NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

LOURENÇO MANUEL VIANA GUARDA SOARES

BSc in Computer Science

FOREST FIRE VISUALISATION IN A VR ENVIRONMENT

MASTER IN COMPUTER SCIENCE AND INFORMATICS

NOVA University Lisbon

February, 2023

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

FOREST FIRE VISUALISATION IN A VR ENVIRONMENT

LOURENÇO MANUEL VIANA GUARDA SOARES

BSc in Computer Science

Adviser: Fernando Pedro Reino da Silva Birra
Auxiliar Professor, NOVA University Lisbon

Co-adviser: João Carlos Gomes Moura Pires
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: Name of the committee chairperson
Full Professor, FCT-NOVA

Rapporteur: Name of a rapporteur
Associate Professor, Another University

Members: Another member of the committee
Full Professor, Another University

Yet another member of the committee
Assistant Professor, Another University

MASTER IN COMPUTER SCIENCE AND INFORMATICS

NOVA University Lisbon
February, 2023

Forest Fire Visualisation in a VR environment

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.10) [32].

This dissertation is dedicated to the brave firefighters of Portugal

Acknowledgements

This thesis would not have been possible without the immense patience and support of my advisor Fernando Birra. His assistance and helpfulness echoes throughout this paper, and his friendliness was a much needed light on the many trials and tribulations that came throughout this paper.

In addition, I also want to extend my gratitude to the NOVA School of Science and Technology, for providing me with the hardware and knowledge that allowed me to develop and finish this project. An additional thank you goes to Professor João Lourenço for freely providing the template for this document.

Finally, it is imperative for me to state just how grateful I am to my friends and family for supporting me throughout the development of this thesis, as well as the development of my life. Many of you with which I have fought duels, climbed difficult walls, made questionable art, and discussed the theory of everything. I would not be where I am today without all of you.

"Cheering people up and spreading happiness is never trivial. In fact, it may be the least trivial thing of all." (Andrew W.K.)

Abstract

Forest fires in Portugal are a longstanding problem, having been exacerbated by the effects of climate change. Being able to effectively combat wildfires requires study of previous fires, to learn how they spread and what can be done to prevent them in the future.

Virtual reality proposes an interesting visualisation method. It allows for an up-close exploration of fire spreading phenomena, which when combined with the inherit immersiveness of the medium, can prove highly educational in a way that is not easily communicated by simply looking at data points in a two dimensional map. Such technology can be implemented through a game engine such as Unreal Engine.

The NOVA School of Science and Technology developed a tool dedicated to the study of past wild fires in Portugal collected using MACFIRE, visualized through virtual reality. It currently allows for a macro visualisation of the fire front's movement in 10 minute intervals.

This dissertation proposes a method for enhancing the micro visualisation of the fire, which should have a minimal impact on the real-time performance of the final tool. These enhancements will allow the users to obtain a better understanding of how a wildfire spreads between trees and underbrush on a micro scale. The work carried out during the duration of this dissertation consisted of a tool for importing landscapes into Unreal Engine, documentation corrections, and an engine upgrade to the already existing forest fire tool. Then, a fire spreading system was written, which allows users to create fires in the world and watch them evolve.

This system was tested throughout multiple stages of its development, as well as tested with a final scenario in virtual reality to stress the graphics processor. The tool's efficiency was evaluated through analysis of its performance, for both the processor and the graphics card. The final API was also analysed and evaluated to show that it is ready to be integrated into the final tool.

While the system that was developed does not directly allow the user to observe the data from an already existing fire, it provides methods for this to be integrated into the final tool.

Keywords: Virtual Reality, Unreal Engine, Real-time, Data Modelling, Data Visualisation, 3D Rendering

Resumo

Em Portugal há muito tempo que os fogos florestais constituem um problema, mas este tem-se vindo a agravar ultimamente devido às alterações climáticas. O combate efetivo aos fogos florestais exige um bom conhecimento dos fogos anteriores e do modo como se espalharam. Só assim será possível perceber como os prevenir no futuro. A realidade virtual propõe um método de visualização interessante que permite acompanhar de perto o fenómeno do avançar do fogo. Quando combinada com a sua natural envolvência, a realidade virtual torna-se muito mais eficaz do que os dados inseridos num mapa a duas dimensões. Esta tecnologia pode ser desenvolvida através de um motor de jogo como o Unreal Engine.

A Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa desenvolveu uma ferramenta para o estudo dos fogos florestais ocorridos em Portugal, utilizando a MACFIRE e visualizando-os com recurso à realidade virtual. Esta ferramenta permite já a macro visualização do movimento da frente de fogo com intervalos de 10 minutos.

Nesta dissertação propõe-se um método para melhorar a visualização de um fogo numa escala micro, com um impacto mínimo no desempenho em tempo real da ferramenta final. Estes melhoramentos permitirão aos utilizadores compreender mais facilmente o modo como os fogos florestais avançam através de árvores e vegetação rasteira numa escala micro.

O trabalho desenvolvido nesta dissertação consiste na criação de uma ferramenta para importar terreno para o Unreal Engine, em correções à documentação já existente e em atualizações da ferramenta de visualização de fogos. De seguida, desenvolveu-se um sistema de simulação de avanço de fogos, que permite aos utilizadores criar fogos e observar a sua evolução.

Este sistema foi testado não só à medida que foi desenvolvido, como também num cenário final em realidade virtual para fazer um teste de stress ao processador gráfico. A eficiência da ferramenta, tanto no processador como na placa gráfica, foi avaliada através da análise ao seu desempenho. A API final foi analisada e avaliada de forma a demonstrar que pode ser integrada na ferramenta final.

O sistema desenvolvido não permite a observação de dados de fogos já ocorridos, mas

disponibiliza métodos para a sua integração na ferramenta final.

Palavras-chave: Realidade Virtual, Unreal Engine, Tempo Real, Modelação de Dados, Visualização de Dados, Renderização 3D

Contents

List of Figures

List of Tables

Acronyms

AABB Axis Aligned Bounding Box *(pp. 16, 36, 41, 42)*
AR Augmented Reality *(p. 4)*

DEM Digital Elevation Model *(pp. xiii, 5, 30–32, 60)*
DLSS Deep Learning Super Sampling *(p. 5)*

FCT NOVA School of Science and Technology *(pp. 1, 6)*
FPS Frames Per Second *(pp. xvi, 4, 26, 44, 61, 65)*
FSR FidelityFX Super Resolution *(p. 5)*

GDAL Geospatial Data Abstraction Library *(pp. 30, 31)*
GIS Geographic Information System *(pp. 2, 5, 21, 28, 60)*
GML Geography Mark-up Language *(p. 5)*
GPU Graphics Processing Unit *(pp. 5, 10, 18, 26, 27, 61, 65, 70)*

HLSL High Level Shading Language *(p. 20)*

JSON JavaScript Object Notation *(p. 5)*

LOD Level of Detail *(pp. 6, 20, 27)*

PBR Physically Based Rendering *(p. 20)*
PPI Pixels Per Inch *(p. 26)*

UE4 Unreal Engine 4 *(pp. 17, 18)*
UE5 Unreal Engine 5 *(pp. xiii, xiv, 17–20, 30–33, 41, 42, 45, 46, 48, 52, 53, 60, 67, 71)*

VR Virtual Reality *(pp. xvi, 1–6, 9, 14, 21, 26, 61, 65)*

XML Extensible Mark-up Language *(p. 5)*

1 Introduction

1.1 Motivation

Wildfires cause unmitigated damage on land, devastating and heavily impacting both human and wild life. In the year 2022, wildfires in Portugal burned more than 100 thousand hectares of land in a single month, close to 1% of total landmass[11]. This is not an isolated incident, as wildfires are an unfortunate yearly occurrence in the country.

Understanding how the wildfires spread, as well as the impact of fire fighting techniques, is paramount to being able to reduce the damage caused by a fire. This requires a study of previous fires, which can be difficult to understand by simply looking through discrete data points on a map, or impractical to perform field research on.

Virtual Reality (VR) is a good candidate for this problem, as it would allow one to experience the event, from the safety of a simulated world. It would, furthermore, allow one to observe fire spreading phenomena up close, further letting the user understand how the environment and firefighting attempts shape the spread of a fire. While graphical fidelity is not typically the end goal of an educational tool, it is still useful to simulate phenomena in a convincing manner.

Since VR is a real time medium, and quite a complicated one to implement from scratch, the project is being developed with a game engine called Unreal Engine. This eases development in numerous ways, taking the burden of dealing with complex systems such as physics, memory allocation, and scene optimization, away from the developer.

1.2 Context

NOVA School of Science and Technology (FCT) has been developing the SI-MORENA project, which contains a suite of tools. One of these tools' goals it to take data points from existing fires and allowing the user to scrub the timeline of events. This tool also allows one to observe the fire from many different points of view, such as from a macro scale, or up-close from a micro one. The project utilizes VR technology to accomplish this.

The visualization tool uses georeferenced data to generate an accurate map of regions

in Portugal, and combines data from MACFIRE to create a 3D visualization of a fire that has occurred in the past. MACFIRE is a Geographic Information System (GIS) which exists to assist in the modelling of the evolution of a forest fire. When a fire occurs and first responders are sent, data is continuously collected by people in the field regarding the current location, intensity, and number of active fire fronts. The system is also used to log firefighting efforts, including tracking the different methods. The fire front is logged as a set of discrete lines on a map of the afflicted area.

At the time of writing, the tool has techniques developed for interpolating the data from MACFIRE to allow visualizing the evolution of the fire front in discrete 10 minute time steps. It also provides an interface that allows the user to place themselves (virtually) in a location of a forest fire, and to observe its spread. This is accomplished through the use of rough graphics to convey the idea of the event.

The end goal of the wildfire visualization tool should be to automate as much as possible. It should be easy to plug in data from MACFIRE, and to provide the tool with geographical data from the affected area, ultimately allowing for a fast observation of recent wildfire.

1.3 Problem Description

As one of the goals of the wildfire visualization tool is to educate people on the methods in which a forest fire spreads at a macro scale, it is desirable to further improve it to show how the fire spreads at a micro scale.

Forests are rather complicated to render due to their scale and density, therefore while it is beneficial to improve the fidelity of the tool allow one to observe how a fire spreads in a realistic manner, one must also be careful with the implementation. VR is performance intensive, so shortcuts and tricks must be used to ensure that the tool behaves under real time constraints. Ultimately, the graphical and simulation improvements proposed in this dissertation cannot harm the educational aspect of the tool.

1.4 Contributions

The following contributions are expected from this project:

- **An improved visualization of the fire spreading at a micro scale**. Users should be able to observe phenomena, such as fire spreading from tree to tree, in a convincing manner.
- **The improved fire spreading should not contradict the final data from MACFIRE**. This point ensures that the data not lose its educational value.
- **The system should provide a solid foundation so that the graphical improvements are easy to implement**. While it would be preferable to also improve the graphics,

implementing a system which is flexible enough as to not impact future work is imperative.

- **The added changes should not impact the use of the tool**. This point is true for both the performance of the tool, as well as for the educational use. Considering that the target system is VR, it is imperative that the simulation not impact performance.

1.5 Document Structure

The document is structured as follows:

- Chapter 1 describes the context of the problem, as well as the motivation for solving it, finishing off with the predicted contributions.
- Chapter 2 attempts to break down the problem of rendering fire and presents the current state of fire rendering and simulation technology. Furthermore, the chapter looks into how fire simulations are accomplished in real time interactive mediums.
- Chapter 3 discusses a possible solution to the problem, which was devised during the study of the current algorithms.
- Chapter 4 presents the implementation that was accomplished, including elements that were unsuccessful or unfinished.
- Chapter 5 evaluates the solution, looking at its flexibility, the viability of the different algorithms, as well as performance testing.
- Chapter 6 concludes the dissertation by discussing the finalised project, evaluating the work against the original stated goals, and describing possible future work.

2

Related Work

Before engaging directly on the topic at hand, it is imperative that one research already existing work to gain an understanding of current technologies and how the techniques on display can be used to solve the problem.

The wildfire visualisation tool has been in development for a few years, thus it will be important to look over previous work that provide the groundwork for this problem. Once the baseline has been understood, further research into scientific journals and similar theses can be done. Finally, because the SI-MORENA project focuses on technologies such as VR and Augmented Reality (AR), it is also interesting to look into video games which employ fire spreading mechanics, and in situations where engine information is unavailable, speculation through careful observation and debugging can prove informative to learning how these effects are optimized for real-time rendering constraints.

2.1 Virtual Reality

Virtual Reality, as a technology, is very computationally expensive. The PlayStation 5, currently one of the most technologically advanced home game consoles, supports two different modes: Quality mode and Performance mode[47]. When it comes to games with high graphical fidelity, the former tends to aim for 4K resolutions at 30 Frames Per Second (FPS), while the latter targets 1080p resolutions at 60 to 120 FPS. VR, on the other hand, requires a minimum of 90 FPS to prevent motion sickness[8], and typically renders at 1080p or, more commonly, near 4K quality *for each eye*. The fact that both eyes need to observe the scene at different perspectives means that the scene has to be fully rendered twice, leading to an additional performance strain.

Currently the easiest way to target such demanding rendering profiles is by improving the hardware used to render. This, of course, is not always a viable solution, so instead some techniques are utilized to trick the eye into thinking it is looking at a higher resolution than it is. Techniques such as interlacing have been present since the existence of motion picture[46], with checkerboard rendering being used by game consoles since 2013[42]. VR, on the other hand, can benefit from a technique known as Foveated Rendering[39],

which can combine eye tracking to increase the resolution of the scene close to where the eye is looking while keeping the peripheral vision at a lower resolution. Unfortunately, very few VR headsets in the market have eye tracking technology, so the foveated area is typically static to the center of the frame. NVIDIA also provides Deep Learning Super Sampling (DLSS), which is a real-time machine learning algorithm that is used to upscale an image[10]. This technology is proprietary to NVIDIA, however AMD provides an open source alternative called FidelityFX Super Resolution (FSR)[1].

One interesting rendering technique that is commonly used in VR is Asynchronous Reprojection[3]. This technique stands out because it does not improve performance itself, rather only the perception of the frame rate. This is accomplished by taking multiple previously displayed frames and using the motion information from a headset's sensors to warp the frames into a prediction of what the next scene will look like, until the Graphics Processing Unit (GPU) is able to catch up and render a brand new frame of information. The biggest downsides of this method are that the edges of the scene tend to be distorted (the use of eye tracking can be used to alleviate this problem) and that the motion of objects in the scene cannot be interpolated.

2.2 Geographic Information Systems

A GIS is a collection of systems, both hardware and software, used to store geographical information. This includes, but is not limited to, terrain displacement, landmarks such as cities and roads, latitudinal and longitudinal coordinates, and even custom data[49]. This extra flexibility allows GISes to be used to map anything cartographically, such as crime rates, weather, or traffic. Most of this data can be stored in an Extensible Mark-up Language (XML)-like format known as Geography Mark-up Language (GML), or as a JavaScript Object Notation (JSON)-like format known as *GeoJSON*. Some popular software which can read and modify GIS data include the proprietary *ArcGIS*[2], and the free and open-source alternative *QGIS*[43].

The data stored in a GIS is represented in two main forms: as either a rasterized image or as a vector graphic, with the latter allowing for more precision. One useful type of data that can be generated from a GIS is a heightmap, which is a greyscale raster image where the value of each pixel represents a height value (where lighter colors represent higher elevations than darker colors). An example heightmap is illustrated in Figure 2.1. Heightmaps are commonly referred to as a Digital Elevation Model (DEM) in the context of GIS'.

One concept that can be challenging is representing the three-dimensional shape of the earth as a two-dimensional plane. All projections of a sphere on a plane will require that the surface be distorted in some form, and so many different map projections exist as a compromise[33]. GIS systems are no different, and usually contain embedded within them information about the projection system it uses.

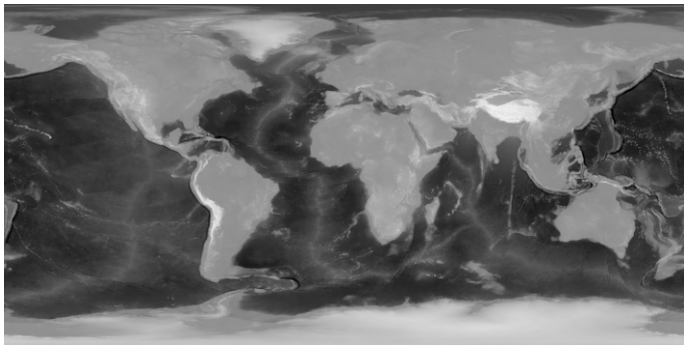


Figure 2.1: A heightmap of the world. The image is 21600x10800 pixels, where each pixel roughly represents 2km (1.8km at the equator) of the earth's surface, while the pixel brightness represents the ratio between the highest point (Mount Everest at 8.8 thousand km above sea level) and the lowest point (The Mariana Trench at 10.1 thousand km below sea level), with sea-level translating roughly to 56% brightness. Image sourced from [4].

.

2.3 MACFIRE and SI-MORENA

Because MACFIRE only contains discrete points when they are manually logged, and said points can vary significantly in location and timestamp, it can be difficult to visualize the event at a macro scale. FCT proposed a VR based visualisation tool to assist in this. One of the important steps was transforming the data into something which could be more easily interacted with. One student[7] proposed a technique for generating polygons from the data, allowing for a greater understanding of how the fire propagates and for filling in the gaps in the timeline into discrete 10 minute steps. This also allowed for the dealing of edge cases such as when two fire fronts unite, which the program struggled to convey previously.

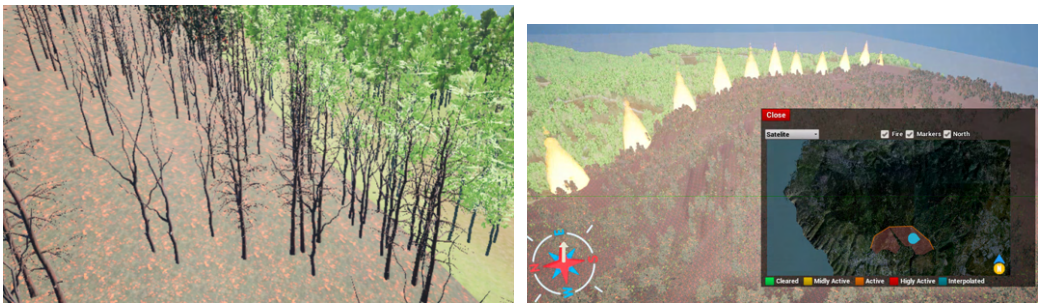
To create a map of the area, the visualisation tool utilizes georeferenced data to create maps of fire afflicted areas. This data is used to both generate terrain and vegetation[36]. The terrain is generated through heightmaps and Unreal Engine's World Composition tool[51], while the vegetation through Unreal Engine's Procedural Foliage Tool[14] and Grass Tool[25]. The data is not used to position every single individual tree or grass object in a given area, rather it provides the program with parameters for vegetation density. The vegetation itself is not placed randomly, rather it uses an algorithm to approximate the natural spread of plant seeds, as well as some filters to prevent the spawning of trees in incorrect areas (such as roads). This can be seen in Figure 2.2.

The plants generated by the Procedural Foliage Tool are instances and have full 3D models, including Level of Detail (LOD)s. The Grass Tool, on the other hand, does not generate instances. Rather, a map is generated to represent the grass object positions, which are only rendered if the camera is close enough.

When burned, the area is changed by darkening the terrain and trees, removing leaves from trees (leaving their skeletons behind), and removing the grass altogether. This is

Figure 2.2: A demonstration of the vegetation filtering in action (Image sourced from [36])

illustrated in Figure 2.3(a). The burned area is represented by a polygon, which is used to filter which areas of the map require switching texture and/or models. Some basic fire particles are placed in the edges of this polygon, as demonstrated in Figure 2.3(b). This polygon is generated by interpolating known data from a forest fire provided by MACFIRE[7]. Therefore, it is not necessary to simulate the actual spread of the fire from the ground up, merely how the fire front transitions between two fixed times in the data.



(a) (b)

Figure 2.3: The current state of the tool's scene rendering. 2.3(a) demonstrates the contrast between an area affected by a fire and one that has not (Image sourced from [36]), while 2.3(b) shows the current method for illustrating the fire front (Image sourced from [7]).

The ultimate goal of this dissertation is not to simulate the propagation of fire in a forest from scratch, but rather attempt to fill the gaps in the 10 minute intervals of the interpolated data in a visual manner. This provides a good compromise between the need for a realistic simulation, and the computational challenges outlined in Chapter 2.1.

2.4 Simulating a Flame

To tackle the subject of simulating fire, one must first look at the physics behind combustion. Fire is an exothermic chemical reaction, where particles quickly exchange bonds with surrounding gases, in a process known as oxidation. The flame is the visible

portion of the fire, and is a mixture of gases that emit light[28]. It would currently be computationally infeasible to simulate all of the chemical processes happening at play, however, since the flame itself is an incandescent gas, it can be modeled as a fluid. This also means that smoke, while not entirely a gas, as it is composed of both solid and liquid particles, can be modeled using the same system (as in, treated like a gas) [37].

There are two main approaches to modeling fluids in a computer. One could choose to model it as individual particles, or as a grid of stationary regions with attributes that represent the average of "imaginary" particles that would be in each grid square. Said grid does not need to be equisized, the size and amount of the grid cells can vary depending on the amount of detail one wishes to represent in a given area (for instance, higher density areas need more accurate computations, so should be represented with more cells). From a mathematical point of view, one can represent the state of a fluid at a given time through a velocity vector. The Navier–Stokes equations are one such representation of the evolution of a velocity field over time[48].

