DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Volume Rendering Simulation in Real-Time

KTH Thesis Report

Enrique Perez Soler

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Enrique Perez Soler <eps@kth.se>
Information and Communication Technology
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
Fatshark AB

Examiner

Elena Dubrova
Place
KTH Royal Institute of Technology

Supervisor

Huanyu Wang
Place
KTH Royal Institute of Technology

Industrial Supervisor

Axel Kinner

Place

Fatshark AB

Abstract

Computer graphics is a branch of Computer Science that specializes in the virtual representation of real-life objects in a computer screen. While the past advancements in this field were commonly related to entertaining media industries such as video games and movies, these have found a use in others. For instance, they have provided a visual aid in medical exploration for healthcare and training simulation for military and defense.

Ever since their first applications for data visualization, there has been a growth in the development of new technologies, methods and techniques to improve it. Today, among the most leading methods we can find the Volumetric or Volume Rendering techniques. These allow the display of 3-dimensional discretely sampled data in a 2-dimensional projection on the screen.

This thesis provides a study on volumetric rendering through the implementation of a real-time simulation. It presents the various steps, prior knowledge and research required to program it. By studying the necessary mathematical concepts including noise functions, flow maps and vector fields; an interactive effect can be applied to volumetric data. The result is a randomly behaved volumetric fog. The implementation issues, intricacies and paradigms are reflected upon and explained. The conclusion explores the results while these illustrate what can be accomplished using mathematical and programming notions in the manipulation of graphic data.

Keywords

Volume Rendering, Real-time Simulation, Computer Graphics (, Shader)

Abstract

Datorgrafik är en filial inom datavetenskap som specialiserar sig på realistisk virtuell representation av verkliga objekt på en datorskärm. Medan de tidigare framstegen inom detta område vanligtvis var relaterade till underhållande medieindustrier som videospel och filmer, hade dessa hittat en annan användning i andra. De har tillhandahållit ett visuellt stöd inom bland annat sjukvård, arkitektur, militär och försvar och utbildning.

Helt sedan deras första applikationer för visualisering av data har det skett en tillväxt av ny teknik, metoder och tekniker för att förbättra datarepresentationen. Idag, bland de mest ledande metoderna, kan vi hitta Volumetric eller Volume Rendering-tekniker. Dessa möjliggör visning av 3-dimensionell diskret sampling av data i en tvådimensionell projektion på skärmen.

Denna
avhandling ger en studie om volumetrisk rendering genom en realtidsimulering. Den presenterar de olika stegen, kunskapen och forskningen som krävs för att simulera den med hjälp av de matematiska grunderna som gör det mycket eftertraktat och även beräkningsproblematiskt. Implementeringsfrågor, komplikationer och paradigmer återspeglas och förklaras medan resultaten illustrerar vad som kan åstadkommas med matematiska och programmeringsmeddelanden när de tillämpas på manipulering av grafisk data.

Nyckelord

Volymåtergivning, realtidssimulering, datorgrafik

Acknowledgements

This project was made possible thanks to the support of my KTH examiner Elena Dubrova and KTH supervisor Huanyu Wang; and to the opportunity being granted by Fatshark Games as well as the assistance received from my supervisor and friend there Axel Kinner, Peder Nordenström and all the employees at Fatshark who have been welcoming and caring. Furthermore, it goes without saying that my deepest gratitude goes to KTH Visualization Studio Project Manager, teacher and mentor of mine Björn Thuresson. His guidance and aid promoted my dream of entering the world of Computer Graphics which this experience so successfully contributed to.

Acronyms

CPU Central Processing Unit

GPU Graphic Processing Unit

ICT Information and Communication Technologies

GLSL OpenGL Shading Language

VS Vertex Shader

PS Pixel Shader

GDC Game Developers Conference

SIGGRAPH Special Interest Group on Computer GRAPHics and Interactive Techniques

Contents

Chapter 1

Introduction

In what could be referred to as the building blocks of the Information and Communication Technologies (ICT), there is a particular Computer Science sub-field that is dedicated entirely to the generation and manipulation of graphic data. This sub-field is called Computer Graphics.

