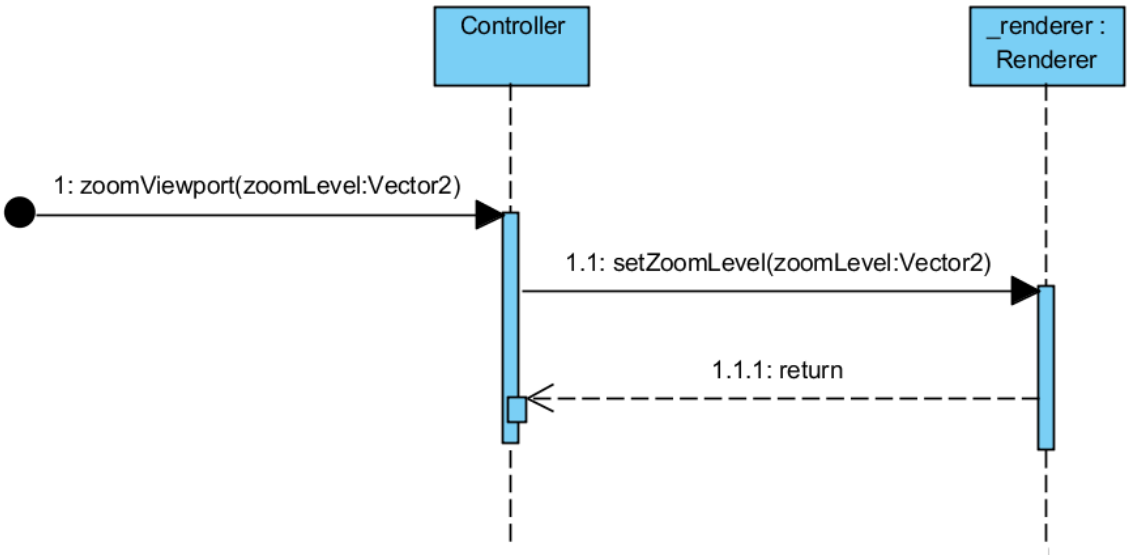


Figure 5.1.10: *Sequence diagram of the terrain designer "zoomViewport()" system operation.*

- ***"createProject()" system operation:***

The *"createProject()"* operation is executed when the user commands the application to create a new project. A new project instance will be set as the active project, and in case a previous project instance existed, it will be discarded. The new project instance is set up using the constructor of the project class.

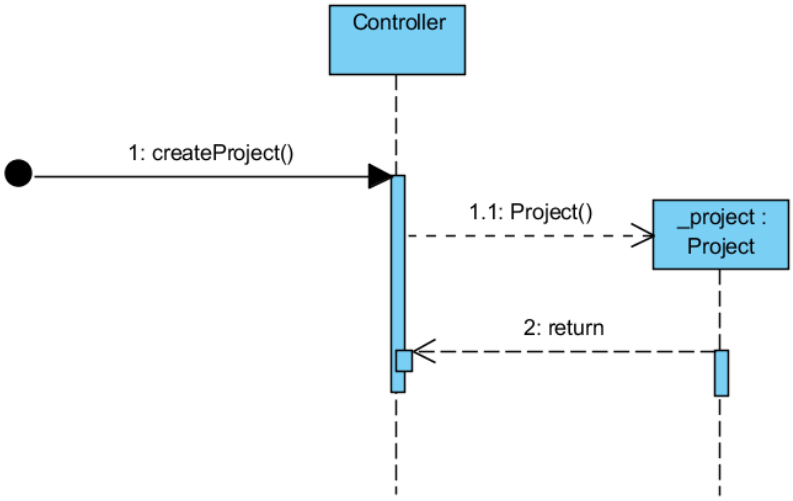


Figure 5.1.11: *Sequence diagram of the terrain designer "createProject()" system operation.*

- ***“openProject(file)” system operation:***

The *“openProject()”* operation executes when the user commands the system to open an existing project located on the specified path. The application will try to load the project, and if it succeeds the selected project will be the active project.

In order to load the project, the application has to load from the project file the information for all the tiles, being latitude, longitude and the color and height texture paths, and then set the color and height texture bitmaps to each tile instance, which will be then added to the project.

Besides the tile information, if the project contains any camera waypoint data the application also has to load from the project file the position of every camera waypoint and add them to the project.

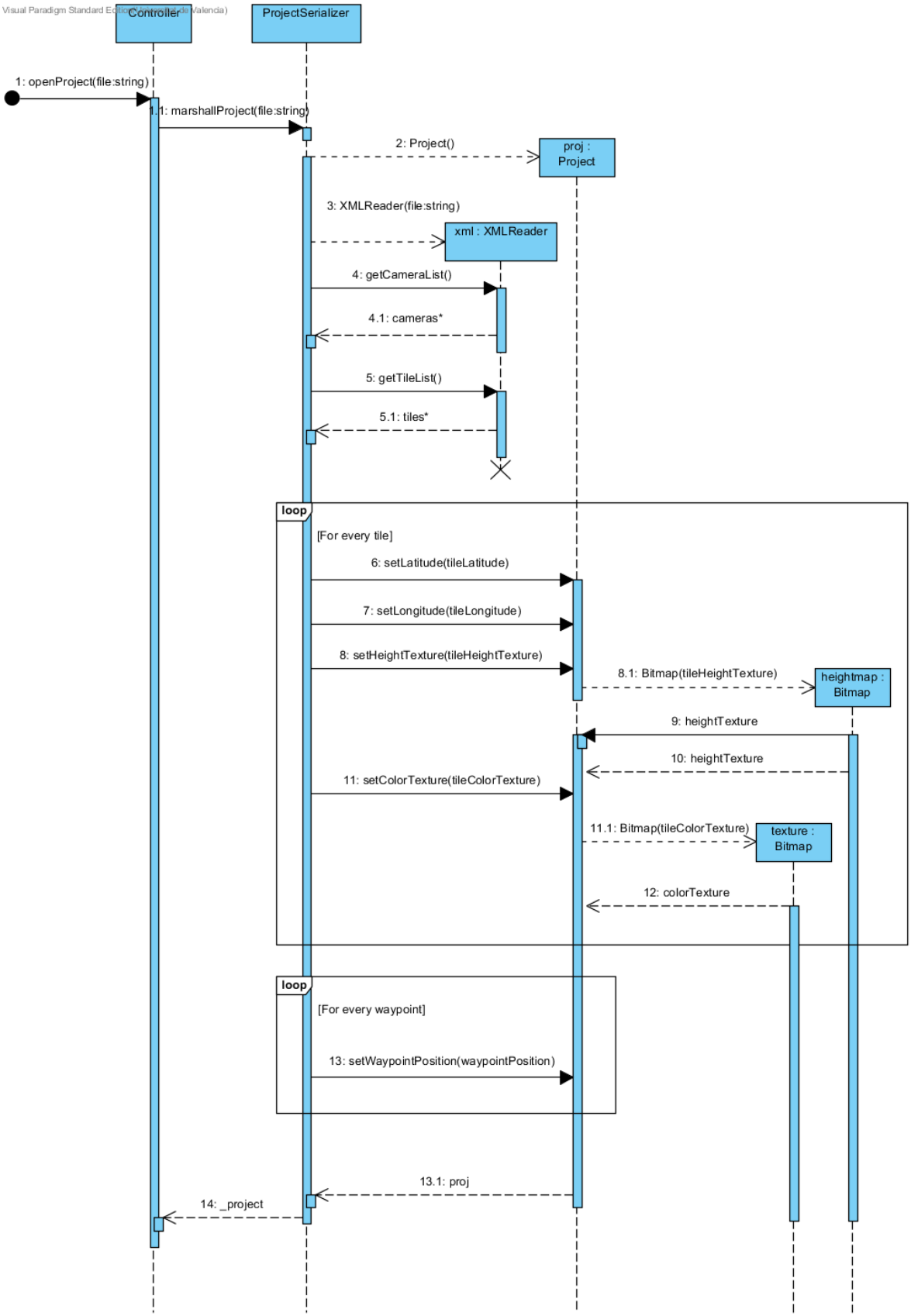


Figure 5.1.12: *Sequence diagram of the terrain designer "openProject()" system operation.*

- ***"importProject(file)" system operation:***

The *"importProject()"* operation executes when the user commands the application to load a project that has already been compiled into a project to be read by the terrain viewer.

Height textures of compiled projects have the height information along the normal vector information encoded in a specific way. Then, this operation is almost the same as the *"openProject()"* operation but taking into account the changes that must be done to the source data.

The sequence diagram of this operation is not represented, since the only difference between this operation and the *"openProject()"* operation is on the way the data is treated on the texture load time.

- ***"exportProject(path)" system operation:***

The *"exportProject()"* operation executes when the user commands the application to export the active project as a compiled set of files to a storage device, in order to use them later with the terrain viewer application.

It will serialize the content of the project, saving the position of tiles and camera waypoints in a file and then exporting the color and height textures, with the appropriate format and content.

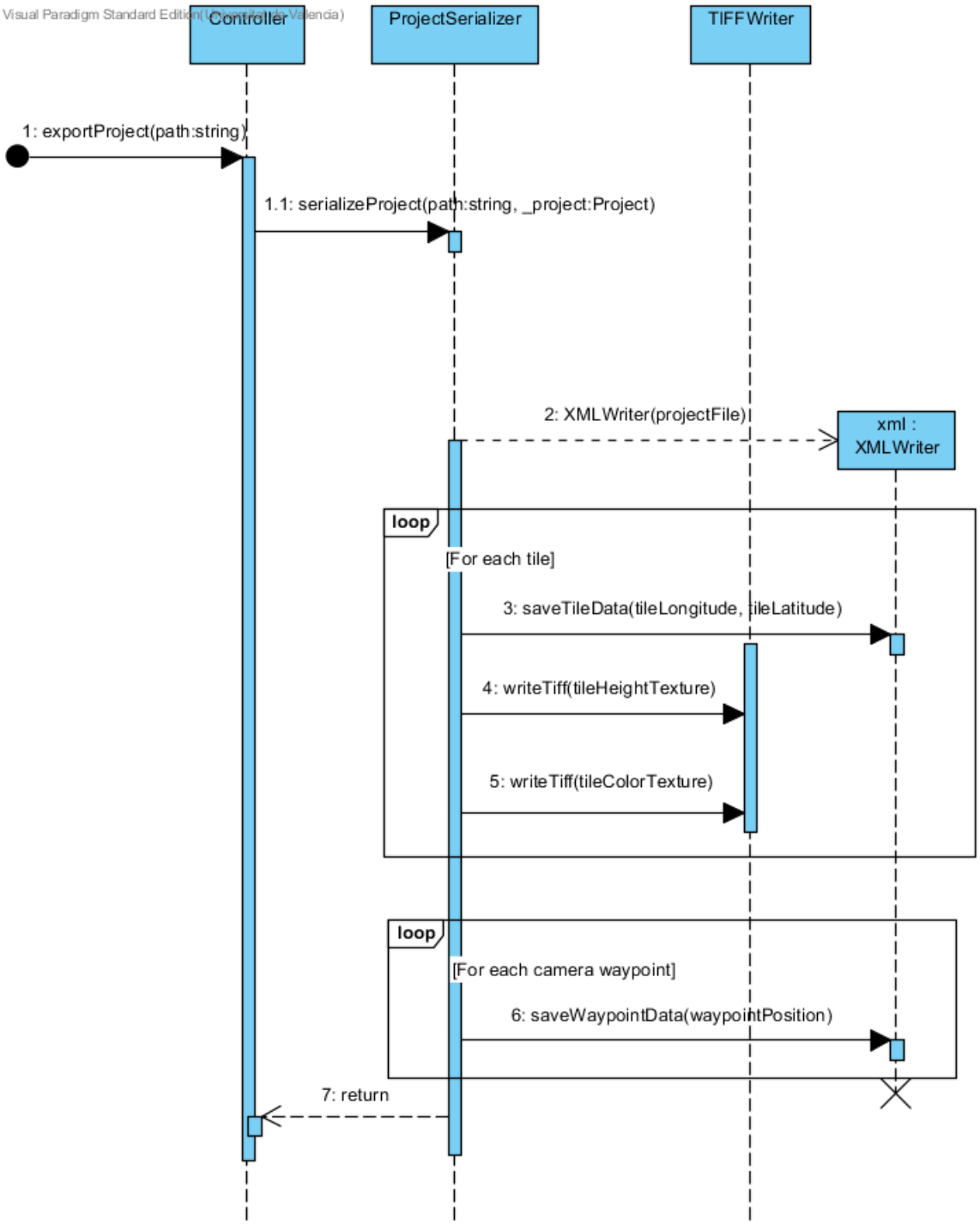


Figure 5.1.13: *Sequence diagram of the terrain designer "exportProject()" system operation.*

- ***"convertSrtm(file)" system operation:***

The *"convertSrtm()"* operation executes when the user commands the application to convert a SRTM file to a TIFF file. This operation will open the file on the specified path, convert it into a native bitmap and then export it as a TIFF file at the same directory.

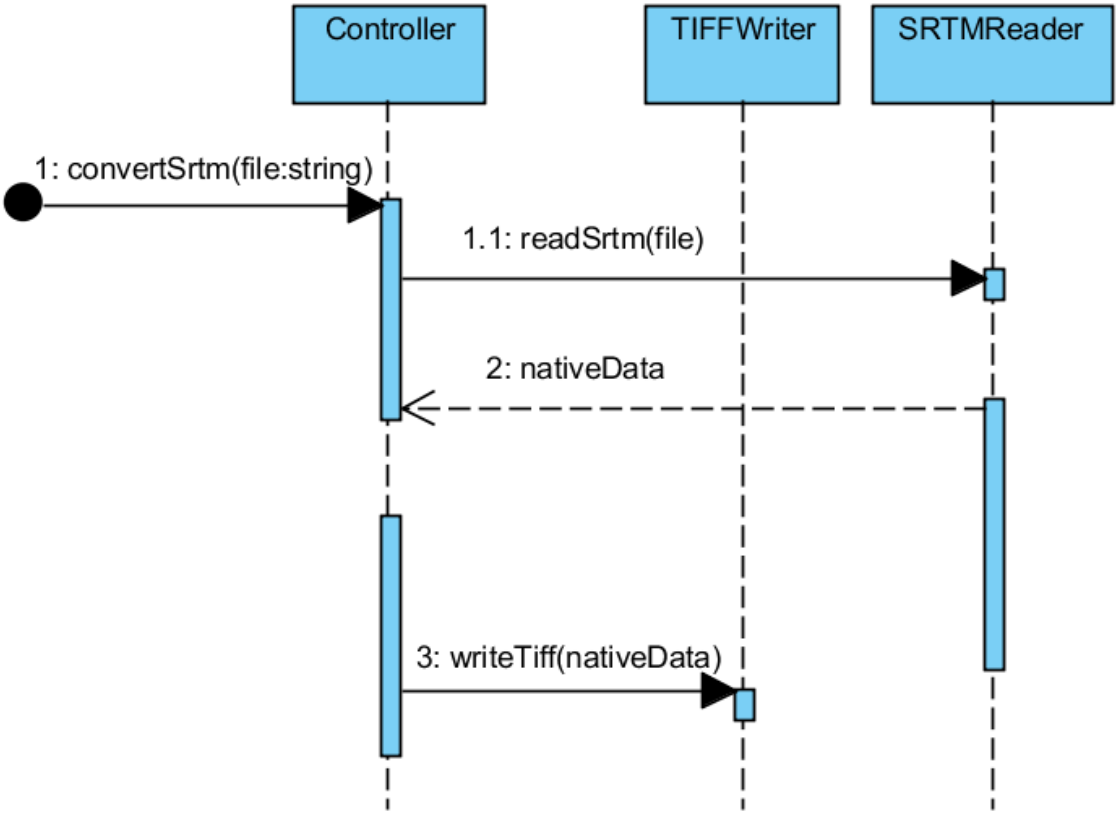


Figure 5.1.14: *Sequence diagram of the terrain designer "convertSrtm()" system operation.*

- ***"createTile()" system operation:***

The *"createTile()"* system operation is executed when the user commands the application to create a new tile on the active project. A new tile instance will be created with its default settings (latitude zero, longitude zero and neither color or height textures), and it will be added to the project.

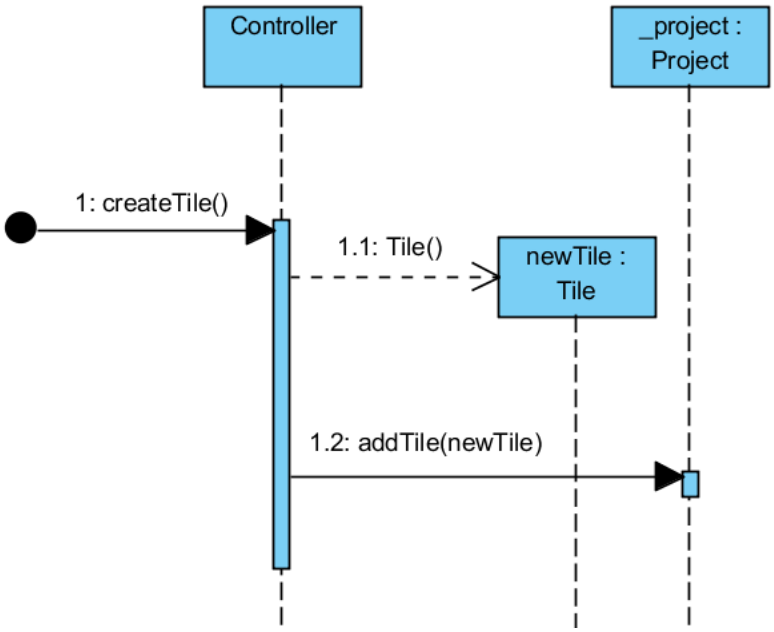


Figure 5.1.15: *Sequence diagram of the terrain designer "createTile()" system operation.*

- ***"setTexture(file, type)" system operation:***

The *"setTexture()"* system operation executes when the user commands the application to set the color or height texture of a tile. The operation loads the texture from the specified file, and sets it as the color or height texture, as specified.

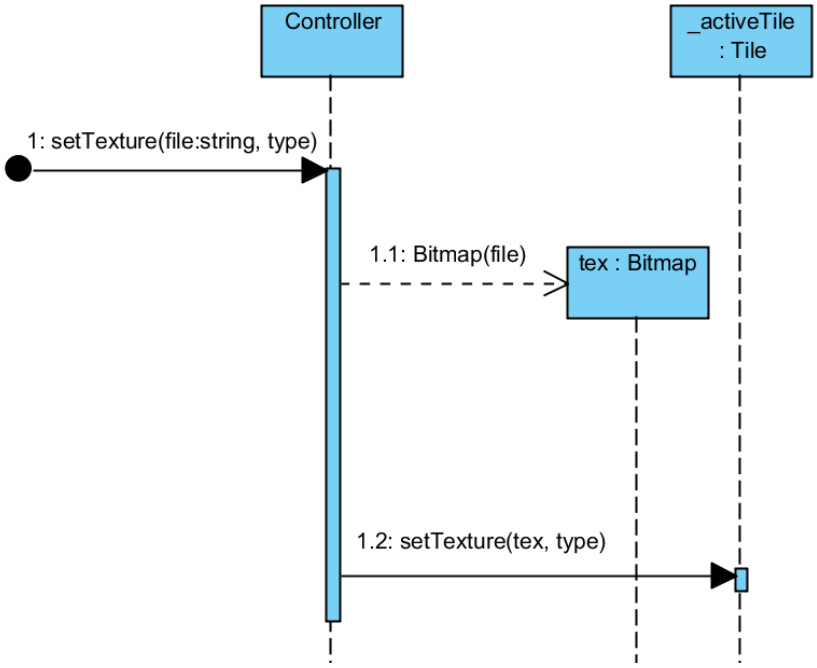


Figure 5.1.16: *Sequence diagram of the terrain designer "setTexture()" operation.*

- ***"moveTile(newPosition)" system operation:***

This operation executes when the user commands the application to change the position of the active tile, setting it to the specified new position. The position contains the two coordinates of the tile, its latitude and its longitude.

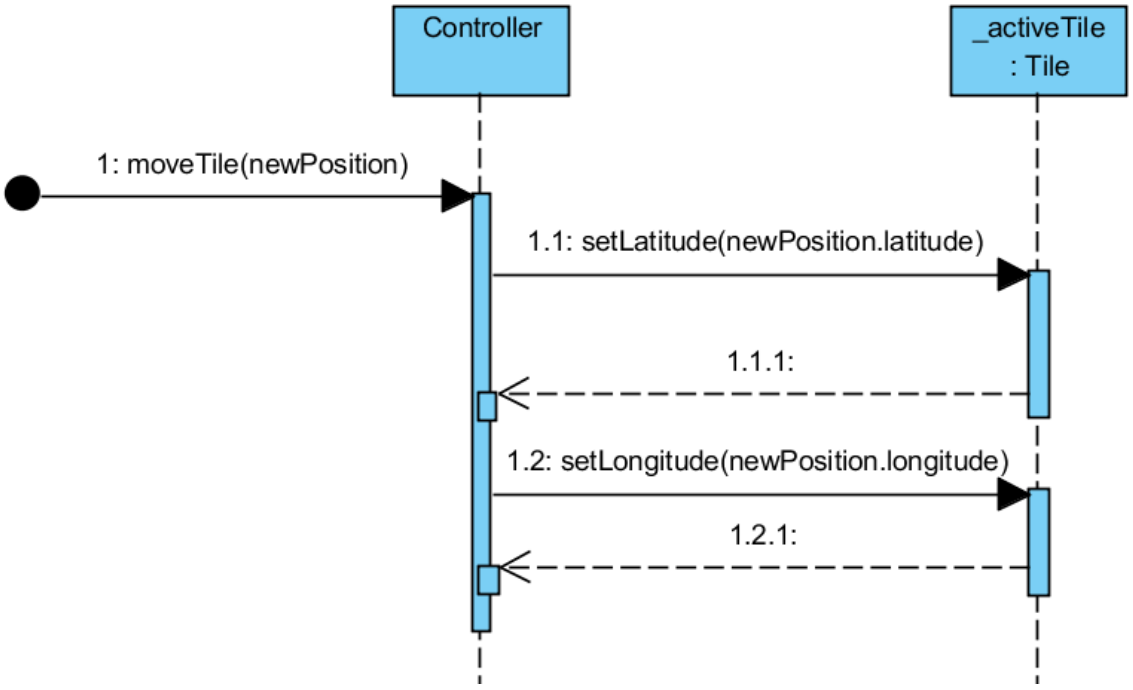


Figure 5.1.17: *Sequence diagram of the terrain designer "moveTile()" system operation.*

- ***"deleteTile()" system operation:***

This operation executes when the user commands the application to remove the active tile from the project. At deletion time, if the tile has an active height or color texture it must free the resources.

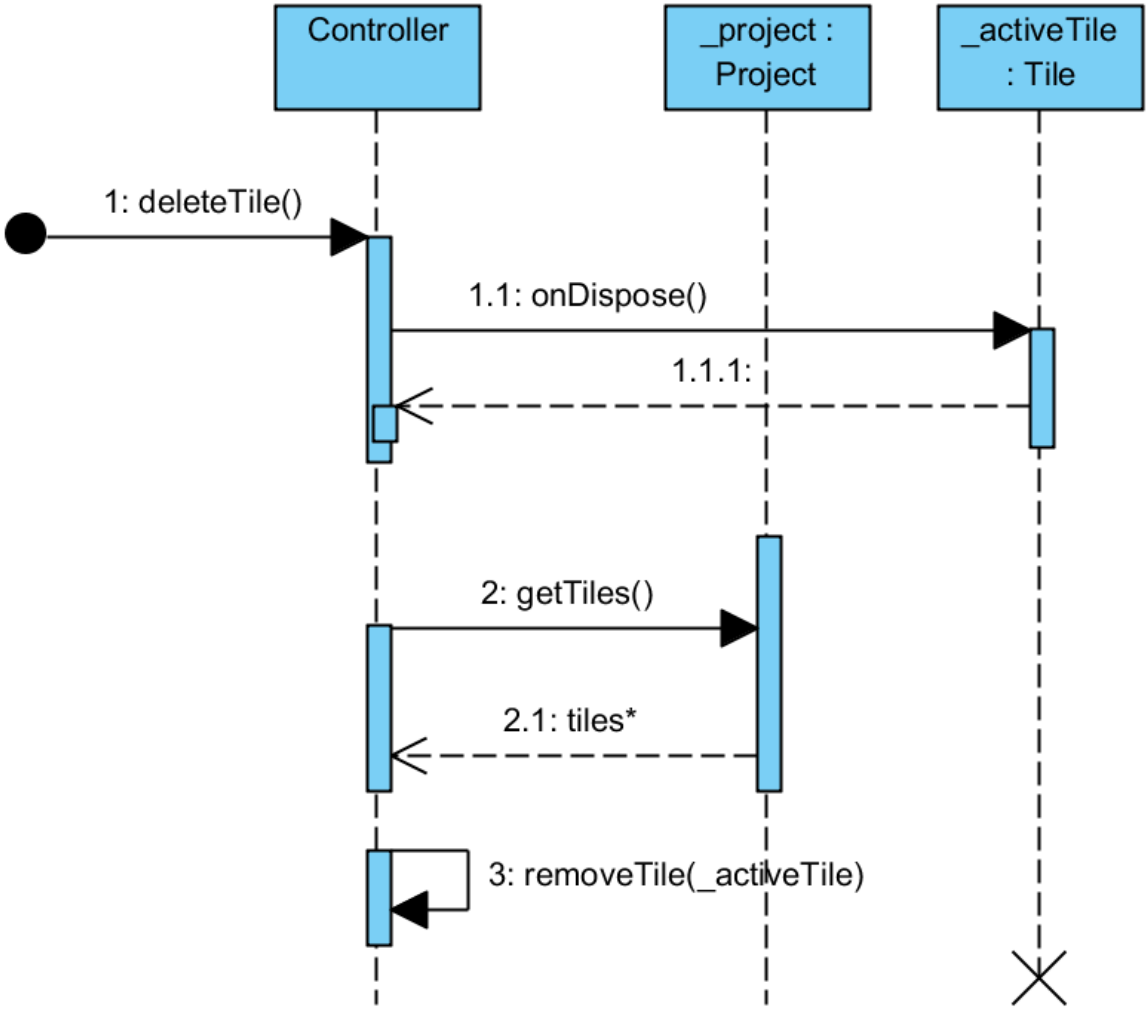


Figure 5.1.18: *Sequence diagram of the terrain designer "deleteTile()" system operation.*

- ***"createCamera()" system operation:***

This operation executes when the user commands the application to create a new camera waypoint. It will be created on the terrain center of coordinates, latitude zero and longitude zero.