Fluid simulations involve numerous steps, such as diffusion, advection, and divergence. They can be further refined by calculating other properties such as foaming, vorticity confinement, surface tension, and even going as far as trying to simulate non-Newtonian fluids. Of course, the more complex the simulation, the more real-time performance is impacted.

2.4.1 Rendering Fluids

Once we have a working simulation, the next step is to render the flame. A good approach to rendering gases is through the use of volumetric rendering, because the common rendering technique of using 3D triangles would prove very computationally challenging for the high quality meshes needed to get a good looking result. Instead, the shape is sliced into isosurfaces and rendered through a collection of billboards. This is typically done using the marching cube algorithm or through ray marching[12].

The rendering of volumetric fire can prove useful for emulating light emission without the need for ray tracing. Using the volumetric fire data and the Radiative Transport Equation, Pegoraro and Parker provided us a method to calculate the evolution of the fire's radiance[40]. Most of the light generated is from soot particles, which are approximated per unit volume of cloud. Color is achieved using Plank's formula. Finally, refraction is calculated using Ciddor's equation. The test scenes used in the paper took 40 to 80 seconds to render.

Much like ray tracing was considered too computationally expensive to perform in real time for decades, advancements in hardware technology has led to fluid simulations being computationally feasible as well. An overview of the performance of three different rendering techniques was demonstrated in a thesis by Y. He[27]. The three methods consisted of: a fluid simulation, a coefficient estimation, and deterministic ray tracing, and each was tested with different solver algorithms and at different grid resolutions. The

thesis shows that the rendering process can be quite heavy, with frame times around the 1ms mark for a single flame at base resolution. The results can be consulted in the table 2.1. Considering that a forest fire will contain multiple fires, as well as many other objects to render, it is unlikely that the final scene would fall under the 11ms minimum for VR.

Module	Block	Time(ms) Under Resolution		
		Half	Base	Double
Fluid Simulation	MacCormack Advection	<1	4±1	>1000
	Pressure Solver (34 iterations)	1±0.2	5±1	50±30
Coefficient Estimation	Fire Color Integral	<1	2±1	60±10
	Local Multiple Scattering Fire (30 iterations)	<1	1±0.1	10±5
	Global Distant Light Propagation	1±0.1	5±0.2	60±10
	Distant Light Local Multiple Scattering (10 iterations)	<1	1±1	4±2
Deterministic Ray Tracing	Rendering (150 samples per pixel)	<1	2±1	143±8

Table 2.1: Performance table showing the different rendering techniques described by Y. He (Sourced from [27])

Fuller et al.[16] demonstrated that it is not necessary to perform complex fluid simulations at all to achieve good looking results with volumetric rendering of fire. The method described in this paper consists of using a base texture to define the color, intensity, and shape of the fire, and then generating the volume from it. The volume is then subdivided horizontally and deformed using a B-spline, as illustrated in Figure 2.4. The texture mapping is challenging as the entire rendering is performed in a pixel shader, so the texture coordinates need to be transformed as well. To give it the volumetric appearance, the fire is sliced into multiple billboards using the cube slicing algorithm, and displaced with Perlin Noise to generate disturbances to the flame.

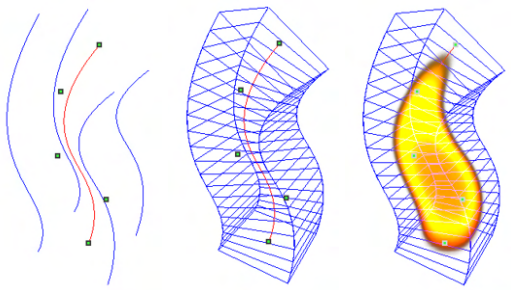


Figure 2.4: A breakdown of the volumetric fire rendering approach from Fuller et al. (Image sourced from [16])

2.4.2 Simpler Ways of Rendering Flames - Particle Systems

Video games have been rendering 3D flames at real-time speeds for decades without the use of fluid solvers. Because Unreal Engine is a general game engine first and foremost, it will be beneficial to look at simpler approaches of rendering flames and smoke. Fluid simulations are great tools, but might be less performant when tested in a large forest

scene. Due to the density of vegetation, the GPU can be very easily bottlenecked, with the CPU being kept busy with other high level engine code.

The simplest form can be through the use of a particle system[30], as shown in Figure 2.5(a). This method allows us to greatly optimize the rendering, by generating billboards (which are flat objects in 3D space that always face the camera) in a given shape. More realistic flames can be rendered by using a higher number of particles, however, since this greatly impacts performance, this method is typically reserved for flames that are closer to the camera. Yet, even in this simple method, if we wish to blend the particle textures for the best possible performance, we must first sort them by distance to the camera, adding another layer of preprocessing and increasing the rendering complexity.

Another common technique for rendering fire is to use animated textures on billboards, illustrated in Figure 2.5(b). This has the benefit of being much lighter than a particle system, but the downside of looking repetitive due to the limited number of frames in the animation. This can be alleviated through the use of deformation curves and noise to visually displace the textures, not too dissimilar to the technique described by Fuller et al.

(a) (b)

Figure 2.5: A demonstration of both methods for rendering fire using particles. 2.5(a) is using a particle system (Image sourced from[30]), while 2.5(b) shows a collection of animated billboards in the game *Half-Life 2*

2.5 Fire Spreading Simulation

As previously mentioned, it is not necessary to perform a simulation of the fire propagation at a macro scale, since we are already provided with the data of an actual forest fire. However there is still the issue of the micro simulation. This would include phenomena such as fire spreading upwards from the base of a tree to the leaves above it, as well as leaves from one tree spreading their embers directly onto the leaves of another

tree. Furthermore, as previously stated, the tool only allows for scrubbing between 10 minute intervals, therefore micro simulations will be needed to fill in the gaps.

It is also worth noting that these micro simulations should only occur if the camera is close enough to the fire to properly observe it. There is no point in processing and rendering simulations of the fire spreading from branch to branch if individual branches cannot be distinguished on the viewer's screen.

Pirk et al.[41] demonstrated an incredibly complete simulation of a tree burning. The method in the paper allows the simulation of heat transfer, temperature changes, and even weakening (and eventual breakage) of individual branches. This was accomplished by representing the tree as a system of connected particles, which store physical and biological attributes, embedded in a physics simulator. The fire itself is treated as a temperature field evolving in a gaseous fluid, represented by a grid, and simulated through fluid dynamics.

Another paper by Hädrich et al.[26] discusses an extensive forest fire simulation tool. The tool simulates, in realtime, the propagation of flames due to numerous factors, such as tree geometry, terrain (downward slopes tend to propagate fires less than upward ones), wind, and different vegetation materials. The simulation of fires is done through a grid-based fluid solver, to allow to transfer heat from the environment to other plants (which in turn heat up the environment, creating a feedback loop). This paper stands out from the fact that the smoke can also generate condensation. This can lead to flammagenitus clouds, phenomena which can cause rain or thunderstorms, as depicted in Figure 2.6.

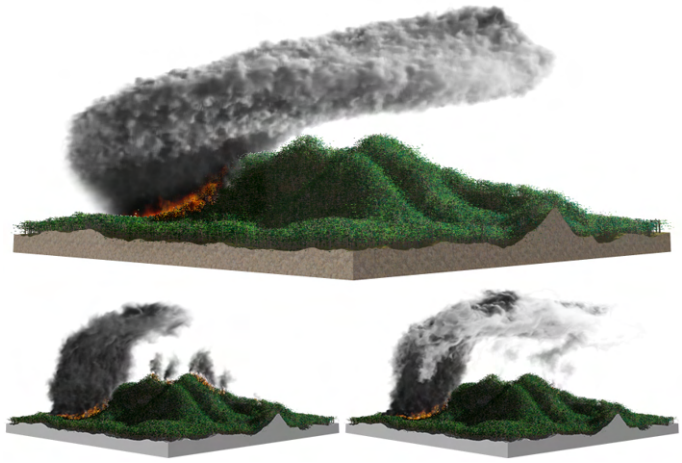


Figure 2.6: A demonstration of the generation of flammagenitus clouds (and their effect on the environment) in the simulation by Hädrich et al. (Image sourced from [26])

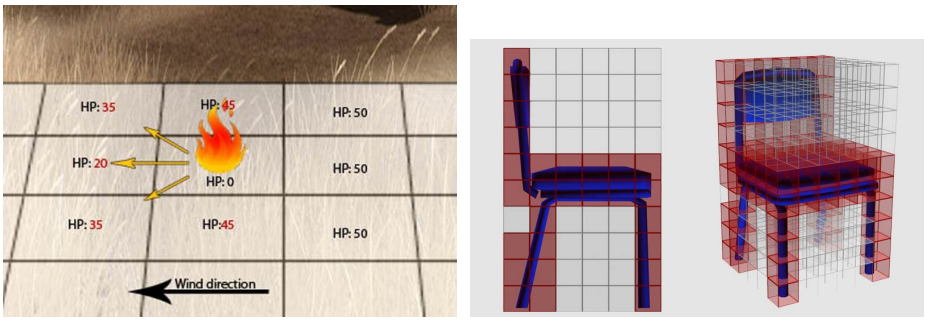
2.6 Games

Once again, fluid simulations have come up as a technique for solving micro-scale simulation of fires, but video games have been doing the same, in a convincing manner,

for decades. Therefore, it is beneficial to look at some concrete examples regarding how some games approach the problem in a simpler, but convincing form.

2.6.1 Far Cry

Far Cry is an Open World shooter series by Ubisoft, best known for its realistic depiction of an incredibly lush forest which players may burn during combat. J. Lévesque, one of the engineers behind the game's engine, described the implementation of the fire mechanic in a blog post[31]. In essence, the world is divided into an equally spaced two-dimensional grid, while objects are divided into a three-dimensional grid, as illustrated in Figure 2.7(b). These grids are generated dynamically when a fire happens. Each grid has a health value, and when its health reaches zero, it ignites. When ablaze, it will damage surrounding grids over time. A cell will also deal more damage in the direction of the game's wind, using dot product to interpolate the directional damage. This is shown in Figure 2.7(a). Different materials are simply just grid values with larger health points.



(a) (b)

Figure 2.7: A visual representation of the fire spreading in Far Cry. 2.7(a) demonstrates the cell health and fire spreading, while 2.7(b) shows how a three-dimensional grid is constructed for objects (Images sourced from [31])

The effect also uses three optimisations:

1. Fire particle emitters are moved to align with the player's view, and the density of emitters is also increased the closer the player is to the fire. When there is a lack of memory, emitters are reduced but the particle sizes increase to compensate.

2. Cells are combined (and separated as needed) into the smallest possible box that encloses them, which allows engine events to be sent per box instead of per cell.

3. Each fire contains a "spreading value", which it reduces over time. The spreading value is passed onto adjacent cells when they burn, to contain the fires into small areas.

In terms of rendering, through observation, it is simple to deduce that the fire in *Far Cry* is composed of animated billboards, and some smoke billboards. Figure 2.8(a) shows

that particles can become wider and taller depending on both the size of the fire, as well how close the player is to them (as previously outlined). The flame billboards will also tilt based on the direction of the wind. After being burned, objects simply disappear, are swapped out for skeletons versions (trees, for instance, do not disappear but rather have their branches emptied of leaves, shown in Figure 2.8(b)), or have their textures darkened to look charred.

(a) (b)

Figure 2.8: A demonstration of fire in the *Far Cry* series. Figure 2.8(a) shows a burning house in *Far Cry 3*, while Figure 2.8(b) shows a burning tree being de-leafed in *Far Cry 4*

2.6.2 Minecraft

Minecraft is a sandbox open world game, developed by Mojang. *Minecraft*'s world is procedurally generated and composed entirely of blocks. Unlike *Far Cry*, the actual implementation of the fire mechanics are not publicly available, but have been observed and documented by the game's community[13]. Every block in *Minecraft* has physical properties, such as flammability and health. Fire spreads to adjacent flammable blocks at a distance of one block sideways (including diagonals), one block downwards, and up to four blocks upwards. The probability of a block catching fire and the duration it burns for is dependent on the material's properties.

When a block is burning in *Minecraft*, its fire is rendered based on which sides have caught fire. Top fires are rendered as an animated cube, lateral fires are rendered as flat, animated textures and bottom fires are rendered as two animated textures facing each other. The fire effect emits very small pixel particles (to represent smoke) that travel upwards a short distance and disappear. All of this is visually demonstrated in Figure 2.9. Once an item has been burned, it simply disappears. *Minecraft* does not employ any material swapping or color darkening on burned materials.

2.6.3 Teardown

Teardown is a sandbox game developed by Tuxedo Labs. *Teardown* stands out due to the fact that its world is composed entirely of voxels (which are not axis aligned), and fully

Figure 2.9: A screenshot of a burning wooden house in Minecraft

ray traced. Unfortunately, similarly to *Minecraft*, the developers have not openly discussed the implementation details, but it can be extrapolated both from developer blog posts and community observations[15]. Each voxel contains physical properties, and are grouped by volumes. Fire, shown in Figure 2.10(a), spreads similarly to *Minecraft* but with the added bonus of also producing volumetric smoke, shown in Figure 2.10(b), while simultaneously heavily affecting structures. *Teardown*, unlike *Minecraft*, simulates flexible joints and block physics, resulting in structures collapsing in their entirety if they are left floating. Similarly to *Minecraft*, due to the voxel approach to the game's engine, fires simply remove voxels from the game world, as opposed to leaving charred remains.

(a) (b)

Figure 2.10: Screenshots of fire (Figure 2.10(a)) and smoke (Figure 2.10(b)) in Teardown.

Teardown, despite all its graphical fidelity (albeit in a stylized look), does not run very well outside of enthusiast-level graphics cards, showing that the described techniques might not be ready for use in a VR environment.

2.7 Spatial Partitioning

Because the project will involve development of a micro-scale fire simulation over a potentially large area, it is imperative to look at ways to perform complex searches fast and efficiently. This will require some sort of spatial partitioning algorithm, which allows

for the culling of objects to reduce the temporal complexity of searches. Typically, space partitioning is accomplished through the use of a tree data structure.

2.7.1 Two-Dimensional

Perhaps one of the easiest approaches to partitioning a space in two dimensions is to break it into an equally spaced grid, this can be simply represented as a two-dimensional array where each cell points to a list. Finding objects in a particular area is as simple as finding which cells intersect with the search shape, and then performing an intersection test on the objects stored in each cell.

This solution presents some issues. Firstly, if objects are concentrated in a given region, the empty cells will needlessly occupy memory, as illustrated in Figure 2.11. The second problem is that this solution might be a bit tricky in the scenario where an object intersects two or more cells, as you will be required to duplicate the object so that it can be found in either cell. Finally, if the object is sufficiently large that it occupies a lot of cells, this means even more duplicated data which needs to be filtered out during the search process.

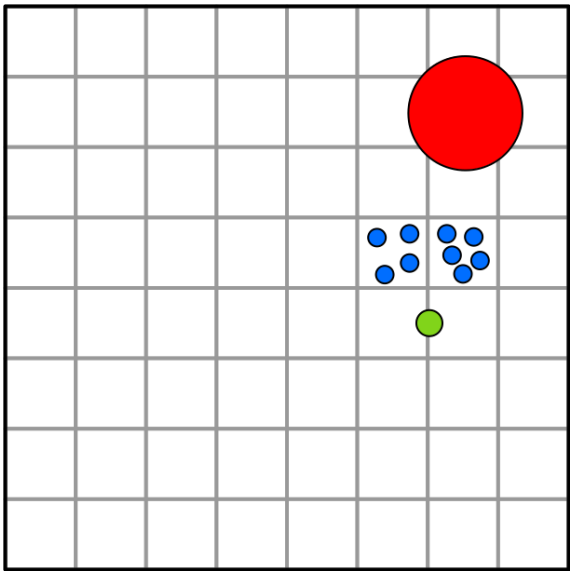
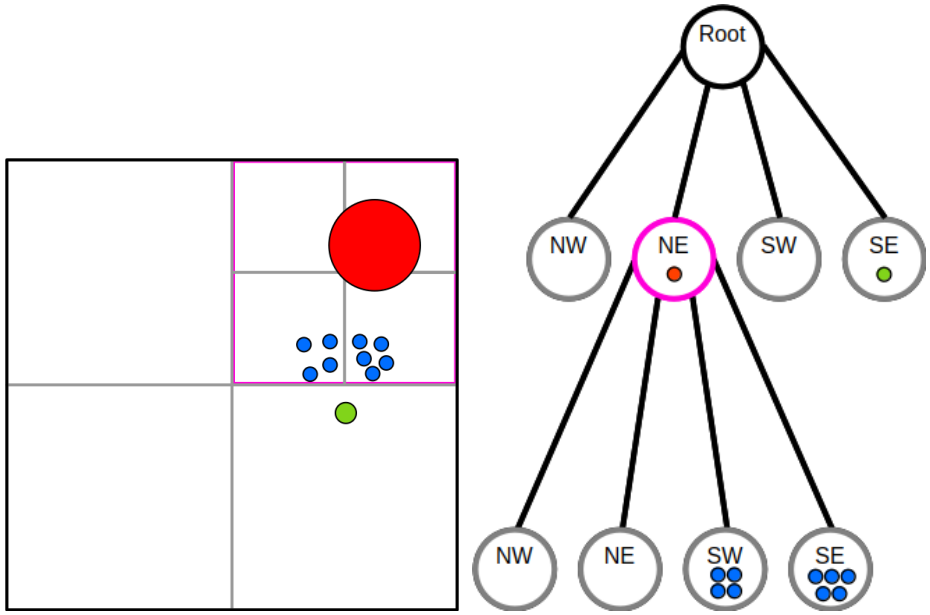


Figure 2.11: An example scene, showcasing the potential problems with the naïve approach to partitioning the space as a grid. The objects only really occupy the top right quadrant of the scene, so a large amount of cells are left empty. The green circle is on the border between two cells, while the large red circle occupies 9 different cells.

A better approach to this idea of dividing the map into equally spaced grids is to instead divide it only when necessary. One could divide the space equally into four sections, and then subdivide those sections again as more granularity is needed. This granularity can be as simple as having a constant "capacity" value that causes the subdivision to occur once the number of objects in the quadrant surpasses said capacity. During the subdivision process, the objects are moved into the children nodes nodes only if said object can be fully encompassed by a given child quadrant. If the object will not fit into any child, then

it is kept in the parent, even if this will result in the quadrant being over capacity. This algorithm is known as a quadtree, and was proposed in a paper by R. Finkel and J.L. Bentley in 1974[44].



(a) (b)

Figure 2.12: The example scene from Figure 2.11, now using a quadtree. The tree on Figure 2.12(b) represents the scene in Figure 2.12(a). A capacity value of 5 was chosen for this particular quadtree.

The squares that make up the space of a quadtree are known as an Axis Aligned Bounding Box (AABB). AABB's are defined by a minimum point, which is typically the corner coordinate of the square with the smallest value on X and Y, and a maximum point with the largest X and Y value. The minimum and maximum coordinate point are always diagonally opposite to each other in a square. The square do not have any rotation to them, hence they are always aligned to the Cartesian coordinate axis.

Searches, insertions, and deletions in quadtrees have a complexity of $O(\log n)$ in both the average and worst case. Deletions, however, can be reduced to $O(1)$ by simply storing a pointer to the node which the element is stored in. Once removed from a grid, the implementation can choose to remove all children nodes from the quadtree if they are all found to be empty. This can be done in order to reduce the memory footprint of the data structure, and can even be performed in a deferred step to reduce the computational load from a deletion.

Quadtrees do not need to be strictly implemented as a perfectly even subdivision of the space. Because performing checks on axis aligned bounding boxes is relatively cheap, a scene can instead be subdivided into more convenient rectangles. For instance, in Figure 2.12(a), the northwest and southwest quadrants could be combined into a rectangle. The northeast-most quadrant could store the largest circle instead of its parent node if the

subdivision was performed at the minimum coordinate of the bounding box encompassing the circle.

One of the pitfalls of a quadtree is that if the scene is highly dynamic in nature, the quadtree requires a lot of removal, insertions, and subdivisions in a short amount of time. A better approach to this would be to loosen the concept of a boundary, focusing more on the minimum bounding rectangle that surrounds a set of objects rather than the strict bounding box of the subdivision of the space. This means that as the objects in the space move, so does the boundary that encompasses them. This variant is known as a loose quadtree [34].

2.7.2 Three-Dimensional

While a quadtree is strictly designed for a two-dimensional space, it is relatively simple to extrapolate it into three dimensions. Instead of subdividing it into four quadrants, it can be subdivided into eight octants. This variant is known as an Octree[35], and is demonstrated visually in Figure 2.13.

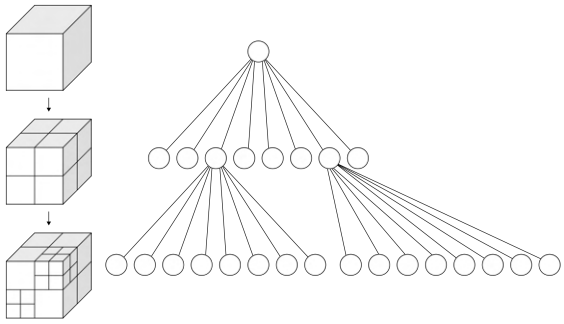


Figure 2.13: An example of an octree, showing its spacial representation on the left, and its subsequent tree on the right. Image sourced from [50].