The multifaceted potential of visualization media has taken over many aspects of people's everyday life. It has been incorporated into many software and hardware projects targeted at different professions. Specially to those focused on providing user experience and visual interaction. Projects with different backgrounds ranging from visually aiding scientific or medical investigation to providing a learning platform for educational and technical purposes. Like pilot training exercises in military and defense.

However, its primary use was found in multimedia entertainment industries like movies or *video games*. These have evolved rapidly since their first appearance in 1958 and were publicly consolidated at the hands of Atari Inc.'s *Pong* game in the 1970s. With each step and new title released showing a higher improvement of the display quality of graphic data in terms of realism and complex representations. Most of these were focused on providing a believable and realistic synthesization of real-life elements such as trees, plants, buildings, cars, etc.

All of these were made possible thanks to the frequent advancements on the Graphic Processing Unit (GPU). Over the past five years, the rendering rate[1], as measured

[1]The speed in which the 3D data is transformed into a screen displayable 2D image through the process of rendering

in pixels per second, has been progressively doubling, increasing the performance ten times. Additionally, the quality of computation and the flexibility of graphics programming have steadily improved during that same time. Facts that were unbelievable back then as PCs and computer workstations had graphics accelerators, not GPU centered entirely to the creation of graphics [4]

1.1 Background

This project's implementation is mainly cemented on the principles of computer graphics' shaders, computer graphic's rendering pipeline through the rendering engine[2] and real-time simulation. From these aspects and concepts will sprout the different software techniques, methods and implementations that would define the functionalities researched for completing it.

A computer graphics' shader or shader program is a computer program that contains the mathematical calculations implemented in one of its possible shading languages[3] to transform 3D data into its 2D representation. The mathematical calculations involve the use of linear algebra by performing transformations such as rotations, scaling and translations over vectors, matrices and even pixels.

The Computer Graphics Pipeline is a modelization of the steps a software environment oriented at implementing graphics, i.e. a rendering engine, needs to perform to transform 3D data to 2D-representations to display on the screen. This model is divided in three main components as seen in Figure 1.1.1 and is commonly used in real-time rendering. The host company provided the *Autodesk Stingray* renderer as well as the necessary tools to develop the simulation of the project.

[2]Also called ”renderer”, it is one of the core functionalities that form the game engine, software-development environment designed for people to build video games.

[3]They are divided into two types depending on whether the rendering is done offline or in real-time. Real-time rendering examples include: OpenGl shading language(GLSL) and High-Level Shader Language (HLSL)

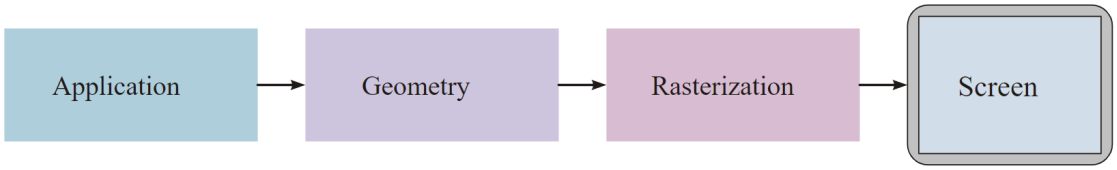


Figure 1.1.1: Graphics Pipeline main parts as portrayed in Wikipedia

The thesis of the project pretends to research on and implement a real-time simulation. These have the complexity of having to perform all the executions at the same rate of the computer's performance. They must execute ready-to-go changes in the simulated environment without having prepared them outside of the normal working clock of the computer. This supposes a challenge as well as an opportunity to learn more of these kind of simulations.

In order to develop such simulation, there must be an understanding on concepts related to virtual physics systems like Flow Maps[4], Vector Fields for aerodynamics' study and Height Maps as these will take part in its implementation. These will be explained in more depth in its due time further on. Furthermore, all the background notions that inspired many of the research areas will be explained more thoroughly with additional references to previous related work in section 2.1. Below you will find a table with information about the software tools required and used for the simulation.