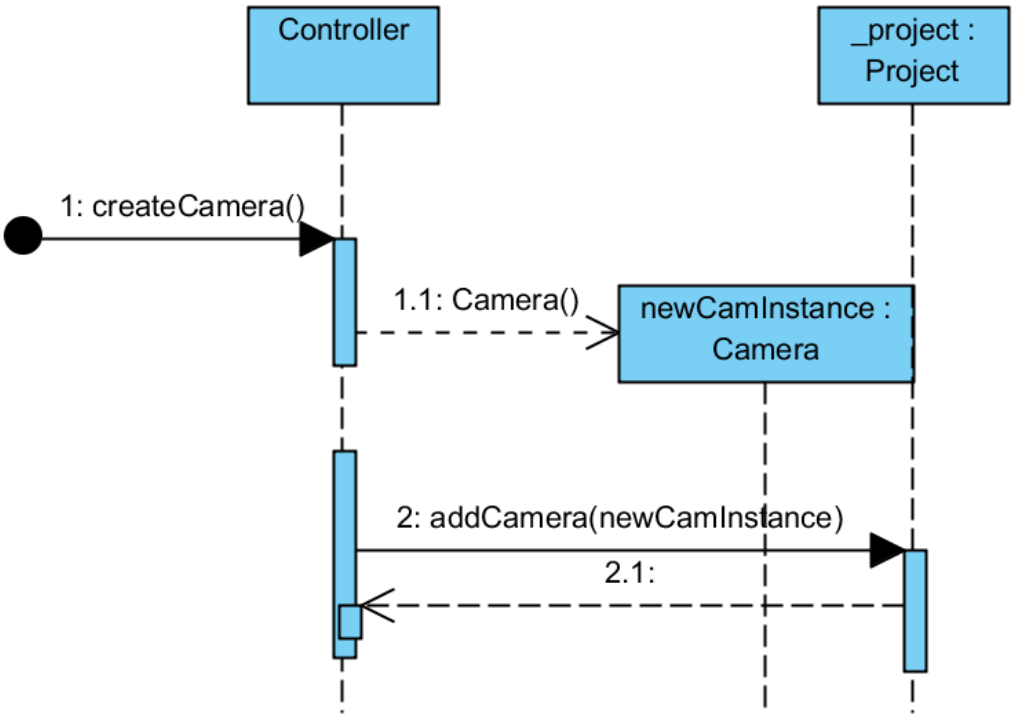


Figure 5.1.19: *Sequence diagram of the terrain designer "createCamera()" system operation.*

- ***"moveCamera(newPosition)" system operation:***

This operation executes when the user commands the application to change the position of the active camera waypoint, setting it to the specified new position value.

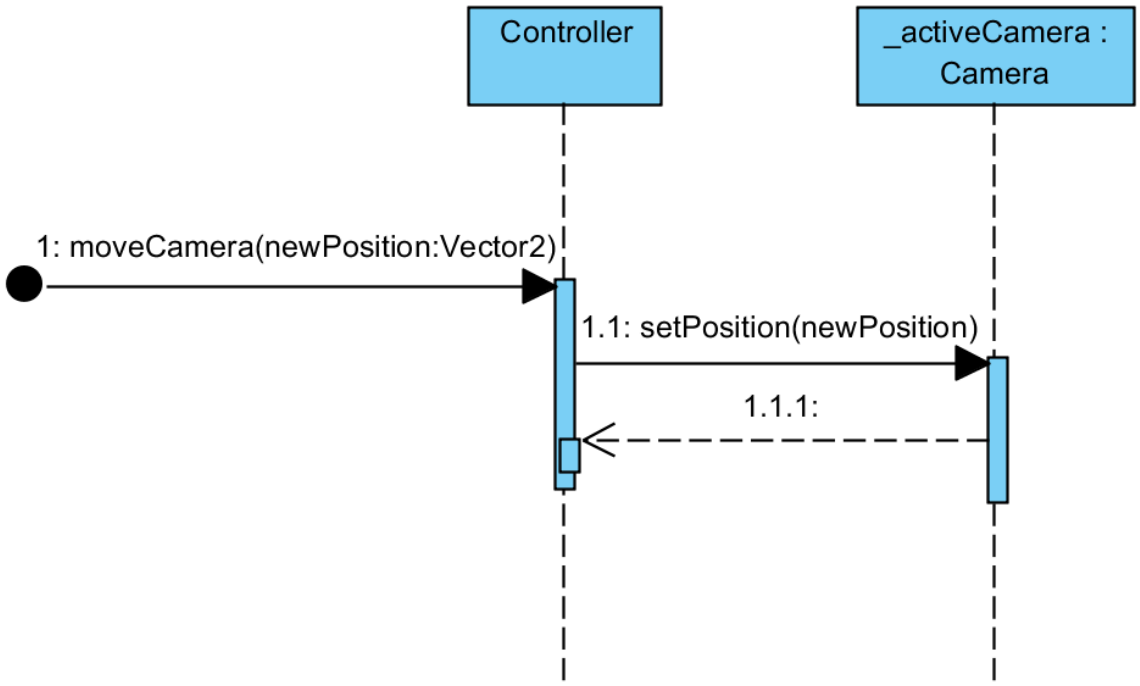


Figure 5.1.20: *Sequence diagram of the terrain designer "moveCamera()" system operation.*

- ***"deleteCamera()" system operation:***

This operation executes when the user commands the application to remove the active camera waypoint from the project.

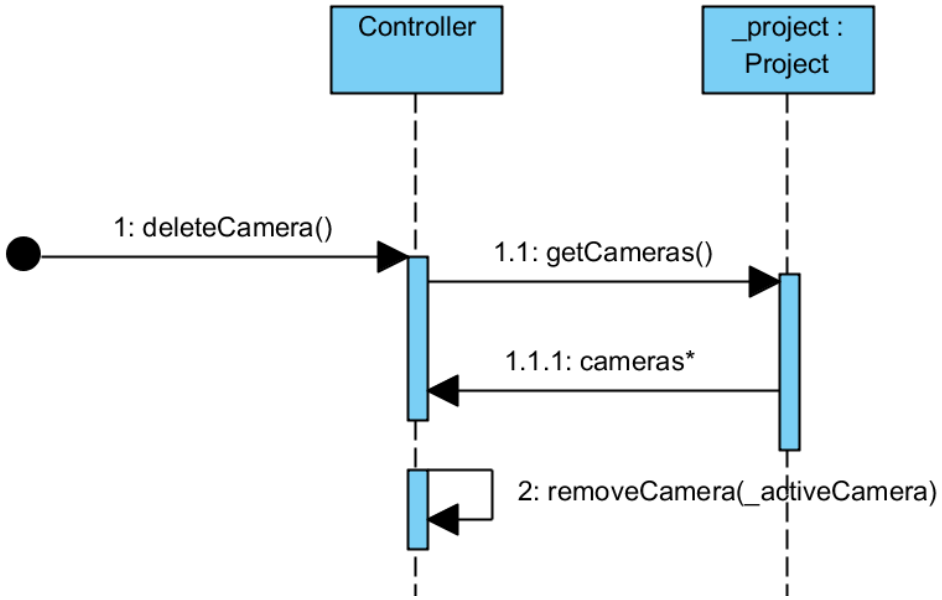


Figure 5.1.21: *Sequence diagram of the terrain designer "deleteCamera()" system operation.*

- ***"iberpixDownload(x0, y0, x1, y1, layer, path)" system operation:***

This operation is in charge of downloading the set of terrain source data from the Iberpix dataset. It connects to the Iberpix server and downloads all the terrain tiles from the tile in the coordinate {x0, y0} to the tile in the coordinate {x1, y1}, downloading the specified layer: color texture layer or height texture layer.

The downloaded data will be then saved to the specified directory, appending the coordinates of the tile to the file name. To be able to download the file, the URI to the Iberpix resource will be built for every tile using its coordinates and layer, and then download it using a HTTP connection.

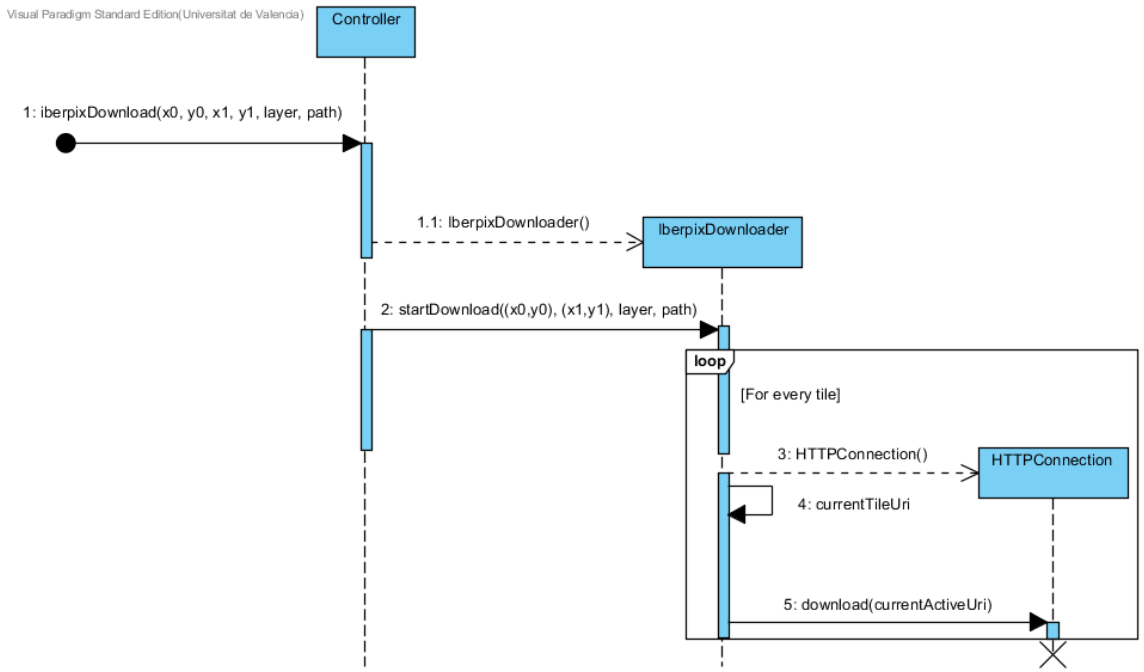


Figure 5.1.22: *Sequence diagram of the terrain designer "iberpixDownload()" system operation.*

- ***"saveProject()" system operation:***

This operation is in charge of saving the current state of the project to a storage device. It saves into a XML file the latitude, longitude and texture paths of all the tiles, along with the position of all the camera waypoints.

The sequence diagram of this operation is not included, as it is the same case as in the *"exportProject()"* system operation, but without exporting the color and height textures as TIFF files.

5.2: Conceptual Design of the Terrain Viewer

The following block diagram illustrates the interaction of the different components of the system:

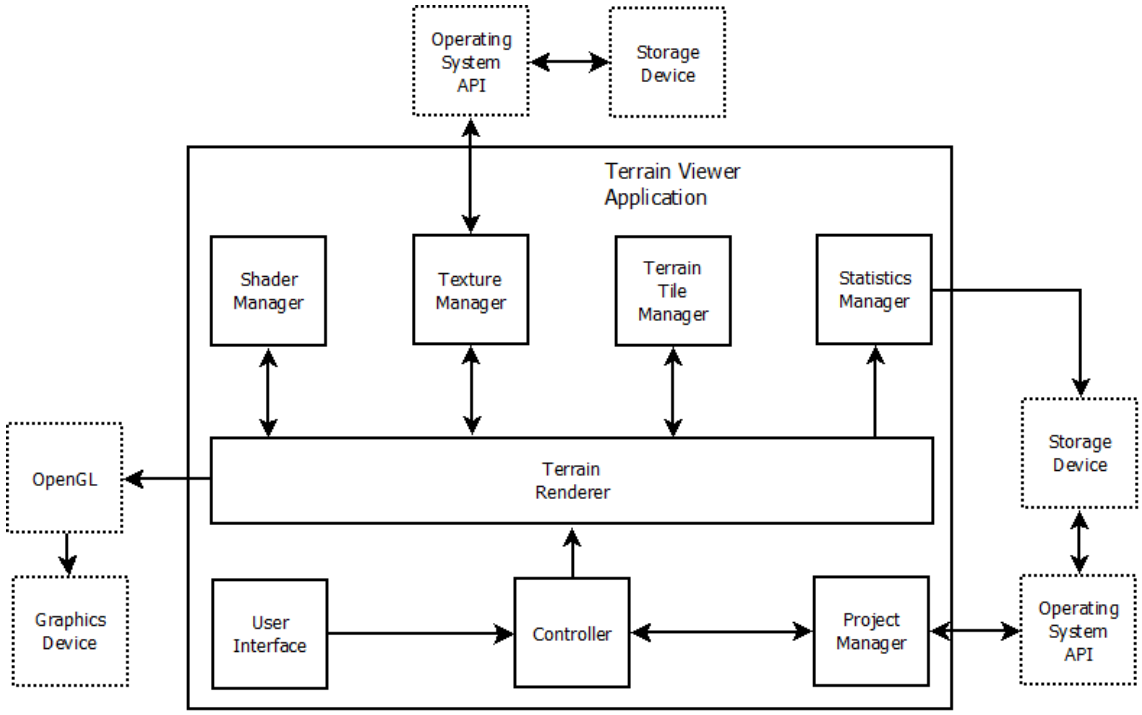


Figure 5.2.1: *Terrain Viewer Block Diagram*

On the following sections, the design of each one of the components of the application is analyzed, to build the conceptual class diagram of the application.

5.2.1: User Interface Design

The design of the user interface of the terrain viewer application is not as crucial as the design of the terrain designer application. In this application, the interface exposes the different parameters of the application to the user, which consist on changing the rendering mode and viewport, and loading a project.

Regarding the structure of the interface, the objective is creating an interface that is as less intrusive as possible for the user, since the main focus of the application is on the terrain renderer.

As stated previously, the technology used to implement the user interface is non-object oriented, where all the logic will be encapsulated into a single class. In the same fashion as in the interface of the designer application, the next step is analyzing the use cases of the application to determine which controls will compose the user interface.

- ***Use Case Analysis Oriented to the User Interface:***

The uses cases relating to controls on the user interface are the load project use case, the toggle flight use case and the toggle render settings use case.

The move view use case and the take screenshot use case are handled by the user interface component but do not have any related control. The class must have methods to handle the keyboard presses and mouse drags to respond to the user commands of these use cases.

  - ***Load Project Use Case:***

On the first step of the main branch, the user commands the application through a button, which will be implemented as a button called "*Load Project*". On the second step, the user introduces the path of the project on a dialog, which will be implemented using a standard open file dialog.

On the error branch, the user is informed of the error through a message dialog. This will be implemented through a standard message box dialog.

- ***Toggle Flight Use Case:***

The toggle flight use case is fired when the user commands the application to start or stop the camera flight. This will be implemented as a button, which will display the “Start” label when the flight is stopped and the “Stop” label when the flight is active.

- ***Toggle Render Settings Use Case:***

This use case is used to change between wireframe and fill polygon modes, to enable and disable automatic level of detail management and to change the minimum distance and maximum factor of level of detail.

The implementation will consist on checkboxes to switch the polygon mode and the automatic level of detail management, and edit fields to change the maximum and minimum level of detail factor and distance respectively.

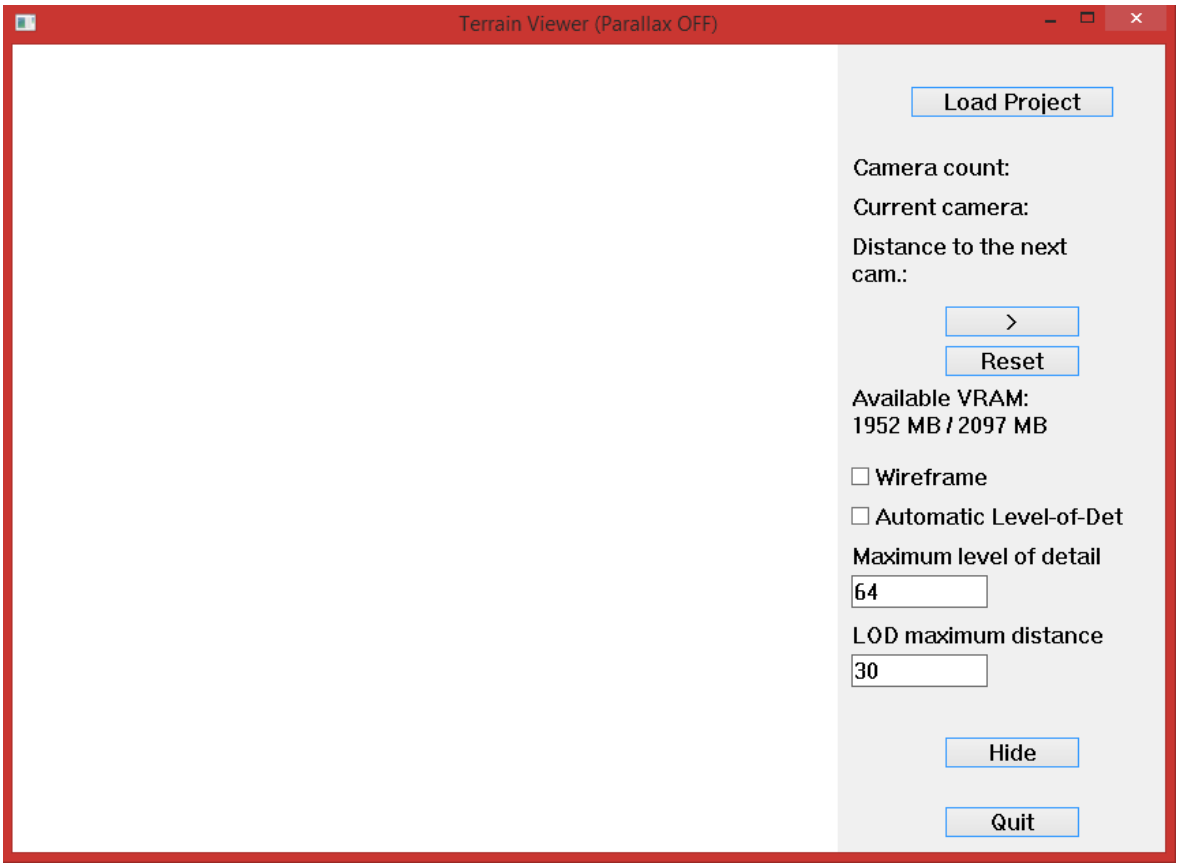


Figure 5.2.2: *User interface of the terrain viewer application.*

- ***Summary:***

This component translates into a single class encapsulating all the code of the user interface, "*WindowsApplication*".

5.2.2: Project Manager Design

The project manager is the component in charge of loading the data from the compiled project, obtaining all the information out of the project file and creating its associated terrain tiles.

It will translate into a project class, containing methods and attributes to set all the information relevant to the project and to load the data from an existing project file.

To convert the height and color texture files into data which can be understood by the terrain renderer, it will communicate with the texture manager, and the same will happen with the terrain tile manager to create the needed geometry for each one of the tiles.

- ***Summary:***

The project manager component will translate into a single class called "*Project".*

5.2.3: Terrain Renderer Design

The terrain renderer is the main component of the terrain viewer application. It is responsible of creating the visualization of the terrain using the data provided by the project manager, and of coordinating and keeping track of the data of all the other components related to the terrain renderer.

It consists of a controller renderer class, containing methods to draw terrain tiles as well as complete projects. The management of the projection of the terrain translates into a camera class, containing all the information about the point of view of the terrain.

To let the user do flights over the terrain, the terrain renderer also contains a class encapsulating the behavior of the terrain flight, which will calculate the position of the camera at every moment following the trajectory created by the waypoints. This trajectory is calculated using classes to evaluate cubic curves, specifically uniform B-Splines and Bézier curves.

- ***Summary:***

The terrain renderer component translates into the *"Renderer", "Camera", "CameraTravel", "CubicCurve", "BezierCurve"* and *"UniformBSpline"* classes.

5.2.4: Shader Manager Design

The shader manager is in charge of loading the shader code from files, compiling and linking the shaders into shader programs and of keeping track of the compiled programs.

This is implemented as a shader class, with methods to load the shader code from the storage device and compiling it into shader object code, and as a shader program class, with methods to link the different shader objects into a complete shader program, which will be used by the renderer to set-up the various variables and attributes used by the shaders.

- ***Summary:***

The shader manager component translates into the *"ShaderProgram"* and *"Shader"* classes.

5.2.5: Texture Manager Design

The texture manager is the component in charge of loading the height and color data of the terrain projects, processing the data as needed and finally uploading it into the graphics card. It keeps track of all the loaded textures, letting the terrain renderer access them as necessary.

Height textures have their data encoded in a specific format and contain not only height data but also the normal vectors of the data. The texture manager must be able to process and generate all needed additional data for the rendering of the terrains.

This will be implemented as a texture class, representing a texture object in the application, and the texture load and data decoding and generation will be implemented on a generic texture loader class, having one specific case to load TIFF image files.

Besides the default texture objects, in order to be able to take captures of the rendering area by the statistics manager the texture manager also implements a memory texture class, where the texture contents are saved on the main system memory instead of being saved on the graphics device memory.

- ***Summary:***

The texture manager component translates into the "*Texture", "ImageLoader", "TIFFImageLoader"* and *"MemoryTexture"* classes.

5.2.6: Terrain Tile Manager Design

The terrain tile manager is the component in charge of creating and storing the underlying geometry and other required data of the terrain tiles, in order to supply them to the terrain renderer component at drawing time.

This is achieved through implementing a terrain tile class, having methods and attributes to store the latitude, longitude and references to the height and color texture objects associated to the tile.

The project manager project class keeps track of the existing tiles on a project by storing references to tiles on the terrain tile manager.

- ***Summary:***

  The terrain tile manager translates into the *"Terrain"* class.

5.2.7: Statistics Manager Design

It is the component in charge of keeping track of statistical data relevant to the study of the performance of the rendering methods. The different components submit a sample of the process to the statistics manager, which will calculate the mean of the submitted data when the application closes. This is implemented through a profiler class, containing all required methods for its operation.

The statistical data collected is: texture loading from the storage device to main system memory time, texture loading from main system memory to the graphics device loading time, system reported and estimated graphics memory used by the application, project loading time, single tile loading time and complete project loading time.

- ***Summary:***

  The statistics manager translates into the *"Profiler"* class.

5.2.8: Controller Design

The controller is the component in charge of the coordination of the rest of the components. Due to the nature of the environment chosen for the development of the application, the controller will be split between the user interface controller class and the main entry point function.

- ***Summary:***

  The controller translates into the *"WindowsApplication"* class and partly into the application main entry point function.

5.2.9: Design Class Diagram

Following the example of the terrain designer application, the high level descriptions of the different components of the application are used to build the design class diagram.

This diagram will be the basis for the implementation of the application, although the methods and attributes of the classes will change, and other helper classes or elements will be added depending on the peculiarities of the implementation.

In the next section, the system operations inside each one of the use cases of the terrain viewer application will be analyzed to model the interaction of each one of the system operations inside them, illustrating the communication between the different classes of the system to solve the different operations.