When it comes to partitioning fluids, octrees can be used, but a more common alternative is the k-d tree[5]. These trees are preferable for this scenario because they are specifically designed to facilitate nearest-neighbor searches, which is the most common type query used in fluid dynamics. K-d trees are, put simply, an octree where instead of splitting the area into an octant it is only split in half by a single hyperplane. K-d trees therefore provide an efficient way to partition and query clusters of points, however they do not provide an efficient way to query arbitrary shapes. k-d trees also struggle with dynamic data as it could easily require rebuilding the entire tree when an object is moved.

2.8 Unreal Engine

While the wildfire visualisation tool was initially written in Unreal Engine 4 (UE4), it will be upgraded to Unreal Engine 5 (UE5) to take advantage of the new features, such as

a much simpler-to-use World Composition workflow which removes a few steps to load map data onto the engine.

Unreal Engine's development can be done in two ways: using a visual scripting language called Blueprint, or through C++. The latter tends to be more flexible, allowing the user full control of the engine, as well as more performant code. UE5, unlike most modern game engines, opts for an object oriented approach over an entity component system.

It is worth pointing out that Unreal Engine uses a left-handed coordinate system, where the positive Z axis points upwards. The easiest way to visualize this is by looking at a paper map: north points towards the negative Y axis, east towards the positive X axis, and positive Z axis being the height of terrain extruding towards the eyes of the reader.

2.8.1 World Partition

Worlds in unreal can be created in two different ways: either by hand or from Heightmaps. In order to create levels from heightmaps, however, the images must go through a pre-processing step where they are split into chunks using an algorithm explained in the development documents[18].

When it came to large levels, UE4 provided a system called "Level Streaming". In essence, the world was split into chunks, which were loaded as the player approached the edges of the play area. The developer would create one "persistent" level which would always be loaded, and this one would decide when to stream a chunk of the level in or out[20].

Later on, Epic Games introduced World Composition as an alternative[23]. It was created to combat one of the main problems of the level streaming system, which was that the level files were locked while they were being worked on by a developer, complicating collaboration. It also allowed worlds to be created which were not limited by the hard-coded size limit in Unreal Engine worlds.

UE5 then proceeded to deprecate the level streaming and world composition systems in favor of a new "World Partition" system[24]. This new system provided better memory management, smaller file sizes, and support for multiplayer (which was missing on World Composition). World partition also introduced a new slew of features, including data layers (allowing actors to be grouped, which could then be toggled as necessary), and a editor that allowed for greater control over how regions are loaded. Porting over a world from the level streaming system is relatively seamless, but converting projects from the world composition system is more complicated.

2.8.2 Particle Systems

Unreal Engine provides two different tools for creating particles systems: Cascade and Niagara. The former has been largely superseded by by the latter in UE5, which provides better control over the emitters, GPU processing support, and even support for

fluid simulations. Although Epic Games advises against the use of fluid simulations due to their heavy nature, suggesting instead to bake the results of the simulations[38]. This is a potentially viable approach, given that the scene will not be dynamically altered by the user.

UE5's Niagra particles are composed of three layers. The first is a system, which contains all the elements (and subsequently is what is spawned in the level). Systems contain emitters, which create the individual particles, and each emitter contain modules which tell the particles how to behave. The tool provides a plethora of features, allowing to modify multiple aspects of a particle while it lives. This includes physics (velocity, momentum, friction, gravity), color, size, collisions, and even event handlers.

2.8.3 Foliage

In order to make a forest scene, one of the most important things that will be needed is foliage. The Unreal Engine provides a Foliage Tool, which allows the user to paint foliage into the world. This, in essence, is a tool for placing objects and models in a world with given random properties, using a given density.

In UE5, the Procedural Foliage Tool was introduced with the aim of significantly speeding up the placing of foliage into the world[21]. However, because this tool is experimental, it must be manually enabled by the user.

One of the features of the procedural foliage tool is that, rather than manually placing objects in a world, it allows the developer to procedurally spawn foliage into the world given a set of parameters and constraints. This is done by simulating the creation of foliage over time with an initial seed density by giving them lifespans and letting them grow, and spread, and die as they would naturally. This results in a more realistic clustering of foliage, and even goes as far as making foliage be less likely to grow in areas where they are not exposed properly to sunlight. The result from the procedural foliage placement is shown visually in Figure 2.14.

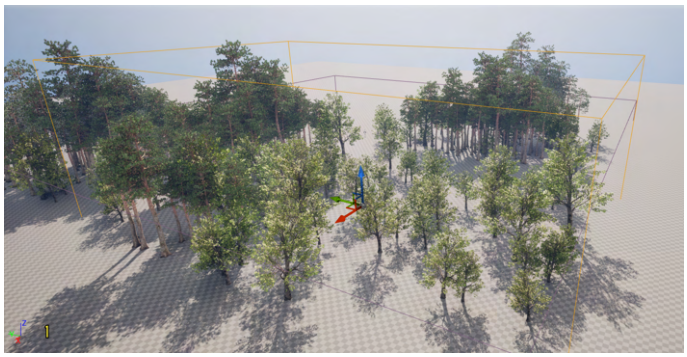


Figure 2.14: A demonstration of Unreal Engine's procedural foliage. Image was sourced from [21].

2.8.4 Assets

In order to have a realistic looking scene, we will require a large number of assets. Creating these 3D models, textures, and animations can take a significant amount of time. As a result, Epic Games provides the Unreal Asset store, where developers can purchase high-quality assets for free and commercial use. The store not only provides assets, but also plugins, code, and even entirely finished games. This can be used to significantly speed up and decrease the cost of development.

Another important tool provided by Unreal Engine is its material system. While shaders can be written from the ground up using High Level Shading Language (HLSL), UE5 provides a node based shading system which support many complex features such as Physically Based Rendering (PBR), layers, and animations through mathematical functions.

The last aspect of UE5 which is relevant to our work is the LOD system. Unreal Engine's LOD system is basic, allowing for the swapping of models at runtime based on their relative size to the player's screen. A common technique for rendering lush forests in video games is to swap them out for billboards at long distances. Unreal does not provide this functionality as part of its LOD engine directly, but rather through an alternative system called Impostors[45].

3

Proposed Solution

Given our analysis of fire simulation and rendering in Chapter 2, as well as our research into practical implementations in professional video games, we may now conceptualize a solution for our initial problem, defined in 1.3. This Chapter presents our design of a program that can meet the performance requirements to simulate a real fire in VR.

3.1 Micro Simulation

As it stands right now, the fire visualisation tool works as such:

1. A server is launched, which contains the data points from a fire that happened, stored in GIS format.
2. The VR game is launched, and it connects to the server.
3. The game requests fire front data from the server. This happens in 10 minute intervals.
4. The data points from the server are used to generate a polygon which is then projected onto the map.
5. Said polygon is used to modify foliage and repaint the terrain.
6. Finally, a set of particles are drawn on the edges of the polygon to simulate a fire front.

The tool already performs a macro-level simulation of the fire, represented by the polygon of the fire front. The goal is to focus on the micro-level simulation, ie:

1. What happens to the foliage when the fire touches it?
2. How thick is the fire front itself?
3. When does said fire in the front burn out?
4. How can we interpolate the current simulation with its final result?

5. Similarly, how can we reverse or skip time in the simulation?

The proposition is as follows. Consider the entire region which we wish to simulate, partitioned as a 2D grid where each element has a side of length S. A grid element can have one of four different states:

1. **Empty** - This grid position is unoccupied by a grid element, thus it does not take up memory in the program.
2. **Ignited** - This grid position represents a fire, and is attempting to spread to surrounding grid cells. These cells are initialized with a lifetime value L.
3. **Damaged** - This grid position is not on fire, but is currently taking damage from a grid element which is ignited. When initialized, this grid will have a health value H.
4. **Burnt** - This grid position represents a fire that has burnt out, and no longer does anything.

At the start of the program, there will be no grid elements present, meaning all positions of the grid are empty. At some point in time, a single grid element will ignite (representing the start of the fire), and it will want to spread to other grids. Since the grids that surround the grid element are **Empty**, the neighbours must first be generated, so the program will generate as **Damaged** grids in empty positions surrounding the **Ignited** grid element. Over time, the ignited grid attacks the neighbouring grids and depletes their H. Once the H hits zero, this grid will turn into an Ignited grid element, and will repeat the cycle of creating neighbours and damaging them. After enough time has passed, a grid element's L will expire, and it will transform into a **Burnt** grid, which no longer does anything and merely exists to ensure that the grid position will not be occupied again in the future. Figure 3.1 demonstrates this system visually.

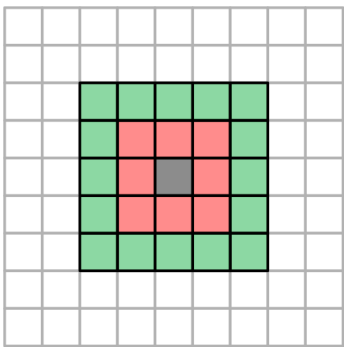


Figure 3.1: An example scene. The fire started at the center and spread evenly in all directions. The green grids represent damaged cells, the red grids represent ignited cells, and the black grids represent burnt cells.

Because the entire terrain is generated from a heightmap (a two dimensional representation of the world), it would make sense to keep this simulation entirely confined within

the worlds of two dimensions. This means that, in order to spread to other elements, grids simply will need to check neighbouring cells on the x and y dimensions.

However, the same does not ring true for foliage, like trees. In this scenario, if a grid element overlaps a piece of foliage, it must also start checking for neighbours in the z axis. Once a grid element no longer overlaps a piece of foliage, it can ignore the z axis once again. Figure 3.2 shows grid elements spreading to foliage.

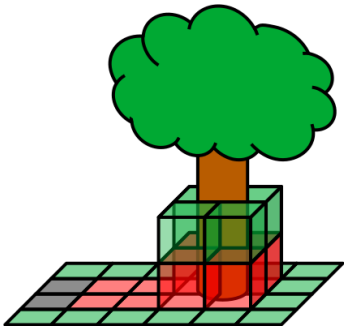


Figure 3.2: An example of the grids spreading to a piece of foliage

3.1.1 Optimizing the Grid

Once the cell spreading system has been implemented, the next step will be to improve its performance. It does not make sense to perform a micro simulation on areas of the map that are not being observed up close by the camera, therefore it would be pertinent to decrease the resolution of the grid based on the distance from the camera.

To keep the system simple, rather than having the grid itself change, it was chosen to have the grid elements themselves shrink and grow with one another. Once the camera is sufficiently distant from a cell, for distance D, the cell is now allowed to grow up to a maximum of $2S$ of its original size. Similarly, if the camera is $2D$ away from the cell, it can occupy $3S$ grid ppositions. However, if the camera's distance returns to D, then the $3S$ cell must split itself back into a set of $2S$ and S sized cells. Figure 3.3 shows the scaling of grid elements based on their distance from the camera.

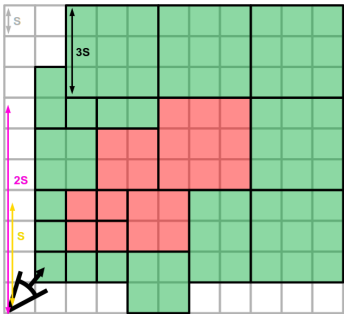


Figure 3.3: An example of the cell sizes changing based on their distance to the observer on the bottom left

When a grid element of size nS ignites, the neighbours it generates should also have a size nS, otherwise large cells would end up creating thousands of small cells, at a rate of $9 + 4 \times n$. These neighbours would have their H value scale based on their distance size, and similarly ignited cells would scale their initial L value too.

When it comes to combining the data from two separate cells, it can be as simple as summing their H or L value, and similarly dividing them in the case of a cell division. Ultimately, the point of the system is that it can be convincing enough at a glance. It does not need to hold up to scrutiny every time the user observes it, as long as the simulation converges with the data of the actual fire.

3.1.2 Connecting the Simulation to the Data

While there will not be enough time to actually integrate the cell spreading system into the final fire visualisation tool, it is important to have a plan for how such a thing would be accomplished. Therefore, this section proposes a method for ensuring that the simulation adheres to the fire front data obtained from the server.

It does not make much sense to simulate a fire if we have data that we wish the fire to match. However, the truth is that we already have everything we need in order to do that. Because the fire front is represented as a polygon, and we can poll the server for the polygon which represents the fire at a future point in time, we can simply scale the L and H value of the cells so that they converge with the target polygon. Not only that, knowing the current position of the cell, we can project the shortest line from it to the nearest edge of the polygon, and we can use that to help guide the fire to ensure its spread converges in a specific direction. An example of a collection of grid elements spreading to converge with the fire front polygon edge is shown in Figure 3.4.

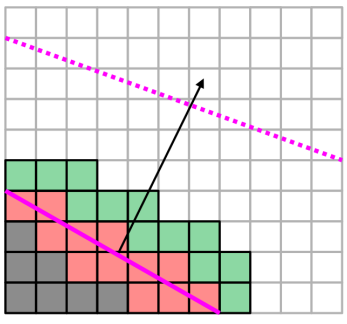


Figure 3.4: The pink line represents the current fire front, while the dotted line represents the front after T time. The black arrow shows the direction which the grid elements will converge towards.

There is an issue with this approach: the edge of the polygon, which represents the fire front, may converge with another edge, or split into two or more edges. One solution here is to spread fire to grid elements according to which edge is closest. However, to

avoid querying for closest polygon edge for each grid element, we can store the polygon data in a quadtree as well, culling the required number of edges that need to be checked.

One benefit of connecting the simulation to the data will be that the burnt grids will no longer be necessary. In the scenario where a fire is being simulated freely, the burnt grids are important to ensure that a grid element will not attempt to spread to an area which has already been completely burned out, so these grids will always have to be stored in memory. In the case where we are tying the simulation to already existing data, we don't need to worry about this phenomena, as there should be enough systems in place to prevent the ignited grids to spread backwards.

3.2 Fire Rendering and Terrain Damage

For rendering the fire front, the simplest approach is to use a billboard-based particle system. A full-on fluid simulation is unnecessarily computationally heavy, especially considering that the scene will likely be GPU-bound as opposed to CPU-bound once the fire has grown sufficiently large.

One of the benefits of using cell combination technique described in section 3.1.1 is that we can use the size of the cell itself to define how large a fire particle should be. It is unnecessary to have a lot of really small particles at distances where they cannot be told apart, so instead we join these into a single large particle of the same size, as shown in Figure 3.5.

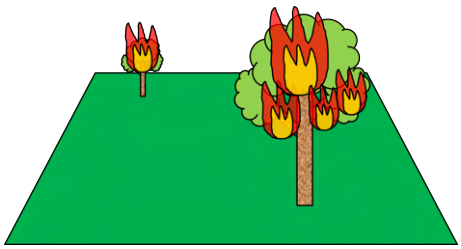


Figure 3.5: A demonstration of scaling particles by distance. Since one might not even be able to see the different branches in a tree that is far away, it would be more performant to draw one large particle than to draw smaller particles on each individual branch.

There exists other phenomena in fire, such as haze, sparks, and smoke, which would be interesting to also render through particles. However, these should be low priority as they can degrade performance significantly, as well as reduce visibility of the scene (which complicates the use of the simulation as an educational tool).

When it comes to the assets themselves, such as the textures of the terrain, the models of the foliage, and the particle effects themselves, these can be acquired from the Unreal Asset Store. Asset development takes a significant amount of time, and is not the focus of this dissertation.

Lastly, it is also worth discussing the ways which we can simulate the damage which the fire causes to the scene. Unreal Engine's materials can be programmed to change dynamically through the use of a shader. Unreal provides a node based programming language for developing materials, combined with material layers, one could create a texture which seamlessly blends between states. For instance, the ground could have a normal grass layer, a burnt layer, and a layer that adds glowing sections to the grass in order to simulate embers.

Another interesting thing worth investigating is Unreal Engine's ability to place decals on surfaces, which are textures which are projected onto surfaces. These are commonly used in video games for bullet holes, blood splatters, and in our scenario, scorches from explosions or fires. These can be used to blend the transition between the fire front and the unburned terrain.

Unreal also supports the ability to switch models at runtime, so a healthy bit of foliage can be replaced with a dead skeletal version once it has been completely consumed. In the case of leaves, grass, and similar fauna, it is sufficient to simply remove these from the world (or at least not render them). This could be done by fading it out, which while not entirely realistic, would look a bit more convincing than simply making it invisible instantly.

3.3 Evaluating the Solution

Due to Unreal Engine's heavy system requirements, as well as the performance requirements of VR, testing will require a relatively performant computer. In a worst-case scenario, it is expected that the software run on consumer hardware, such as 64-Bit Windows 10, with 16GB of Random Access Memory (RAM), and on the GeForce RTX 2060 6GB, one of the most popular consumer graphics cards[9]. The VR head mounted display will be a HTC Vive Pro, with a 615 Pixels Per Inch (PPI) resolution, using the included controllers which have 24 sensors.

Ideally, the project would be tested in a scene that represents the area of the Mação region, but it can be relegated to a smaller test scene if there are problems regarding integration with the SI-MORENA project. Even if it is impossible to test using real data from a fire, the simulation itself can be tested at many different resolutions and grid sizes.

Visual improvements will lead to an impact in the performance of the tool. Therefore, it is important to profile the implementations as they are completed. Unreal Engine provides a pretty extensive GPU profiler tool called Unreal Insights, which can be used to aid in this step. An example of the profiler is shown in Figure 3.6.

Unreal's profiler can function in multiple ways. The first way is a simple time for each step in the creation of a scene: game code running on the CPU, and rendering code running on the GPU. Both return values in milliseconds, with our target being to keep the scene total under 11.11ms (for our 90 FPS target). The profiler lets the user collect data over a single frame, or during a set of frames, and it breaks down everything during said

time. Of course, this profiling will impact performance while it is occurring, but not to the point where the data it provides becomes useless. Nevertheless, the profiler will not be running on the final application, so the concern is moot.

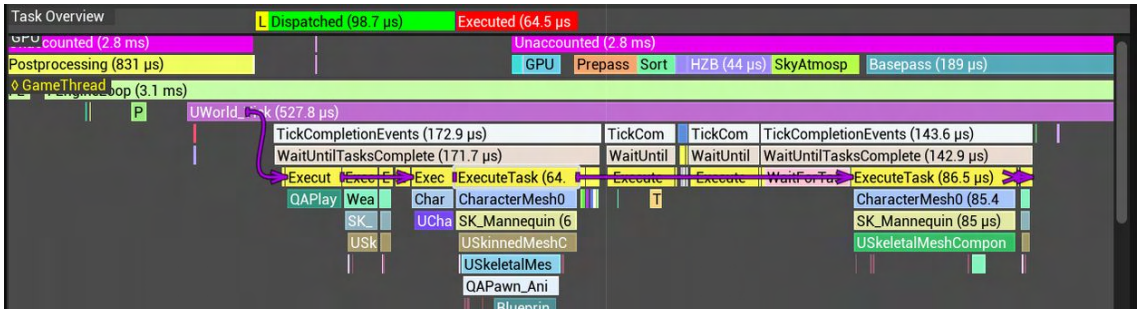


Figure 3.6: An scene breakdown from Unreal Insights, which shows the time it took for the CPU to process different actors in a single game tick. Image taken from [22]

The rendering of the scene involves multiple steps on the GPU, from shadow passes, visibility culling, screenspace effects, and reflection mapping. The profiler is able to break down the time taken in each step, and even allows to show which objects are requesting which step of the render pipeline, as well as graphic memory usage. Profiling the CPU is not much different using Unreal's profiler. It breaks down function call times in ms, shows which objects called which functions, when threads were paused, and other useful insights.

It is assumed that the rendering of the forest scene in the already developed tool is currently optimal, but further improvements can be performed to improve it even further, such as the entire removal of small objects (such as bushes) at far distances, the use of billboards for far away trees, and texture LODs.

As the project is being developed, there will come points where a theoretical study of a set of algorithms will not yield sufficient information to make a decision on which to implement, so the Unreal profiler will serve to assist in choosing the best performing ones, as well as make decisions on which part of the program will need to be optimisations. The tool will also serve to show whether or not the final solution will be beneficial, as well as figure out future optimisations or potential bottlenecks.

4

Implementation

Following the design of the program outlined in Chapter 3, we outline the development process, including challenges faced and decisions made to address these challenges at each step.

The repository containing all the implemented work can be found at https://gitlab.com/si-morena/vr-ar/lourencothesis.

4.1 Initial Work

In order to integrate the micro simulation, we first setup the existing fire visualisation tool. This posed an immediate challenge due to issues with dependencies.

4.1.1 Setting up the Previous Work

The MACFIRE server application runs inside a Docker container, and is composed of Flask (a Python web framework) alongside a PostGIS and PGAdmin databases to store GIS data and user account data, respectively. The use of a web application allows offloading of the fire front calculations to a separate system, and the use of Docker allows for easier distribution.

However, despite the use of Docker, the existing build script did not initially compile due to incorrect paths for some system files. This likely arose from changes to the project structure as it developed. The fix was applied and published along with some comments on the README regarding how to get the container compiling on Linux.

Following these corrections to the Dockerfile, both the container and the server built and ran correctly. While no issues arose from compiling the Unreal 4.2 project, the state of the fire visualisation tool itself presented the following issues:

1. The tool interfaced with the server, but did not appear to reflect changes from the data.

2. The tool did not allow for the manipulation of time in the simulation.

3. Assets, such as models and textures, were missing. This was due to the fact that the asset packs which the project used originally were once free, but have since become paid. There were other assets which were referenced in the code but not published to the Git repository itself, likely due to file sizes.