Development environments for Real-Time Simulation
Computer Graphics Development Environment
Game Engine: Autodesk Stingray and Autodesk Bitsquid (discontinued)
Programming Languages
Main: Lua
Graphics: Direct3D High-level Shading Language (HLSL)

[4]In cartography, flow maps are a mix of maps and flow charts, that show the movement of objects from one location to another. However, in computer graphics it is a 2D texture that would be applied to an object, normally a fluid, that contains the flow directions that the fluid will follow

1.2 Problem

There are many problems in the area of Computer Graphics that contribute to the exhaustion or other technical issues that harm the efficiency and increase the payload of the Central Processing Unit (CPU) and GPU. The resolution of these issues are normally assigned to different subareas of expertise based on their nature. Some are encountered in the movie industry when attempting offline rendering techniques to improve realism and others are found in the gaming industry while researching real-time rendering techniques. The Special Interest Group on Computer GRAPHics and Interactive Techniques (SIGGRAPH) is annually organised for companies to provide their advancements in the area and also provide issues that are holding them back.

Some of the issues presented in the past years [1] revolved around rendering subfields like *Physically-based rendering*, which is an approach that attempts to render graphics more photorealistically[5], *Global Illumination*(GI) algorithms based on the basic physics notions behind light reflection and refraction and meant to add more realistic lighting to 3D simulated environments; and finally the issues caused by the application of deep-learning[6] to real-time graphics simulation.

Among some of the most complex of them all we can find those produced by real-time rendering. These kind pose a heavier number of calculations and instructions for the GPU when simulating the graphics in real-time. One of the most recent challenges found in this subfield [8] was the possibility of simulating physically based animations of fluids such as smoke, water, and fire and how they can be integrated into real-time applications. These effects have previously belonged to the offline high-definition rendering domain due to its great computational cost and because the volumetric data produced does not fit easily into the standard rasterization-based rendering paradigm.

However, the main issues are explained perfectly in Chapter 39 of GPU Gems [8] where the limitations in volumetric rendering that create issues are listed and explained in more depth. Among them we can find the following performance considerations:

- **Rasterization Bottlenecks**

[5] An accurate representation of the flow of light in the real world. Seeming as lifelike as possible

[6] Subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called *artificial neural networks*.

Unlike most graphics applications, texture-based volumetric renderers use a small number of relatively large geometric primitives. The rasterizer produces many fragments per primitive, which can easily become the bottleneck in the pipeline.

- **Fragment Program Limitations**
  The performance of volumetric rendering is highly due to the complexity of the fragment shader. This is the *OpenGL*[7] pipeline stage after a primitive is rasterized. For each sample of the pixels covered by a primitive, a ”fragment” is generated. Each fragment has a screen space position, a few other values, and it contains all of the interpolated per-vertex output values from the last Vertex Processing stage. However, texture reads can result in pipeline stalls, reducing rendering speed significantly. Achieving peak performance requires finding the correct balance of texture reads and fragment operations, which can be a challenging profiling task.

- **Texture Memory Limitations**
  The trilinear interpolation used in volume rendering requires at least eight texture lookups processes, making it more expensive than the bilinear interpolation used in standard 2D texture mapping. Moreover, when using large 3D textures, the texture caches may not be as efficient at hiding the latency of memory access as they are when using 2D textures. When the speed of rendering is critical, smaller textures, texture compression, and lower precision types can reduce the pressure on the texture memory subsystem.

However, the question that this thesis presents is none that need be answered directly. It is not trying to prove, test or solve any previously established problems present in the issues aforementioned. By using an empirical research method, an investigation is carried out. Findings, advancements and current technologies and their specifications are researched and observed to understand their nature. Not in the sense of studying and analysing the performance issues or efficiency but by compiling the procedural steps and the background research needed to carry out a similar experiment.

In other words, the question to be answered is none that is born from the need to question the technology but rather a formulation of many others that provide an

[7] A cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics

understanding of the work being done. As an observatory and experience-driven research that it is. But if a question had to be synthesized to include all others it would be something along the lines of *What is needed, in terms of knowledge and technology, to implement a real-time simulation of volumetrically rendered data?*

1.3 Purpose

This thesis provides a study on real-time simulation. It presents the various steps, knowledge and research required to generate a simulation that utilises the basic concepts of volume and graphics rendering. Through the intricacies and problems that appeared throughout the implementation, there can be found the lessons learned and the progression that took place.