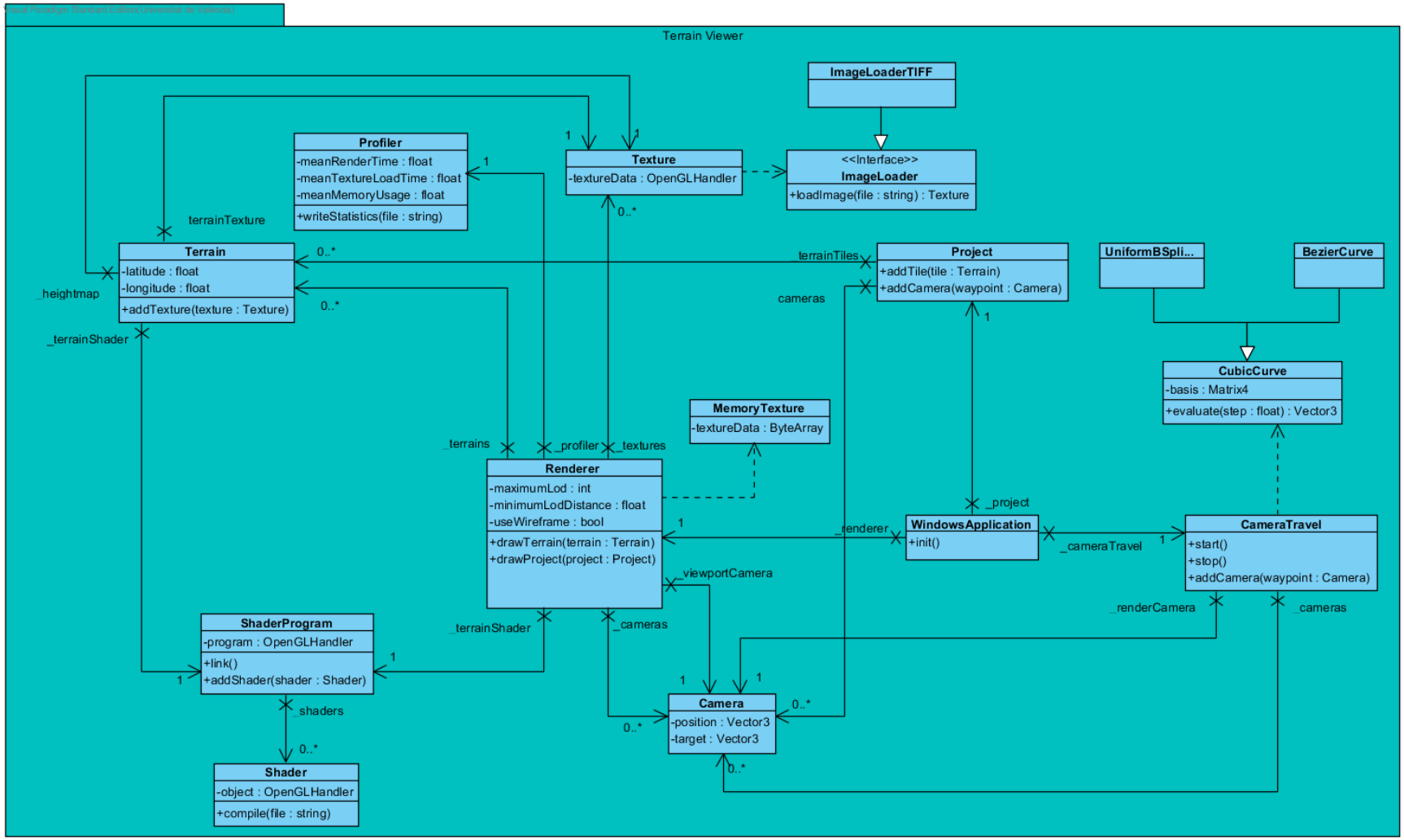


Figure 5.2.3: *Design class diagram of the terrain viewer application.*

5.2.10: System Operation Interaction Diagrams

In this section, the system operations are analyzed to create the interaction diagrams of the different operations, showing how the different classes communicate to obtain the solution to the user commands, along with their contracts.

- ***"init()" system operation:***

This operation executes the first time the user loads the application. It is responsible of loading all the subsystems required for the working of the application, initializing the user interface and creating the terrain rendering context, loading all the required shaders and creating the rendering viewport camera.

The initialization of the user interface involves creating the application dialog with all its controls and setting their callbacks. The initialization of the terrain renderer involves the creation of the required hardware context to access the graphics device. Both will be explained further in the implementation chapter.

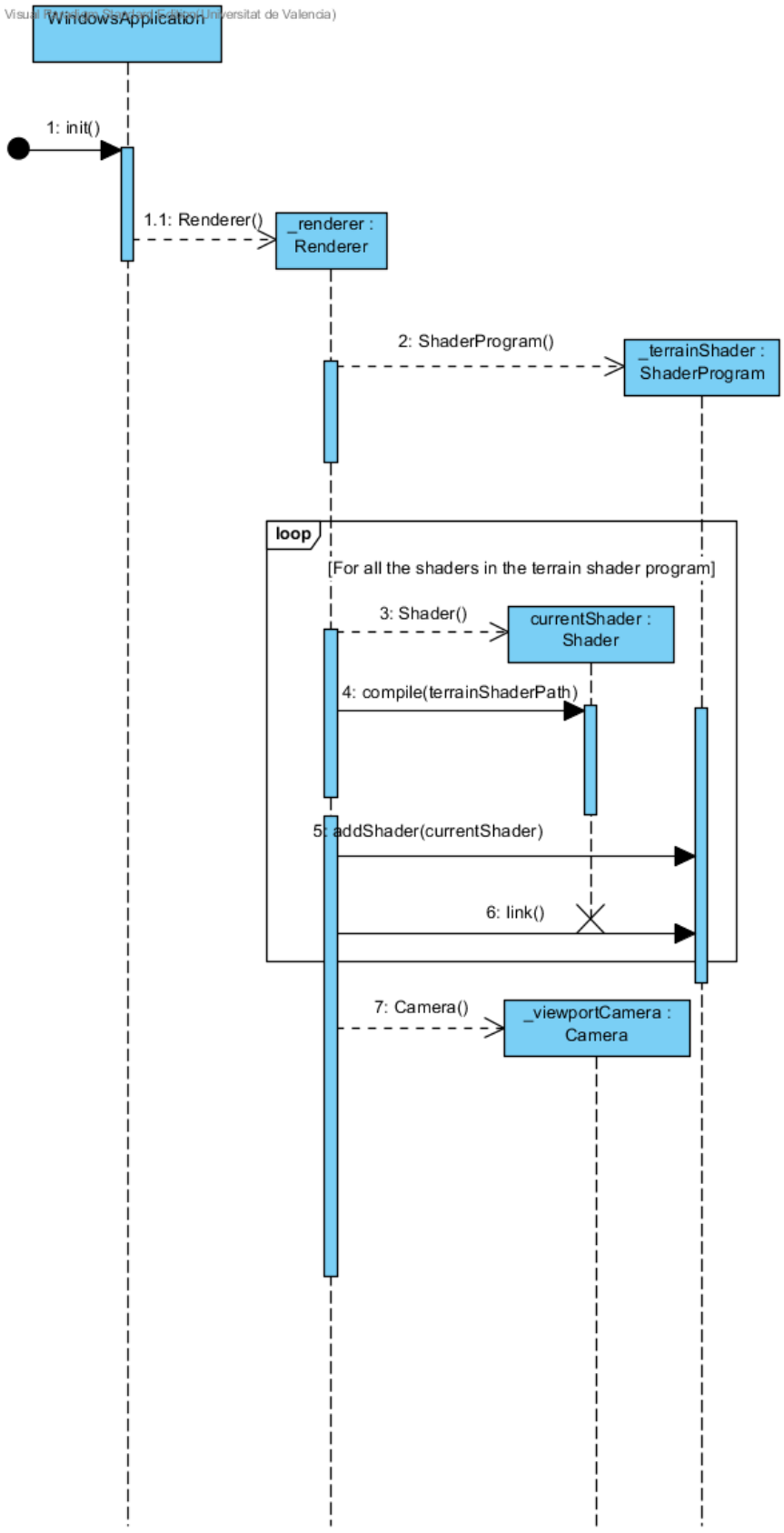


Figure 5.2.4: *Sequence diagram of the terrain viewer "init()" system operation.*

- ***"loadProject(file)" system operation:***

This operation executes when the user commands the application to load a terrain project. It is in charge of creating the project instance, reading from the XML file the information about all the tiles and camera waypoints and adding them to the project, loading the height and color textures from the storage device and setting them to each one of the terrain tile instances.

At the texture loading time, the height texture must be treated correctly since the data is exported with a specific structure, which will be explained further in the implementation chapter.

When the project is loading the camera flight waypoints, it also adds the camera instances of each waypoint to the camera travel instance, which will enable the user to do a camera flight animation when the project has finished loading.

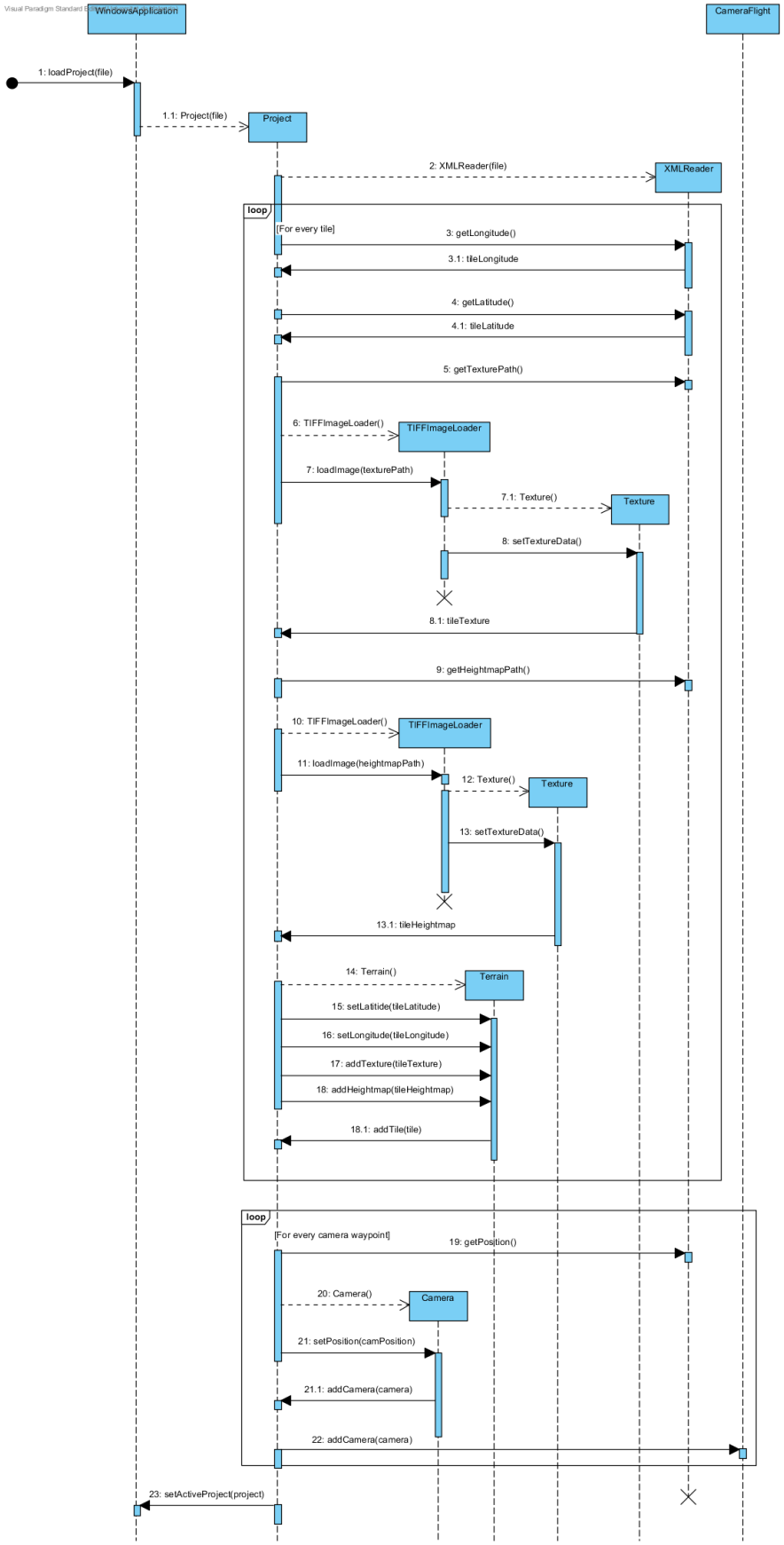


Figure 5.2.5: *Sequence diagram of the terrain viewer "loadProject()" operation.*

- ***"moveView(newPosition)" system operation:***

This operation is executed when the user commands the application to change the position of the viewport camera on the renderer.

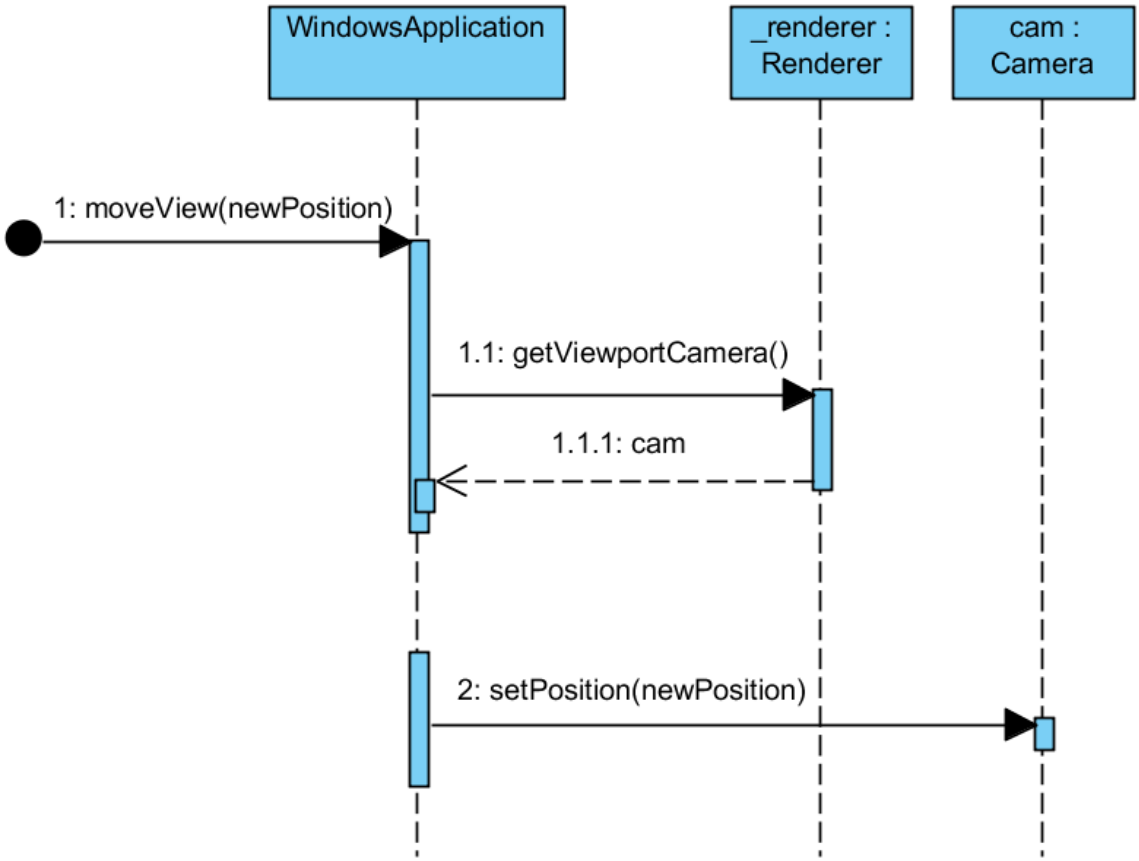


Figure 5.2.6: *Sequence diagram of the terrain viewer "moveView()" system operation.*

- ***"startFlight()" system operation:***

This operation executes when the user commands the application to start the camera flight animation. To be able to do so, the loaded project must have enough waypoints to enable the camera flight, which will be explained further in the implementation chapter.

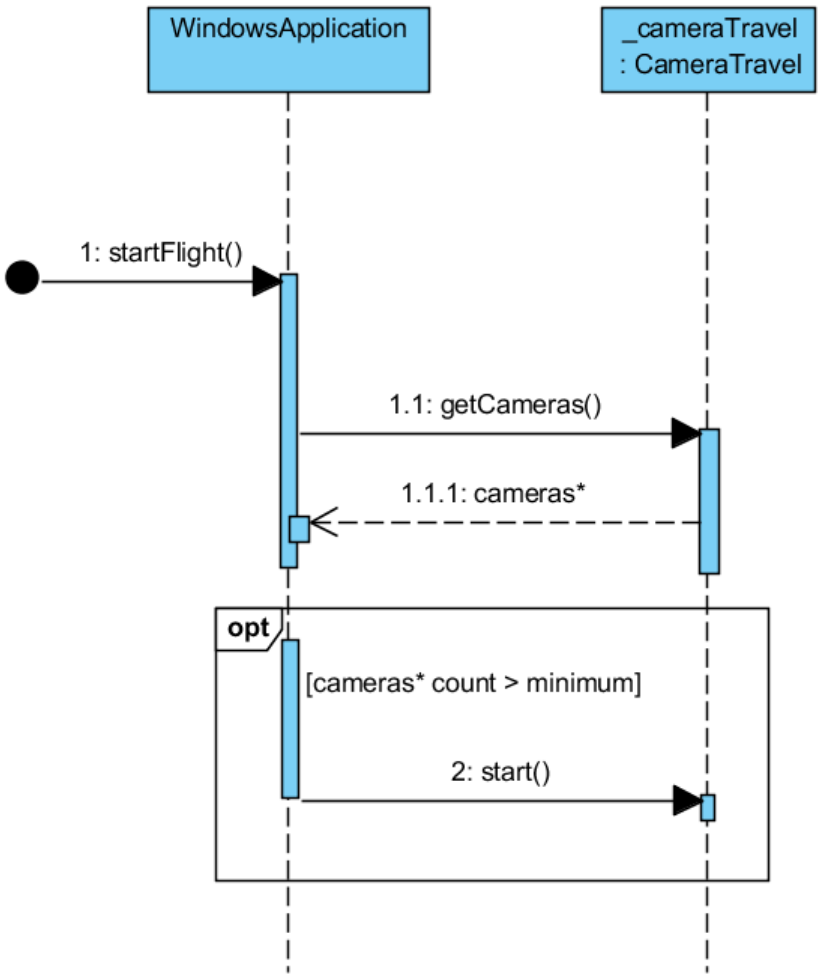


Figure 5.2.7: *Sequence diagram of the terrain viewer "startFlight()" system operation.*

- ***"stopFlight()" system operation:***

This operation is executed when the user commands the application to stop the camera flight animation. It will only be stopped if the camera flight animation is active.

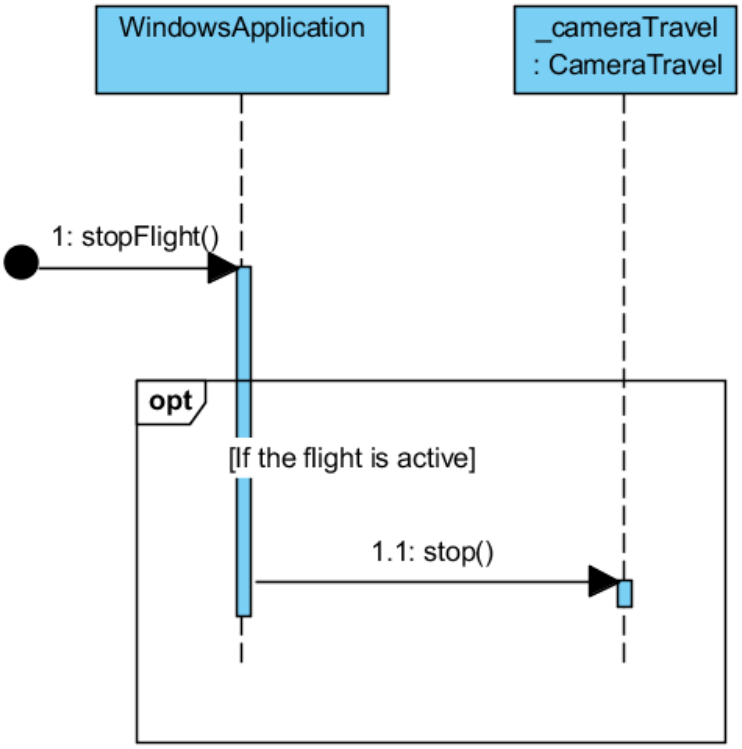


Figure 5.2.8: *Sequence diagram of the terrain viewer "stopFlight()" system operation.*

- ***"takeScreenshot()" system operation:***

This operation is executed when the user commands the application to take a screenshot of the rendering viewport. To achieve this, the renderer creates a copy in the system memory of the rendering viewport, and it is saved to the storage device as a TIFF image file.

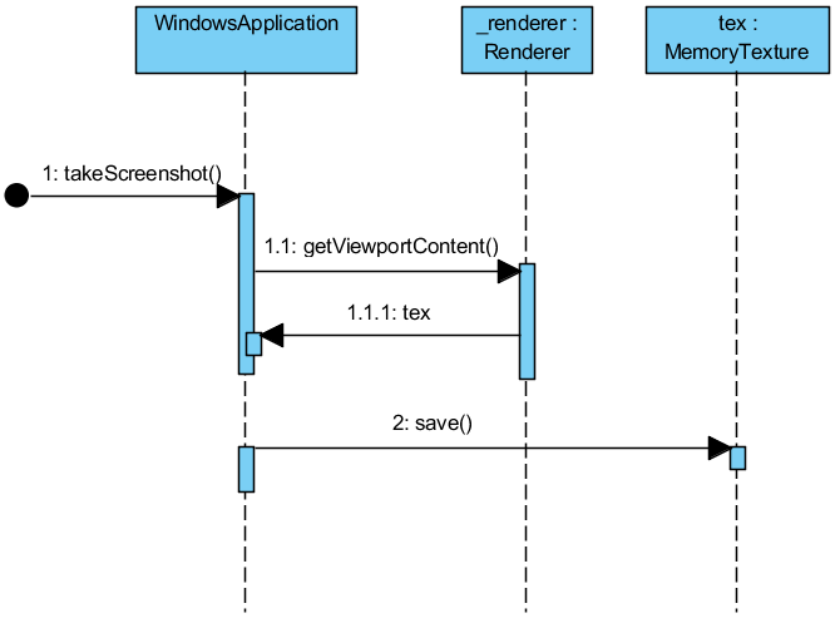


Figure 5.2.9: *Sequence diagram of the terrain viewer "takeScreenshot()" system operation.*

- ***"toggleWireframe(newState) system operation:***

This operation is executed when the user commands the application to change the wireframe mode of the renderer to the specified state.

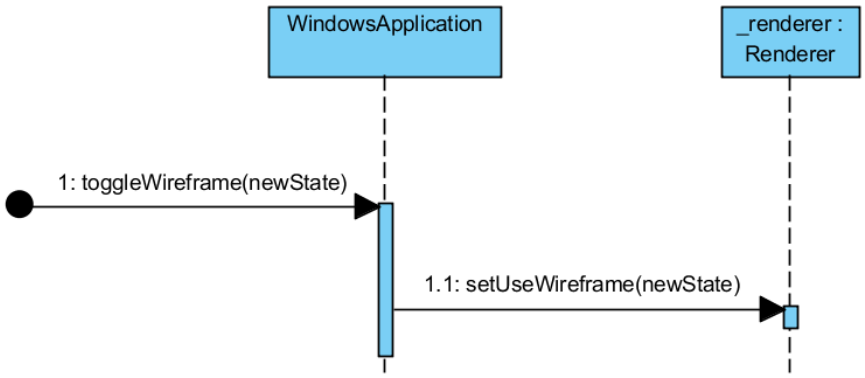


Figure 5.2.10: *Sequence diagram of the terrain viewer "toggleWireframe()" system operation.*

- ***"toggleAutoLod(newState)" system operation:***

This operation is executed when the user commands the application to change the automatic management of the level of detail of the renderer to the specified state.

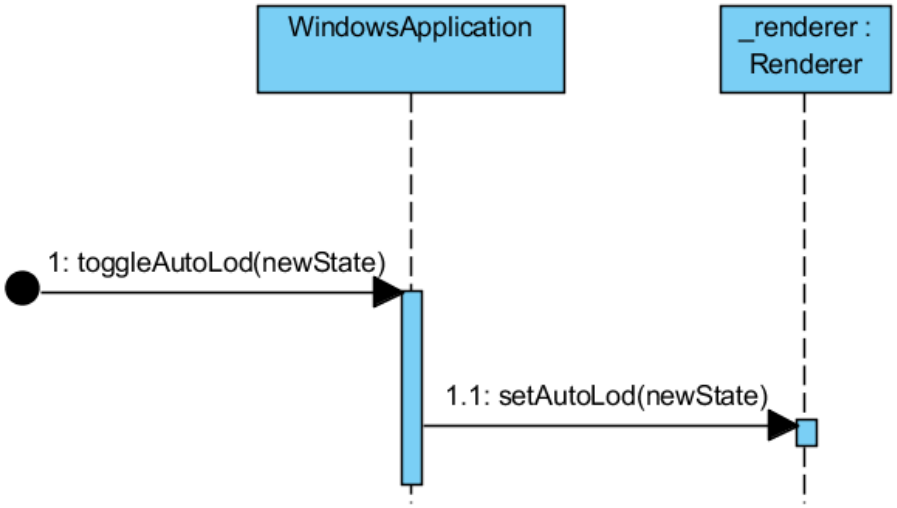


Figure 5.2.11: *Sequence diagram of the terrain viewer "setAutoLod()" system operation.*

- ***"setMaxLod(newLodLevel)" system operation:***

This operation is executed when the user commands the application to change the maximum level of detail of the renderer to the specified value.