VR functionality, such as headset and controller tracking, as well as player movement, worked without much issue, although it did require the installation of multiple programs such as Steam and HTC VIVEPORT. Many of these problems were ignored because there was another student who was tasked with integrating all the previous work from other students into one project, and it was assumed that many of these troubles would be corrected in time.

4.1.2 Updating Unreal Engine

It was requested that the Unreal Engine 4.2 project be upgraded to Unreal 5.1, which was the latest version at the time, in order to capitalize on the potential performance improvements of 5.0's World Partition.

Unreal Engine provides a "Project Upgrade Wizard", which attempts to convert the project files to the newest format, resolve potential conflicts, and warn of deprecations. Unfortunately, the Upgrade Wizard was not an option for the project, due to the use of plugins without an available corresponding 5.0 version.

To solve this, the plugins had to be manually recompiled for Unreal 5.0, which was possible due to the fact that their source code was included in their store pages. Most of the plugins were relatively easy to convert, however one plugin named KantanCharts was using deprecated functions, and these had to be replaced to get the plugin to compile successfully.

Once the plugins which were not upgraded by their authors had been repaired, they were placed manually in the project's repository, and the project's build system was tweaked to fetch them locally as opposed from the asset store. Once this was done, the Upgrade Wizard succeeded in converting the project.

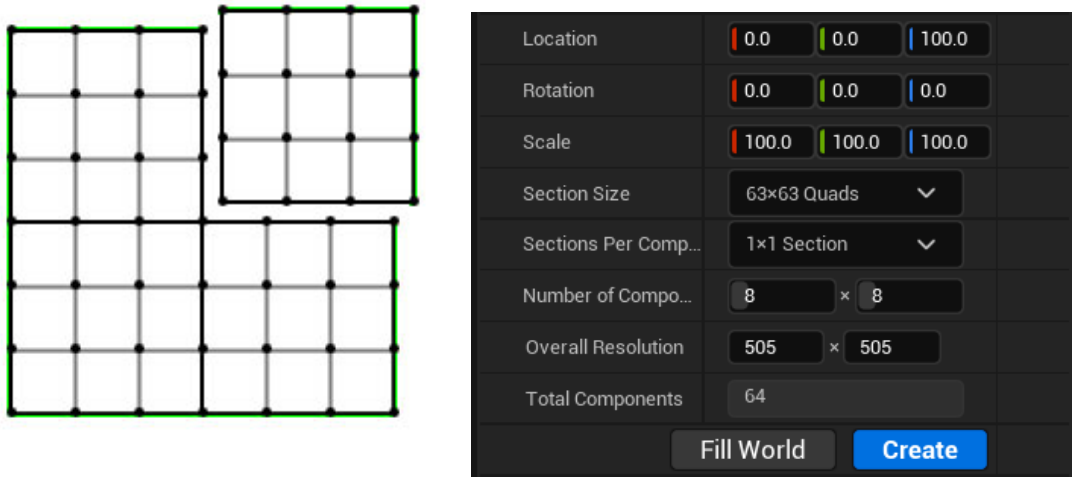
The upgraded project was tested, and it was found to be in a similar state to how it was before. Due to the fact that the student performing the merging had still not finished their work, it was decided that the project would be left as is, since it was not known whether it was fully functional or not. The upgraded project was published to git, and it was decided that instead of losing further time waiting for other student's work, that this dissertation would be developed in a fresh repository, and its integration into the final fire visualisation tool could be done at a later date.

4.2 Terrain Tool

Starting from scratch, one of the first things that is needed is to import the terrain of the area where the fire occurred, which in this case is the Mação region in Santarém. A

DEM of the region was provided in image format (as a heightmap) so that it could be imported into Unreal Engine.

In Unreal Engine, a landscape is divided into multiple 'Landscape Components', which are subdivided first into sections, and then further subdivided into quads. Figure 4.1(a) shows an example of how this division is done. More sections and quads increases the resolution of the landscape, while more landscape components results in larger landscapes. Epic Games features a table in their landscape documentation which recommends a combination of input values to use[18], to be filled out in the form shown in Figure 4.1(b). UE5 also requires that large landscapes be split into multiple tiles, with each individual image following a name pattern.



(a) (b)

Figure 4.1: Figure 4.1(a) shows a landscape made of four components, each one split into 3 sections. Figure 4.1(b) shows the import settings in UE5.

When importing heightmaps, UE5 assumes that one pixel in the heightmap image is equivalent to 1 meter, and so in order to properly import the landscape, we must know the scale of each pixel[19]. QGIS allows us to import the heightmap and view information about what its pixels represent. However, upon importing the heightmap, it stated that the pixel size was 3.91×10^{-5} degrees. This meant that a pre-processing step would be required to convert the DEM to a unit which could be used by UE5.

It was decided that a tool would be written to perform the reprojection of the DEM, rescale its pixels to match UE5's assumptions, tile, apply the name pattern, and finally to output the recommended values to use when importing into UE5. Python was chosen as the programming language for the tool for the following reasons:

1. It has support for the Geospatial Data Abstraction Library (GDAL), which would simplify the reprojection step.

2. The Python interpreter that executes the scripts is cross-platform.

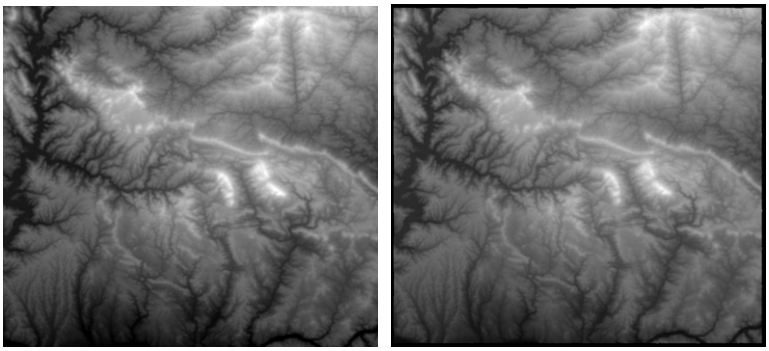
3. Python is a relatively easy language to prototype in.

4. The execution speed of the program is irrelevant as it is an offline tool.

The reason that the original DEM uses degrees is because it is represented using the geodetic system EPSG:4326. Because EPSG:4326 is a two-dimensional ellipsoidal representation of the world, it needs to be converted into a geodetic system which represents the world in Cartesian coordinates. This unfortunately is not something that can be easily automated, at least not accurately, because the curvature of the earth is different based on both latitude and longitude. Thus, a specific geodetic system must be selected, of which EPSG:32629 was the best choice since it is specifically meant to be used within 12°W and 6°W of the northern hemisphere, where the Mação region is located.

GDAL also provides functions for scaling the heightmap, however because the heightmap is made of discrete pixels, this process will be destructive when scaling down or will require extrapolation to fill in the gaps when scaling up. Therefore, to accomplish the resizing, an image scaling algorithm must be used. By default, the tool uses nearest neighbour resampling, but it also supports bilinear, cubic, and Lanczos resampling, among other methods. To scale, the user simply provides the desired size of each pixel (in meters) on X and Y as an input value to the program, optionally providing the preferred scaling algorithm.

For the height, UE5 requires that the absolute minimum of the heightmap have a brightness value of 0, and the absolute maximum to have a brightness value of 65535 (meaning the heightmap should have 16-bit precision). Hence, it is also necessary to get the minimum and maximum brightness, so that all pixels can be scaled to the required interval. Figure 4.2(a) shows the input DEM, and Figure 4.2(b) shows the outputted heightmap which has been slightly rotated and had its brightness values normalized.



(a) (b)

Figure 4.2: Figure 4.2(a) is the raw DEM, while Figure 4.2(b) is the reprojected and rescaled result from the Python tool (before tiling).

Finally, the program tiles the output image and names each one with the pattern `tile_xA_yB.png`, where `A` and `B` represent the tile number in that specific axis. Because

every tile must have the same size, any pixels of the heightmap that are outside of the bounds of the original DEM are given a brightness of zero.

The tool was added to the same Git repository [as this project], and includes a README that explains how to install the dependencies, use the tool,background knowledge about geodetic systems, and how to switch the one used by the script. An example of the tool being used to generate a heightmap is shown in Figure 4.3.

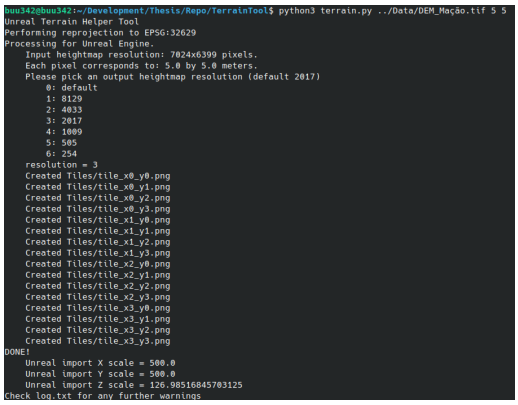


Figure 4.3: A demonstration of the Python tool being used to generate a heightmap tile set, as well as the required scale values to import the heightmap into UE5.

The region of Mação represented in the DEM has an area of around 35×32 kilometers, and was initially attempted to import into Unreal with a pixel scaling of 1 meter. Unfortunately, UE5 would crash while importing this heightmap; therefore, a pixel scaling of 5 meters was used instead. While this did not change the region represented in the DEM, it did reduce the resolution of the detail slightly. It could be that, on a system with more RAM, the import with 1 meter scaling would be successful, but the 5 meter scaling was sufficient for our purposes. Figure 4.4 shows the imported terrain, running in UE5.

Figure 4.4: The terrain, imported into UE5 to scale. The highlighted square is a single landscape component.

Finally, with the map successfully imported into UE5, we proceeded to add foliage to burn. Assets were brought over from the previous project, and a simple grass texture was applied to the whole scene. Using the procedural foliage spawner, conifer trees were added to a small area in the center of the map. While it would have been preferable to have the foliage spawner affect the entire map, this would require multiple spawners to avoid running out of memory in UE5. Since creating lots of different spawners would take a lot of time to set up due to the map's size, it was decided to leave this as future work, since a fully decorated map was unnecessary for our tests, and doing so would require extensive analysis of the data to provide an accurate reproduction of the real Mação environment. Figure 4.5 shows the final product.

Figure 4.5: The terrain, now textured and with trees scattered about it.

4.3 Naïve Implementation

Due to the author's familiarity with C++, this language was chosen as the development language for the implementation of the fire spreading. This required the use of Microsoft Visual Studio, which is a Windows only program. Once a Windows environment was set up, work could begin.

During this project's development, subsequent versions of Unreal Engine were released, with the last non-beta version (as of the writing of this dissertation) being 5.3.2. The project was kept up to date with the releases, and is working on the latest stable release of the engine.

In order to keep this text clear, two important distinctions must be clarified. The "grid" refers to an imaginary grid that partitions the world, and the size of said grid is static everywhere. Meanwhile, a "grid element" or "cell" refers to a singular object, aligned to the grid, which occupies one or more positions of said grid. Confusingly, the code listings may refer to a collection of grid elements as "grids", rather than "cells".

First, an actor was created called `GridController`, which is the main class that manages the scene and handles the logic of all grid elements. A constant `GRID_RESOLUTION`, which describes the smallest possible size of the grid in Unreal's map units, was defined with a value of `64` map units, equivalent to 64 cm. The actor was also given two methods required by Unreal: `BeginPlay` which executes when the actor is constructed and `Tick` which runs every engine tick (which is a constant value of time). The actor also contains two important attributes, a list named `BurningGridsList` which contains pointers to all currently burning cells, and `FloorGridsMap` which maps 2D coordinates to cell pointers.

Another class named `GridElem` was created, which represents a single cell of the grid. The definition of the structure is shown in Listing 4.1.

```
typedef struct {
    int hp; // Health before combusting into fire. If this is zero, the grid is on fire.
    FVector coord;   // The coordinate of the grid
    double lifetime; // How long this fire will burn for
    TArray<GridElem*> neighbours; // List of neighbouring grids
    bool generatedneighbours; // Have the neighbours been generated? (default false)
} GridElem;
```

Listing 4.1: The structure definition for a single grid element

One might wonder why the grid element was written as a struct, rather than as an actor. This was done because the game might need to handle hundreds of different elements, thus it was written as a struct to minimize the memory footprint, which in turn would improve cache performance during iteration.

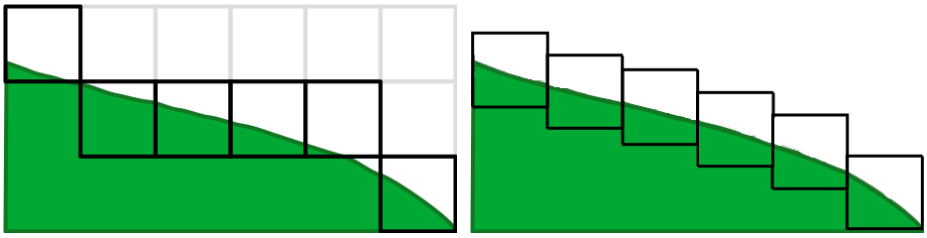
Lastly, an actor called `FireStarter` was created and placed in the world, responsible for starting fires in the area during this initial implementation stage. The actor, much like the `GridController` class, has the two main methods required by Unreal, but it also contains a third method `OnCollide`, which the engine executes when a collision has occurred with another actor in the world. The actor, during initialisation, is given a sphere collider and physics are enabled on it. equently, when the game is executed, the `FireStarter` object will fall to the ground and collide with the Landscape actor, triggering the `OnCollide` method. The `OnCollide` method searches for the first occurrence of a `GridController` actor in the world and executes a method from it called `CreateFireAtXYZ`, which takes the collision coordinate as an argument.

The `CreateFireAtXYZ` method starts by aligning the coordinate so that it is centered with the grid defined by `GRID_RESOLUTION`. This is shown in the Listing 4.2, where a grid element's X and Y values are aligned.

```
FVector pos_aligned = FVector(
    FMath::Floor(pos.x/GRID_RESOLUTION)*GRID_RESOLUTION + GRID_RESOLUTION/2,
    FMath::Floor(pos.y/GRID_RESOLUTION)*GRID_RESOLUTION + GRID_RESOLUTION/2,
    pos.z + GRID_RESOLUTION/2 // No need to align Z, just need to push it up
);
```

Listing 4.2: A sample of code showing how cells are aligned to the grid

The reason that the Z value is not grid aligned is because doing so would lead to potential issues with grid visibility, as demonstrated in Figure 4.6(a). The goal was to have the center of the grid function as the bottom of the fire, such as these could be correctly aligned with the terrain, as per Figure 4.6(b). The landscape is guaranteed to be two-dimensional as it was generated from a heightmap, thereby ensuring grids would not spawn on top of other grids, as you would expect if the map contained caves.



(a) (b)

Figure 4.6: Figure 4.6(a) shows what would happen if the grid elements were aligned to the grid's Z. Many elements would end up spawning inside the terrain as they would fail to spawn in the adjacent position due to not colliding with anything. Figure 4.6(b) shows the preferred result.

After aligning, a `GridElem` struct is created with a `hp` value of zero (since it starts on fire), `generatedneighbours` as `false`, and a `lifetime` value of `GetTime() + FIRE_LIFETIME` (where `FIRE_LIFETIME` is defined as `10`, while `GetTime()` returns the current time of the program). The pointer to this `GridElem` is added to the `GridController`'s `BurningGridsList` and `FloorGridsMap`.

Later, when this implementation is moved over to the fire visualisation tool, the values of `hp` and `lifetime` will need to be flexible and dependent on the progression of the fire front polygon. However, these values were left as static constants to make debugging and performance testing easier.

To perform the logic of the fire advancing and spreading, the grid controller must iterate through the list of burning cells and generate neighbours, attack them, and remove elements from the `BurningGridsList`. The `GridController`'s `Tick` function, which performs this logic, is described in Listing 4.3.

```
for (GridElem* ge : this->BurningGridsList) {
    if (!ge->generatedneighbours)
        this->AttemptSpreadGrid(ge);
    for (GridElem* nge : ge->neighbours)
        this->AttackNeighbour(ge, nge);
    if (ge->hp == 0 && ge->lifetime < GetTime())
        this->BurningGridsList.Remove(ge);
}
```

Listing 4.3: The main `Tick` loop of the `GridController`

The `AttemptSpreadGrid` method is what handles the generation and assignment of a cell's neighbours. It does this by taking the cell's center coordinate, and adding or subtracting `GRID_RESOLUTION` on both the X and Y axis to ensure that each neighbouring cell is tested. First, the `FloorGridsMap` is checked to ensure that a grid element already exists at a given coordinate (the Z value is irrelevant since the hash map uses two-dimensional coordinates to perform the mapping). If an element exists at the coordinate, then it is added to the cell's `neighbours` list.

It is important that, when a cell is added to a grid element's list of neighbours, the assignment must be done in both directions. This allows us to iterate over a cell's neighbours and prevent dangling pointers whenever a cell is removed from the world. During the assignment, the grid element should check whether the element it wishes to insert does not already reside within the list.

In the case where a grid element did not already exist, then the area it wishes to spread to must be tested to ensure that the landscape exists at the given coordinate. An AABB collision check is performed at this coordinate using the grid element's shape as a reference. To successfully do the collision check, the minimum coordinate of the AABB is subtracted by the element's height in order ensure that a downward slope would not cause the grid to miss the landscape, and the maximum coordinate is increased by three times the grid's height for a similar reason with upward slopes. The maximum coordinate is significantly larger to simulate the fact that fires tend to spread faster uphill than downhill, since the heat from the flame rises.

Figure 4.7 demonstrates how the collision check is used to calculate the position of a grid element's Z value on the terrain. It is worth pointing out that despite the grid elements touching each other on the X and Y plane, this is not guaranteed to happen on the Z axis. This was ignored at this point of the implementation, only being corrected later.

It is not enough to check if a collision occurs. Unreal Engine divides objects into "Collision Groups" so that they can be easily filtered, and both landscape and foliage must share collision groups in order for the Procedural Foliage Tool to work properly. So once a collision is found, the components of the overlapping actors must be checked to confirm if they contain a Landscape component.

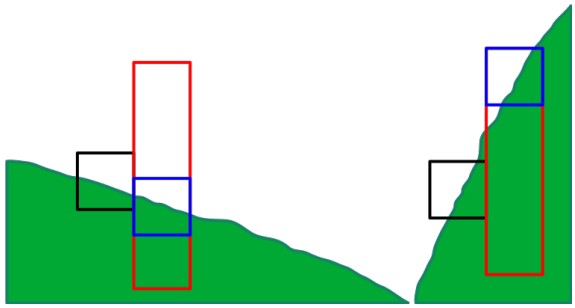


Figure 4.7: Demonstration of the collision check (in red) performed by the grid (in black), and the resulting grid (in blue) against a downward and upward slope. If the element was projected without any height changes in the left example, it would fail to collide with the terrain due to the grid residing in the air. In the example on the right, the element would have failed to collide with the edges of the terrain, meaning a collision would not have been found.

If a collision with a Landscape component is successful, a `GridElem` is created at the intersection coordinate, with `hp` value initialized to `FIRE_HP` (defined as `10`), `lifetime` as zero, and the element is added to the `FloorGridsMap`. Once all neighbouring coordinates have been tested, the grid element's `generatedneighbours` value is set to `true` in order to ensure it is not run again, and the method finishes.

The next method in the `GridController`'s tick function is `AttackNeighbour`. This function is relatively straightforward: it checks if the neighbouring cell has `hp` larger than zero. If so, then `1` is subtracted from the neighbour, and if its `hp` now falls to zero, then the neighbour's `lifetime` value is set to `GetTime() + FIRE_LIFETIME` and the cell is added to the `BurningGridsList` attribute.

One interesting aspect in the implementation is the decision to store grid elements twice, in both a dictionary and a list. This is done because a grid element is inert when its `hp` value is nonzero (meaning it is a damaged cell which has not yet ignited), and when its `hp` value is zero and its `lifetime` value has expired (meaning the cell has burnt out). There is no interest in iterating through these cells, thus any element which is active (on fire) is placed in the `BurningGridsList` to reduce the number of iterations necessary by the `GridController`.

As it is written right now, every time the `Tick` function is called by the engine, the cell will attack its neighbours and spread very rapidly. It is ideal to put a cap on the simulation speed, thus a `NextTick` attribute is added to the `GridController`, a `FIRE_TICKTIME` constant is created with a default value of `0.5`, and the loop inside the tick function is wrapped with a timer. This is shown in Listing 4.4.

With this change, if a `GridElem` only contains one neighbour which is on fire, it will take five seconds to have its health fully depleted to zero.

To render the cells, one can just iterate through all the elements in the `FloorGridsMap` and render a cube with a color, as shown in Listing 4.5.

The rendering code is placed outside of the `NextTick` check. If it is placed inside, then

the boxes would only render for a single tick.

```
if (this->NextTick < GetTime()) {
    this->NextTick = GetTime() + FIRE_TICKTIME;
    for (GridElem* ge : this->BurningGridsList) {
        ...
    }
}
```

Listing 4.4: The main `Tick` loop of the `GridController`, now wrapped with a timer

```
for (TPair<FVector2D, GridElem*> pair : this->FloorGridsMap) {
    GridElem* ge = pair.Value;
    FColor col = FColor(0, 0, 0);
    if (ge->hp > 0)
        col = FColor(0, 255, 0);
    else if (ge->hp == 0 && ge->lifetime > GetTime())
        col = FColor(255, 0, 0);
    DrawDebugBox(ge->coord, FVector(GRID_RESOLUTION)/2, col);
}
```

Listing 4.5: The debug rendering function for grid elements.