It illustrates the results that can be accomplished using mathematical and programming notions applied to the manipulation of graphic data. Also, it means to present a personal journey of discovery to unravel the mystery behind computer graphics data display.

The main goal is the successful implementation of a volume rendered data real-time simulation. This data is provided by the company as well as the tools necessary to carry out the simulation. The thesis is meant to use these and the research carried out of background knowledge in shaders, linear algebra, fluid mechanics and vector fields to implement it.

The result of the project lies not on the simulation itself but also on the research carried out to understand it. To further understand what is needed and what it would take to render or represent graphic data in real-time. The deliverables will include visual media like demonstration videos and images aside from the code. All pertaining to the capturing of the simulation to showcase its gradual evolution throughout its development. These will be relevant to visualize the different stages and phases of the research and the conclusions and discussions resulted in their evolution.

1.4 Benefits, Ethics and Sustainability

The degree project provides a window into the complex and ever-changing world of computer graphics' real-time rendering. It doesn't contain any ethical or sustainability

issues *per se* as it focuses solely on the knowledge needed to generate such technology.

The perspective is borrowed from that of someone who would be inexperienced or a neophyte to this field. For the kindred minded, this thesis would serve as an introduction or experimental attempt in the implementation of a project belonging to a different realm of the ICTs.

For those finding an interest in this subfield, this thesis would hopefully provide a clear explanation on real-time simulation, volume rendered data, shader coding and mathematical notions in an applied use case scenario. For advanced or well-versed individuals it might merely reflect the inexperience of a novice. Or perhaps a window to a simpler time and a retreading of basic common procedures and implementations where they once started with.

1.5 Methodology

The proceeding of *Portal of Research Methods and Methodologies for Research Projects and Degree Projects* by Anne Håkansson [7] considers the different possible research methods that can be carried out in a project. A research method establishes the framework for the research and the research strategies are the steps carried out to conduct it.

The most common research methods are:

- **Experimental research** method studies causes and effects while dealing with and establishing variables and relationships between them to find causalities. This research method is often used when investigating systems' performances.
- **Non-experimental research** method examines existing scenarios and draws conclusions from the situations. It's a more user-experience oriented method as it tries to predict behaviour or opinions but does not set out or test causalities between variables. It is mainly used for studying users' behaviour or opinions over systems.
- **Descriptive or Statistical Research** method studies phenomena and describes characteristics for a situation but not its causes or occurrences. It uses either quantitative or qualitative methods such as surveys, case studies and

observations to produce accurate representations of people, events or situations. It can be used for all kinds of research or investigations in computer science genre that aim to describing phenomena or characteristics.

- **Analytical Research** method tests pre-existing hypotheses. It focuses on validating them by performing a critical analysis using facts and knowledge already collected about the topic. The method aids decision making processes in areas such as product design and process design.
- **Fundamental Research** method is an applied research based on the interest and observation to obtain new insights into the nature of the topic being researched. It creates new ideas, principles and theories focused on innovating and solving previously established problems. It's the most common and basic method of research.
- **Applied research** method involves answering specific questions or solving known and practical problems. It focuses on applying existing research, data and circumstances to solve problems and develop practical applications and technology.
- **Empirical research** method uses experiences and observations to derive knowledge and test predictions by focusing on real people and situations. It forms a body of well-formed theories by collecting and analysing data and experiences which it later uses to evaluate and reveal relationships between technologies and practises by means of experimentation. The data is then analysed with either quantitative or qualitative methods to explain the intrinsic situations.

As for the research approaches, they are used for drawing conclusions and establishing what is true or false. The most common are the *inductive* and *deductive* approaches but there are hybrid approaches like *abductive*.

The *inductive* approach establishes a general proposition from particular facts and formulates theories and propositions with alternative explanations from observations and patterns. Data is collected, commonly with qualitative methods, and analysed to gain an understanding of phenomenon and establishing different views of the phenomenon. The outcome is based on behaviours, opinions, and experiences and must contain enough data to establish why something is happening, which are the

reasons for the theories or requirements for an artifact.