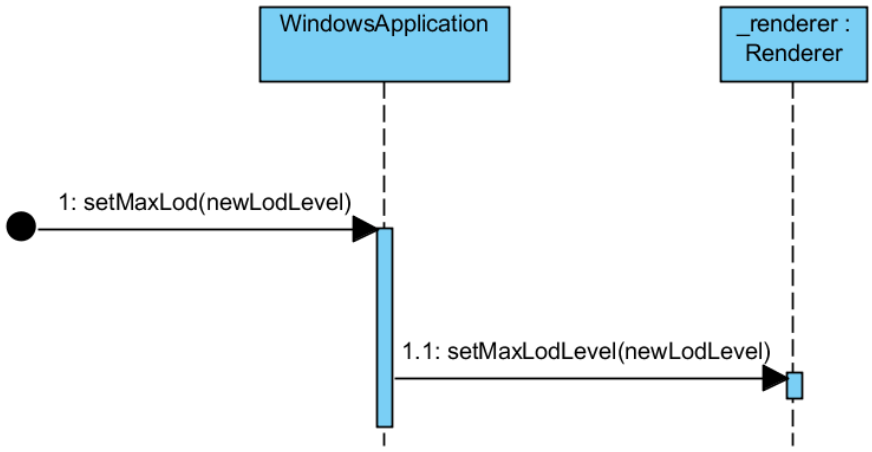


Figure 5.2.12: *Sequence diagram of the terrain viewer "setMaxLod()" system operation.*

- ***"setMinLodDistance(newDistance)" system operation:***

This operation is executed when the user commands the application to set the minimum level of detail distance for automatic LOD management of the renderer to the specified value.

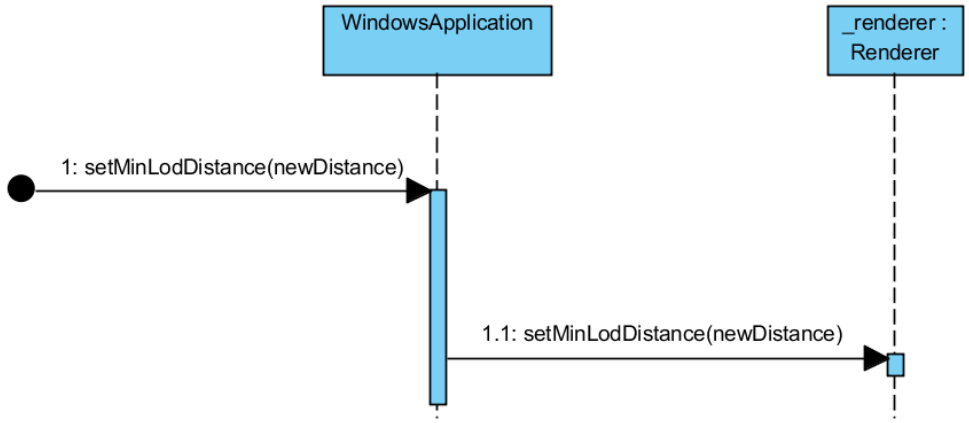


Figure 5.2.13: *Sequence diagram of the terrain viewer "setMinLodDistance()" system operation.*

5.3: Summary

In this chapter, the different system operations were analyzed to model their behavior using sequence diagrams. These diagrams will be the basis for the implementation of the application.

The sequence diagrams give a high level representation of the behavior of the application, although many characteristics of the application are left for the implementation stage, since they may change depending on the target platform.

In the next chapter, the implementation of both applications will be tackled, showing the peculiarities and difficulties of the different methods and algorithms of the system.

CHAPTER 6: SYSTEM IMPLEMENTATION

This chapter introduces the implementation details of the system, introducing first the details of the terrain viewer application and then the details of the terrain designer application.

To implement the applications using the technologies selected in the state of the art chapter, there are many peculiarities due to the programming environment, libraries and graphics subsystems selected to develop the applications.

On the implementation details are explained the difficulties encountered when translating the classes of the systems into the final application, the mathematical background behind the approaches taken to get to the solution of the different algorithms and details of the implementation of said algorithms.

The applications have been designed using a block structure as stated in the previous chapter, the system design chapter. This chapter will follow the different blocks of both applications to explain their implementation.

6.1: Implementation of the Terrain Viewer Application

The terrain viewer application is the main focus of the project. It is responsible of rendering the terrain projects and of generating the needed statistics to do the performance analysis of the different rendering methods that have been implemented.

For this application, the selected implementation environment was C++ along with Visual Studio and the selected graphics rendering backed was OpenGL. Many of the decisions taken during the implementation stage are due to the selected technology.

Following are the implementation details for the different blocks that compose the terrain viewer application. Since in this application the user interface is encapsulated inside the controller, on the *WindowsApplication* class, both blocks are described together.

On the other hand, the renderer block and the different blocks that are dependant of it (texture, terrain tile, shader and statistics managers) are also described together.

6.1.1: User Interface and Controller Implementation

The user interface and the controller blocks are encapsulated into the *WindowsApplication* class as well as on the main entry point function of the application. It is in charge of coordinating the different classes of the application and of receiving the user commands from the user interface, translating them into the system operations.

- ***Structure of the controller and user interface:***

In the terrain viewer application, the method selected to create the user interface of the application is using the Windows API directly. In applications using the Windows API, applications are structured using asynchronous design, where the application receives messages created by the operative system and reacts to them.

When the application is initialized, the different windows composing the application user interface are instantiated. In an asynchronous fashion, when the windows are created, the operative system issues a message to the application notifying the creation. The application is then responsible of creating all the controls which compose the dialog that has just been created.

Messages are received on a callback method, where the application decodes the messages that have been posted by the operative system, and reacts appropriately.

The application must check for any queued operative system messages in a loop. When there is an available message, it will be dispatched, firing the callback function to handle it, as illustrated in the figure *6.1.1*.

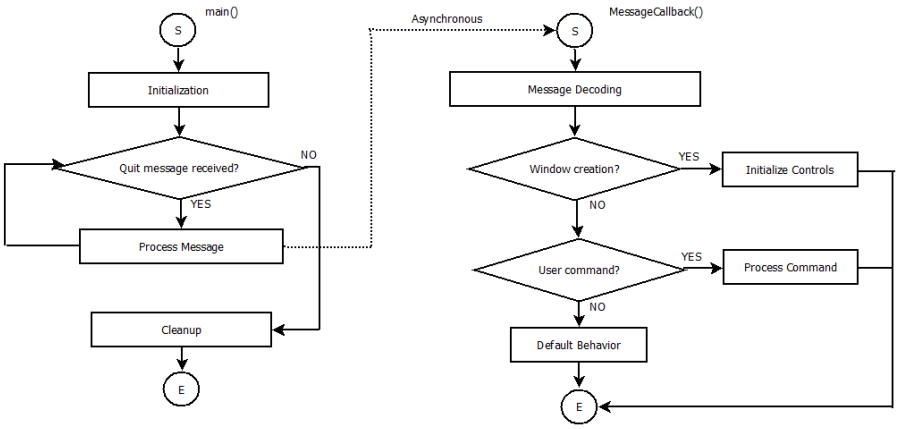


Figure 6.1.1: *Barebones structure of an application using the Windows API.*

Taking profit of this structure, the different components of the application have been implemented using a similar structure. In this case, the components have a *"DoFrame()"* method and optionally an *"EndFrame()"* method.

The first is called on every iteration of the main loop, before the messages are processed. The second is called on every iteration of the main loop, after the messages are processed, although this second method is only on the components which must do any kind of processing when the frame finishes.

Taking this into account, the structure is as follows: first, the controller is instantiated on the main entry point of the application. Then, on a loop checking whether the application should keep working or not, the main loop calls the *"DoFrame()"* and *"EndFrame()"* methods of the controller, which in turn will call the corresponding methods on the components controlled by it, the *Renderer* class, the *CameraTravel* class and the *Profiler* class.

The message decoding is encapsulated on the *WindowsApplication* class, leaving the main loop only with the responsibility of calling these two methods. On the other hand, the message callback is also handled by the *WindowsApplication* class, where the system operations will be called depending on the issued command. The figure *6.1.2* illustrates the initialization of the viewer application.

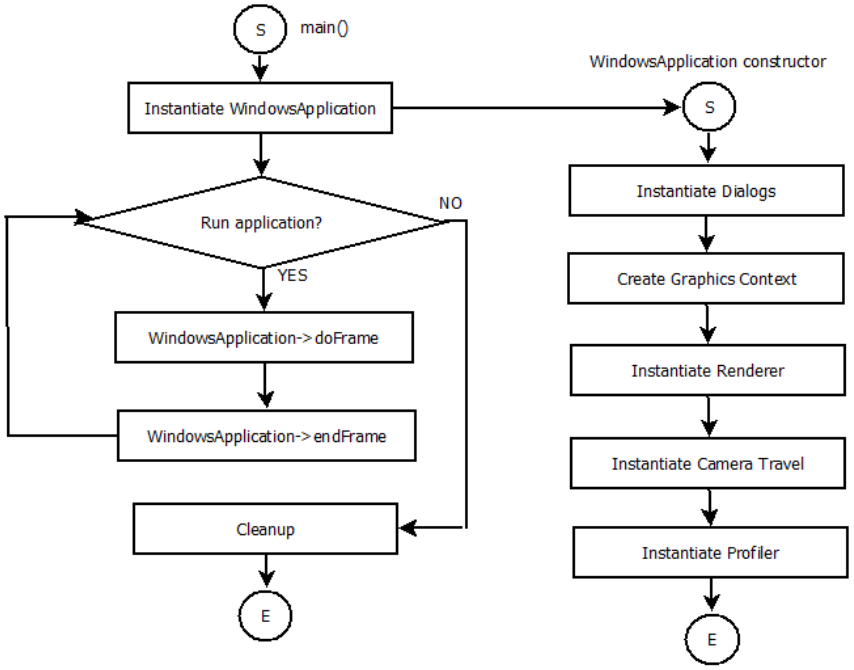


Figure 6.1.2: *Initialization of the viewer application.*

Following is explained in detail the implementation of the operations done to initialize the *WindowsApplication* class and the structure of the *"DoFrame()"* and the *"EndFrame()"* methods.

- ***Dialog Instantiation:***

In the previous chapter were described the various controls that compose the user interface, and it was stated that it should be as less intrusive as possible. To achieve this, the final structure of the user interface is using a single dialog with a small side bar. The dialog exposes the viewport of the terrain renderer, while the sidebar exposes all the controls stated on the system design.

To achieve this, the implementation requires using three dialogs: a main dialog, which serves as a container for the rendering area and for the side bar area, another dialog which will hold the graphics context for the terrain renderer and an additional dialog which will serve as the side bar holding the controls of the application. The two later dialogs are child dialogs of the container dialog.

As explained on the structure of a general Windows API application, the creation of the dialogs is not a blocking operation. When the creation commands are issued, the operating system will start creating the dialogs, and when the dialogs are created, the message callback function will be executed notifying the application about it. It will be then time for the application to instantiate the controls of the side bar. This process can be seen in the figure *6.1.3*.

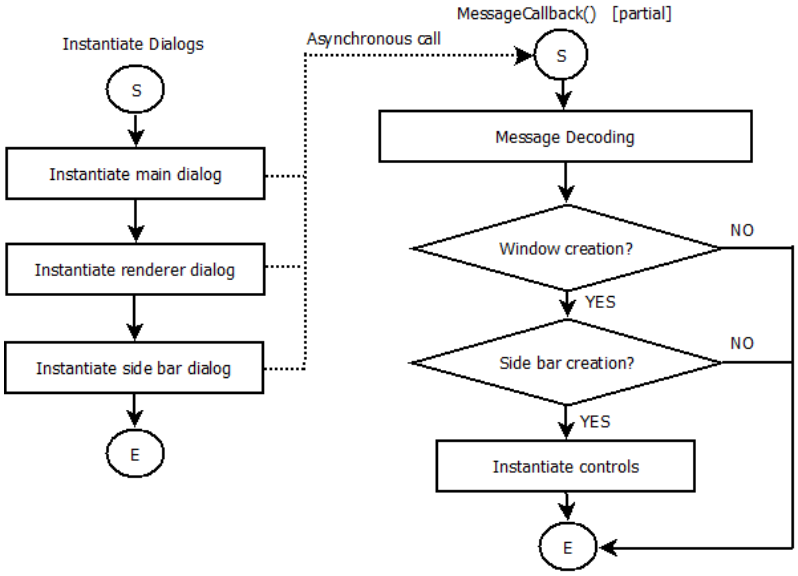


Figure 6.1.3: *Instantiation process of the dialogs of the application.*

- ***Context Creation:***

As stated on previous chapters, the selected technology to use as the 3D rendering backend is the OpenGL library. To be able to use the library a graphics context must be created first.

A graphics context is a link between the actual surface on the screen and the operations issued to the OpenGL library. OpenGL does not have any notion of screen surfaces, monitors or windows, since this different for every platform.

In this application, the renderer dialog is linked to a graphics context, which lets the operating system pipe the scene rendered by OpenGL to the dialog, letting the user visualize the rendered scene.

This is not a trivial task, and to better understand why a brief summary of the history behind OpenGL is explained first.

OpenGL was born in the 1990's as an evolution of an early 3D graphics technology introduced by Silicon Graphics in the 1980's, IrisGL, to build an industry standard for 3D graphics rendering. OpenGL offers a low level, platform agnostic interface to the graphics pipeline, exposing the hardware capabilities of the hardware to the application.

The structure of OpenGL is similar to a state machine: the application issues commands with the information of the scene: the position of the geometry to be renderer, the position and intensity of the different lights, the coloring of the geometry, etc. The appearance of the rendered scene will depend on the set-up of the state machine at render time.

On early versions of OpenGL, the commands to change the state machine were issued at one data item at a time, which soon deemed too slow for the amount of data used in 3D scenes. Following versions started adding improvements, first issuing data in arrays and later introducing several kinds of objects encapsulating the data to be issued to the graphics hardware.

One of the major changes to the library was the introduction of user programmable shading, through *shaders*. Shaders are small programs defining how the vertex data should be processed, and how the rasterized fragments should be colored. In its early versions, the transform and shading algorithms were fixed and only accessible by issuing commands to the state machine. Now the algorithms are user programmable, giving complete customization to the programmer.

Although these additions supposed a major improvement to the library, many critical applications still relied on its earliest features. To allow old applications to keep working on modern systems, it was decided to make the new versions of the library backwards compatible with its previous versions. This was done introducing the new features of the library as extensions to the bare bones early version of the library.

After several iterations, this decision became a problem since the library was becoming too cumbersome to use, with many extensions which may be or may be not implemented on the target graphics hardware, and with varying behaviors depending on the target system.

To resolve this problem, the library was separated into two profiles, the compatibility profile and the core profile. The compatibility profile was based upon the early behavior of the library and its extensions, but would not be further improved, while the core profile supposes a start from the ground up with all the newer features of OpenGL, and is not compatible with the early features of the library. The OpenGL 3.0 version is the last compatible version, while all the posterior versions belong to the core profile.

In the viewer application, the required minimum OpenGL version is the 4.3 version, which exposes the hardware tessellation features of the graphics card. The graphics context used by the application must be able to use this core profile version.

The Windows API exposes a set of functions to create graphics contexts to draw an OpenGL rendered scene on a window dialog. To do so, a handle to the graphics device being used to draw the target dialog to the screen is obtained, and then the context is created. The following pseudo code illustrates how the graphics context is created.

```
Function createContext
    deviceHandle <- GetDC(rendererDialog)
    graphicsContext <- wglCreateContext(deviceHandle)
End
```

At clean up time, the graphics context is released, removing the link between the dialog and the graphics hardware.

The version and profile created by the *"wglCreateContext()"* function depends on the target platform, operative system version and environment. Usually, the created context will be a compatibility profile, exposing the latest compatibility profile version supported by the graphics hardware.

In this application the required OpenGL version is the 4.3 version. To ensure that the created context is a core profile context compatible with said version, the *"CreateContextAttribsARB"* OpenGL extension is used. This extension enables the application to fine tune the details of the context creation. Since it is an OpenGL extension, a graphics context must be created before using it.

The following pseudo code illustrates how the required graphics context is created, using said extension.

```
Function CreateContext
    deviceHandle <- GetDC(rendererDialog)
    temporaryContext <- wglCreateContext(deviceHandle)

    If CreateContextAttribsARB extension is supported
        setup <- {Use version 4.3, core profile}
        graphicsContext <- wglCreateContextAttribsARB(setup)
    End

    wglDeleteContext(temporaryContext)
End
```

- ***"DoFrame()" and "EndFrame()" methods:***

These two functions are in charge of notifying the *Renderer* and *CameraTravel* classes that a new frame has started and that the frame is about to end. This way, both the rendered viewport and the camera flight animation can be updated continuously.

The first is also in charge of translating and dispatching the operative system messages to the message callback, and of commanding the renderer to draw the active project, if there is one. When the method starts, a timer starts tracking the time needed to complete a frame. When the method ends, the elapsed time is submitted to the profiler, to be able to create the statistics of the mean rendering time.

The second method is in charge of commanding the graphics device to swap buffers. When drawing commands are issued, the rendered scene does not get directly drawn to the screen. It is drawn on a buffer stored in the graphics device memory.

When a frame ends and all the drawing commands have been issued, the screen buffer and the memory buffer are swapped, showing the rendered scene on the screen.

The following pseudo code illustrates the behavior of both methods from the *WindowsApplication* class.

```
Function DoFrame
    Start tracking render time

    If there are operative system messages queued
        Dispatch messages to the message callback
    End

    rendererInstance.DoFrame()
    cameraTravelInstance.DoFrame()

    If there is an active project
        rendererInstance.DrawProject(activeProject)
    End

    profilerInstance.AddRenderTime(elapsedTime)
End
```

```
Function EndFrame
    SwapBuffers()
End
```

- ***Message Callback:***

The message callback is in charge of reacting to the operative system messages, which are in turn generated when the user interacts with the application. These interactions may be in the form of using the commands on the user interface, i.e. pushing buttons or editing edit fields, using the keys or the mouse or moving or resizing the window.

Each one of the three dialogs is responsible of reacting to a different set of user commands: the main dialog is responsible of reacting to resizing commands, the side bar dialog is responsible of reacting to user interface interactions and the renderer dialog is responsible of reacting to keystrokes and mouse commands.

The main dialog must notify the other two dialogs of any resizing event. When the user changes the size of the main dialog, the rendering dialog and the side dialog must change their sizes accordingly. It also is in charge of firing the clean up process when the user wants to close the application.

The side bar dialog contains the controls for all the available operations. When the user interacts with one of the controls, the side bar dialog will fire up the system operation related to the command issued by the user.

And finally, the renderer dialog will react to keystrokes and mouse commands done over the rendering area, to fire the system operations related to the renderer, namely moving the viewport and taking screenshots.

The following pseudo code illustrates the structure of the message callback of the application.

```
Function MessageCallback
    Decode Message

    Switch Source Dialog
        Case Render Dialog
            If Mouse Event
                moveViewport()
            Else If Keystroke Event
                takeScreenshot()
            End
        End

        Case Side Dialog
            If Control Event
                Switch Control
                    Case Load Project
                        loadProject()
                    End
                    ··· Repeat for all system operations
                End
            End
        End

        Case Main Dialog
            If Resize Event
                Resize Side Dialog
                Resize Render Dialog
            Else If Close Event
                Notify clean up
            End
        End
    End
End
```

- ***Controller Cleanup:***

When the application is going to close, the *WindowsApplication* class destroys the renderer instance, destroys the graphics context and the profiler and

finally destroys all the created window dialogs. The renderer is responsible of the destruction of all its elements at clean up time.

Just before the application closes, the profiler has to write all the collected statistical data to the storage device. To do so, it commands the profiler to write its statistics before destroying it.

The diagram of the figure *6.1.4* illustrates the clean up process of the *WindowsApplication* class.

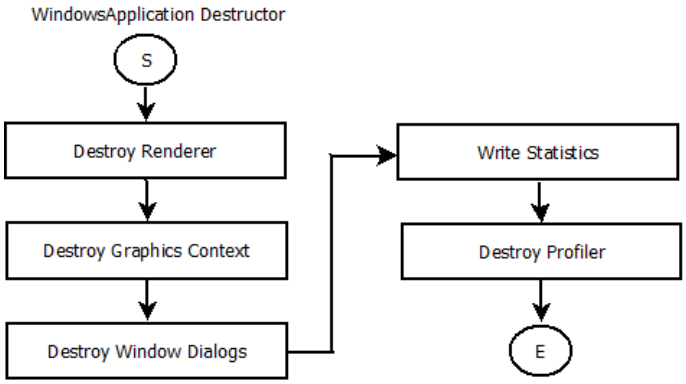


Figure 6.1.4: *Controller cleanup process diagram.*

6.1.2: Terrain Renderer Implementation

The terrain renderer is the most important component of the viewer application. It is in charge of creating the terrain scene and keeping track of all the data used on the rendering: the terrain geometry and textures, and the shader programs used to set up the rendering.

In this section, the implementation details of the renderer are introduced, explaining the different rendering procedures, the generation and decoding of the texture data, and the generation of the geometry for the terrain tiles.

- ***Renderer Initialization:***

When the controller instantiates the renderer, the "*init()"* system operation is fired. This operation is in charge of setting up the necessary components of the renderer for it to work correctly. Specifically, the renderer creates the required shader programs and the viewport camera.

The shader program source code for each shader is loaded from the storage device, to let the user modify the source as wanted.

When the shaders are compiled, the process may fail, if there is an error on the shader code itself. Then, when the renderer is creating the shader objects it must check for exceptions generated by the compilation, and if it catches an exception, show an error message dialog to the user with the generated error and close the application.

- ***ShaderProgram and Shader Implementation:***

The *Shader* class encapsulates the source code loading and compiling of each one of the shader objects which compose the shader program. It must check for any compilation error, in case there is any the compilation method will throw an exception with the compilation error.

On the other hand, the *ShaderProgram* encapsulates the linking of a set of *Shader* objects inside a single shader program. As with the compilation of said

objects, the linking of the shader programs may fail, and the linking method will throw an exception with the linking error.

The following pseudo code illustrates the implementation of the *"compile()"* method of the *Shader* class and the implementation of the *"link()"* method of the *ShaderObject* class:

```
Function Compile(file: string)
    Descriptor <- Open(file)

    While Not end of file
        Source <- AppendLine(Descriptor)
    End

    Close(Descriptor)

    ShaderHandle <- glCreateShader()
    glShaderSource(ShaderHandle, Source)
    glCompileShader(ShaderHandle)

    If Compilation error
        Throw error
    End
End
```

```
Function Link()
    ProgramHandle <- glCreateProgram()

    For Each Shader s In _shaders
        glAttachShader(s, ProgramHandle)
    End

    glLinkProgram(ProgramHandle)

    If Link error
        Throw error
    End
End
```

The shader code used for the terrain rendering will be explained later. When the terrain rendering method is called, the renderer will request to the *ShaderProgram* the OpenGL handle of the shader program.

Through this handle, the renderer can send to the graphics card the references to the data attributes to be used on the rendering, and can set the uniform variables of the shaders.

- ***Instantiation of Texture, Camera and Terrain objects:***

The terrain and texture objects are supposed to be instantiated independently. These objects must be instantiated through the *Terrain* class. This way the renderer will be able to keep track of their creation, and to be able to destroy all the objects at clean up time.

To achieve this, the *Renderer* class has the *"CreateCamera()"*, *"CreateTerrain()"* and *"CreateTexture()"* methods. These methods create an instance of the corresponding object with its specified parameters (position and target for the first; latitude, longitude, height texture and color texture for the second; path for the third), add the instance to the object stack of the renderer and finally return a reference to the instance so it can be accessed.

The next pseudo code illustrates the structure of the *"CreateCamera()"* method. The other two methods have the same structure, but create instances of the corresponding objects.

```
Function CreateCamera(Vector3 position, Vector3 target)
    CameraInstance <- New Camera(position, target)
    _cameras.AddElement(CameraInstance)
    Return CameraInstance
End
```

- ***Renderer Cleanup Sequence:***

At clean up time, the renderer loops over the three stacks to destroy all the instantiated camera, texture and terrain objects. Using this method, these classes can be instanced, without having to worry about possible memory leaks due to losing references to the instances.

Besides cleaning up all the instanced terrains, textures and cameras, it also deletes the shader program that has been previously created at the initialization time.