With a simple controller class and a grid element structure, a basic fire spreading system has been implemented. Figure 4.8 shows the progression of the system after running for twenty seconds.

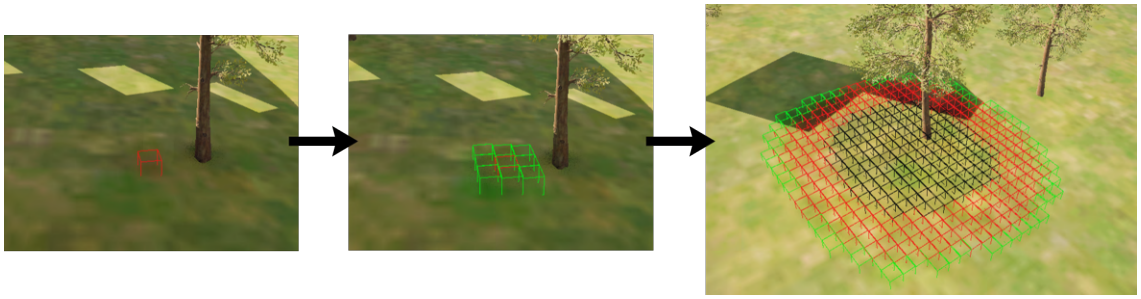


Figure 4.8: A demonstration of the naive simulation, starting as a single grid and quickly spreading outwards in all directions.

One of the benefits of keeping the elements aligned to a grid is that it lets this implementation support multiple fires without any additional work. Placing multiple fire starters in the map will cause multiple fires to break out, and each one will spread out by itself. When a burning element attempts to spread to a zone where a separate element already resides, it simply assigns it as a neighbour and continues without problems.

The last thing worth tackling is adding support for wind. This can be used to force the fire to trend towards a specific direction, and ensure that a grid element will tend towards the edges of the fire front polygon. Wind was specified as a single, global 2D vector value for the entire map.

If the position of both the attacker and victim grid element is known, a direction vector can be extrapolated by subtracting both positions and normalizing the vector. Then, instead of just damaging a grid element by a static value of 1, the damage value is multiplied by the dot product of the direction vector with the wind vector. If the wind direction vector is nonzero, and the result of the dot product is positive, then the damage application can proceed. The fire spreading with a global wind value is shown in Figure 4.9.

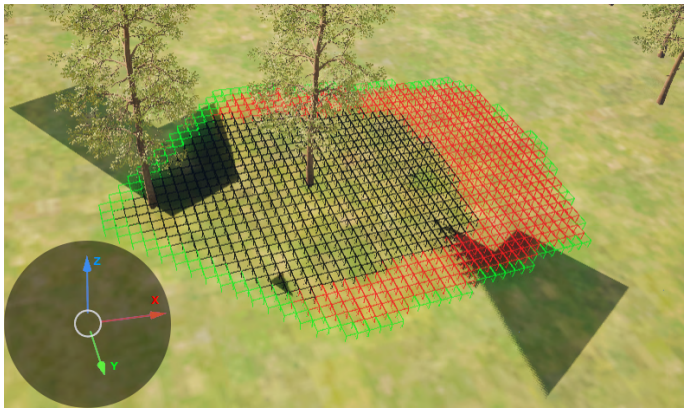


Figure 4.9: A demonstration of the fire spreading with the global wind value set to the static value `(1, 0)`. This causes the fire to spread towards the positive X axis, which is the right side of this image.

4.3.1 Adding Support for Foliage

Now that a basic simulation was written, the next step was to add support for burning foliage and other flammable objects. Because these objects exist in three dimensions, it requires some minor changes to the `GridElem` struct and to the `GridController` actor. First, a second struct was created called `ObjectBurn`, and its defintion can be seen in Listing 4.6.

```
typedef struct {
    FTransform transform; // The transform data of this object
    TMap<FVector, GridElem*> gelems; // The map of grids
    OBJType type; // The type of the object
    void* object; // A pointer to the object being burned
} ObjectBurn;
```

Listing 4.6: The burning object struct definition.

In order to allow for support of burning multiple different types of objects, the `ObjectBurn` struct uses a `void*` along with the enumeration `OBJType`.

Next, the `GridElem` struct was modified to add a `burnobj` attribute (which points to a `ObjectBurn`), and the `GridController` actor had `BurningObjectsMap` added to it in order to map 3D coordinates to `ObjectBurn`'s.

The only method which requires changing is the `GridController`'s `AttemptSpreadGrid` method. Instead of checking the X and Y axis only, now the Z axis must be checked as well, but only if the cell's `burnobj` pointer is not null.

As stated in section 4.3, the components of overlapping objects are checked to ensure collision with a landscape, so this must obviously be changed to check if a foliage actor was hit as well. Unfortunately, getting which specific foliage was hit is non-trivial. When foliage is generated by the Procedural Foliage tool, it creates `InstancedFoliageActors` in the world, which contain multiple foliage components, which themselves contain multiple `FoliageInstancedStaticMeshComponent`. This inner-most component features a function for testing box overlaps, and it returns a list of indices of the foliages that were hit. From that, one can then obtain information about the actual individual foliage instance.

Once the key of the overlapped foliage instance is known, the transformation of the foliage is obtained, and the instance's coordinate is checked against the `BurningObjectsMap`. If the map entry does not exist with the given coordinate, then it is created and added. Next, a `GridElem` is created similar to how it would be on a landscape, but now its `burnobj` pointer is assigned to the found/created `ObjectBurn` and the element is added to the `ObjectBurn`'s `gelems` map. The grid element is not added to `FloorGridsMap` since it does not belong to the terrain.

Using a map with cell positions comes with a problem, which is the fact that the grid elements are aligned to the terrain's Z. This means that the positions of grid elements that spread to the foliage could differ if they are not aligned in the same way. See Figure 4.10 for a visual demonstration of the problem.

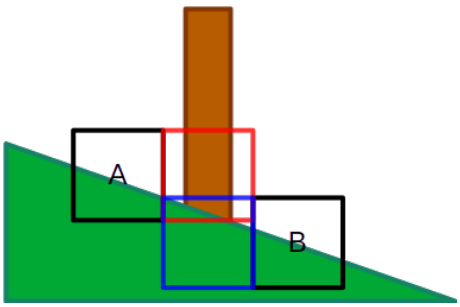


Figure 4.10: The brown rectangle represents a tree trunk (foliage). Grid A will attempt to create a grid aligned to itself (the red grid), while grid B will attempt to do the same (the blue grid). Because the only way for the grid to know if a position is occupied is by checking the projected grid's center coordinate against an `ObjectBurn`'s map of grids, both grids will be created. This will result in multiple overlapping grid elements.

There are two ways to resolve this issue. The first way would be to align the grid elements that belong to foliage to some global grid. This would work well but would have a few consequences, namely that two similar foliages placed in different locations could have vastly different grid shapes, as it is dependent on whether the foliage overlaps the

boundary between cells. The second solution would be to keep grid elements aligned to the center of foliage instance itself. The downside of this is that grid elements that belong to the foliage and grid elements that belong to the landscape might overlap. It was decided that the second solution would work best, because it could also allow for a second potential optimisation.

Since foliage meshes do not change, it was interesting to check whether there were any performance benefits in precalculating all of the possible grid element positions in a foliage instance. This was tested but ultimately scrapped for two reasons:

1. If all the elements were created at once, the program would hitch due to all the cell memory allocations for large objects (such as trees, which could fit almost three hundred grid elements). Hitching would have a significant usability impact in a VR environment.

2. Precalculating would require foliage meshes to be the same everywhere, this would mean no rotation or scaling could be applied to the models. This would create a very unrealistic scene.

Therefore, dynamically calculating the grid element positions was decided to be the better option, since it would allow for more flexibility, and require less work from developers integrating this API into the final fire visualisation tool since they would not need to create the pre-calculated cell position structure for every type of combustible object placed on the map. The fire simulation, extended to foliage, can be seen interacting with trees in Figure 4.11.

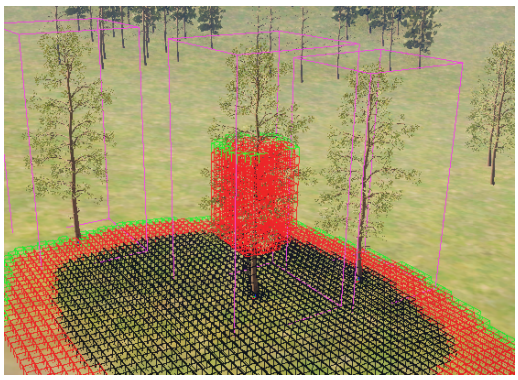


Figure 4.11: The fire simulation affecting a tree.

The last thing worth discussing when it comes to foliage is regarding how the collision check is performed to begin with. 3D meshes are composed of many small triangles, and checking an AABB against every single one to isolate a collision would be computationally heavy for complex models. Therefore, UE5 attempts to represent the object as a collection of primitive shapes such as capsules and rectangles.

As can be seen in Figure 4.12, the collision shape does not take into account every branch of the tree. For a more accurate simulation of fire spreading up close, a more

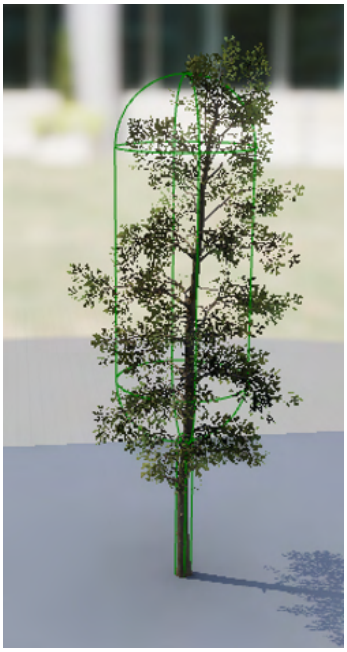


Figure 4.12: The conifer model and the collision shape defined automatically by UE5.

accurate collision mesh would be required, which could have not just an impact on the collision checking of the grid elements, but an overall impact on game performance as well since the same collision boxes are used for everything that can interact with them in the game engine.

4.3.2 Adding Support for Inflammables

Lastly, support for checking against inflammable actors and materials would be desirable, as in the fire visualisation tool there are structures such as houses and roads placed on the map which do not combust.

Adding support for inflammable actors is relatively trivial. As discussed in Section 4.3, since Unreal Engine uses collision groups to filter collisions, the same data can be leveraged here. The developer can define custom collision groups, assign them to objects, and perform an overlap check against that specific collision group. If the AABB overlaps with an object in the inflammable collision group during the `AttemptSpreadGrid` method, it is stopped early.

When a collision check is performed against an object, Unreal provides a lot of collision information, such as the normal vector of the collision, and the physical material that was hit. Physical materials are attributes given to materials, which are used for object interaction. One such example is the physical materials being used to decide which footstep sounds to play when players walk over terrain. One could paint a material with the inflammable physical material assigned to it on sections of the landscape (for instance, a rock material which would not combust), and then simply check if the collision

information responds with the inflammable physical material.

An implementation of the fire spreading system with the inflammable material system working is shown in Figure 4.13.

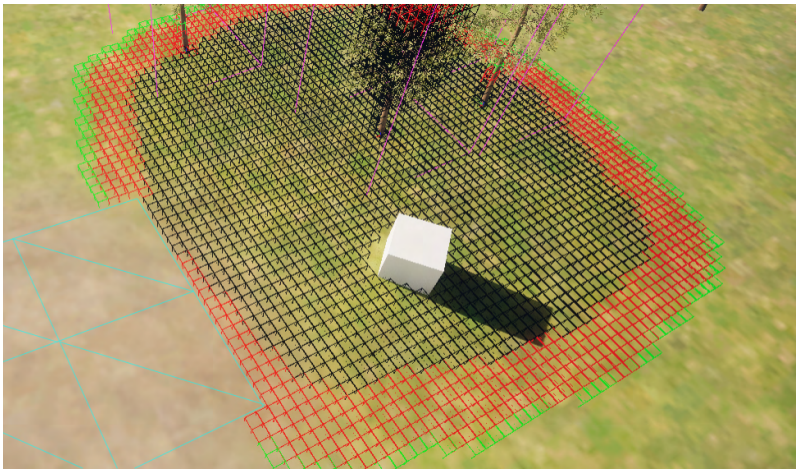


Figure 4.13: The fire simulation avoiding an actor given the inflammable collision group (the cube) as well as an inflammable material (The dirt on the left side). The triangles of the landscape mesh marked as inflammable are highlighted in cyan.

Due to how materials are implemented in Unreal Engine, there is a noticeable abrupt rectangular gap in the transition between the flammable and inflammable materials in Figure 4.13. This is because despite the materials blending together visually, internally unreal only stores one physical material per quad of the terrain mesh. The only workaround for this is to have a higher resolution landscape mesh, at the cost of performance and storage space.

It is worth pointing out that the grid is not necessarily aligned to the quads of the terrain itself, so there may be situations in which a grid element overlaps both a quad which contains a flammable physical material and another which does not. In this scenario, the grid element will choose not to spread to this location.

4.4 Structural Optimisations

With the naïve implementation finished, it was time to refine it. First, a base benchmark was necessary to compare against, so the simulation was run as-is on a machine with an AMD Ryzen 7 4800H. The performance testing will be focusing entirely on the performance of the CPU, thus all rendering of grid elements was disabled.

In order to measure the performance, a static scene was set up with foliages and terrain, and a fire starter was placed. The fire starter would drop and create a burning grid element, which would spread outwards. Every 45 seconds, a snapshot would be taken with Unreal Insights to measure the processing time of the `GridController` tick in that frame, as well as the object's memory usage. The total number of grid elements in memory, as well as

the number of elements that were burning, was also recorded. The average of 5 tests was taken and compiled into Table 4.1.

Total Cells	Burning Cells	Processing Time	Memory Usage
648	426	1.42ms	96 940
11 221	1625	3.45ms	1 678 661
28 026	4702	7.83ms	4 192 689
45 746	6966	10.53ms	6 843 601
115 242	10387	17.26ms	17 274 611

Table 4.1: The processing time (of the CPU) for grids, and the memory usage (in bytes). For 90 FPS, the processing time must fall below 10ms. GPU performance was not measured because it are not relevant to the processing of grids, only rendering.

4.4.1 Eliminating Redundant Cells

The first obvious optimisation that was done to the algorithm was to remove the use of the burnt grids, because they are taking up significant memory. The reason for the burnt grids existing is to prevent the burnt grid elements from spreading back towards an already burnt area. When it comes to integrating the simulation into the fire visualisation tool, the burnt grids will not be necessary at all as the spread of the grids will be dictated by the fire front polygon. But for the time being, a simple check can be performed to cull the majority of burnt grids.

When a grid element burns out, its neighbours are iterated over. If a neighbour grid is itself burnt out, and all the neighbours of said grid are themselves burned out, then the neighbour grid can be safely removed. After finishing the iteration, if the grid no longer contains any neighbours, then itself can also be removed.

The same test was run, with the same methodology, and the results were compiled into Table 4.2. It is worth pointing out that the timer used real-time, thus it does not take into account time lost due to hitches or lag when starting the program. This means that the number of burning grid elements might not match exactly with the previous table, but using the average of five tests should help even it out.

Total Cells	Burning Cells	Processing Time	Memory Usage
648	426	1.12ms	96 940
6 175	1 729	3.23ms	923 780
27 787	4 729	8.36ms	4 156 935
49 234	6 971	10.21ms	7 113 928
72 025	9 354	16.63ms	10 080 496

Table 4.2: The performance table for grids after removing unnecessary burned objects

As can be seen in the table, the memory usage went down significantly, and as a result it was possible to have large fires represented with less memory. Of course, this had

minimal impact on the processing time since the amount of burning grid elements was left unchanged.

4.4.2 Making Grid Elements into Actors

The next potential performance improvement was to decouple grid logic from the `GridController`. Instead of the controller actor performing the fire spreading logic on each grid element, one could make grid elements into actors with their own `Tick` function, and only have it use the controller to retrieve neighbours. The idea behind doing this came from the observation that UE5 tends to defer heavy operations on its data structures (such as deletions), and the documentation claims that "many actors can be updated in parallel if they're in the same group"[17].

This was implemented, and tested using the same methodology described in Section 4.4. The results of the test can be seen in Table 4.3.

Total Cells	Burning Cells	Processing Time	Memory Usage
628	462	1.22ms	102 230
6 312	1 693	3.71ms	976 248
26 789	4 638	8.56ms	4 316 455
47 323	6 479	10.27ms	7 312 239
71 353	9 623	16.82ms	10 814 423

Table 4.3: The performance table for grid elements split into separate actors

The performance numbers showed that the change had little impact on the execution of each grid element's logic with a slight increase in memory usage. However, in large scenes the performance would stutter more due to actor creation and deletion being a much heavier operation. This is likely because the engine requires more bookkeeping regarding the managing of actors and events, compared to simply calling `malloc` and `free` on some small C structs.

4.4.3 Switching from Maps to Trees

One big problem with the usage of a dictionary is the fact that when the grid combination and division is implemented, using the center coordinate of the grid element as the key value will no longer work. A significantly large grid element will occupy the position of many cells, thus a redundant key value pair will be needed for every one. Quadtrees are optimized for storing and searching arbitrarily sized rectangles, making it the obvious choice.

UE5 provides a quadtree data type, however the provided implementation was not very flexible (for instance, it lacked the ability to manually traverse the tree). It was decided that writing a custom one would also be more efficient because it was possible to leverage the fact that elements are always aligned to a grid. By design quadtrees will split the space

they represent evenly, which could result in many elements being unable to be stored in children nodes, significantly impacting search performance. Though it is worth pointing out that this optimisation will not benefit scene with large grid elements which overlap multiple boundaries. Figure 4.14 demonstrates the problem visually.

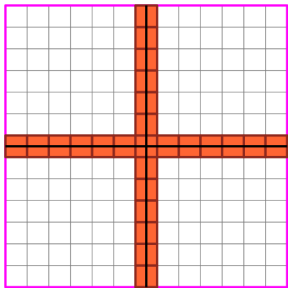


Figure 4.14: The red elements, because they reside on the borders of the children nodes, they must instead be placed on the root node (in pink), which would reduce the search performance of the quadtree.

Because UE5 is a complex program, it was decided that the quadtree implementation would be easier to write and debug on a smaller visual application, and then ported to Unreal once finished. Since the data structure would need to be written in C++, it was that decided that the implementation would be written with a library that the author of this dissertation was already familiar with: wxWidgets.

The quadtree implementation should be as efficient as possible:

1. During searches and removals, the quadtree tests which quadrant the element most likely belongs inside and searches it first

2. When an element is inserted, a pointer to the child node is returned so that removals can be done in $O(1)$.

3. As previously mentioned, the quadtree is aligned to the global grid, so elements crossing tree edges should be reduced.

4. When an element is removed, the quadtree's structure is unchanged because cleanup is deferred. In the scenario where an element moves, it must be removed and reinserted into the quadtree. By deferring the cleanup, the case where the children nodes are destroyed and recreated right after can be minimized. This also results in stutters being reduced since the quadtree is not required to recursively destroy all its children in a single step.

The implemented test program is shown in Figure 4.15.

Bringing the implementation over to UE5 was relatively trivial, the only thing which required changing was switching from the standard library's `std::vector` over to Unreal's `TArray`.

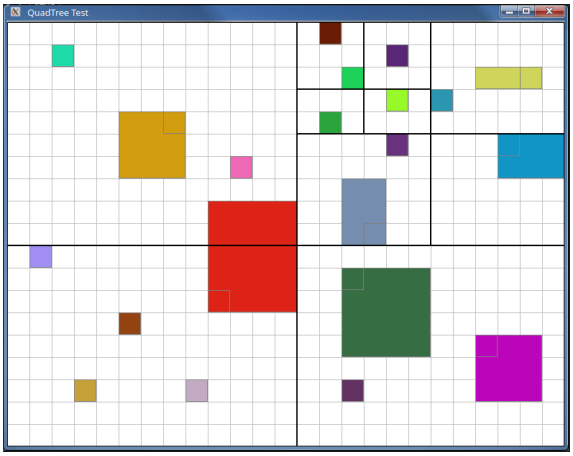


Figure 4.15: The implemented quadtree running in a wxWidgets application. The quadtree used a maximum capacity value of 4.

To test the performance of the new data structure, a test was devised where 100,000, 500,000, and 1,000,000 randomly generated grid elements (with varying sizes and positions) were created. Then, all elements were inserted into both a `TMap` and quadtree, and the time to perform all insertions was measured. Then, a selection of random points equivalent to the number of elements were tested, as well as randomly generated rectangles for area tests, and time time taken for the entire operation to be performed was recorded. For testing searches in an area, the quadtree test used rectangles with length/width between 0 and 1000, while the map used 0 and 15 (any larger and the time would explode due to the operation being $O(n^2)$). Finally, all elements were removed from the structures, and the processing time for the entire operation was recorded. Five total tests were performed, and the average processing time for each one of the data structure's methods was recorded. Finally, the times were normalized based on the number of actions that were performed (for instance, the average times for 500,000 elements was divded by 5 to be easier to compare to the times of 100,000 elements), and compiled into Table 4.4.

	100 000 Elements		**500 000 Elements**		**1 000 000 Elements**	
	TMap	**Quadtree**	**TMap**	**Quadtree**	**TMap**	**Quadtree**
Insert	16ms	42ms	31ms	57.4ms	33.8ms	82.4ms
Find (Point)	12ms	97ms	10.8ms	91.8ms	12.4ms	93.6ms
Find (Area)	881ms	657ms	2 719ms	886.6ms	6 117ms	1 463.1ms
Remove	11ms	6ms	18.6ms	12.2ms	20.2ms	14.9ms

Table 4.4: Comparison of operations on a TMap versus the implemented Quadtree.