The *deductive* approach on the other hand derives conclusions from known premises and theories and it tests them to verify or falsify hypotheses. It mostly uses quantitative methods with large data sets to test them and the outcome is a generalisation that must be based on the collected data establishing what is happening, aside from the explanations of causal relationships between variables.

The *abductive* approach uses both deductive and inductive approaches to establish conclusions from an incomplete set of observations. The approach starts with an incomplete set of data or observations and uses preconditions to infer or explain conclusions. The outcome is a likely or possible explanation and is, hence, useful as heuristics.

A detailed description of the different research strategies and methods are analysed more in depth in Chapter 3. However, the research method followed will be the empirical. This consists of investigating the findings, advancements and current technologies that allow the implementation of the described simulation. It requires the collecting and understanding of the data, specifications and other paradigms of the subfields and areas relevant of computer graphics. It then reflects on the different studies and research carried out and also mentioned throughout this report such as Philip Sköld's Volumetric Lighting paper [17], Ubisoft's Volumetric Fog proceeding at SIGGRAPH 2014 [21] or Siim Raudsepp's research on Volumetric fog rendering [14]. Moreover, it applies notions of the mathematical foundation of previous investigations by other experts in the field like Inigo Quilez [13] or the *Book of Shaders* [11]. All while compiling the procedural steps and different issues encountered along the way when carrying out such a procedure.

The research approaches to these are at times inductive and other times deductive. When the issue or step at hand is not been understood, a deductive approach to its hypothesis is taken using the previously acquired knowledge to solve it. However, if it is understood but it is not clearly explained why it works or its reason to exist an inductive approach is applied where an inference or an analysis of the procedure taken is carried out.

1.6 Stakeholders

As previously mentioned, the people that will be affected by this research are the ones involved in the implementation and innovation of both this particular subarea and overall Computer Graphics subfield.

My experience in the company of the same industry has lead me to believe those who work in this subfield have an interest in its different subareas. The main reason is their jobs generally require them to adapt and develop similar technology in a case by case basis. Moreover, due to the mathematical foundations of most of the concepts and principles found throughout this research, these can be applied or adapted to other projects. Proof of that can be found on the different subareas of the references that were taken as inspiration.

Nevertheless, the focus and tone of this report is meant to serve as an insightful first experience-based compilation of all the knowledge needed to implement the simulation at hand. It would serve as a template or guideline to interested parties who, like me, have found an interest in developing or learning more about such technology.

1.7 Delimitations

The study is mainly limited by the time provided to complete the thesis. Not in terms of a regulation that is limiting but rather the fact that it is a wide and extensive area. So a complete compilation of all the knowledge needed to fully explain all the intricacies of the subfield being researched requires more time.

Furthermore, the level of expertise and familiarity with the researched area may have taken part in the lack of efficiency during the research. The scope set at the beginning of the thesis was significantly bigger only reduced by the level of expertise needed to fulfill the project. Through the research and learning of the different tools and concepts, a notion of the complexities and unfamiliar mathematical principles deemed the project initially unreachable.

But after the assistance of my company supervisor and other experts of the field, we refocused the scope of it and managed to synthesize a well-rounded and insightful thesis. One that would aid anyone interested in learning about its field of research

and the different issues than can arise in it.

1.8 Outline

The following chapters will compile the relevant summarized notions and practises that directly and indirectly influenced the research carried out.

Starting with background information of Volumetric rendering and Real-Time simulation in Chapter 2. Their mathematical foundations and its applications to similar projects have influenced the present thesis. Projects that helped in the comprehension of this background knowledge but also provided a basis or template to our own implementation.

Later, moving on to the research methods being applied in Chapter 3 and followed by the step by step implementation done in Chapter 4. Finally, chapters 5 to 6 will go through the overall results and conclusions of the project itself.

Chapter 2

Volumetric rendering and Real-Time simulation: Rendering 3D fluids

Fluid simulation constitutes the building block of different types of natural phenomena simulations ranging from water to smoke or, in our particular case, fog. Because of the large amount of parallelism in graphics hardware, the simulation we describe runs significantly faster on the GPU than on the CPU[4].