The following pseudo code shows the structure of the clean up sequence of the renderer.

```
Function OnDispose()
    For Each Camera cam In _cameras
        Delete cam
    End

    For Each Terrain ter In _terrains
        Delete ter
    End

    For Each Texture tex In _textures
        Delete tex
    End

    Delete _terrainShader
End
```

- ***Texture Loading Implementation:***

The renderer can create textures from TIFF image files to use them as the terrain color and height textures, using the *TIFFImageLoader* class. Depending on its intended usage, as a height texture or as a color texture, the loading process is different.

Color textures use three channels, each one for the red, green and blue color channels, and a density of eight bits per channel. These textures are uploaded to the graphics hardware as is, preserving the same number of channels and its pixel density.

Height textures on the other hand only use two out of the three channels inside the image file. The red channel contains the height information for each sample of the terrain, and the green channel contains the up component of the normal vector for each sample of the terrain. The normal vector is used by the post processing effects in the fragment shader. When these textures are uploaded to the graphics hardware, all channels except red and green are discarded.

To load the texture data to the graphics hardware, the textures have to be loaded into the system main memory, using the *LibTIFF* library. It offers an easy to use, efficient and open source implementation of the *TIFF* image format, enabling applications to save and load many kinds of *TIFF* format images.

When loading height textures, the texture loader creates a bounding box using the height samples in the texture. The bounding box contains the minimum and maximum height for each patch primitive in the corresponding terrain mesh. This bounding box is used by the post processing effects, and the number of patches per terrain tile is obtained through a specified value.

To find the maximum and minimum height for each patch, the process is as follows:

1. The texture height and width is fetched.

2. The number of patches is obtained depending on the specified number of subdivisions of the terrain tile geometry data.

3. The image is looped for every subdivision, scanning all the texels on the texture surface corresponding to each subdivision. The maximum and minimum heights for each subdivision are obtained and added to the texture object.

When the textures have been loaded to the graphics card, their mipmaps are created by the graphics card. After all the necessary processing where the copy of the texture data in the system main memory has completed, the copy is discarded, since it is not needed for the application to work once the texture has been loaded to the graphics device.

The following pseudo code illustrates the process used to load textures from the storage device.

```
Function LoadImage(File: string)
    Tex <- New Texture
    Image <- TIFFOpen(File)
    Bpp <- TIFFGetField(BitsPerSample)
    TexHandle <- glCreateTexture()
    MemoryData <- TIFFReadData()

    If Bpp == 8
        glTexImage2D(SourceRGB, DestinationRGB, MemoryData)
    Else If Bpp == 16
        glTexImage2D(SourceRGB, DestinationRG, MemoryData)
        HeightArray <- New Vector2[]

        For Each Patch
            Find Minimum height, Maximum height
            HeightArray[Patch] <- Minimum, Maximum
        End

        Tex.SetBoundingBox(HeightArray)
    End

    glGenerateMipmaps()
    Tex.SetTextureData(TexHandle)

    TIFFClose(File)
    Delete MemoryData

    Return Tex
End
```

- ***Terrain Tile Geometry Generation***

To be able to render the terrains, the graphics device must be supplied with an underlying geometry, which will be uploaded to the graphics device and tessellated to generate the final triangles that create the terrain mesh.

To better understand the procedure used to generate the underlying geometry, following is a brief explanation about how is the data going to be treated by graphics device.

The graphics device receives a set of control points, constituting a point cloud without a defined polygonal structure. Using the tessellator, the graphics device triangulates the patches defined by the point cloud using the specified level of detail (tessellation factor).

The first stage of the tessellation pipeline is the tessellation control shader; where the number of input control points per patch, number of output vertices per patch and the tessellation factor are set. In the next stage, the tessellation evaluation shader, the triangulated mesh is received and the vertices are transformed to create the corresponding terrain mesh.

When using tessellation to render terrain tiles, the easiest approach is using a single control point as the underlying geometry, which in turn will generate a number of triangles proportional to the tessellation factor value.

However, the tessellation factor has a maximum value, dependant on the hardware implementation of the tessellator. In recent commercial hardware, the tessellation factor is usually limited to 64 subdivisions. The following pseudo code shows how the maximum tessellation factor supported by the device can be polled.

```
Function GetMaximumTessellationFactor()
    Return glGetInteger(GL_MAX_TESS_GEN_LEVEL)
End
```

The maximum triangle density of the final surface is greatly limited due to this restriction. To understand this problem, let an example heightmap have 1024x1024 height samples. With a maximum tessellation factor of 64, the generated mesh would have 64x64 vertices, meaning that 99,6% of the samples in the heightmap would not be represented on the terrain mesh.

To solve this problem, the selected solution is creating a grid of square equally spaced control points, creating more subdivisions to increase the maximum number of triangles on the mesh.

With this approach, taking into account a maximum tessellation factor of 64, a terrain tile with a single control point would have 64x64 vertices; one with four control points would have 128x128 vertices, and so on. Assuming a square heightmap, the number of subdivisions needed to create the point cloud would be:

$$subdiv = \frac{heightmap\ width}{maximum\ tess.factor}$$

Formula 6.1: *Subdivision count for point cloud generation*

The heightmap of the previous example, with 1024x1024 samples, would need 16 subdivisions to be able to create a mesh with all the terrain samples represented, which will in turn be generated by a point cloud with 256 elements.

The maximum number of vertices that can be rendered in the scene is still limited by the maximum number of triangles that can be rasterized by the graphics device in real time. However, this limitation is usually bounded by the tens of millions and does not suppose a major problem like the tessellation factor limitation.

The following pseudo code shows the algorithm used to build the point cloud for the terrain tiles. The position of the control points is bounded between {0,0} and {subdiv, subdiv}.

```
Function GenerateControlPoints(subdiv: int)
    dx <- 1.0
    dz <- 1.0
    x <- 0.0
    z <- 0.0

    ControlPoints <- new Vector4[]

    For i <- 0:1:subdiv + 1
        For j <- 0:1:subdiv + 1
            ControlPoints.Add({x, 0.0, z, 1.0})

            x <- x + dx
        End

        x <- 0.0
        z <- z + dz
    End

    Return ControlPoints
End
```

- ***Terrain Tile Instantiation:***

At instantiation time, terrain tiles are responsible of creating its underlying point cloud geometry, uploading the point cloud and the bounding box of the tile to the graphics device, and of setting up the OpenGL state machine with said data to let the renderer draw it when the project is going to be rendered.

The point cloud data is generated as stated previously, and the bounding box data is fetched from the specified height texture object. The terrain object then creates a vertex buffer object or VBO for short to upload the data to the graphics device.

VBO's are a data structure which contain arbitrary data used in rendering. It is one of the objects introduced in the newer versions of OpenGL, and reduces greatly the bandwidth usage between the graphics device and the system memory. In this case, the point cloud and the bounding box are uploaded once to the GPU into their respective VBO's, and at render time the data is directly accessed from the graphics memory.

To further reduce the bandwidth usage, the state of the OpenGL state machine is stored inside a vertex array object or VAO for short. VAO's are used to store the state of the OpenGL state machine, letting the application issue all the commands to the graphics device, and then only reference the state to activate it.

In the VAO set up, the two VBO's are linked with a specific name. That name is used in the shader pipeline to access the data stored in the VBO, as an attribute in the vertex shader, the working of such will be explained later.

The following pseudo code illustrates the instantiation of the terrain tile objects.

```
Function TerrainConstructor(subdiv: int, heightmap: Texture)
    ControlPoints <- GenerateControlPoints(subdiv)
    BoundingBox <- heightmap.GetBoundingBox()

    ControlPointsVBO, BoundingBoxVBO <- glGenBuffers()
    TerrainVAO <- glGenVertexArray()

    glBufferData(ControlPoints, ControlPointsVBO)
    glBufferData(BoundingBox, BoundingBoxVBO)

    glBindVertexArray(TerrainVAO)
    glUseProgram(TerrainShader)

    glVertexAttribPointer(ControlPointsVBO, "aVertices")
    glVertexAttribPointer(BoundingBoxVBO, "aBoundingBox")
End
```

- ***Terrain Rendering Implementation:***

In this section, the implementation of the different terrain rendering algorithms is explained in detail. To do so, the shader code used in all the shader pipeline stages of the rendering is explained, stating the shader code used for each rendering algorithm, tessellation alone and tessellation including the selected post processing effects. The algorithms used in the terrain tessellation method were introduced in the state of the art chapter.

The first step is explaining the set up for the basic terrain tessellation algorithm. To be able to use terrain tessellation, the shader pipeline must be configured with the appropriate shaders, and the renderer must have a method to render terrain tiles, which will be called by the controller at the beginning of each frame.

- ***Renderer Class Rendering Methods Implementation:***

The *Renderer* class has two methods to draw terrain tiles, the *"DrawTerrain()"* method and the *"DrawProject()"* method. The former is used to render a single tile at a given latitude and longitude, and the later is used to render all the tiles in a terrain project in a batch.

The implementation of the *"DrawTerrain()"* method is fairly simple, since all the logic of the terrain rendering algorithm is encapsulated in the shader code. The method is responsible of setting up the uniform variables of the shader program to and of commanding OpenGL to start the rendering.

The data committed to the OpenGL library is stored inside the vertex array object (VAO) of the terrain tile, which is fetched from the *Terrain* object to be rendered. The uniform variables required for the shader programs to work will be explained later, when the shader code algorithms is introduced.

The following pseudo code shows the implementation of the *"DrawTerrain()"* method.

```
Function DrawTerrain(terrain: Terrain, position: Vector2)
    textureHandle <- terrain.GetTextureHandle()
    textureUniformHandle <- glGetUniformLocation("uTexture")
    glUniform(textureUniformHandle, textureHandle)

    ··· Repeat for all the uniform variables

    glBindVertexArray(terrain.GetVertexArrayObject)
    glDrawArrays(Patches, 0, PointCloudCount)
End
```

The implementation of the *"DrawProject()"* method is also very simple. This method loops over all the existing tiles in a project, and commands the renderer to draw them using the previous method.

The next pseudo code shows the implementation of the *"DrawProject()"* algorithm.

```
Function DrawProject(project: Project)
    tiles <- project.GetTerrainTiles()

    For Each Terrain t In tiles
        latitude <- t.GetLatitude()
        longitude <- t.GetLongitude()

        This.DrawTerrain(t, {latitude, longitude})
    End
End
```

- ***Shader Program for the Tessellation Rendering Algorithm:***

In this section, the shader program pipeline used to implement the terrain tessellation algorithm is described. To be able to use terrain tessellation, the shader pipeline must have programs of the following stages: vertex shader, tessellation control shader, tessellation evaluation shader and fragment shader.

The shader pipeline for the bare tessellation shader algorithm only needs a single input attribute, the control points of the terrain tile. All the required data to render the geometry, such as vertices, texture coordinates or normal vectors are obtained dynamically on the tessellation evaluation shader. The control points are passed directly to the next stage without any transformation.

The following pseudo shader code shows the implementation of the vertex shader:

```
in vec3 aControlPoints;
out vec3 vControlPoints;

main()
{
    vControlPoints = aControlPoints;
}
```

The next stage in the shader pipeline is the tessellation control shader. It is in charge of setting the level of detail of the tile, through the inner and outer tessellation factors, and of culling whether patches should be rendered.

- ***Per-patch Culling***

Per-patch culling is used to check if the control points of an input patch will be inside of the viewport area once the primitive generator stage has tessellated the patch. If the output vertices are not going to be rendered the patch can be skipped, saving computation time.

The patch control point coordinates are passed to the tessellation control shader in world space coordinates. The method used to determine whether a patch falls into the viewport area or not is converting the world space coordinates to screen space coordinates and check if they bound inside the screen limits.

As stated in the terrain tile initialization section, patch control points are bounded between zero and the subdivision count plus one. To obtain the control point position in screen space, it has to be normalized to model space bounds, [-0.5, 0.5] on all axes.

This is achieved by normalizing the control point coordinate by the subdivision count, and then translating it by -0.5 on the XZ plane, to center it on the model space center of coordinates. The next formula illustrates this operation.

$$|\vec{p}|_{model} = \frac{\vec{p}}{subdiv + 1} - \left(0.5\,\hat{\imath} + 0.5\,\hat{k}\right)$$

Formula 6.2: *Control point position normalization to model space coordinates.*

In the next stage patches that fall into the screen limits will get a tessellation level greater than zero, while patches that are outside the screen limits will get a tessellation level equal to zero.

The following pseudo shader code shows how the patch culling is calculated:

```
bool cullPatch(vec3 p, int subdiv, mat4 mvpMatrix)
{
    vec3 pNorm = p / (subdiv + 1) - vec3(0.5,0,0.5);
    pNorm *= mvpMatrix;

    if (pNorm.xz > -1 || pNorm.xz < 1)
        return false;
    else
        return true;
}
```

- ***Calculation of the level of detail***

Setting the inner and outer tessellation levels define how many vertices will be generated from each patch. Outer levels define the number of subdivisions on the edge created by two control points and inner levels define the number of concentric subdivisions between the patch center and the control points.

The method used to calculate the level of detail is a heuristic method, setting-up a function relating the tessellation level with the distance from the viewer to the patch. In a first approach it could resemble the following function:

$$TL(d) = \text{clamp}\left\{-\frac{TL_{max}}{d_{max}} \cdot d + TL_{max},\quad 1,\quad TL_{max}\right\},\qquad d \in [0, +\infty)$$

Formula 6.3: *Tessellation level formula.*

TL_{max} is the maximum tessellation level desired, often the maximum tessellation level supported by the rendering device. d_{max} is the distance where the tessellation level should fall to the minimum level (one). The *clamp* operator restricts the result of the operation *f* in the interval [a,b], expressed as:

$$\text{clamp}\{f, a, b\} = \min\{\max\{f, b\}, a\}$$

Formula 6.4: *expression for the clamp operator.*

When the distance between the viewer and the patch is minimal the tessellation level should be the maximum possible level. As the distance grows the tessellation level should shrink, lowering the level of detail until the patch is outside the viewport back clipping plane.

Another adjustment is forcing the function to take only values on powers of two up to the maximum tessellation level. This ensures that the patch will be always subdivided in power of two dimensions, which is useful when using mip-mapping on the terrain height map textures. This can be expressed as follows:

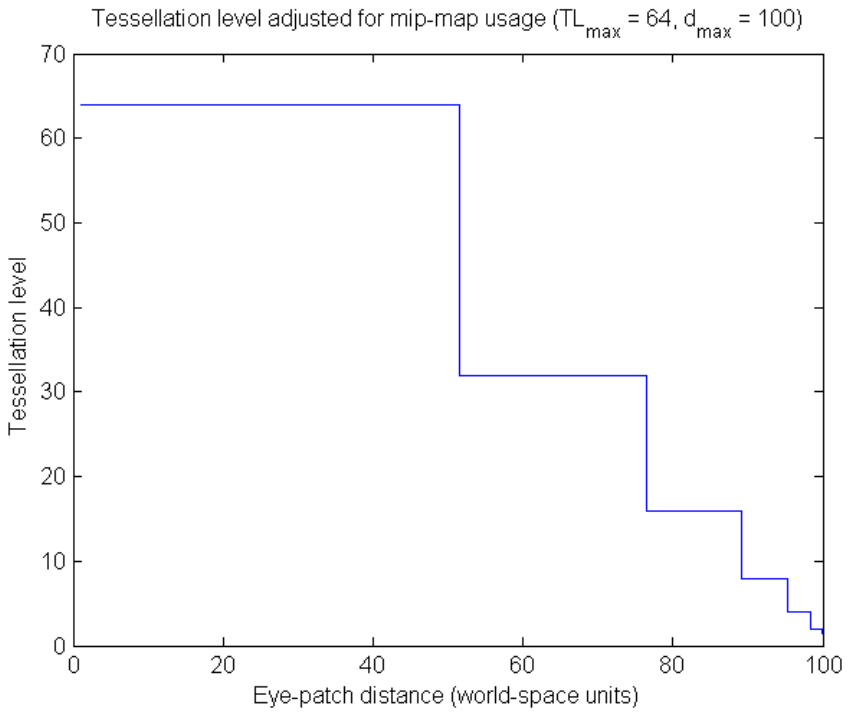


Figure 6.1.5: *Tessellation level adjusted for mip-mapping.*

$$TL_{mip\ map}(d) = \text{clamp}\{2^{\lfloor \log_2 TL(d) \rfloor}, \quad 1, \quad TL_{max}\}, \quad d \in [0, +\infty)$$

Formula 6.5: *Tessellation level adjusted for mip-mapping formula.*

The following pseudo shader code shows the implementation of the tessellation factor adjusted for mip-mapping usage.

```
float tessLevel(float d, float md, int sd)
{
    float tess = sd - ((sd / md) * d) + 1;
    tess = pow(2, ceil(log2(tess)));
    return clamp(tess, 2, sd);
}
```

As for the calculation of the distance between the patch and the viewing point there are two main approaches: taking the distance between the viewing point and the center of the patch and taking the distance between the viewing point and a sphere which contains the outer control points of the patch.

The later method ensures that adjacent patches will have the same number of vertices on sharing edges, preventing the apparition of seams, which has been implemented.

The following pseudo shader code shows the implementation of the tessellation control shader, with per-patch culling and the calculation of the tessellation factors adjusted for mip-mapping usage. Control points are passed forward unaltered to the next stage.

```
in vec4 vControlPoints;
out vec4 tcControlPoints;

uniform int uSubdiv;
uniform float uMinLodDist;
uniform vec3 uEye;
uniform mat4 uMvpMatrix;

main()
{
    if (!cullPatch(aControlPoints, uSubdiv, uMvpMatrix))
    {
        float d = 0.5 * (uSubdiv + 1);
        vec3 p = aControlPoints;

        InnerFactor = tessLevel(dist(p, uEye, uMinLodDist,
                                uSubdiv);
        OuterFactorUp = tessLevel(dist(p + vec3(0,0,d), uEye);
        OuterFactorLf = tessLevel(dist(p + vec3(-d,0,0), uEye);
        OuterFactorDw = tessLevel(dist(p + vec3(0,0,-d), uEye);
        OuterFactorRt = tessLevel(dist(p + vec3(d,0,0), uEye);
```

```
    }
    else
    {
        InnerFactor = 0;
        OuterFactorUp = 0;
        OuterFactorLf = 0;
        OuterFactorDw = 0;
        OuterFactorRt = 0;
    }

    tcControlPoints = vControlPoints;
}
```

The next step on the shader program pipeline is the tessellation evaluation shader. It receives the triangulated mesh created from the control points, generated using the specified tessellation factors.

The evaluation shader is responsible of transforming the received vertices into the final terrain mesh, sampling the height data from the heightmap texture. To do so, the vertex texture coordinates are generated. It also is responsible of passing the final texture coordinates to the fragment shader, to be able to create the shaded image of the terrain.

The tessellator generates the vertices and assigns it a tessellation coordinate to identify the position of the generated vertex in relation to the original control points. When triangulating the point cloud as quads, these coordinates are given as Cartesian coordinates.

The following pseudo shader code shows the implementation of the tessellation evaluation shader:

```
in vec3 tcControlPoints;

out vec2 teTexCoord;

uniform mat4 uMvpMatrix;
uniform float uSubdiv;

uniform sampler2D uHeightmap;
uniform float uScale;

main()
{
    float u = TessellationCoordinate.u;
    float v = TessellationCoordinate.v;
    float x = tcControlPoints.x;
```

```
    float z = tcControlPoints.z;
    vec3 vertex;

    teTexCoord.s = (u + x) / (uSubdiv + 1);
    teTexCoord.t = (v + z) / (uSubdiv + 1);

    vertex.x = ((u + x) / (uSubdiv + 1)) - 0.5;
    vertex.y = texture(uHeightmap, teTexCoord).r * uScale;
    vertex.z = ((v + z) / (uSubdiv + 1)) - 0.5;

    VertexPosition = uMvpMatrix * vertex;
}
```

The last stage of the shader program pipeline is the fragment shader stage. In the case of simple terrain tessellation, the fragment shader is very simple as its only responsibility is shading the rasterized fragments with the terrain color texture.

The following shader pseudo code shows the implementation of the fragment shader.

```
in vec2 teTexCoord;
out vec3 fColor;

uniform sampler2D uTexture;

main()
{
    fColor = texture(uTexture, teTexCoord).rgb;
}
```

- ***Implementation of the Post-processing Effects***

The main part of the implementation of the post-processing effects is done in the fragment shader stage, since its purpose is modifying the rasterized image to enhance the visual fidelity of the terrain.

However, the post-processing algorithms make use of additional data which must be provided by the shader stages preceding the fragment shader. The needed data is the tile bounding box, which was introduced in the texture loading section, the vector between the camera and the vertex to be rendered, and the sampled height of the vertex. The last data has been integrated into the texture coordinate vector.

The following pseudo shader code shows how the eye-vertex vector is calculated.

```
...
out vec3 teTexCoord;
out vec3 teEyeVector;
...

main()
{
    ...
    teTexCoord.z = vertex.y;
    teEyeVector = (vertex * uMvpMatrix) – teEyeVector;
}
```

Following are the explanation for the implementation of each one of the post-processing effects in the fragment shader. The selected technologies in the state of the art chapter were simple parallax mapping, parallax mapping taking into account the slope, binary parallax mapping and secant parallax mapping.

- ***Implementation of simple parallax mapping:***

The concept behind parallax mapping is modifying the texture coordinates of the terrain geometry, which represents the terrain macrostructure, to approximate its mesostructure, as introduced in the state of the art chapter.

A ray is traced from the eye to the target in the macrostructure. The intersection between the ray and the mesostructure is obtained. This is illustrated in the figure *6.1.6*.

The target texture coordinates are translated by the projection of the intersection point on the macrostructure, as stated by the following formula, where the *uv* vector represents the texture coordinates and the *v* vector represents the eye-target vector:

$$\overrightarrow{uv}_{parallax} = \overrightarrow{uv} + H\left(\overrightarrow{UV}\right)\frac{v_x\hat{\imath} + v_z\hat{k}}{v_y\hat{\jmath}}$$

Formula 6.6: *Simple parallax mapping solution.*

The following pseudo shader code and figure show the implementation of the simple parallax mapping method.

```
float hsb = texture(uHeightmap, teTessCoord.st).r;
uv = teTessCoord + hsb * eye.xz / eye.y;
```

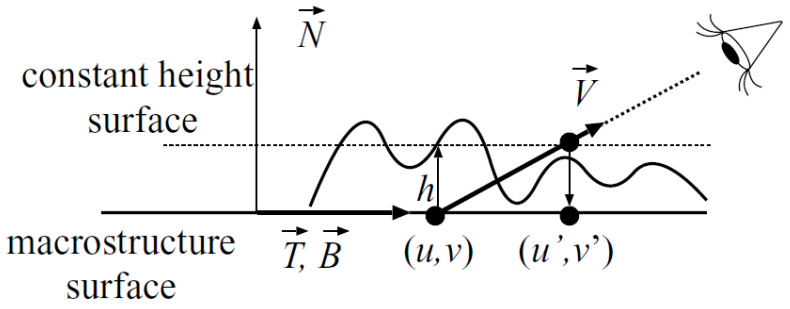


Figure 6.1.6: *Finding the intersection point between the eye and the mesostructure.*

- ***Implementation of parallax mapping taking into account the slope:***

Parallax mapping taking into account the slope is an improvement over simple parallax mapping where the slope of the macrostructure plane is taken into account.

In simple parallax mapping, the macrostructure is assumed to be coplanar to the XZ plane at height zero. In this improvement, the height of the target is taken into account to obtain an intersection closer to the real mesostructure intersection point. This is illustrated in the figure *6.1.7.*

Following is the implementation of the parallax mapping taking into account the slope algorithm:

```
float hsb = texture(uHeightmap, teTessCoord).r;
uv = teTessCoord.st + (hsb - teTessCoord.z) * eye.xz / eye.y;
```

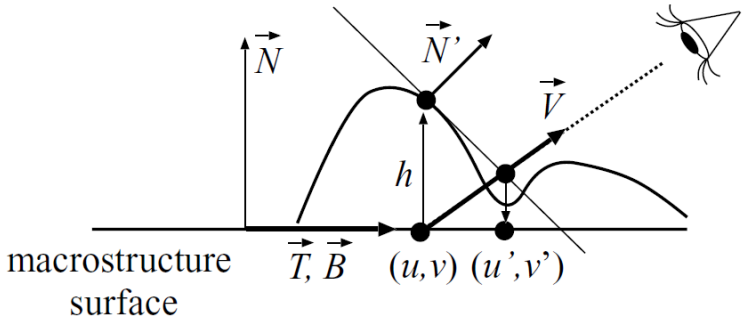


Figure 6.1.7: *Finding the intersection taking into account the slope.*

- ***Implementation of binary parallax mapping:***

Binary parallax mapping is based on doing a binary search on the ray between the eye and the target to estimate the intersection point. The algorithm initializes two points, one known to be above the structure and the other known to be below the structure.

The interval is halved on every iteration, always remaining with the interval that is more probable to have the intersection with the ray. The more iteration the algorithm is executed, the closer the estimated intersection point will be to the real intersection point, as shown in the figure *6.1.8*.