One more optimisation needed to be done to the quadtree implementation, which is to figure out the optimal `CAPACITY` value which will cause the quadtree to subdivide. This was tested in Unreal by inserting 1 million elements and recording the time taken for the whole operation. The average of 5 tests was compiled for each test, and the performance

table 4.5 was the result, with the final value of 25 being chosen.

Capacity	Insert	Find (Point)	Find (Area)	Remove
5	804ms	999ms	20 279ms	101ms
10	1 032ms	956ms	16 338ms	114ms
15	796ms	902ms	15 810ms	134ms
20	593ms	901ms	14 766ms	150ms
25	551ms	920ms	15 008ms	155ms
30	653ms	970ms	16 391ms	164ms
50	618ms	958ms	16 746ms	166ms
100	526ms	1 546ms	22 987ms	144ms

Table 4.5: Comparison of quadtree performance for 1 million entries based on the `CAPACITY` value. While in a real world scenario the quadtree of the simulation will have clustered data, the points were inserted at random locations to force more recursion (for a worst case scenario).

Once the quadtree was fully implemented and tested to be working, the next step was to also implement an octree. Since an octree is just a quadtree but with an extra dimension, the implementation was relatively trivial as it was just a matter of copying the quadtree implementation and replacing the usage of `FBox2D` to `FBox`, as well as adding a few more lines of code to handle the Z axis division.

Next, the data types in the `GridController`'s `FloorGridsMap` and `ObjectBurn`'s `gelems` had to be replaced, with the former being renamed to `FloorGridsQT`. Iterating the quadtree's elements for rendering now had to be done recursively instead of using a single for loop. While less efficient, the rendering code is merely for debugging purposes and will not be used in the fire visualisation tool.

The constructor of the quadtree requires that the user specify the boundary of the root node. Because the landscape is static, it would be beneficial to automatically compute the quadtree boundary from it. Unfortunately, UE5 provides limited documentation on its landscape calculations, both on the documentation pages and in source code comments. As a result, the size of the quadtree must be passed to the `GridController`'s constructor, otherwise a warning will be thrown during game execution. he attempt to dynamically calculate the quadtree bounds was left commented out in the `GridController`'s constructor.

One last optimisation can be done to the code as a result of the quadtree. Previously, to find the neighbouring elements, it was required to check each coordinate around the element by iteratively looking at all neighbour coordinates, one by one, as shown in Listing 4.7.

```
1 int startz = (ge->objburn == NULL) ? 0 : -1;
2 for (int z = startz; z <= 1; z++) {
3     for (int y = -1; y <= 1; y++) {
4         for (int x = -1; x <= 1; x++) {
5             if (x == 0 && y == 0 && z == 0)
```

```
                continue;
            FVector testpos = FVector(
                ge->coord.X + GRID_RESOLUTION*x,
                ge->coord.Y + GRID_RESOLUTION*y,
                ge->coord.Z + GRID_RESOLUTION*z);
            GridElem* found = this->FloorGridsMap.Find(testpos);
            if (found != NULL)
                ...
        }
    }
    if (ge->objburn == NULL)
        break;
}
```

Listing 4.7: The old neighbour searching algorithm.

This is incredibly costly as the map would need to be iterated over 8 (in the case of 2D iterations) or 26 times (in the case of 3D iterations). The searching can be improved from $O(n^3)$ to $O(\log n)$ by simply performing a single search in the quadtree, as shown in Listing 4.8.

```
FBox gebox = GridElem_GetFBox(ge);
FBox testbox = gebox;
TArray<GridElem*> found;

testbox.ExpandBy(GRID_RESOLUTION);
if (ge->objburn == NULL)
    this->FloorGridsQT->FindElementsInRect(FBox2D(testbox), &found);
else
    ge->objburn->gelems->FindElementsInCube(testbox, &found);

found.Remove(ge);
for (GridElem* nge : found) {
    if (!GridElem_GetFBox(nge).Intersect(gebox))
        ...
}
```

Listing 4.8: The new neighbour searching algorithm.

4.5 Grid Grouping

With the grid elements now stored in a better data structure, the next step is to optimize the scene further by implementing a system where grids will choose to combine with one another depending on the camera distance.

4.5.1 Combination

It was decided, for simplification and optimisation reasons, that only burning grids would concern themselves with performing joins/divisions. Three modifications were

done to the `GridElem` struct, the first being that the `coord` was changed to be the minimum coordinate of the grid instead of the center, the second was the addition of an integer `lod` attribute to the structure, and the third was the addition of a vector `size` value which represents the number of grids which the grid element expands towards in a given direction (for example, a size value of (1, 0, 0) means that the grid occupies two `GRID_RESOLUTION` worth of space on the x axis).

When iterating over the burning elements list, the distance of the element from the camera is calculated, then the distance is divided by a constant value of `GRID_RESOLUTION*4` and stored in the `lod` attribute of the grid element. Two new methods were created, and were named `Combine_GridElements` and `Separate_GridElements` respectively. These were added to the `GridController`, and called after a grid element has performed an attack on all its neighbours.

In order for a grid to combine, its neighbour must fulfil the following prerequisites:

1. The `burnobj` pointer in both elements must be equal. This is to prevent landscape grid elements from combining with foliage elements, as well as different foliage objects from having their grid elements combine with one another.

2. The neighbour must have its `hp` value set to `0` to ensure it is on fire, and its `lifetime` value must be larger than `GetTime()` to ensure it is not burnt out.

3. The sum of the grid and its neighbour's unscaled width and height values divided by four must be smaller than its `lod` value

4. The bounding box generated by the combination of both grid elements must not overlap any other grid element

If these prerequisites are met, then the two grid elements can be combined. This is as simple as setting the grid element's `coord` value to the bounding box's minimum value, setting the `size` value to the bounding box's maxmimum subtracted by the minimum divided by `GRID_RESOLUTION`, combining the `hp` and `lifetime` values of both elements, and then deleting the now redundant neighbour element.

One last tweak is necessary to the combined grid, which is that its Z value needs to be adjusted. One problem that was common during testing as the grid grid element expanded in size was that it would commonly end up in scenarios where the element was buried under the landscape mesh.

To adjust the height, the element performs a line trace on the center coordinate, as well as on all four of its corners so that it can obtain the minimum and maximum Z value of the terrain. First, the grid element's centre is adjusted to be tangent with the landscape, and then its height is adjusted, given the known minimum and maximum coordinates. The grid element will always have a height of at least `GRID_RESOLUTION`, but its height does not need to be aligned to a grid (since the landscape is, in essence, two-dimensional). It is also worth pointing out that if the grid element belongs to a foliage object, then its

minimum and maximum coordinate must be aligned to the 3D grid, thus cannot have variable height.

Because the size of the grid element has changed, it needs to be removed and re-inserted into the quadtree/octree. However, because the `GridElement` stores within it a pointer to the tree child node it belongs to, it is not necessary to perform the insertion from the top of the tree. Since the grid element has increased in size, it will never be inserted into the children of the current tree node (as the children will always be smaller), therefore we can insert the element back into the same node or into one of the parent nodes (which are guaranteed to always be bigger). This bottom-up approach is much more efficient than traversing the tree all the way from the top.

4.5.2 Separation

With the grid combination implemented, the next step was to implement the grid separation. For a grid element to seperate, it must fulfill the following requirements:

1. The grid element cannot have all of its `size` components equal to zero, as a grid element cannot be smaller than `GRID_RESOLUTION`.
2. The sum of the grid element's unscaled width and height divided by two must be smaller than its `lod` value.

Separating the grid element is relatively trivial, as it is an incredibly similar operation to what is done in the quadtree/octree implementation: divide the space equally in the center, and align the final split location to the grid. If one of the quadrants/octants of the division yields a value of zero, then no grid is created there.

After separation, the grid elements must also have their height values readjusted to prevent overly tall and skinny elements from being created. Figure 4.16 demonstrates the problem visually:

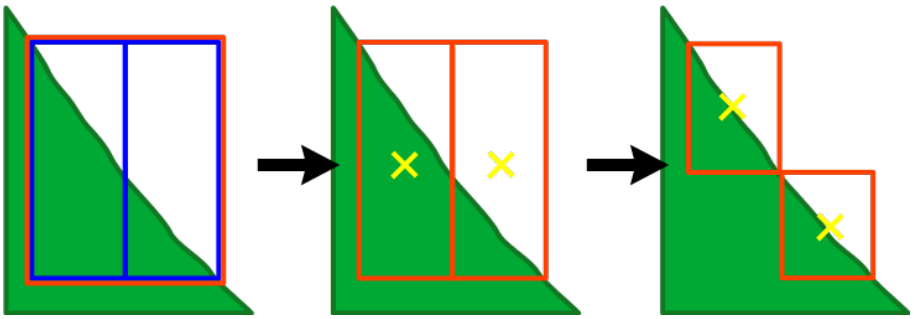
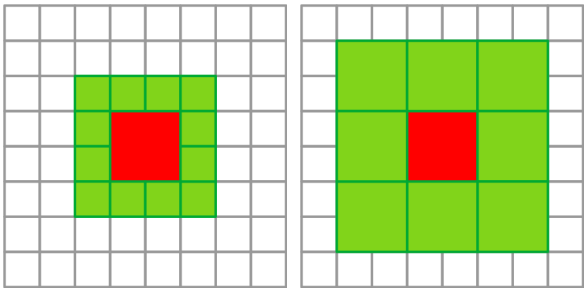


Figure 4.16: On the left figure, the red cells form the blue bounding box. The middle figure shows that the center coordinate of the combined result is no longer tangent with the landscape, thus it must be moved upwards as shown in the right figure.

The Z height adjustment is performed the same way as during the cell combination. Also similar to the cell combination algorithm, there is no need to fully traverse the

quadtree/octree to reinsert the element and the new elements that were formed as a result of the division. Since the grid element is now smaller, it is guaranteed to fit in the same tree node it was inside of, or in one of its children.

One problem arises from the use of large grid elements: defining how the fire will spread. It would not make sense for a large grid element to generate small neighbours, because sufficiently large cells would try to create hundreds of really small ones, causing stutters and increasing memory usage significantly. Instead, it was decided that the better option would be to have the large grid elements attempt to spread into large grid elements. Figure 4.17(a) demonstrates the undesired neighbour generation algorithm, while Figure 4.17(b) demonstrates the preferred method.



(a) (b)

Figure 4.17: Figure 4.17(a) shows the undesired grid spreading method, while Figure 4.17(b) shows the preferred alternative.

This presents a challenge, what should happen if a large grid element attempts to spread into an area that is already occupied by other grid elements? In the case of there already existing burning elements, it would not make sense to remove them as that would be the equivalent to extinguishing a fire. Furthermore, if unburnt cells are present with a depleted `hp` value, it wouldn't be productive to replace it with a new cell with replenished `hp`. Therefore, it would be wise to make holes in the cell so that those already existing elements are not replaced.

Subtracting a box from another is relatively simple, and the algorithm is outlined in Listing 4.9.

UE5's `FBox` class provides an `Overlap()` method to retrieve the overlapping rectangle, but one can be easily written by getting the maximum value X and Y component of the minimum coordinate of the two rectangles, and the minimums of the maximum coordinate. Those four values give the minimum and maximum value of the intersection rectangle. If the subtraction of the smallest maximum coordinate with the largest minimum coordinate gives a value smaller or equal to zero on either the X or Y component, the rectangles do not intersect.

```
1 // Get the overlapping box between the two
2 FBox2D intersection = subtrahend.Overlap(minuend);
3
```

```
// Edge case: no overlap
if (intersection.GetSize().X == 0 ||
    intersection.GetSize().Y == 0)
    return;

// Min is top left corner of the rectangle, Max is the bottom right
// ------------------
// |        A       |
// |----------------|
// | C |  Hole  | D |
// |----------------|
// |        B       |
// ------------------

int rectAHeight = intersection.Max.Y - minuend.Max.Y;
if (rectAHeight > 0)
    NewBoxes.Insert(FBox2D(FVector2D(minuend.Min.X, minuend.Min.Y),
                           FVector2D(minuend.Max.X, minuend.Min.Y + rectAHeight)));

int rectBHeight = minuend.Max.Y - intersection.Max.Y;
if (rectBHeight > 0)
    NewBoxes.Insert(FBox2D(FVector2D(minuend.Min.X, intersection.Max.Y),
                           FVector2D(minuend.Max.X, intersection.Max.Y + rectBHeight)));

// The same logic applies to C and D
// 3D is no different, we just need two more if checks to handle the Z value
```

Listing 4.9: The rectangle subtraction algorithm.

To simplify debugging and testing, like with quadtrees, the box cutting algorithm was implemented in the wxWidgets application before being ported over to UE5, shown in Figure 4.18.

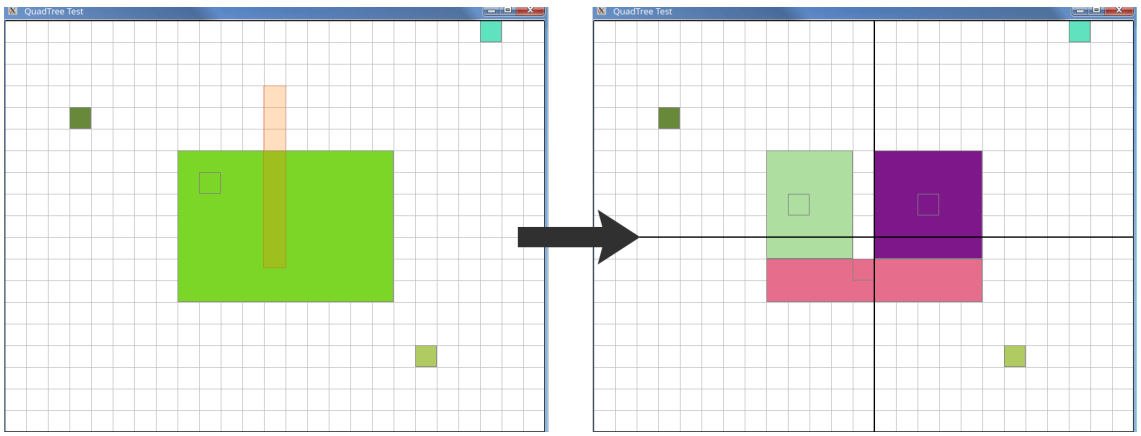


Figure 4.18: The rectangle splitting algorithm working in the wxWidgets application.

It is worth mentioning that this implementation also works on burnt cells, but they were not brought up because they will not be present in the fire visualisation tool. Generated

grid elements will have their `hp` value equal to `FIRE_HP` multiplied by the number of cells which the element takes up in the grid. Similarly in terms of damage, cells will multiply the damage by the number of cells they occupy.

One aspect that has not been covered yet regarding the combination and splitting is its effect on a grid's knowledge of its neighbours. In the scenario of a grid combination, the list of neighbours of the grid can be easily fixed by combining both lists (and removing the since-deleted grid/duplicates), but the scenario of the grid splitting it would be a lot more complicated. Furthermore, when a grid is combined with another, it needs to generate new neighbours with the same size, which could potentially modify the neighbours of nearby grids.

Currently, the only way to either regenerate grids or to force a recheck of neighbours would be to modify the `generatedneighbours` boolean of a `GridElem`, so it was decided to replace it with a `neighbourflags` value instead that can store whether a grid needs to check for neighbours and when a grid needs to generate neighbours. In the scenario where a combination happens, the resulting grid needs to both regenerate and recheck neighbours. In the scenario where a grid is split, it only needs to recheck for neighbours. Finally, when a grid is generating new neighbours which will overlap an already burning cell, said cell will have its recheck neighbours flag cleared.

In the `AttemptSpreadGrid` method, a grid element sets its `NEIGHBOURFLAG_GENERATED` flag to `true`. Outside of the method, a new one was added called `FindGridneighbours` which is used to find neighbours and enable the `NEIGHBOURFLAG_FOUND` flag.

When a grid element has grown sufficiently large, it is possible that it will be larger than any foliage it attempts to overlap. In this scenario, the grid element which is to be created and stored in the `ObjectBurn` will have a size equal to the foliage's bounding box, rounded to the nearest `GRID_RESOLUTION` to ensure it can be evenly divided if the camera gets too close.

The last point worth mentioning about the grid LOD algorithm is that, in its current state, the `GridController` is doing a lot of heavy operations per grid element in a tick: it needs to generate neighbours, find neighbours, attack neighbours, check and perform a separation, and check and perform a combination. The controller is spending quite a lot of time idle and then performing everything on every grid element every `FIRE_TICKTIME` seconds. This led to a lot of stuttering, so it was better to defer these steps.

The controller now performs two steps: The first is a a logic step where grid elements generate, find, and attack neighbours. The second is a optimisation step, where grid elements combine and separate. The logic step occurs once at the start of every `FIRE_TICKTIME`, while the optimisation step happens up to 8 times before the next `FIRE_TICKTIME`. The reason that it does multiple optimisations per `FIRE_TICKTIME` is because one pass is not enough to fully optimize a group of grids. Figure 4.19 illustrates why.

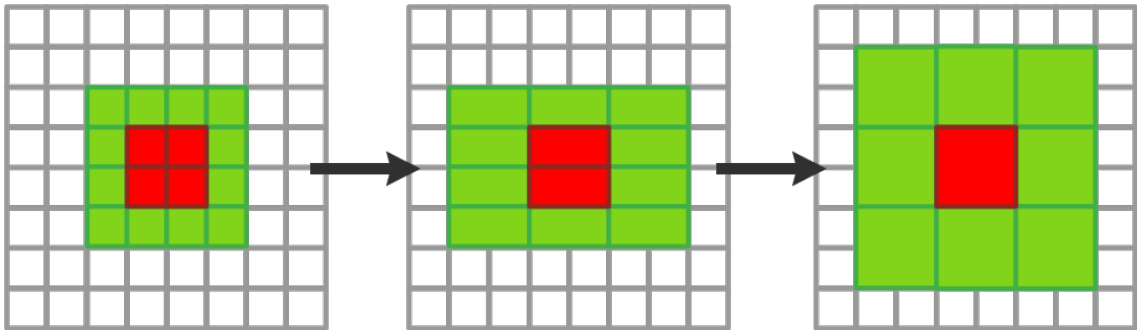


Figure 4.19: In order for the collection of grids on the left figure to be optimal, they need at least two ticks to combine.

4.5.3 Difficulties

One problem that was run into a lot was the simulation's tendency to combine elements into very skinny ones, which in turn would create skinny neighbours, combine with them, and form even skinnier elements. These unwanted grid elements are shown in Figure 4.20.

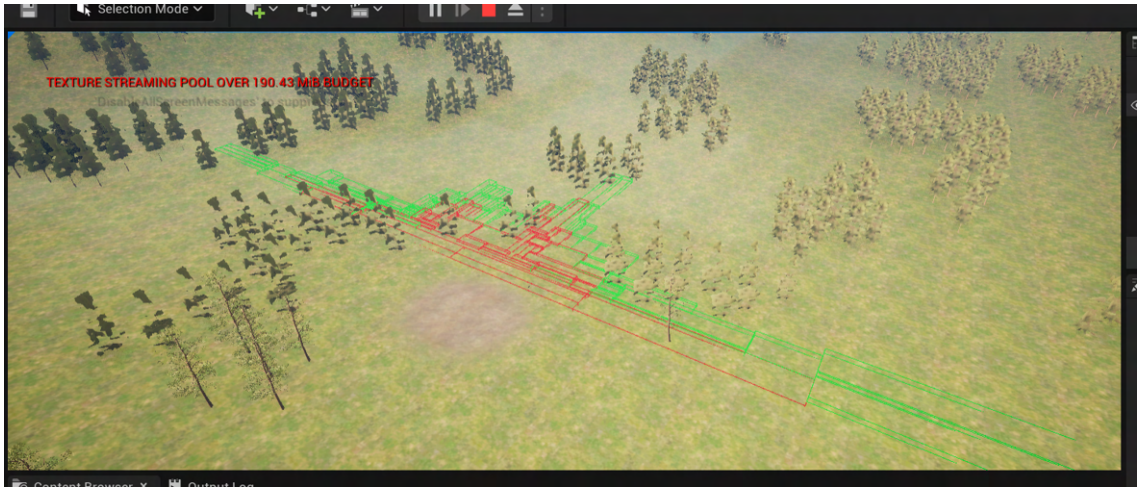


Figure 4.20: A screenshot showing the skinny grids problem

There were multiple different attempts at solving this issue. The first solution that was attempted was to put a limit on how big a grid could extend in a given direction. For instance, a grid element's width could not be larger than thrice the height. While this solution did fix elements getting too long, it had the unintended side effect of sometimes preventing optimal grid elements from forming in multiple steps, and this led to lots of small interlocking grids in a brick wall pattern which could not combine because the bounding box between them overlapped other grids. This is shown visually in Figure 4.21.