2.1 Fluid Simulation Principles

While a thorough explanation of fluid simulation is presented in this chapter most of it was mainly used for understanding its behavior. Moreover, its research did not apply directly to the implementation of the project except for specific concepts which will be highlighted when explained. Furthermore, most of the explanations are referred to the related work or similar research papers.

2.1.1 Chapter 38 of NVIDIA'S GPU Gems Book

There are two basic elements that need to be established in order to represent a realistic fluid simulation.

- A mathematical representation of the state of the fluid through time.
- The velocity of the fluid

The velocity is of paramount importance as it would determine the behavior of the fluid

and its interactions with the environment. It varies in time and space so the perfect way to represent it is through a *vector field*[1] as shown in figure 2.1.1

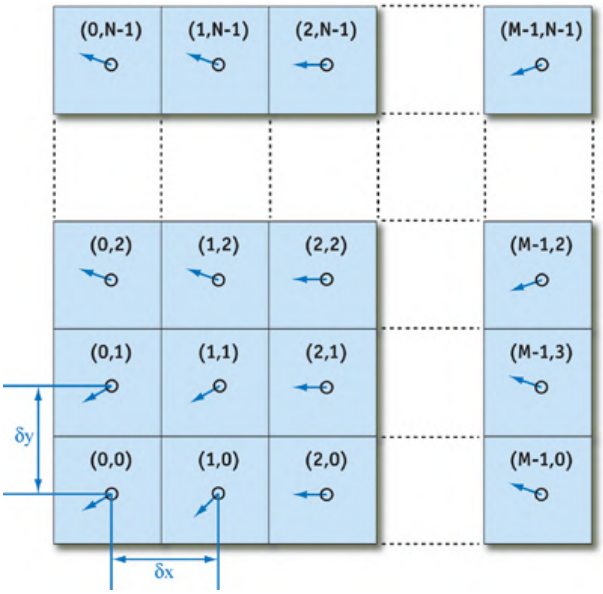


Figure 2.1.1: A reproduction of the Vector field representation from the book. Adapted from *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics* (Figure 38-2), by R. Fernando

Where for every position ***(x,y)*** there's an associated velocity at time ***t***. The time parameter is used to calculate the vector field on each state of the fluid. These calculations involve solving the *Navier-Stokes Equations* for incompressible and homogeneous fluids.

According to R. Fernando,

> *A fluid is incompressible if the volume of any subregion of the fluid is constant over time. A fluid is homogeneous if its density, is constant in space. The combination of incompressibility and homogeneity means that density is constant in both time and space. These assumptions are common in fluid dynamics, and they do not decrease the applicability of the resulting mathematics to the simulation of real fluids, such as water and air.*

We won't go deeper over the explanation of the equations as its use is limited to the understanding of the sort of algorithm needed to proceed. To solve these equations it is needed an algorithm that can monitor the evolution of the flow over time. Therefore, there's a need for an incremental numerical solution.

[1]A mapping of a vector-valued function onto a parameterized space, such as a Cartesian grid[4]

Such solution can be found through the *Helmholtz-Hodge Decomposition* algorithm which finds its basis in vector decomposition and applies it for vector field decomposition.