The following pseudo shader code and figure illustrate the implementation of binary parallax mapping.

```
float hsb1 = teBoundingBox.x - teTexCoord.z;
vec2 ST1 = teTexCoord.st + hsb1 * eye.xz / eye.y;

float hsb2 = teBoundingBox.y - teTexCoord.z;
vec2 ST2 = teTexCoord.st + hsb2 * eye.xz / eye.y;

for (i = 1:Iterations) {
    STm = (ST1 + ST2) / 2;
    hsbm = (hsb1 + hsb2) / 2;
    hsb = texture(uHeightmap, STm).r - teTexCoord.z;

    if (hsb < hsbm) {
        ST2 = STm; hsb2 = hsbm;
```

```
    } else {
        ST1 = STm; hsb1 = hsbm;
    }
}

uv = STm;
```

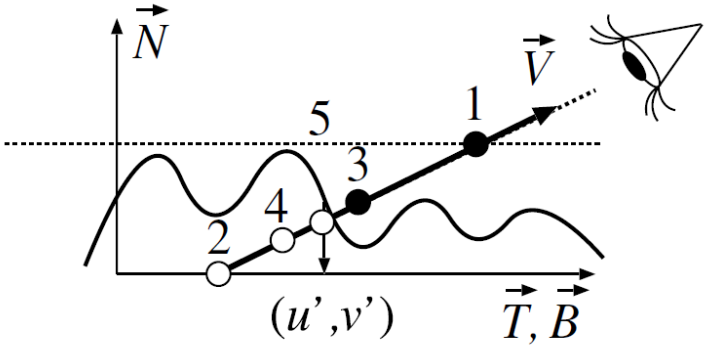


Figure 6.1.8: *Binary search with five iterations.*

- ***Implementation of secant combined with linear parallax mapping:***

Secant parallax mapping is an improvement over binary parallax. The binary method assumes that the macrostructure is coplanar to the XZ plane when finding the intersection of the ray between the eye and the terrain.

On the other hand, secant parallax mapping assumes that the macrostructure is planar between the two points of the interval and calculates the intersection there.

To do this, the algorithm obtains the estimated point, and then calculates the intersection between the plane between the two points and the ray, taking into account the slopes.

For every iteration, the heightmap is checked to make sure that one of the points of the interval is above the surface and that the other point is below the surface.

The following pseudo shader code shows the implementation of the secant parallax mapping algorithm.

```
float Ha = teBoundingBox.x - teTexCoord.z;
float Hb = teBoundingBox.y - teTexCoord.z;
float Dh = (Ha - Hb) / Iterations, Da = 0, Db = 0;

vec2 STa = teTexCoord.st + Ha * eye.xz / eye.y;
vec2 STb = teTexCoord.st + Hb * eye.xz / eye.y;
vec2 Ds = (STa - STb) / Iterations;

Ha = Hb;
STa = STb;

for (i = 1:Iterations) {
    STa += Ds;
    Ha += Dh;
    Da = Ha - (texture(uHeightmap, STa).r - teUvCoordinates.z);

    if (Da < 0)
        break;

    Db = Da;
    STb = STa;
}

if (Da == Db)
    discard;
else
    uv = STb * (Da / (Da - Db)) + STa * (Db / (Db - Da));
```

- ***Implementation of the camera travel***

The camera travel is in charge of animating the viewport camera, to make it follow the waypoints in the project, creating the camera flight. To find the points where the camera should go, the camera travel class creates a cubic curve adjusted to the waypoints.

Then, it obtains the solution of the curve and the solution of its first derivative, storing the obtained data on two vectors. At render time, the camera travel class is messaged through its *"DoFrame()"* method. Then, the position and the target of the viewport camera are updated.

Cubic curves are a grade three polynomial functions which can be expressed as the following matrix product.

$$Q(t) = GMT(t) = GM \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

Formula 6.7: *General expression for a cubic curve.*

Where *t* is the time step, *G* is the geometrical constraint matrix associated with the cubic curve and *M* is the constant basis matrix. The chosen curve to do the camera flight path interpolation is a Bézier curve, which can be expressed with the following expression.

$$Q(t) = [P_0 \quad P_1 \quad P_2 \quad P_3] \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

Formula 6.8: *Expression for the Bézier curve.*

This expression was evaluated in MATLAB to obtain its solution for the equation and its first derivative, and implemented using the following pseudo code.

```
float Evaluate(float t)
{
  return (M14 * P0 + M24 * P1 + M34 * P2 + M44 * P3) * (t*t*t) +
    (M13 * P0 + M23 * P1 + M33 * P2 + M43 * P3) * (t * t) +
    (M12 * P0 + M22 * P1 + M32 * P2 + M42 * P3) * t +
    M11 * P0 + M21 * P1 + M31 * P2 + M41 * P3;
}
```

```
float Derive(float t)
{
  return (3*t*t) * (M14 * P0 + M24 * P0 + M34 * P0 + M44 * P0) +
   (M12 * P0 + M22 * P0 + M32 * P0 + M42 * P0) +
   (2 * t) * (M13 * P0 + M23 * P0 + M33 * P0 + M43 * P0);
}
```

The curve solution and derivative are sampled at each point of the camera flight. Then, the camera position is set to the sampled curve solution at that specific time step, and the camera target is set to the direction given by the curve first derivative.

6.2: Implementation of the Terrain Designer

This section introduces the details of the implementation of the terrain designer application. Since the implementation of the terrain designer is not as critical as the implementation of the terrain viewer application, only the most important algorithms are described.

- ***Iberpix Dataset Access***

The first algorithm to be described is the algorithm used to download data from the Iberpix data set. Iberpix is a project of the Spanish *Instituto Geográfico Nacional* to create a computer model of the Iberian Peninsula. It allows users to access it through a simple to use website. The model contains several information layers, like aerial and satellite color imagery, heightmap data, political and statistical geographic data, among others.

In this application, the interest is on obtaining the color and heightmap data from a particular set of tiles to use them as the source data. The web interface accesses the tile data using a Web service, which has been reverse engineered to obtain the URI pointing to the resources.

The URI of the Iberpix data has the following structure, where the parameters are described next.

```
http://www2.ign.es/iberpix/tileserver/n=<layer>;z=<UTM>;r=<Resolut
ion>;i=<Latitude>;j=<Longitude>.jpg
```

The first argument is the layer to be downloaded. The layers of interest in this project are the color and height data, which are fetched with the "*spot5"* and *"relieve"* tokens. The second argument is the UTM of the data, which is set to 30 by default. The third argument is the resolution of the data in centimeters per sample, which is set to 4000 by default. And finally, the last two arguments are the latitude and longitude of the tile.

- ***SRTM File Conversion***

SRTM files are the geographical data files from the Shuttle Radar Topology Mission of the NASA. The objective of these files was obtaining high resolution geographical data from the Earth, and finding publicly available terrain height data in this format is common.

The SRTM format has two variants: SRTM1 and SRTM3. The former have 3601 samples per file spaced every 1 arc-second, and the later have 1201 samples per file spaced every 3 arc-seconds.

The files do not have any kind of header; the format is then obtained by the size of the file. Samples are encoded into 16 bits signed integers in big-endian format, on the range [-32767, 32767] meters. Data is sampled in row-major order.

The following pseudo code shows the algorithm to decode files with the SRTM format.

```
Function ConvertSRTM(Int16[] stream)
    If stream.Length == 3601 * 3601
        Format <- SRTM1
        Length <- 3601
    Else
        Format <- SRTM3
        Length <- 1201
    End

    Data <- New Int16[Length * Length];
    BufferIndex <- 0
    Int16 temp <- 0

    For i = 0:Length
        For j = 0:Length
            temp[Hi] <- stream[BufferIndex]
            BufferIndex <- BufferIndex + 1

            temp[Lo] <- stream[BufferIndex]
            BufferIndex <- BufferIndex + 1

            If System is Little Endian
                temp.Swap()
            End

            Data.Add(temp)
        End
    End

    Return Data
End
```

6.3: Summary

In this chapter, the most important details of the implementation of the terrain rendering system were described, explaining the algorithms used to obtain the terrain visualization and the reasoning behind them.

In the next chapter, the implementation will be put to a series of tests to do a comparison of the quality of the different rendering methods, under different tests.

CHAPTER 7: TESTS AND RESULTS

In this chapter, the different rendering methods are tested using a series of experiments. First, a data set will be created to do the experiments, to ensure that the results are not caused by a change in the input data. Then, a visual inspection of the rendering obtained with each one of the algorithms is done, to check whether there is any algorithm that is not suitable for this task.

Finally, the suitable rendering algorithms are tested to see the obtained performance and visual fidelity. To do so, a base level is set with one of the algorithms and a comparison between them is done from this base.

The experiments consist on comparing the mean square error of the rendered terrain using the different algorithms with the base level. Then, the algorithms can be sorted by its visual fidelity and performance, having the mean square error of the visualization at a target frames per second.

The results obtained on these experiments will be discussed, obtaining conclusions about the behavior and performance of the algorithms and about the suitability of the implemented methods for terrain rendering.

Finally, an evaluation of the costs at the end of the project will be done, comparing them to the cost estimation of the third chapter, and checking if there is any kind of variance between the original estimation and the final result.

7.1: Test Description

This section introduces the test battery done to obtain the performance statistics of the terrain viewer application. All tests will be conducted under the same system, to prevent the results from being affected by changes on the test system hardware or software.

The table *7.1* shows the hardware and software configuration of the system used to run the tests.

Identification	***Machine 1***
Graphics Device	Nvidia GeForce 740M
Graphics Memory	2 GB
System CPU	Intel 4700MQ (4 x 2,4 GHz)
System Memory	12 GB
Operating System	Windows 8.1

Table 7.1: *Hardware used to conduct the tests.*

Along with the test hardware, the data set used to conduct the tests is introduced. The test data set is a subset of the digital model of the Puget Sound, in Washington State, USA. This data set has been thoroughly used in the testing of terrain rendering algorithms, and was introduced in the early 2000's [Li01].

The subset consists of a 4x8 tile terrain, with each tile having 1024x1024 samples with a sample spacing resolution of 20 centimeters. This leaves a total surface of 819 meters wide by 1638,5 meters long. The color textures have a resolution of 2048x2048 pixels. The test data takes around 141 MB of disk space in its compressed form and around 671 MB of video memory space in its decompressed form.

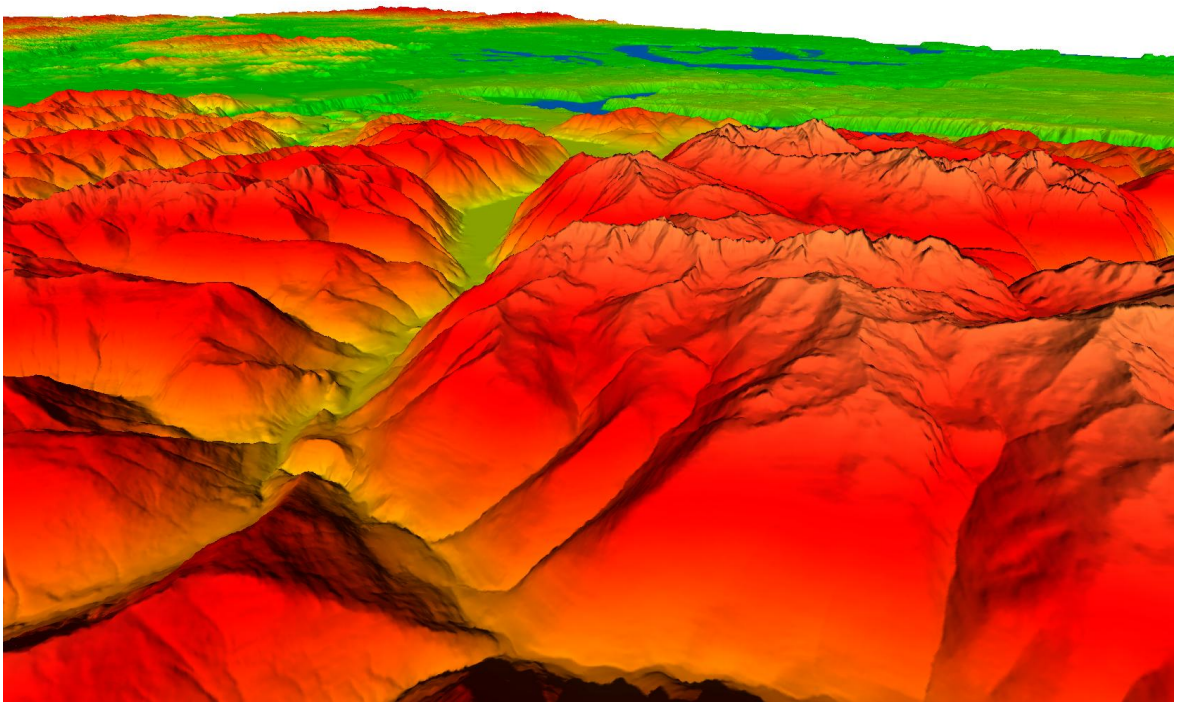


Figure 7.1.1: *Rendered view of the test data set.*

The data set includes a path to do a camera flight, which will be used to obtain screen captures of the rendered terrain, to later process them and obtain the mean square error between the different algorithms.

The next step consists on creating the base line for the tests, using the tessellation rendering algorithm alone. Using this algorithm, the average frames per second are obtained for a given number of triangles. This will be used as the base FPS for the comparison with the other algorithms.

- ***Tessellation Algorithm Base Test***

The tessellation algorithm is the base for the terrain rendering on this project. In this section, the algorithm is tested with different levels of detail (tessellation factors) to see how the algorithm scales with the number of triangles being rendered.

The terrain data set has 8x4 tiles and the underlying geometry has been set to 8 subdivisions, which translates into a 162 triangle mesh with the minimum tessellation factor. The number of triangles scales linearly as the tessellation factor increases, until it reaches a total of 331776 triangles at the maximum tessellation factor, 64.

Using this data, the average frames per second depending on the target rendered triangles are obtained. The figure *7.1.2* shows the obtained data, as can be observed the frames per second is reduced as the number of triangles escalates.

The data has been obtained using doing several runs of the application using the test data. The level of detail is fixed, taking samples from the minimum value to the maximum value in steps of five levels. Five realizations of the test have been done, obtaining the mean frames per second as the final value, which have been adjusted.

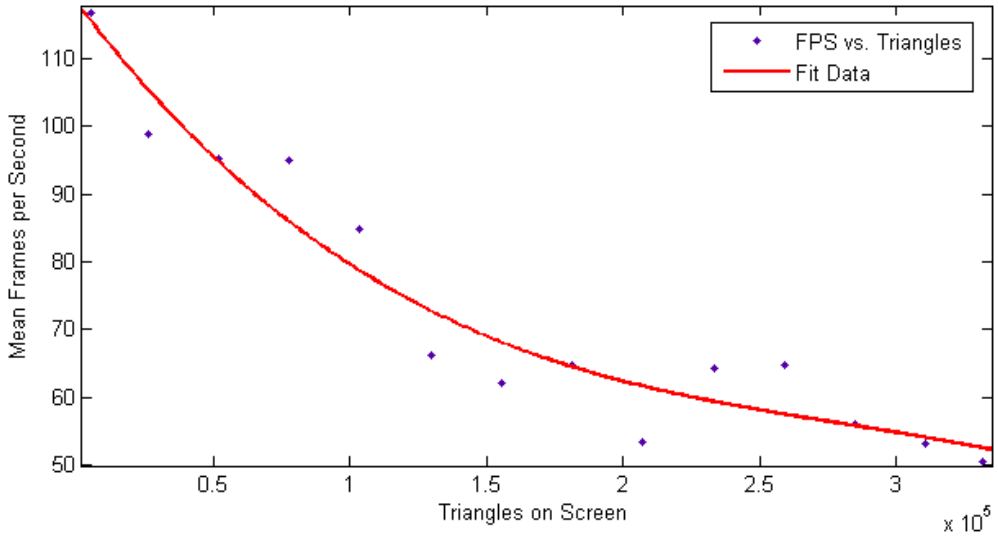


Figure 7.1.2: *Triangle count versus frames per second.*

The highest triangle counts represent the highest tessellation factor values, while the lowest ones represent the lowest tessellation factor values. With very high triangle counts, the frames per second go under the interactive level of 60 frames per second.

To compare this behavior with the usage of automatic management of the level of detail, the same calculation has been done but using automatic LOD management with a minimum distance of 5 units. The figure *7.1.3* shows the obtained result.

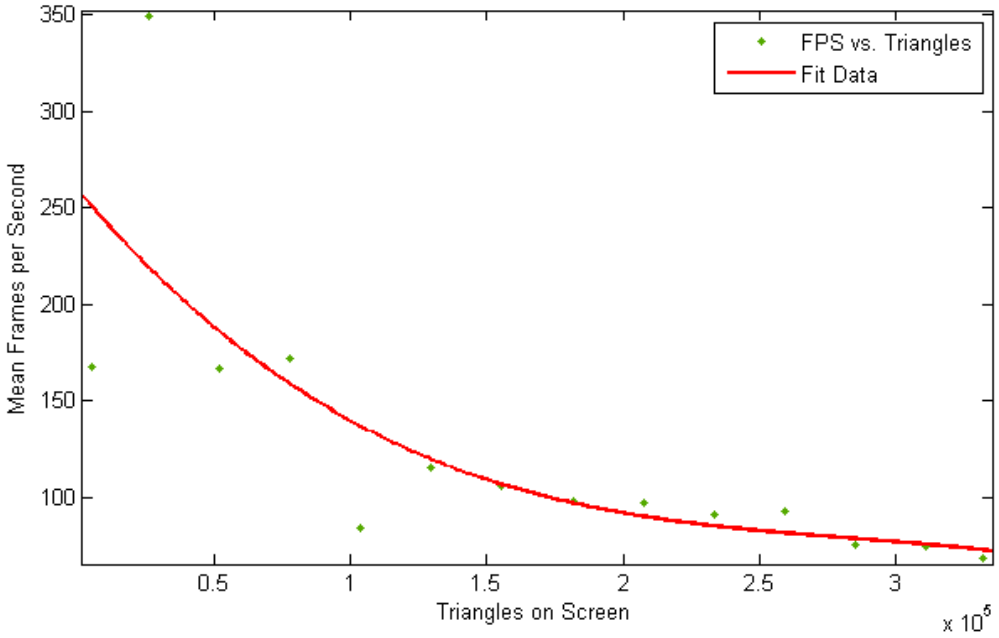


Figure 7.1.3: *Triangle count vs. FPS using automatic LOD management.*

The usage of automatic management of the level of detail increases the mean frames per second, obtaining frame rates around two fold the rates obtained tessellation with a fixed level of detail.

This the use of dynamic control of the level of detail to decrease the polygon count of the terrain, which in turn will in turn increase the frames per second obtained by the application.

These results set the base level for the next comparatives, where the post-processing effects will be compared with the tessellation method to have a measurement of their visual fidelity and performance.

- ***Visual comparison of the post-processing effects***

The objective of the post-processing effects is increasing the visual fidelity of the rendered terrain at a lower triangle count, which means that the objective is minimizing the mean square error between the rendering with simple tessellation and maximum tessellation factor, and post-processed rendering with a lower tessellation factor.

Following, the different post-processing are compared visually with the tessellation method at the maximum level of detail. Through this comparison any existing visual artifacts can be detected.

The right half of the pictures show the terrain rendered using simple terrain tessellation, while the left half of the pictures show the same terrain rendered using the corresponding post-processing effect.

Methods that create any kind of visual artifacts that are unpleasant to the eye are not suited for the solution, even if the statistical comparison deems the methods as good when compared with other solutions. To be able to see the differences, the methods are compared head to head, representing the same view of the terrain.

- ***Simple parallax mapping:***

The first method tested is simple parallax mapping. The figure *7.1.4* shows a head to head visual comparative of rendering using simple tessellation and using simple parallax mapping.

As can be seen from the picture, the output image has many visual artifacts in high elevation areas, which make this algorithm not valid for terrain rendering.

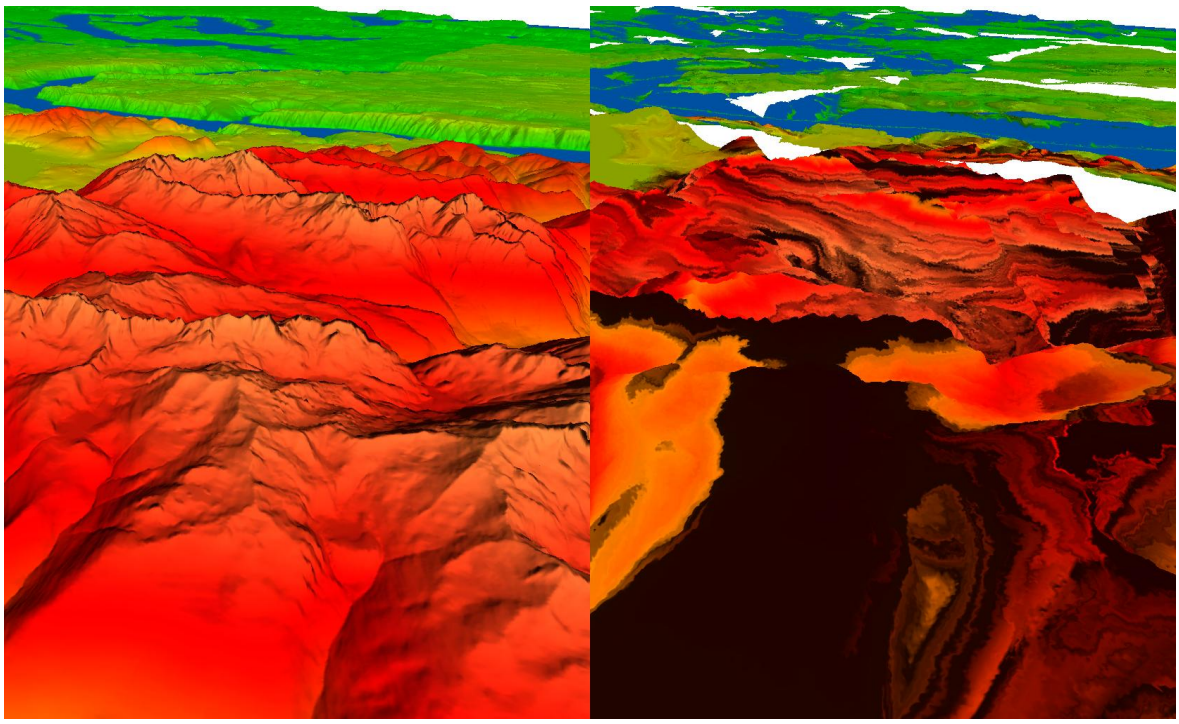


Figure 7.1.4: *Tessellation versus simple parallax mapping.*

- ***Parallax mapping taking into account the slope:***

The second method tested is parallax mapping taking into account the slope. In the figure *7.1.5*, the left side shows rendering using simple tessellation, while the right side shows the rendering adding the parallax mapping method.

As can be seen from the image, this can be a good candidate to be used as a post-processing method, since even though there are visual artifacts visible in the picture, these do not represent a visual annoyance severe enough as to discard it.

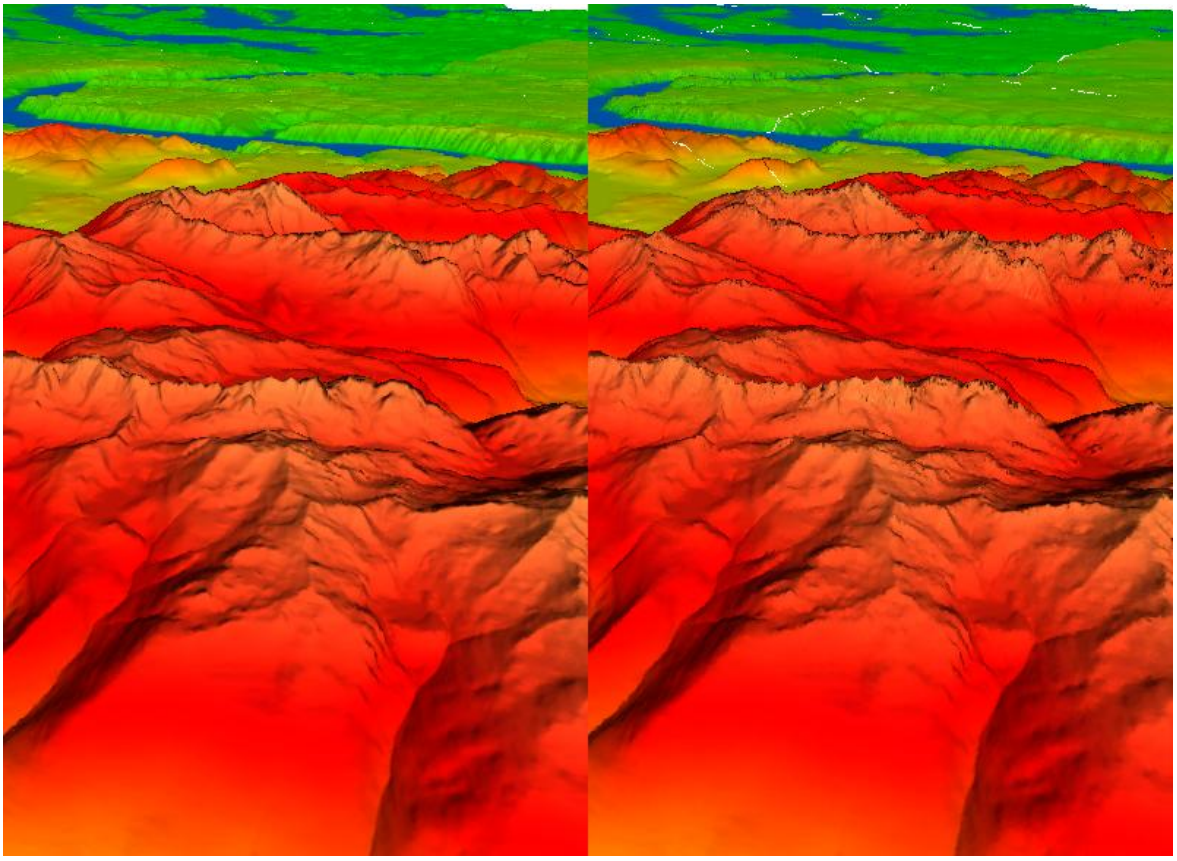


Figure 7.1.5: *Tessellation versus parallax taking into account the slope.*

- ***Binary Parallax Mapping:***

The third method put to testing is binary parallax mapping. In this case, an iteration count of 10 has been used. As can be seen in the figure *7.1.6*, the algorithm generates an image with no noticeable visual artifacts, which makes this algorithm a good fit for the task.