The second method that was attempted was to force the grid elements to alternate the merging direction. For example, in the first step, a grid element would only merge with

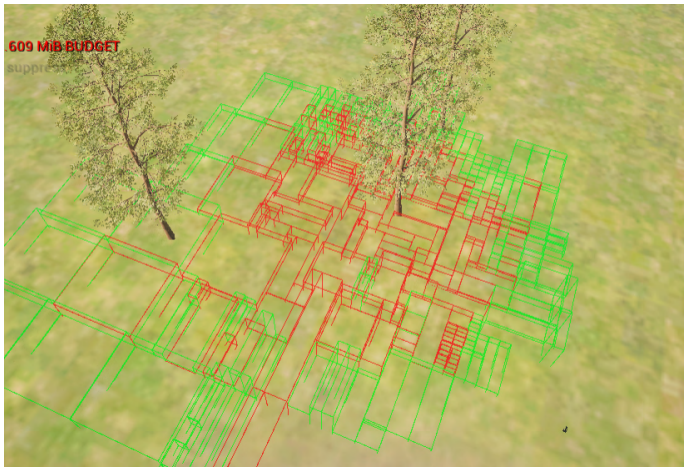


Figure 4.21: A screenshot showing the interlocking grids problem. Despite the camera being sufficiently distant from the grids to allow them to combine, they refuse to due to the bounding box of attempted combinations overlapping with other grid elements. This problem is especially exacerbated when the camera pulls close to a set of grid elements and forces them to divide, since the division might not be fully symmetrical.

elements to its left or right, but then after combining it would now only combine with elements above or below it. This also, once again, resulted in situations where lots of small grids should join, but ultimately did not.

One of the main reasons that grid elements would end up forming these long pieces was due to how the the damage calculation was being performed. The grid would use its entire area as a multiplier for the damage application on its neighbours, which is counterintuitive as the area of contact between the two elements can be very small. Figure 4.22 demonstrates the problem visually.

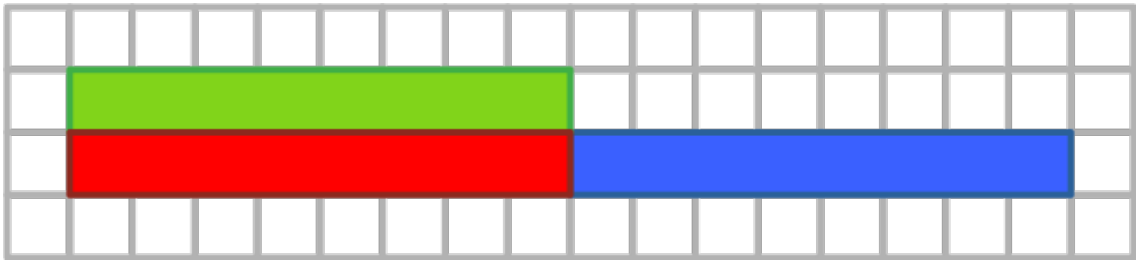


Figure 4.22: The contact area between the red and blue elements is only one grid, while the green element shares 8 grids of contact. Because of the larger contact area, the red element should be impacting more damage on the green one than the blue one. But because of the logic of the current implementation, the red element imparts the same damage on both, and during the combination step tends to combine with the blue element first.

To fix this, it was required to figure out the edge of contact between two grid elements by using the dot product of the two grid's center points, and then multiplying the damage by the number of grids shared between the contact edge, rather than the whole shape's

size.

While the change to the damage calculation did fix the generation of very skinny grid elements, it did not solve the interlocking grids problem. The only way to resolve that was to modify point 4 of the prerequisites of the grid combination algorithm described in section 4.5.1. Instead of outright failing the merge if another grid element is present in the bounding box of two grids, instead the combination can be allowed if the overlapping grid element would have an area smaller than the combination. If that was the case, the overlapping grid element(s) can be subtracted and split into smaller ones (or outright removed) in a fashion similar to what was described in the final grid spreading algorithm. This results in one grid element usually becoming large enough to always dominate over the others.

4.6 Grid Generation from Geometry

With the grid simulation in somewhat working order, the last thing required for porting the grid system over to the fire visualisation tool was to implement a method of converting an arbitrary fire front polygon into a collection of grids. This is a similar problem to the process of rasterization in 3D graphics, except in this scenario it is ideal to represent the fire front polygon as accurately as possible with the least amount of grids.

Because the only thing which interests us is the fire front, we can very easily create bounding boxes which enclose every edge of the polygon to represent it. Since our rectangles are aligned to a grid, the polygon's vertices will also need to be aligned. In the case of overlapping rectangles, subtracting one from another will fix that. This works for both concave and convex polygons, as demonstrated in Figure 4.23. Said polygon generation algorithm was also ported over to the Unreal project, and uses the polygon normals to create burnt elements behind the grid elements, to ensure that the fire does not spread backwards.

The obvious problem with this approach is that the boxes will also overlap many grid positions that do not belong to the fire front. Of course, when the camera is sufficiently far away, this won't be a big problem, but as the camera approaches, we can divide the grid evenly and eliminate the ones which do not intersect any edge of the polygon. This process can be repeated until a satisfactory resolution is achieved. Said partitioning is shown in Figure 4.24.

This has all been implemented into the test wxWidgets program, but whether the implementation works as expected is yet to be seen, as it needs to be ported over to the fire visualisation tool.

In theory, to implement this API into the fire visualisation tool, when the grids are generated from the polygon, the grid elements need to store the list of edges that they are intersecting. An efficient way to do this is to store the edges of the polygon in a quadtree, so that searches can be done in $O(\log n)$ as opposed to $O(n)$. When a subdivision occurs, the rectangle checks each edge in its list, and eliminates edges which no longer intersect. If

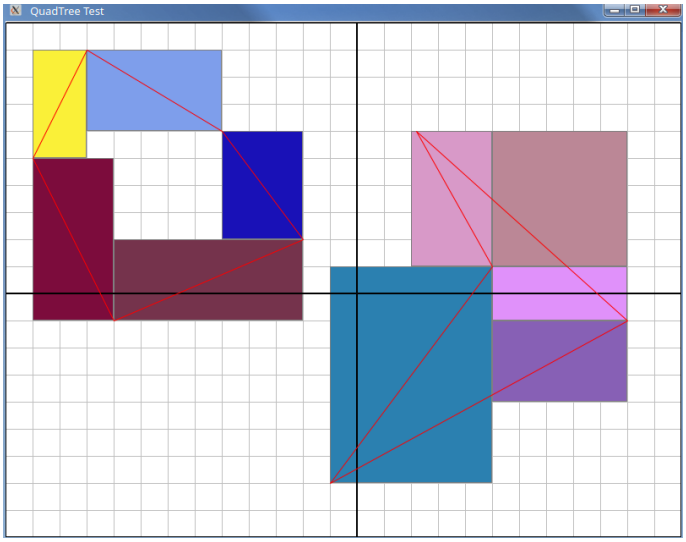


Figure 4.23: A basic fire front rectangle coverage algorithm, working for both a concave and convex shape, shown implemented in wxWidgets.

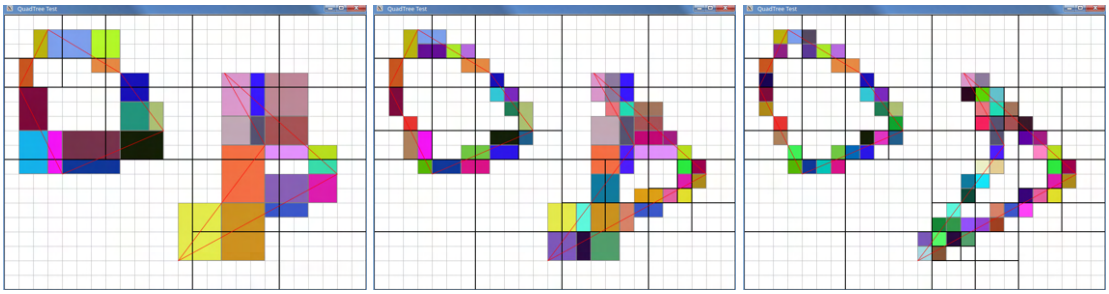


Figure 4.24: Demonstration of the further subdivision of the grids that represent the polygon.

the list is empty, the rectangle is removed. Checking if a line segment intersects a rectangle is done efficiently using the Cohen–Sutherland algorithm [6]. And of course, as outlined in Section 3.1.2, the polygon might have an increase or decrease in the number of edges, which might complicate the implementation in this manner. So the grid elements must find the nearest edge of the future fire-front polygon and modify its data accordingly. This potential integration of the API has been documented in the project's README to potentially facilitate the integration process.

4.7 Simulation Culling

It is inefficient to perform simulations on grid elements which are not visible to the camera, so finding a method to cull them is a good idea. Because our world is projected onto a rectangle (our computer screen), we can figure out the shape it projects onto the scene and then test the grid elements against it.

In order for a point in world space coordinates to be transformed to screen space

coordinates, the operation defined by Equation 4.1 is performed on it.

$$v_{screen} = P_{camera} \times V_{camera} \times v_{world} \tag{4.1}$$

Where:

- v_{screen} is the vector of the 3D point in screen space
- P_{camera} is the camera projection matrix
- V_{camera} is the camera view matrix
- v_{world} is the vector of the point in world space

To obtain the camera frustum vertices in view space, the equation is solved for v_{world}, as shown on Equation 4.2.

$$v_{world} = P_{camera} V_{camera}^{-1} \times v_{screen} \tag{4.2}$$

We replace v_{screen} with the 8 points that represent the corners of 3D screen space, and we divide the resulting v_{world} by the W component to go from homogeneous to 3D coordinates.

With the shape of the view frustum obtained, we can calculate the minimum bounding box of it and use it to return the list of burning grid elements that are visible in the quadtree.

This gives us the list of landscape elements that are burning, but not the elements that belong to foliages. This, however, is resolved by having a second quadtree containing the bounding boxes of `ObjectBurn`'s. The ability to retrieve grid elements from a rectangle and cube was added to the `GridController`.

With this, in theory, the `BurningGridsList` is no longer required since testing the quadtrees ($O(\log n)$) is more efficient than iterating through the entire list and testing the grids against the screen frustum ($O(n)$).

Since the actual culling has not been implemented into the final program, one can only speculate regarding how it would work in the fire visualisation tool. One of the main questions one might have regarding the culling of objects is: "What if a user spends a long time not looking at grid elements (say, for an hour) and attempts to look back at the scene?". Obviously, since these grids are not being updated anymore, their simulation will be very far behind.

To solve this, theoretically, when a grid element is simulating we store the `GetTime()` value in an attribute called `lastseen`. If we view a grid element who's `lastseen` value is too long (say, one minute), we delete the grid element and force it to be regenerated from the fire front polygon. This obviously means that there will be inconsistencies between what the user last saw and what was expected if the simulation were to proceed normally. However, this is a compromise that must be taken in the scenario that we are rendering a potentially huge forest fire in real-time.

Worth discussing is also the fact that foliages are still being simulated, even when the grid element that surrounds it is gigantic. Consider an area completely overlapped by one large grid element, where there are thousands of tiny grass objects. In the implementation discussed in section 4.3 and 4.4, the overlap test will return all found foliage objects and generate grids from them. The solution to this is to calculate the area of the bounding box of the foliage model, and compare it against the bounding box of the grid element. If the area of the bounding box of the foliage model is smaller than an eighth of the grid element's, then the generation of the foliage's `ObjectBurn` can be skipped. This foliage culling feature *has* been implemented into the `GridController` logic, however. A note was left in the repository's README discussing the possible integration of this API into the fire visualisation tool.

4.8 Summary

In summary, the following components were implemented:

1. The existing fire visualisation tool was ported to UE5, but it was unable to be properly tested.
2. A terrain import tool was written in Python, to allow the conversion of GIS DEM into heightmaps supported by UE5.
3. A micro scale fire-spreading system was implemented.
    a) The micro scale simulation supports burning of three dimensional objects, while ignoring inflammable objects and materials
    b) The micro scale simulation is optimized by grouping cells together, essentially also turning in into a less-accurate macro-scale simulation
    c) Functions for testing grid elements in an area were implemented, but are currently not actually used by the program as a functional system would rely on the polygon data to be retrieved from the fire visualisation tool. The potential use of these methods were documented in the repository's README.
4. A test program was written in wxWidgets to demonstrate the creation grid elements from fire front polygons, and said functions were ported to Unreal.
    a) Since the polygon data is currently not dynamically retrieved from the server, the functions are used by the program, only provided as a basic API for creating a single fire front. Said API does not take an evolving fire front into account.
    b) The potential extensions were documented in the repository's README.

While we were unfortunately unable to integrate the micro scale API with the existing fire visualisation tool, our solution acts as a working proof-of-concept, and provides a strong foundation to assist future work with this task.

5 Evaluation

This section evaluates the implemented architecture, looking at its performance, bottlenecks, and overall flexibility. The previous chapter focused on the CPU side of the implementation, and justified design decisions based on their impact on performance. This section will investigate and evaluate the solution under the major bottleneck, the GPU. The program was tested on a machine with an NVIDIA RTX 2060 6GB.

5.1 Performance Testing

As has been highlighted in section 2.1, VR is an incredibly performance intensive application, and thus the simulation is much more likely to run into performance constraints as a result of the graphics processor. So spending a great deal of time reducing the number of active grid elements on screen, which would be emitting particle effects, was necessary. The GPU being the performance bottleneck was found to be clear during the development of the API.

Figure 5.1 shows a snippet from Unreal Insights. The top graph starts green, signifying that the scene took under 16.6ms to render, but eventually starts showing a downgrade in performance (signaled by the yellow bars). A lag spike was highlighted, and focused on the bottom of the tool. The highlighted section of the program took 26.6ms to render a frame (equivalent to 37 FPS). The graph shows that the GridController loop, (which was confirmed to have around two thousand elements on fire) took 2.2ms to process. The CPU spent 13ms waiting for the GPU to finish its rendering task, highlighting that the GPU was the bottleneck.

Before the actual testing of the simulation could take place, it was first necessary to set up the scene to work in VR. By default, Unreal's graphics settings are incredibly demanding, which resulted in the scene struggling to run above 10 FPS. The following tweaks had to be done:

- All ray-tracing features were disabled.
- Lumen Reflections were switched to Screen Space Reflections.

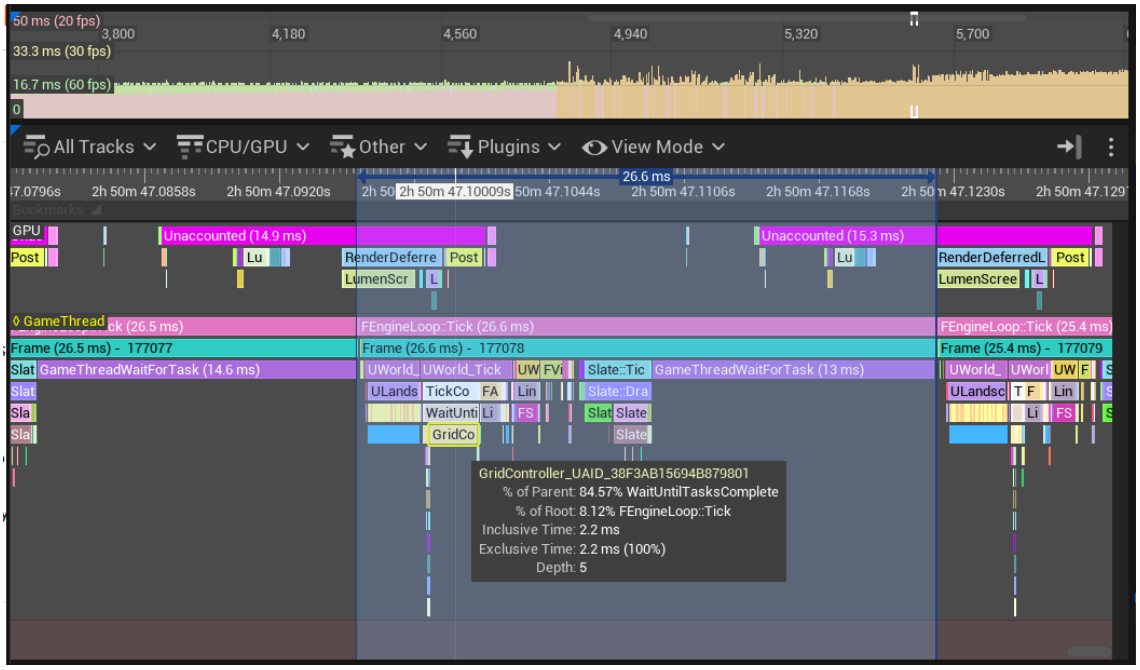


Figure 5.1: A graph from Unreal Insights.

- The engine's renderer was changed from a deferred renderer to a forward renderer.
- Bloom, HDR, and dynamic shadows were disabled.

With these changes, the scene shown in Figure 5.2 was able to run at an average of 11.1 ms, the value recommended for VR, but just barely. The performance would fluctuate between 12 and 10 milliseconds, thus the next step was to reduce the resolution of the VR screen.

By default, SteamVR, which is the program that interfaces the VR hardware with the software running on the machine, uses a resolution of 150% (equivalent to 1852x2056 for each eye). Reducing it down to 100% (1512x1680) allowed the scene to run stably at 10 ms. However, to give the program the best chance it could get, the scene was also tested at 80% (1352x1500), and at 50% (1068x1188). These yielded an average frametime of 9 ms and 8 ms respectively. Figure 5.3 shows the performance graphs of the test scene at different resolutions.

Because only 50% scaling was able to keep the scene fully stable, it was kept for all subsequent tests despite the impacted visuals.

Next, the scene was tested with the grid elements being rendered using debug cubes, to simulate as light a rendering load as possible. Because these tests are supposed to provide a worst-case scenario, the scene was tested with cell grouping deactivated. It took about one thousand on-screen cells to increase the frame rendering time past the 11.1 ms threshold, as shown in Figure 5.4.

Next, it would make sense to simulate the scene using more accurate visuals, so a free asset pack was acquired from the Unreal Marketplace called *M5 VFX Vol2. Fire and*

Figure 5.2: The baseline scene, being rendered in VR.

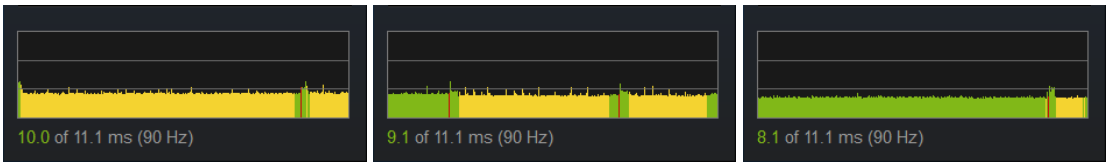


Figure 5.3: These graphs show the scene running at 100%, 80%, and 50% resolution respectively. The graph is split into three sections, with the bottom third representing frame times under 11.1 ms, and the top third representing frame times over 33.3 ms. Green bars represent frames which were able to be produced with the 11.1 ms limit, while yellow bars represent frames which did not, thus the software was required to reproject the previous frame and smooth the transition (to avoid noticeable frame drops).

Flames[29]. This effects pack was chosen because it was free, had fire particles of high quality, had good reviews, and contained both Niagara and Cascade[1] versions of the particle systems, which would allow the performance of both to be compared.

Rendering particle systems actually provides a slight performance boost to the CPU, because rendering debug cubes required iterating through each burning grid element and manually calling the render function. With particle systems, we are only required to create the system when the grid element ignites, and destroy it when the grid element runs out of fuel. The engine handles everything else.

To accomplish this, a pointer to the particle system was added in the `GridElem` struct, and a method to create and destroy particle systems for specific grid elements was added to the `GridController`. These methods are run when the grid element changes state, is

[1]The two particle systems provided in Unreal, described in Section 2.8.2

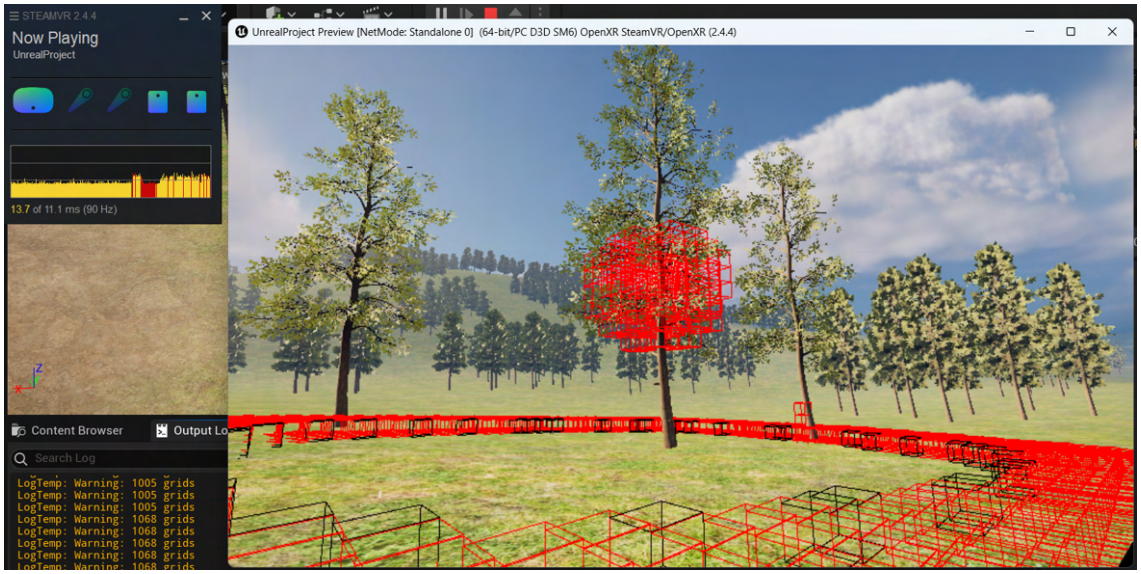


Figure 5.4: The scene running with the simulation enabled. The grid elements are being rendered using debug cubes.

destroyed, or is resized. Since Unreal does not provide an easy method for scaling the visuals of the particle systems at runtime without modifying them to have dynamic user variables, it was decided to keep it easy and just destroy and respawn a particle system but with a size change. Figure 5.5 shows the particle system running for a grid element with size 1, compared to a grid element of size 10.