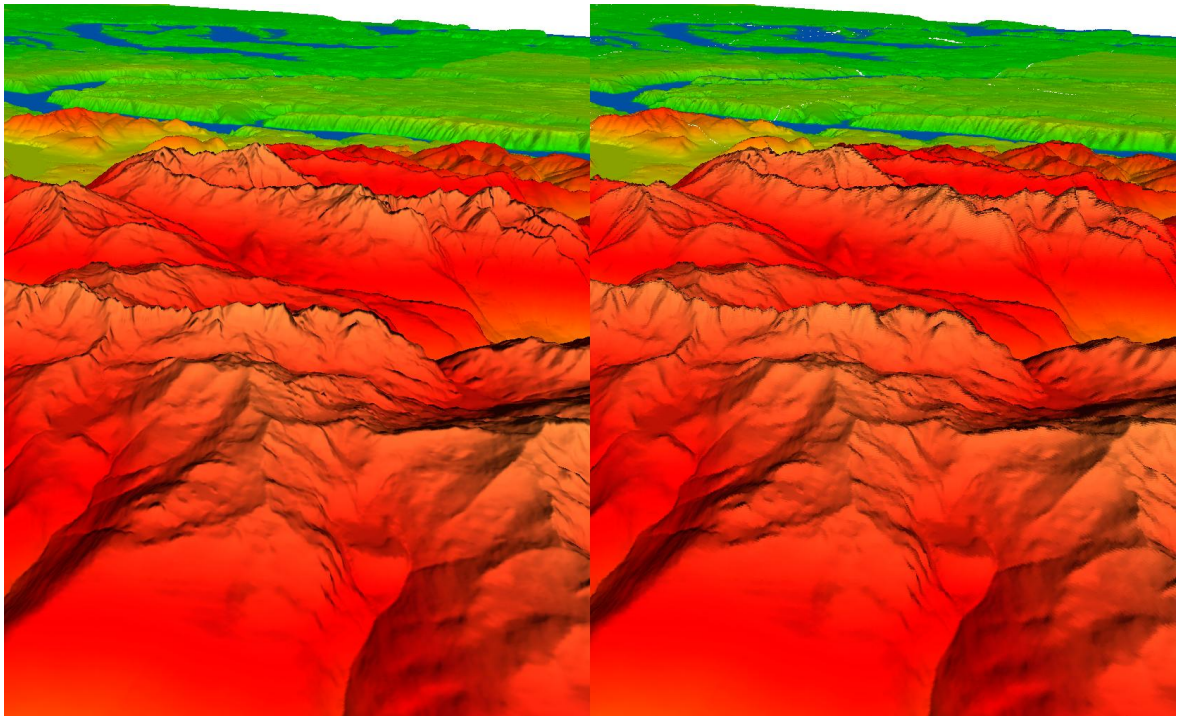


Figure 7.1.6: *Simple tessellation versus binary parallax mapping.*

However, in the figure *7.1.7* the point of view of the camera has been lowered to a point under the maximum height of the terrain. As can be seen, horizontal artifacts have appeared, resulting into a stair like appearance of the terrain.

Moreover, in some parts of the rendered picture the ray between the camera eye and the terrain structure does not intersect, creating *holes* in the terrain mesh. For this reason, this algorithm is only suitable when the camera is situated in a point which is over the maximum height of the terrain.

This is caused by the way the intersection of the ray between the eye and the structure is calculated, which assumes that the structure is coplanar with the XZ plane.

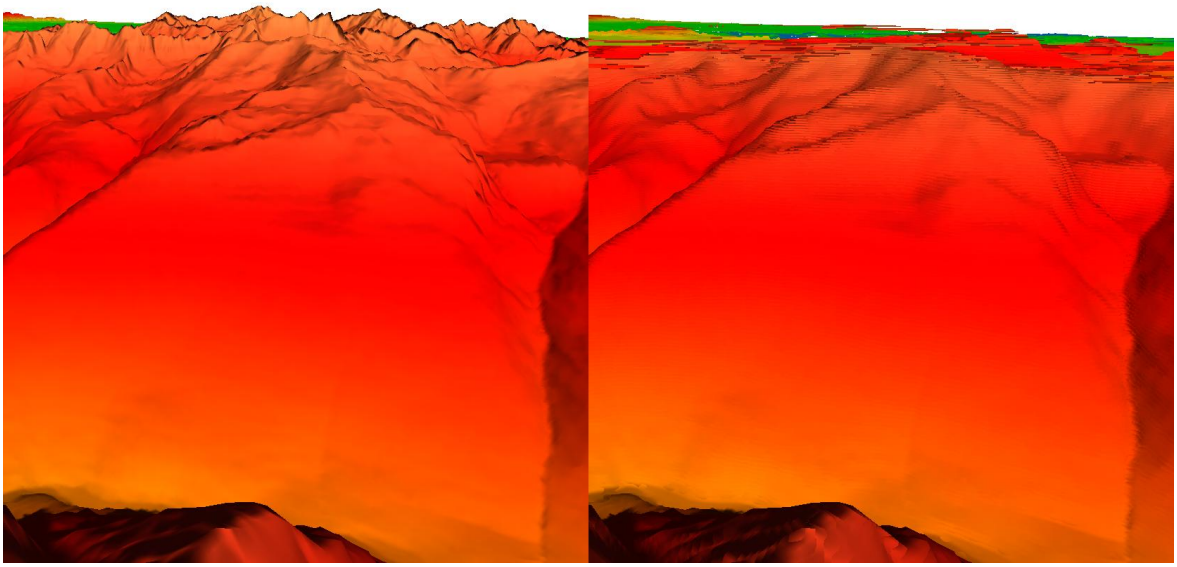


Figure 7.1.7: *Artifacts caused by the binary parallax mapping algorithm.*

- ***Secant parallax mapping with linear search:***

The last algorithm put to test is the algorithm mixing linear search and the secant parallax mapping algorithm. This method produces good results for very high iteration counts, over one hundred iterations, as can be seen in the figure *7.1.9*, but many visual artifacts on low iteration counts as in the figure *7.1.8*.

The iteration count must be kept as low as possible, since the fragment shader is executed once for each screen fragment. Multiplying by the hundreds its execution may decrease greatly the performance of the application.

To analyze whether a high iteration count will decrease the performance of the application, a test has been conducted obtaining the mean frames per second with varying iteration counts.

The figure *7.1.10* shows the test for iteration counts ranging from 10 to 160 in steps of 30 iterations. The management of the level of detail has been set to automatic management with a minimum distance of 5 units, and the test has been run three times.

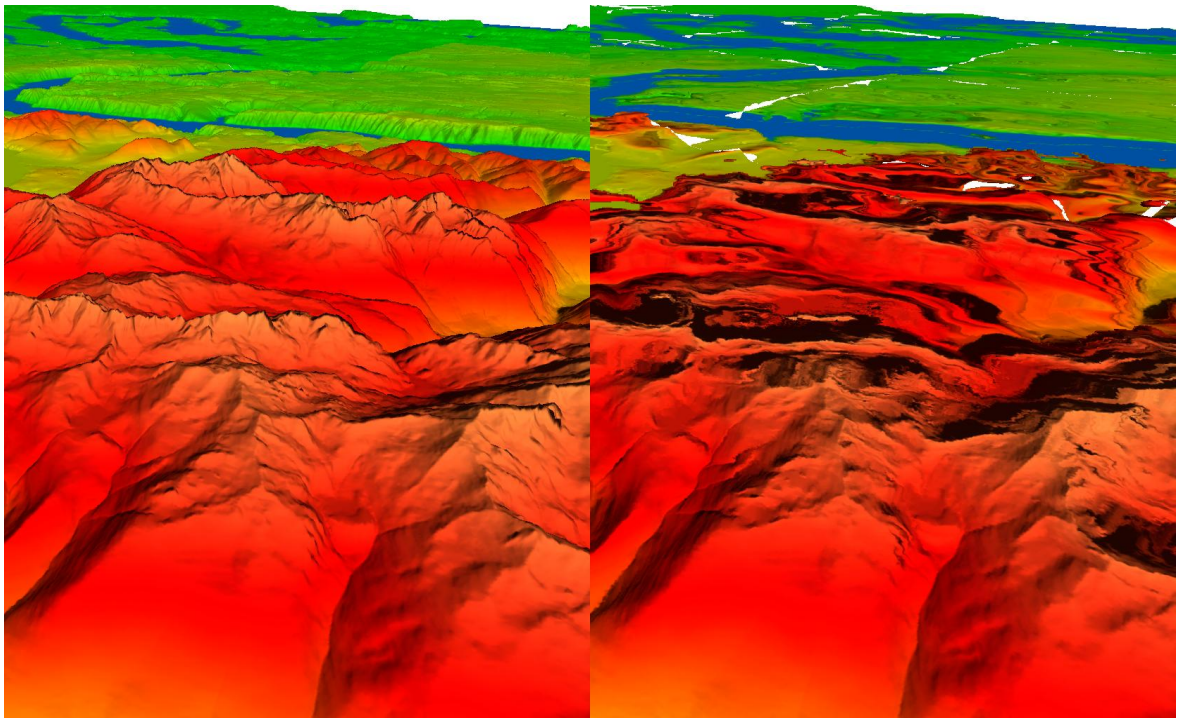


Figure 7.1.8: *Secant parallax mapping with 10 iterations.*

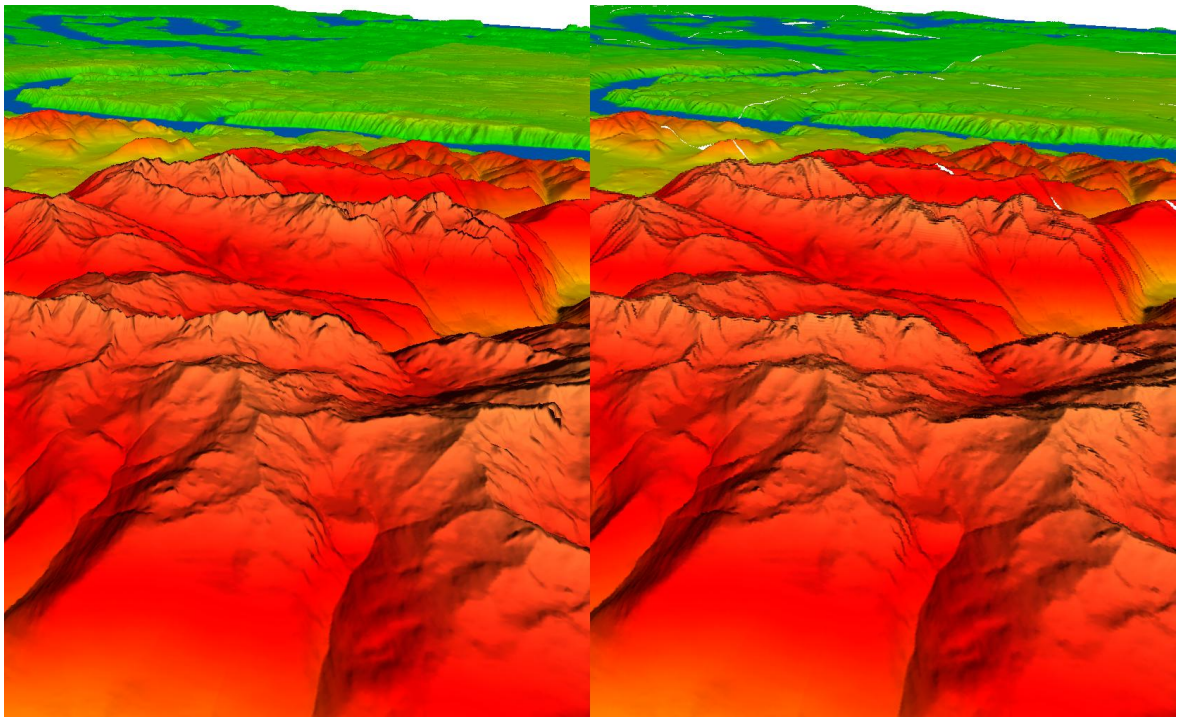


Figure 7.1.9: *Secant parallax mapping with 150 iterations.*

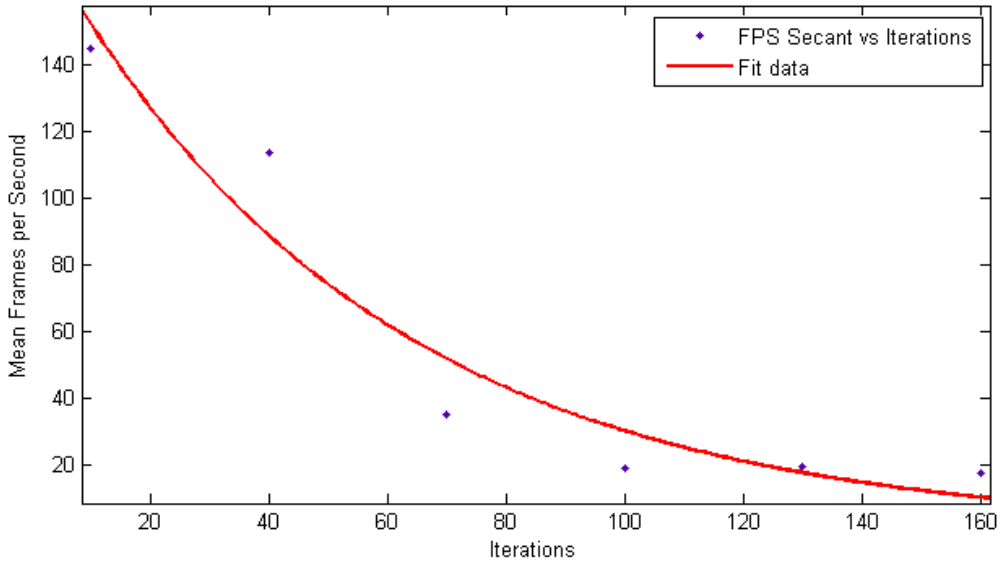


Figure 7.1.10: *Mean FPS of the secant plus linear method with variable iteration counts.*

The results from the test in the figure *7.1.10* show how the increase of the iteration count does also reduce greatly the mean frames per second, quickly getting results below the interactive levels. With iteration counts around the hundreds, which were deemed the only where visual results were acceptable, the mean frames per second are too low to enable interactivity.

For this reason, this implementation of the secant plus linear search algorithm is not a valid method for real time terrain rendering.

- ***Visual tests results***

With this information at hand, the post-processing methods that are going to be tested on the next experiment are the parallax mapping taking into account the slope method and the binary parallax mapping method, which will analyze the performance and quality of the post processing effect.

The simple parallax mapping method has been deemed not valid for terrain rendering, and the secant parallax mapping with linear search algorithm is not valid in its current implementation for interactive terrain rendering.

- ***Testing of the Post-processing Effects***

The objective of this test is comparing the quality and performance of the post-processing methods with the simple tessellation method. The main reason for the implementation of the post-processing methods was obtaining a better visual quality with a lower computational cost, achieving better performance. This test assesses whether this is achieved or not.

To do so, the application is run with the test data set, saving captures of the renderer viewport to the disk. This captures will be the base data for the comparison. This first run is done using simple tessellation with the maximum level of detail possible.

Then, the application is run with the different post processing methods and the simple tessellation method with automatic management of the level of detail, also saving the captures of the process. These runs are done at maximum level of detail values ranging from 64 to the 4, in steps of ten levels.

With this data at hand, the mean square error is obtained between the viewport captures of the tessellation method data at the maximum detail and the other methods.

The objective is obtaining a relationship between the mean square error of the obtained images and the target frames per second.

If the post-processing methods suppose an improvement over the simple tessellation method, the mean square error of the post-processing methods should be lower than the mean square error of the simple tessellation method with automatic management of the level of detail.

The minimum distance of the automatic management of the level of detail has been set to 5 units.

Using the captured data, the mean square error for the binary parallax mapping method and the parallax mapping taking into account the slope method have been calculated. The figure *7.1.11* represents the mean square error in the vertical axis and the tessellation factor in the horizontal axis.

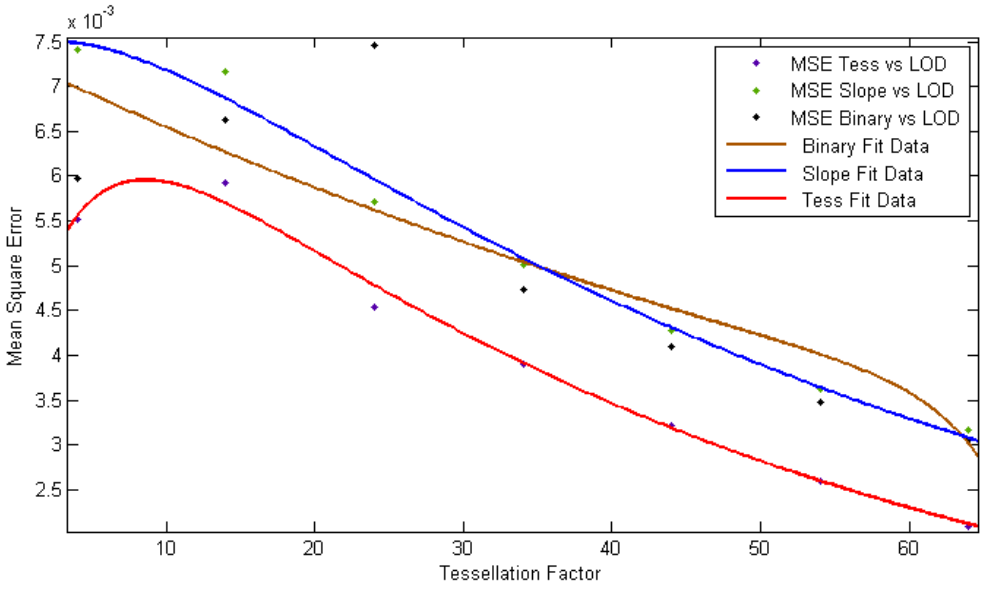


Figure 7.1.11: *Mean Square Error as a function of the tessellation factor.*

The results show how even though the Mean Square Error decreases for all the methods, the MSE of the slope parallax mapping and binary parallax mapping is always higher than the MSE of the simple parallax mapping method using automatic management of the level of detail.

Therefore, according to the obtained results the use of one of the implemented post-processing algorithms does not improve the image quality of the rendered image.

7.2: Results Discussion:

In this section, the results obtained from the tests are discussed, giving an interpretation of the obtained statistical data.

In the previous section, two statistical tests were conducted: a test of the performance of the simple tessellation method and a relation between the mean square error between the visualization with a variable level of detail and a post-processing effect and the visualization with simple tessellation and the maximum level of detail.

- ***Results of the tessellation algorithm tests:***

The result of the first test can be seen in the figure *7.1.2*. The frame rate decreases as the number of triangles being rendered increases.

For most of the time the frame rate stays over the rate set in the begging of the project, 60 FPS. Only when using the maximum level of detail the frame rate drops below the set interactive target.

Since this level of detail has been fixed for all the terrain tiles in the project, the tiles far from the camera have been over tessellated. The use of automatic level of detail management solves this issue.

The results in the figure *7.1.3* show how the average frames per second increase with the usage of automatic management of the level of detail, and the frame rate is kept over the minimum interactive levels most of the time.

With these results at hand, the tessellation method has been proven as a valid technology for terrain rendering, achieving stable interactive speeds when using a level of detail management system.

- ***Results of the visual comparison of the post-processing effects:***

From the visual inspection of the rendering using the post-processing effects, the simple parallax mapping is proven to be not appropriate for terrain rendering from a visual point of view, due to the large number of visual artifacts rendered.

The parallax mapping taking into account the slope method produces a rendering without too many noticeable visual artifacts, proving this technology as a valid technology from a visual point of view for terrain rendering.

The binary parallax mapping method produces a rendering with even less visual artifacts. However, the effect produces stair like artifacts when the point of view is lowered to a point under the maximum height of the terrain. For this reason, this method is valid for terrain rendering applications in cases where this restriction is taken into account.

And finally, the secant parallax mapping with linear search algorithm only produces acceptable results with very high iteration counts.

As can be seen on the figure *7.1.*10, t is not appropriate for real time terrain rendering as is, however it is a good starting point to research ways of modifying the algorithm to obtain a good result using low iteration counts.

- ***Results of the post-processing effects tests:***

The tests done with the post-processing effects consisted on obtaining the mean square error of the image rendered by each one of the effects, taking as reference the image rendered by the simple tessellation method with the maximum level of detail.

From the figure *7.1.11* it can be seen how the mean square error of the images is related with the tessellation factor used while rendering with the post-processing effects.

If the post-processing effects were improving the quality of the rendered picture, the MSE of the simple tessellation method should be below the MSE of the post-processing methods. However, for all the ranges of tessellation factor the MSE of the simple tessellation method is lower than for the post-processing methods.

This means that the post-processing methods do not improve the visual quality of the image, but rather worsen the rendered image.

7.3: Cost Evaluation

In this section, the time cost of the finished project is evaluated to analyze whether there have been differences between the original time cost estimation and the real time needed to complete the project tasks.

With this new cost evaluation, the economic cost of the project will be reassessed, obtaining the final cost of the complete project.

The starting date of the project was the 10th of January 2015, and the due date was estimated to be the 23rd of September 2015.

The times needed to complete the tasks of the complete project have been as shown in the table *7.2*. The final completion date has been advanced to the 2nd of July 2015.

During the development of the project, the times estimated by the expert committee for the study of the state of the art and for the implementation of the different systems of the project have been almost identical to the estimated times.

However, the analysis and design of both applications, together with the processing of the tests results and the writing of the documentation have taken less time than estimated, allowing the project to be finished in an earlier date.

With this final task duration, the project spans a total of 124 days. The figure *7.3.1* shows the updated Gantt diagram, with the final task times of the project. The next step is the calculation of the updated economic cost of the project.

The single personnel resource of the project had a cost of **13€ / hour**. The project has been finished after 124 days, which translate into **992 hours**, using 8 working hour days. The cost of the developer is then **12896€**.

The material resources are unchanged except for the addition of a *MATLAB r2015a Student License + Tool Fitting Toolbox* software license, with a combined cost of 89€ including taxes. The final cost of the project is shown in the table *7.3*.