Figure 5.5: The fire particles, rendering with size 1 and size 10 respectively.

The fire particle effect themselves were composed of multiple parts: a tall flame animation, some small flames that stretch out across the floor, smoke, sparks, view distortion from heat haze, and a dynamic light. Table 5.1 shows the number of particle systems emitters the scene handled before taking longer than 11.1 ms to render a frame,

	Niagara	Cascade
Default	24	31
No Lights	88	103
Only Smoke and Flames	103	112
Only Tall Flames	436	1052

Table 5.1: Table showing the number of grid elements emitting particles before the performance dropped below the 90 FPS recommended for VR.

with different parts disabled. These different quality settings are also demonstrated in Figure 5.6.

From the table, it can be concluded that for simple particle effects, Cascade is the much superior option for the application. It also shows that in a worst case scenario of all particle systems being rendered on screen, the program managed to handle a thousand grid elements at once. What the table also lets us conclude is that having high-quality particles is possible for fires which are happening close to the user, while less quality ones can be used for distant grid elements.

It is possible that more performance can be squeezed by tweaking Unreal Engine settings and modifying the particle system further, but ultimately the importance of the grid grouping algorithm was demonstrated. By keeping the number of visible grids in the scene to a minimum, the quality of the particles can be increased. The program is more likely to be bottlenecked by the GPU than by the CPU, meaning the simulation implementation has been shown to be quite efficient. Of course, better performance can be achieved with a better graphics card, as the 2060 is a mid-range card from the year 2019. But using a more powerful (and expensive) card ends up contradicting one of the goals of the fire visualisation tool, which is that it can run on inexpensive hardware.

5.2 Flexibility

Lastly, it is worth discussing the flexibility and documentation of the implementation. Since the project is ultimately going to be merged into a larger tool by someone else, it was imperative that the final codebase be easy to extend and modify.

The `GridController` was designed from its inception to have lots of its functionality split into several methods, so that when the time comes to port it over to the fire visualisation tool, it could be done without much hassle. Each stage of a `GridElem`'s life is handled by a specific private method, in case anything needs to be modified in the final tool. A collection of public methods, completely unused by the program developed in the dissertation, are also available to facilitate both the integration of the controller as well as test its functionality.

The full API of the `GridController`, including the methods which were left unused in the final program, is described in Listing 5.1.

Figure 5.6: The fire particles, at full quality, smoke and flames only, and tall flames only respectively. Screenshots were taken once the performance had degraded past 16.6 ms.

```
class UNREALPROJECT_API AGridController : public AActor
{
    private:
        void      FindGridNeighbours(GridElem* ge);
        void      GenerateGridNeighbours(GridElem* ge);
        void      AttemptSpreadGrid(GridElem* ge, FBox box, TArray<GridElem*>* created);
        GridElem* CreateGridAtXYZ(FVector pos, ObjectBurn* object = NULL, FVector size=
    ↪ FVector::One());
        bool      AttackGrid(GridElem* ge, GridElem* attacker);
        void      GridBecomeFire(GridElem* ge);
        void      GridBecomeBurnt(GridElem* ge);
        bool      Combine_GridElements(GridElem* ge1, GridElem* ge2);
        bool      Separate_GridElements(GridElem* ge, TArray<GridElem*>* newburning);
        void      FireParticles_Enable(GridElem* ge);
        void      FireParticles_Disable(GridElem* ge);

    public:
        void CreateFireAtXYZ(FVector pos, FVector size = FVector::One());
        void CreateFireFrontFromLines(TArray<FVector>* polygon);
        void CreateFireFrontFromPolygon(TArray<FVector>* polygon);
        FBox GetCameraFrustum(UCameraComponent* camera);
        void FindLandscapeGridsInRect(FBox2D rect, TArray<GridElem*>* found);
        void FindObjectInRect(FBox2D rect, TArray<ObjectBurn*>* found);
        void FindAllElementsInCube(FBox cube, TArray<GridElem*>* found);
        void RemoveGrid(GridElem* ge);
        void RemoveGridAtXY(FVector2D pos);
};
```

Listing 5.1: The class definition of the `GridController`, showing the different methods provided by the API.

While the `GridElem` itself is not a class, a collection of extra functions were developed to facilitate the retrieval of key information about it, manipulate it, and convert between it and UE5's internal `FBox` structure. These functions also serve a second purpose, which is to minimize the changes needed to the program if a future user of the API would need to modify the `GridElem` struct. This API is shown in Listing 5.2.

```
GridElem* CreateGridElem(FVector pos, int firehp, ObjectBurn* object = NULL, FVector size=
    ↪ FVector::One());
ObjectBurn* CreateObjectBurn(OBJType type, FTransform transform, void* object);

double  GridElem_GetLeft(GridElem* ge);
double  GridElem_GetRight(GridElem* ge);
double  GridElem_GetUp(GridElem* ge);
double  GridElem_GetDown(GridElem* ge);
double  GridElem_GetBottom(GridElem* ge);
double  GridElem_GetTop(GridElem* ge);
double  GridElem_GetWidth(GridElem* ge);
double  GridElem_GetLength(GridElem* ge);
```

```
12 double  GridElem_GetHeight(GridElem* ge);
13 double  GridElem_GetWidth_Unscaled(GridElem* ge);
14 double  GridElem_GetLength_Unscaled(GridElem* ge);
15 double  GridElem_GetHeight_Unscaled(GridElem* ge);
16 FVector GridElem_GetMin(GridElem* ge);
17 FVector GridElem_GetMax(GridElem* ge);
18 FVector GridElem_GetArea(GridElem* ge);
19 FVector GridElem_GetArea_Unscaled(GridElem* ge);
20 FVector GridElem_GetCenter(GridElem* ge);
21
22 void GridElem_MakeNeighbour(GridElem* a, GridElem* b);
23 void GridElem_RemoveNeighbour(GridElem* a, GridElem* b);
24 void GridElem_ClearNeighbours(GridElem* ge);
25
26 bool GridElem_IsDamaged(GridElem* ge);
27 bool GridElem_IsOnFire(GridElem* ge, UWorld* world);
28 bool GridElem_IsBurnt(GridElem* ge, UWorld* world);
29
30 // Shape conversion and manipulation
31 FBox2D    FBox2DFromGridElem(GridElem* ge);
32 FBox      FBoxFromGridElem(GridElem* ge);
33 GridElem* GridElemFromFBox2D(FBox2D bound, GridElem* reference);
34 FBox      FBoxFromObjectBurn(ObjectBurn* obj);
35 FBox      FBoxAlignedToObj(ObjectBurn* obj, FBox box);
36 void      AdjustGridElemHeight(UWorld* world, GridElem* ge);
37 BoxSubtractStatus BoxSubtract(FBox minuend, FBox subtrahend, TArray<FBox>* results, bool
       ↪ is2D=false);
38
39 // Unreal helper functions
40 TArray<AActor*> CollidedWithTerrain(UWorld* world, TArray<FOverlapResult> overlaps);
41 bool TraceHitInflammableMaterial(UWorld* world, FHitResult hit);
42 TArray<Combustibles> CollidedWithCombustible(FVector pos, FCollisionShape colshape, TArray
       ↪ <FOverlapResult> overlaps);
43 bool TerrainZAtXY(UWorld* world, FVector2D pos, double* out, double tracez=0, double
       ↪ tracesize=DBL_MAX);
```

Listing 5.2: The list of helper methods for retrieving data from `GridElems`, as well as various helper functions.

Also worth pointing out is that because the `ObjectBurn` struct stores the object it is affecting as a `void*`, it allows the fire simulation to be extended to affect many different types of objects, and all that needs to be extended is some small switch statements inside the `CollidedWithCombustible` and `AttemptSpreadGrid` methods. This is documented in the repository's README.

Lastly, to further facilitate testing and debugging, a single file named `FireConfig.h` was created which features an array of constant values to enable or disable functionality of the grid controller, modify the simulation, or enable various debug features. Every constant in this file contains a small comment which explains what the constant does.

One aspect of the API that could use improvement is that it is, unfortunately, not drag and drop and requires minor tweaking on a per-project basis. Specifically, the bounds of the map need to be set on the grid controller object once placed in the map, as there were difficulties getting the class to dynamically calculate them. These bounds are necessary, because they are used to generate the quadtree.

To assist incoming developers, the API is extensively documented, with explanations avaiable both in the Git repository and in code comments, to help future integration with fire visualisation tool. While the API provides is a simple fire spreading simulation, it is written in a way to allow the grid elements themselves to adapt to the requirements of the fire front polygon in the final tool.

6

Conclusion and Future Work

This chapter consolidates the discussions and conclusions of the work that was developed in this dissertation. We conclude with a list of possible extensions and enhancements.

6.1 Final Overview

The main objective of this dissertation was to improve the visuals of a macro-scale forest fire data visualisation tool, by allowing the fire front to be observed at the micro scale.

Thanks to an implementation of a naïve fire spreading system, it is possible to have very large fires represented by a small number of grid elements. While following really simple rules, the system allows for a plethora of interesting and realistic phenomena to be simulated, such as the possibility of fires spreading from two separate objects, the ease of fires regarding the climbing slopes versus descending, and the inability of fire to spread to inflammable materials. The cell grouping system and provides a method for spatially partitioning the simulation, allowing the controller to skip unnecessary calculations and optimize the scene for what the user is observing.

While the ability to take a set of lines directly from the MACFIRE server and integrate it with the dynamic simulation from the grid controller was not able to be implemented in the given timeframe, a potential solution to the problem was shown, and a suite of functions were written to facilitate the integration with the fire visualisation tool. While the culling methods might result in incongruences with what a user observed in a previous location, the overall macro-scale fire should be unaffected.

The grid controller system provides a solid foundation so that further graphical improvements can be done to the fire visualisation tool. With the provided simulation culling methods, as well as leveraging functionality already present in the API, a developer can implement very realistic fires with smoke and sparks spreading in front of the viewer while keeping the simulation and performance impact light for distant flames.

From the evaluation of the performance of both the CPU and GPU, the overall usability of the fire visualisation tool should be minimally impacted as long as the way in which

particles are rendered is done strategically. The tests showed that the controller can render thousands of flames at once while rendering fires of varying quality, so higher quality particles should be used for close-proximity embers. Virtual Reality is, ultimately, a very performance intensive medium, and care must be taken if one wishes to have a realistic looking scene.

6.2 Limitations and Future Work

When comparing the finished project to the original stated objectives, the key point that remains is the integration of the project with the existing fire visualisation tool. This point is the natural next step for the project.

Additionally, there are a number of possible optimisations and improvements which can be done to the existing system:

1. Unreal Engine provides methods for taking advantage of multi-core processors. It would be interesting to attempt a parallelization of the `GridController`'s main tick loop, as it currently sequentially iterates through each burning grid element to perform its necessary logic. The challenge of this step would be that the neighbour states might not be fully synchronized (for instance, one grid element can ignite a damaged cell, but another grid element that shares the same neighbour might not have enough strength to do so), so the merging of data must be done carefully.

2. Currently, the system was only tested with a single particle system. It would be interesting to extend this further by having multiple different particle systems for different situations a grid element might find itself in. For instance, a grid element which is attempting to climb foliage (such as a tree) could have a slightly warped appearance to the direction it is climbing. Fire particles could also be tilted in order to look like they are being pushed by the global wind value.

3. UE5 allows for the dynamic modification of assets. Since the grid element is a box, one can query the materials which intersect it, and modify them accordingly. For instance, the leaves on trees could be made invisible by querying the triangles that are overlapped by a grid element, and the triangles belonging to a tree's trunk can be swapped for a charred texture. By also leveraging the ability to send custom data to material shaders, one could also implement smoothing between texture changes.

This dissertation approached the problem by developing a micro-simulation of a fire with individual grid elements performing their own discrete logic. However, another equally valid approach could be performed taking the polygon of the fire front, placing it on the world, and then dividing the space in quarters. Said spaces would be further subdivided as the difference between the fire front, the burnt areas, and the unburnt areas starts to converge. This would result in a partition not too dissimilar to a quadtree. Then,

as data is queried from the server regarding the fire front's next position, the partitions are modified to correlate with the new polygon. The key difference between this approach is that the cells do not behave individually to converge towards the fire front's polygon (so no simulation is occurring), rather they are always forcibly changed as the data from the fire front polygon comes in.

Lastly, it would also be interesting to provide a selection of different graphics settings and particle systems, and then perform a collection of user tests to query which graphical options are preferred.

Bibliography

[1] *AMD FidelityFX™ Super Resolution*. AMD. URL: https://www.amd.com/en/technologies/fidelityfx-super-resolution (cit. on p. 5).

[2] *ArcGIS*. ESRI (cit. on p. 5).

[3] *Asynchronous Reprojection*. Google VR. URL: https://developers.google.com/vr/discover/async-reprojection (cit. on p. 5).

[4] Avsa. *World elevation map*. Wikimedia Commons. 2023-07. URL: https://commons.wikimedia.org/wiki/File:World_elevation_map.png (cit. on p. 6).

[5] J. L. Bentley. "Multidimensional binary search trees used for associative searching". In: vol. 18. 9. New York, NY, USA: Association for Computing Machinery, 1975-09, pp. 509–517. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007 (cit. on p. 17).

[6] W. M. N. Bob Sproull. "Principles of Interactive Computer Graphics". In: McGraw-Hill Education, 1979. ISBN: 0070664552 (cit. on p. 58).

[7] M. B. B. Carlos. "Data For VR: Visualização e Interação com informação temporal geoespacial em mundos virtuais". MA thesis. Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2021 (cit. on pp. 6, 7).

[8] "Combating VR Sickness: Debunking Myths and Learning What Really Works". In: (2018-03). URL: https://vr.arvilab.com/blog/combating-vr-sickness-debunking-myths-and-learning-what-really-works (cit. on p. 4).

[9] V. Corporation. *Steam Hardware & Software Survey: January 2023*. Steam. 2022-01. URL: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam (cit. on p. 26).

[10] *Deep Learning Super Sampling (DLSS) Technology*. NVIDIA. URL: https://www.nvidia.com/en-eu/geforce/technologies/dlss/ (cit. on p. 5).

[11] N. Donn. "Wildfires burn 106,500 hectares in Portugal by end August". In: (). URL: https://www.portugalresident.com/wildfires-burn-106500-hectares-in-portugal-by-end-august/ (cit. on p. 1).

[12] R. Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. London: Addison-Wesley Professional, 2004 (cit. on p. 8).

[13] *Fire*. Minecraft Wiki. URL: https://minecraft.fandom.com/wiki/Fire (cit. on p. 13).

[14] *Foliage Tool*. Unreal Engine 4.27 Documentation. URL: https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/Foliage/ (cit. on p. 6).

[15] D. Foundry. *Teardown Is Stunning: Next-Generation Physics + Lighting!* YouTube. 2020. URL: https://youtu.be/7Tu4u_feR4g?t=271 (cit. on p. 14).

[16] A. Fuller et al. "Real-time procedural volumetric fire". In: 2007-04, pp. 175–180. DOI: 10.1145/1230100.1230131 (cit. on p. 9).

[17] E. Games. *Actor Ticking*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/actor-ticking-in-unreal-engine (cit. on p. 45).

[18] E. Games. *Landscape Technical Guide*. URL: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/Landscape/TechnicalGuide/ (cit. on pp. 18, 30).

[19] E. Games. *Landscape Technical Guide*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/landscape-technical-guide-in-unreal-engine?application%5C_version=5.0 (cit. on p. 30).

[20] E. Games. *Level Streaming*. URL: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LevelStreaming/ (cit. on p. 18).

[21] E. Games. *Procedural Foliage Tool*. URL: https://docs.unrealengine.com/5.0/en-US/procedural-foliage-tool-in-unreal-engine/ (cit. on p. 19).

[22] E. Games. *Unreal Insights*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-insights-in-unreal-engine?application%5C_version=5.1 (cit. on p. 27).

[23] E. Games. *World Composition*. URL: https://docs.unrealengine.com/5.0/en-US/world-composition-in-unreal-engine/ (cit. on p. 18).

[24] E. Games. *World Partition*. URL: https://docs.unrealengine.com/5.0/en-US/world-partition-in-unreal-engine/ (cit. on p. 18).

[25] *Grass Quick Start*. Unreal Engine 4.27 Documentation. URL: https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/OpenWorldTools/Grass/QuickStart/ (cit. on p. 6).

[26] T. Hädrich et al. "Fire in Paradise: Mesoscale Simulation of Wildfires". In: *ACM Trans. Graph.* 40.4 (2021-07). ISSN: 0730-0301. DOI: 10.1145/3450626.3459954. URL: https://doi.org/10.1145/3450626.3459954 (cit. on p. 11).

[27] Y. He. "A Physically Based Pipeline for Real-Time Simulation and Rendering of Realistic Fire and Smoke". 2018 (cit. on pp. 8, 9).

[28] A. M. Helmenstine. *What state of matter is fire or flame?* ThoughtCo. 2020-01. URL: https://www.thoughtco.com/what-state-of-matter-is-fire-604300 (cit. on p. 8).

[29] JeongukChoi. *M5 VFX Vol2. Fire and Flames*. URL: https://www.unrealengine.com/marketplace/en-US/product/m5-vfx-vol2-fire-and-flames (cit. on p. 63).

[30] M. de Kruijf and M. Schneider. "firestarter – A Real-Time Fire Simulator". In: 2007 (cit. on p. 10).

[31] J.-F. Levesque. *Far Cry: How the Fire Burns and Spreads*. Jean-Francois Levesque | Engineering Manager. 2012-12. URL: https://jflevesque.com/2012/12/06/far-cry-how-the-fire-burns-and-spreads/ (cit. on p. 12).

[32] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[33] *Map Projections*. Axis Maps (cit. on p. 5).

[34] S. Marius Wolfensberger Zürich. "Improving the Performance of Region Quadtrees". In: 2013-04, pp. 12–16. URL: https://www.ifi.uzh.ch/dam/jcr:ffffffff-96c1-007c-ffff-fffff2d50548/ReportWolfensbergerFA.pdf (cit. on p. 17).

[35] D. Meagher. "Geometric modeling using octree encoding". In: vol. 19. 2. 1982, pp. 129–147. DOI: https://doi.org/10.1016/0146-664X(82)90104-6. URL: https://www.sciencedirect.com/science/article/pii/0146664X82901046 (cit. on p. 17).

[36] R. M. Negrão. "Geração automática da vegetação a partir de informação em SIG". MA thesis. Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2022 (cit. on pp. 6, 7).

[37] D. Nguyen, R. Fedkiw, and H. Jensen. "Physically Based Modeling and Animation of Fire". In: *ACM Transactions on Graphics* 21 (2002-05). DOI: 10.1145/566570.566643 (cit. on p. 8).

[38] *Niagara fluids in Unreal Engine*. Unreal Engine 5.1 Documentation. URL: https://docs.unrealengine.com/5.1/en-US/niagara-fluids-in-unreal-engine/ (cit. on p. 19).

[39] P. Pang. *NVIDIA VRSS 2: Dynamic Foveated Rendering, No Assembly Required*. NVIDIA. URL: https://developer.nvidia.com/blog/nvidia-vrss-2-dynamic-foveated-rendering-no-assembly-required/ (cit. on p. 4).

[40] V. Pegoraro and S. G. Parker. "Physically-Based Realistic Fire Rendering". In: *Eurographics Workshop on Natural Phenomena*. 2006 (cit. on p. 8).

[41] S. Pirk et al. "Interactive Wood Combustion for Botanical Tree Models". In: *ACM Trans. Graph.* 36.6 (2017-11), 197:1–197:12. ISSN: 0730-0301. DOI: 10.1145/3130800.3130814. URL: http://doi.acm.org/10.1145/3130800.3130814 (cit. on p. 11).

[42] Pramath. "Sony explains how checkerboard rendering works on the PS4 pro". In: *GamingBolt* (2016-10). URL: https://gamingbolt.com/sony-explains-how-checkerboard-rendering-works-on-the-ps4-pro (cit. on p. 4).

[43] *QGIS - A Free and Open Source Geographic Information System*. QGIS (cit. on p. 5).

[44] J. L. B. Raphael A. Finkel. "Quad Trees: A Data Structure for Retrieval on Composite Keys". In: 1974-03, pp. 1–9. DOI: 10.1007/BF00288933. URL: https://doi.org/10.1007/BF00288933 (cit. on p. 16).

[45] *Render 3D Imposter Sprites*. Unreal Engine 4.27 Documentation. URL: https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/RenderToTextureTools/3/ (cit. on p. 20).

[46] F. Schroeter. DE574085C (cit. on p. 4).

[47] K. Shinkle. "PS5 explained: What resolution & performance mode differences are". In: *ScreenRant* (2021-08). URL: https://screenrant.com/ps5-resolution-performance-mode-visual-differences-fps/ (cit. on p. 4).

[48] J. Stam. "Real-Time Fluid Dynamics for Games". In: (2003-05) (cit. on p. 8).

[49] *What is GIS? Geographic Information Systems*. GISGeography. 2024-03. URL: htpps://gisgeography.com/what-is-gis/ (cit. on p. 5).

[50] WhiteTimberwolf. *Octree*. Wikimedia Commons. 2010-10. URL: https://commons.wikimedia.org/wiki/File:Octree2.svg (cit. on p. 17).

[51] *World Composition*. Unreal Engine 5.0 Documentation. URL: https://docs.unrealengine.com/5.0/en-US/world-composition-in-unreal-engine/ (cit. on p. 6).