Task Name	Final Task Times	Estimated Task Times
1. Study of the State of the Art	22 days	22 days
2. System Definition	3 days	7 days
2.1. Requirement Capture	3 days	7 days
2.2. Cost Estimation	2 days	3 days
3. Analysis of the Designer Application	5 days	8 days
3.1. Analysis of the Viewer Application	5 days	10 days
4. Design of the Designer Application	5 days	11 days
4.1. Design of the Viewer Application	5 days	13 days
5. Implementation of the Designer Application	15 days	22 days
5.1. Implementation of the Data Structures	3 days	6 days
5.2. Implementation of the Renderer (Viewer Application)	13 days	16 days
5.3. Implementation of the Tessellation Rendering Method (Viewer Application)	8 days	10 days
5.4. Implementation of the Post-processing Effects (Viewer Application)	10 days	10 days
6. Result Processing	5 days	14 days
7. Writing of the documentation	15 days	24 days

Table 7.2: *Final task times of the project.*

Element	Cost
Personnel Resources	12896€
Material Resources	1775€
Total before taxes	14671€
Total after taxes (21%)	**17751,91€**

Table 7.3: *Complete cost of the finished project.*

Id	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		Study of the State of the Art	22 días	lun 12/01/15	mar 10/02/15	
2		**System Specificatio**	**8 días**	**mié 11/02/1**	**vie 20/02/1**	**1**
3		System Definitio	3 días	mié 11/02/1	vie 13/02/1	
4		Requirement Cap	3 días	lun 16/02/1	mié 18/02/1	3
5		Cost Estimation	2 días	jue 19/02/1	vie 20/02/1	4
6		**System Analysis**	**10 días**	**lun 23/02/1**	**vie 06/03/1**	**2**
7		Analysis of the Designer	5 días	lun 23/02/15	vie 27/02/15	
8		Analysis of the Viewer	5 días	lun 02/03/15	vie 06/03/15	7
9		**System Design**	**10 días**	**lun 09/03/1**	**vie 20/03/1**	**6**
10		Design of the Designer	5 días	lun 09/03/15	vie 13/03/15	
11		Design of the Viewer	5 días	lun 16/03/15	vie 20/03/15	10
12		**System Implement**	**54 días**	**lun 23/03/1**	**jue 04/06/1**	**9**
13		Implementation of the Designer	15 días	lun 23/03/15	vie 10/04/15	
14		Implementation of the Data	3 días	lun 13/04/15	mié 15/04/15	13
15		Implementation of the Renderer (Viewer	13 días	jue 16/04/15	lun 04/05/15	14
16		Implementation of the Tessellation Rendering	10 días	mar 05/05/15	lun 18/05/15	15
17		Implementation of the Post-processing	13 días	mar 19/05/15	jue 04/06/15	16
18		Result Processing	5 días	vie 05/06/1	jue 11/06/1	12
19		Writing of the Documentation	15 días	vie 12/06/15	jue 02/07/15	18

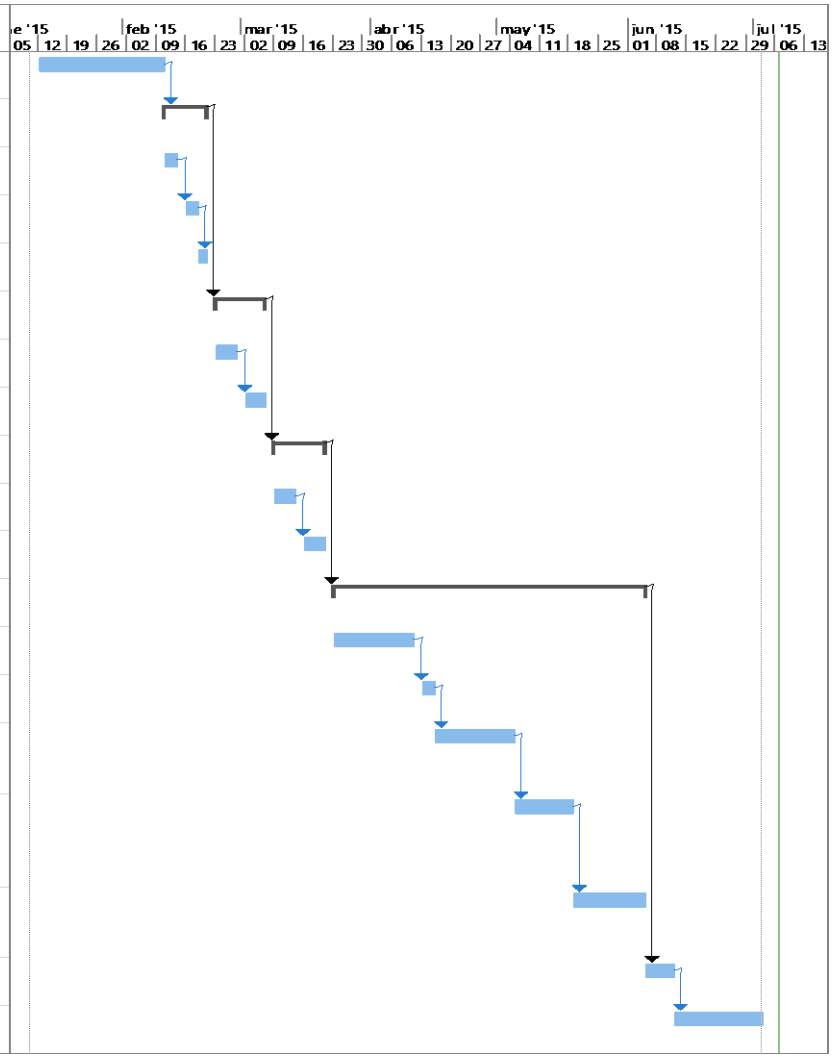


Figure 7.3.1: *Updated Gantt diagram of the project.*

CHAPTER 8: CONCLUSIONS

At this point of the project, the state of the art was investigated, to find the latest technologies in terrain rendering and a selection was done of several rendering technologies based on the GPU to find out whether these are suitable or not for terrain rendering, and to establish a comparison between the methods and find out which one is more appropriate for the task.

To be able to conduct these tests, a terrain viewer application was developed, which allows visualizing the terrains using these technologies and obtaining the statistical data to do the comparison, along with a terrain designer application to prepare the input data used in the terrain viewer application.

The results of the comparisons between the tested methods did prove using tessellation as a valid method for terrain rendering, obtaining interactive speeds for most of the time when using an automatic level of detail management system.

However, post-processing methods using the algorithms implemented on the project have been proven to not do any kind of improvement over the rendered image, proving them as a non valid method for terrain rendering.

The application set is not only valid for the conducting of the performance tests but also is valid as a general purpose terrain visualizing application. With the terrain designer and viewer set, geographical data can be set up to create a terrain project and later visualize it, creating flight animations.

8.1: Future Work

Even though the project is at a very advanced stage, there is still a lot of room for improvement and research.

However, the applications might not have the correct structure for an immediate release, since the design has been done catering to the objectives of this project. The SRTM conversion system has already been packaged as an open source application, released as an open source tool called *srtm2tiff*.

One of the possible future developments using the developments done in the application is the creation of a stand-alone library for terrain rendering, which would enable developers to integrate a terrain visualization engine to their applications.

Another point of interest is the improvement of the different methods to obtain even better results. The secant parallax mapping with linear search presented very good results at a high iteration count. A possible improvement is continuing with the research of this method to get good results at low iteration counts.

Another improvement to the developed methods is reducing the visual artifacts caused when the point of view of the camera goes under the maximum height of the terrain. A possible point of research is modifying the algorithms to be able to render the terrains at an arbitrary point of view.

The applications have been developed to work under the Windows operative system. An interesting future improvement could be porting the developed systems to work with other operating systems and platforms.

The latest improvements in 3D graphics for mobile technologies enable the usage of GPU tessellation with extensions of the OpenGL ES 3.0 graphics backend library. Thanks to the availability of this technology, another path could be creating a terrain rendering system for mobile devices using the technologies developed in this project.

REFERENCES

[Bli78] James F. Blinn: Simulation of wrinkled surfaces. In SIGGRAPH '78 Proceedings of the 5th annual conference on Computer graphics and interactive techniques (1978), pp. 286-292.

[Bon11] Xavier Bonaventura: Terrain and Ocean Rendering with Hardware Tessellation. In GPU Pro (2011).

[Ca11] Ian Cantlay: DirectX 11 terrain tessellation. Nvidia whitepaper (2011).

[Di10] C. Dick, J. Kriigerg, R. Westermannl: GPU-aware hybrid terrain rendering. In Computer Graphics and Visualization Group, Technische Universität München, Germany (2010).

[Du97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein: ROAMing terrain: real-time optimally adapting meshes. In VIS '97 Proceedings of the 8th conference on Visualization (1997), pp. 81-88.

[Kan01] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami: Detailed Shape Representation with Parallax Mapping. In ICAT, December 5-7 (2001).

[Li01] Peter Lindstrom, Valerio Pascucci: "Visualization of Large Terrains Made Easy", in IEEE Visualization '01 Conference Proceedings (2001).

[Lo04] Frank Losasso, Hugues Hoppe: Geometry clipmaps: terrain rendering using nested regular grids. In SIGGRAPH, ACM Transactions on Graphics (TOG) - Proceedings of ACM (2004), Volume 23 Issue 3, pp. 769-776.

[MM05] Morgan McGuire, Max McGuire: Steep Parallax Mapping. I3D Poster; Brown Technical Report (2005).

[Mu89] F. K. Musgrave, C. E. Kolb, R. S. Mace: The synthesis and rendering of eroded fractal terrains. In SIGGRAPH '89 Proceedings of the 16th annual conference on Computer graphics and interactive techniques (1989), pp. 41-50.

[Pol05] Fábio Policarpo, Manuel M. Oliveira, João L. D. Comba: Real-time relief mapping on arbitrary polygonal surfaces. In Proceedings of the 2005 symposium on Interactive 3D graphics and games (2005), pp. 155-162.

[Pre06] Matyas Premecz: Iterative Parallax Mapping with Slope Information. In Central European Seminar on Computer Graphics (2006).

[Tat06] Natalya Tatarchuk: Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering. In SIGGRAPH '06 ACM SIGGRAPH Courses (2006), pp. 81-112.

[Wa01] I. Wald, P. Slusallek: State of the art in interactive ray tracing. EUROGRAPHICS (2001).

[We04] Terry Welsh: Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces. Infiscape Corporation (2004).

[Ye04] Keith Yerex, Martin Jagersand: Displacement mapping with ray-casting in hardware. In Proceeding SIGGRAPH '04 ACM SIGGRAPH Sketches (2004), p. 149.

[Yus11] E Yusov, M Shevtsov: High-performance terrain rendering using hardware tessellation. In GPU Pro (2011).

BIBLIOGRAPHY

OpenGL Programming Guide, Eight Edition.
By *Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane.*
Khronos Opengl ARB Working Group.

Mathematics for 3D Game Programming and Computer Graphics, Third Edition.
By *Eric Lengyel.*
CENGAGE Learning.

Graphics Shaders, Theory and Practice, Second Edition.
By *Mike Bailey, Steve Cunningham.*
CRC Press.

OpenGL 4.0 Shading Language Cookbook.
By *David Wolff.*
Packt Publishing.

Windows Desktop Applications Programming Guide.
Microsoft Corporation.

OpenGL 4 Reference Pages.
Khronos Group.

ANNEX: USER MANUAL

This section describes how to use the applications of the project to create a working terrain project, explaining step by step how to use the source data to compile the project, and how to visualize it in the terrain viewer.

- ***File structure of the applications:***

The terrain viewer and terrain designer are included in the online package and in the CD-ROM appended to the printed version of the terrain. The applications must have the following files in their directory to work properly.

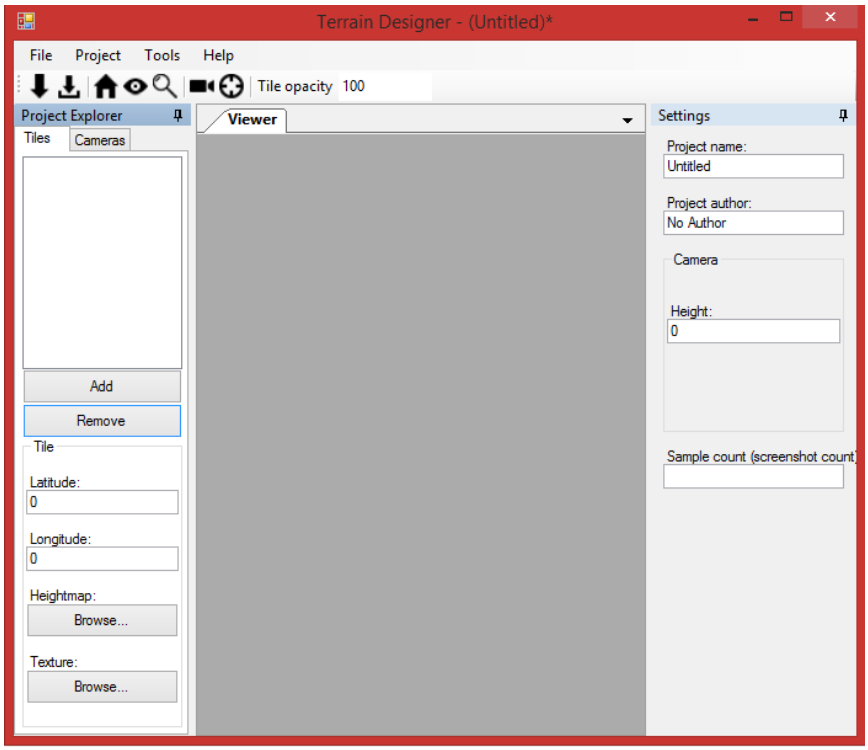
Terrain Viewer	***Terrain Designer***
Designer.exe	Viewer.exe
BitMiracle.LibTiff.NET.dll	zlib1.dll
BitMiracle.LibTiff.NET.xml	libtiff.dll
OpenTK.Compatibility.dll	glfw3.dll
OpenTK.dll	glew32.dll
OpenTK.GLControl.dll	shaders/ f_Color.glsl
WeifenLuo.WinFormsUI.Docking.dll	shaders/ f_TerrainSimple.glsl
data/ camera.png	shaders/ f_Texture.glsl
data/ missing.bmp	shaders/ f_TextureTessellation.glsl
	shaders/ tcs_PassThrough.glsl
	shaders/ tes_MatrixTransformQuads.glsl
	shaders/ tes_PassThrough.glsl
	shaders/ tes_TerrainTess.glsl
	shaders/ v_MatrixTransform.glsl
	shaders/ v_MatrixTransformColor.glsl
	shaders/ v_PassThrough.glsl
	shaders/ v_TerrainSimple.glsl
	shaders/ v_TerrainTess.glsl

If any of the files in this list is missing from the installation, the application should be reinstalled to ensure that it works properly.

Terrain Designer User Manual

- ***Creating a new terrain project:***

To begin the creation of a terrain project, open the terrain designer application and go to the menu option *File -> New Project*. A new project empty project will be created. The figure shows the state of the application when a new project has been just created.



- ***Saving the work of the project:***

To save the progress done in the project, use the menu option *File -> Save Project*. Select the destination folder in the dialog and set the file name. Then, press *Save*. This can also be accessed through the toolbar ⬇ button.

- ***Loading a saved project:***

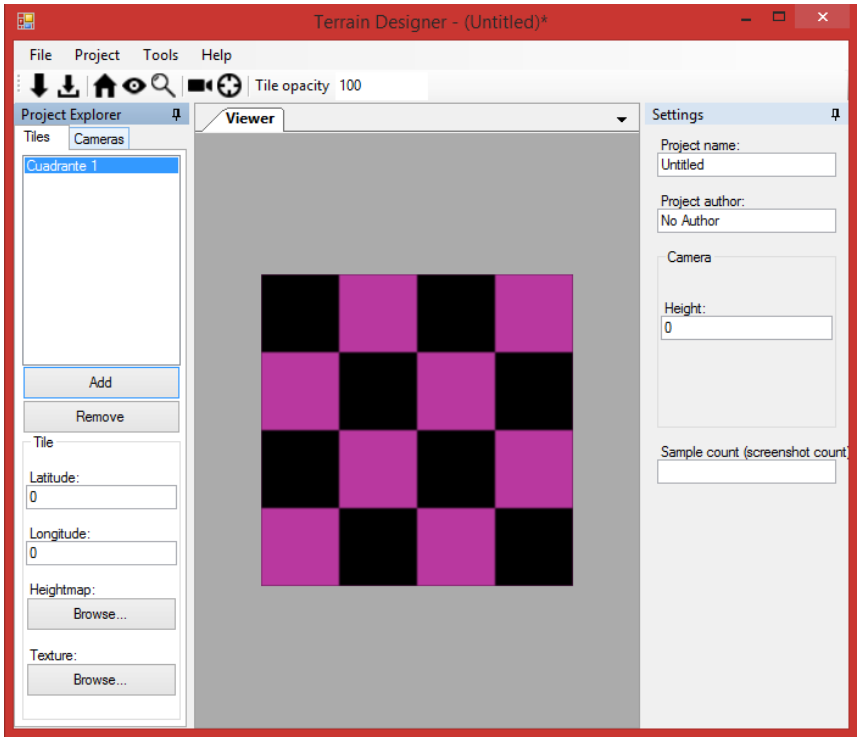
To retrieve the work done in a saved project, use the menu option *File -> Load Project*. Select the project file in the dialog, and press *Open*.

- ***Editing the terrain project:***

The editing of the terrain project is done through the creation of terrain tiles and camera waypoints. In the left side of the application is the project explorer toolbox. From this toolbox, terrain tiles can be created and modified.

- ***Creating a new tile:***

To add a tile to the project, go to the *Tiles* tab in the left toolbox and use the *Add* option. The created tile will have latitude zero, longitude zero and no heightmap or color texture, as can be seen in the next figure.



- ***Editing a tile:***

To edit an existing tile, select one of the tiles from the tile list in the project explorer toolbox. Edit the *Latitude* and *Longitude* values to change the position of the tile, and use the *Heightmap* and *Texture* buttons to select the file for the height and color texture respectively.

- ***Removing a tile:***

To remove a tile, select the target tile from the tile list and press the *Remove* button.

- ***Moving the viewport area:***

To move the content area, press the toolbar button. Then, click on the viewport with the mouse and drag it to set the viewport to its new position. If the viewport has been moved too far, use the button to set the viewport to the origin of coordinates of the terrain.

- ***Zooming the viewport area:***

To zoom in or out the viewport area, press the button. Then, click on the viewport with the mouse and drag it up to reduce the zoom level or down to increase the zoom level.

- ***Adding a camera waypoint:***

To add a new camera waypoint, go to the *Cameras* tab in the project explorer toolbox and press *Add*. This option can also be accessed through the button. The new waypoint will be set in the center of coordinates of the terrain project.

- ***Moving a camera waypoint:***

To change the position of the camera waypoint, select a camera from the camera list and edit the *Position* fields in the project explorer toolbox. Alternatively, select a camera from the list and press the button. Then, drag the mouse to set the new position of the waypoint.

- ***Exporting the terrain project:***

To export the terrain project to a compiled version use the menu option *Project -> Build* or the toolbar button. Select the destination directory for the project and press *OK*.

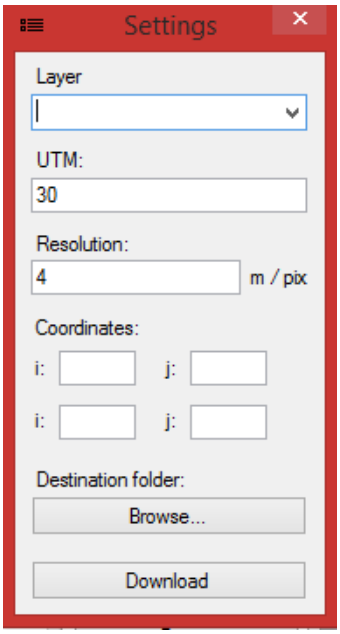
The application will generate a file called *project.xml*, with the project information, and two folders, *heightmaps* and *textures*, containing the height and color textures of the terrain. For this reason, exporting the project to an empty directory is recommended, to avoid any possible data loss.

- ***Downloading a set from Iberpix:***

To start the download of an Iberpix dataset, use the option *Tools -> Download from Iberpix.* In the settings dialog, set the layer to *SPOT5 Texture* or *Heightmap* to download the color or height textures respectively.

Set the UTM and resolution, the start coordinates and the end coordinates and a destination directory with the *Browse* button.

Press *Download* to start the process. A dialog will be shown with the download progress, and the dataset textures will be saved to the destination directory. For this reason, it is recommended to set an empty folder as the destination directory.



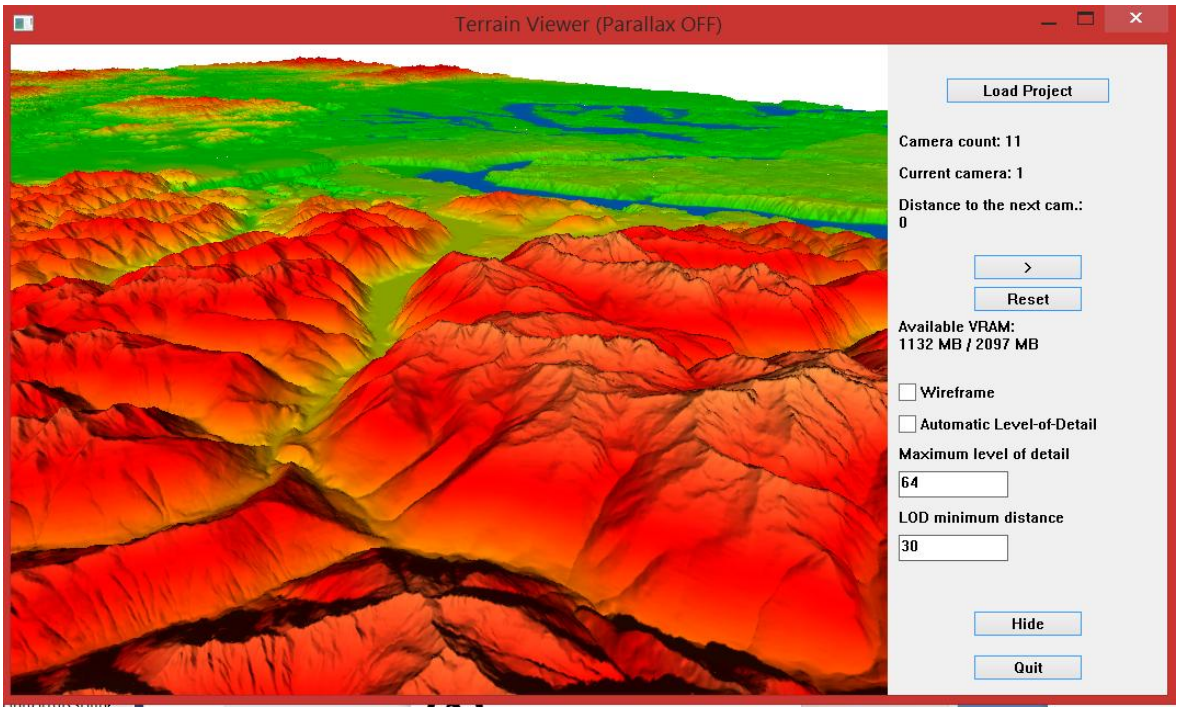
- ***Converting a SRTM file to a TIFF file:***

To convert a SRTM file, use the menu option *Tools -> Convert SRTM to TIFF*. Select the file to be converted and press *OK*. The application will convert the file to TIFF and save it in the same directory as the original file, but with *.tiff* extension.

Terrain Viewer User Manual

- ***Visualizing a compiled terrain:***

Open the terrain viewer application and use the *Load Project* option. Select the XML file of the exported project, and the application will load the terrain project.



- ***Changing the viewport camera:***

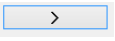
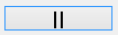
To change the position of the camera, use the arrow keys of the keyboard to move forward, move backwards, strafe left and strafe right. Use the W key to go up and the S key to move down.

To change the target of the camera, click on the viewport and drag the mouse to set the new target.

- ***Taking a screenshot of the rendering:***

To take a screenshot, press the *F2* key on the keyboard. The screenshot will be saved on the project root directory with TIFF format.

- ***Starting and pausing a camera flight animation:***

To toggle the camera flight animation if the waypoints are available, use the  button to start the animation. When the animation is running, the same button turns into the  button. Press it to stop the camera animation.

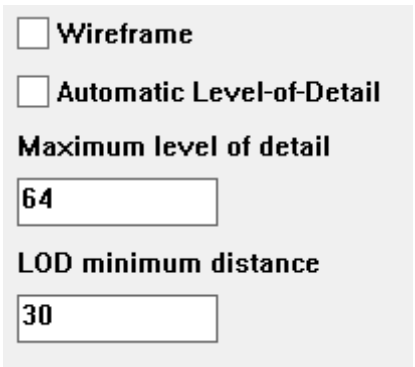
- ***Changing the render settings:***

The render settings of the application can be changed using the controls in the side bar.

The *Maximum LOD Level* edit field sets the maximum level of detail of the application. The *Minimum LOD Distance* edit field sets the minimum distance the automatic LOD management should get turned on.

Use the *Wireframe* and *Automatic Level-of-Detail* to change the rendering from fill to wire mode and to change from manual to automatic management of the level of detail and vice versa.

Use the *P* key to switch from using post-processing to using simple terrain tessellation.



ANNEX: TERRAIN XML FORMAT

This section describes the tags existing in the XML format used to serialize the project by both the terrain designer application and the terrain viewer application.

Tag	***Description***	***Mandatory***
<project>	Container of the terrain project. It must contain a *name*, *author*, *date*, *resolution* and *height*. It might contain a set of *tiles* and / or a set of *cameras.*	Yes
<name>	Name of the project.	Yes
<author>	Author of the project.	Yes
<resolution>	Resolution of the project.	Yes
<date>	Creation date of the project.	Yes
<height>	Height for the camera flight animation of the project.	Yes
<tiles>	Container of all the *tile* objects.	No
<tile>	Container of a single tile. Must contain *latitude*, *longitude*, *texture* and *heightmap*.	No
<latitude>	Latitude of a tile. Must be a real integer.	Yes
<longitude>	Longitude of a tile. Must be a real integer.	Yes
<texture>	Path to the color texture. Saved project expect to contain the absolute path to the source data. Compiled projects should contain the relative path to the texture.	Yes
<heightmap>	Path to the Height texture. Saved project expect to contain the absolute path to the source data. Compiled projects should contain the relative path to the texture.	Yes
<cameras>	Container of all the *camera* objects.	No
<camera>	Container of a single camera waypoint. Must contain *x* and *y*.	No
<x>	Position of the camera waypoint in the X axis. Must be a real float.	Yes
<y>	Position of the camera waypoint in the Y axis. Must be a real float.	Yes