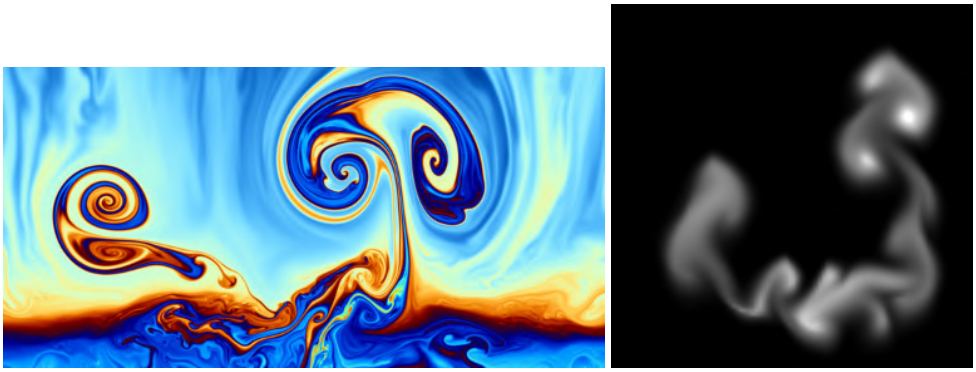


Figure 2.1.2: Examples of 2D fluid simulations over a range of colored pixels

What is important to pick up from these equations and algorithm is not as quite their mathematical basis but rather the pseudo code structure. The volumetrically rendered data is treated as a fluid for it finds similarities to their behavior but it really isn't considered as such conventionally. This data is provided by the company and it holds certain properties that prevents it from behaving entirely as a fluid. Therefore, these principles are studied and applied to our data in a particular way. Leaving some aspects and considerations out of it as they are of no use to our particular type of data. As mentioned previously, further explanations over the different subareas of computer graphics rendering will particularized the use of fluid simulation background.

2.2 3D Fluids Rendering and Real-Time Simulation

As previously seen, the implementation of a 3D fluid simulation is costly in terms of performance and capabilities of the GPU. Mathematically, the most important notion to have in mind is the partial differential equations and the forward Euler integration method. This is used for establishing the rates at which variables change in time.

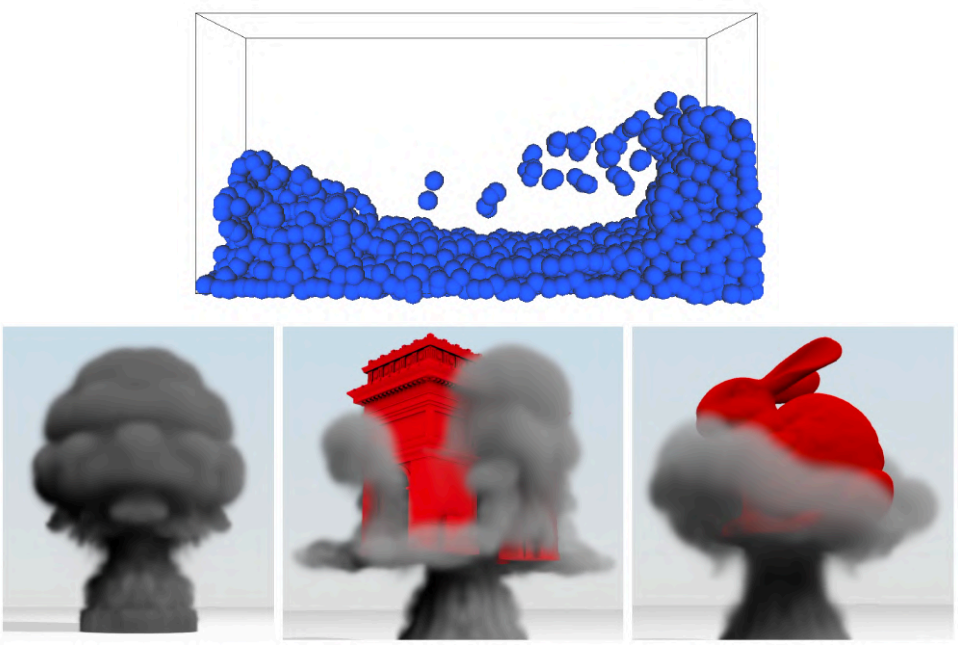


Figure 2.2.1: Examples of 3D particle fluid and smoke simulations

With this concepts we can represent the velocity of a fluid as a function dependant of time and space (position). This function can be given by the *momentum equation*[2]

An clear example can be found in Chapter 30 of Hubert Nguyen's *GPU Gems 3* [8] where the velocity function $\boldsymbol{u}$ can be represented as the momentum equation:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{f},$$

Where $\boldsymbol{p}$ is the pressure or mass density, $\boldsymbol{f}$ represents an external force such as gravity and the upside down triangle is the differential operator given by:

$$\left[\frac{\partial}{\partial x} \quad \frac{\partial}{\partial y} \quad \frac{\partial}{\partial z}\right]^T.$$

The main goal is to compute a numerical approximation of $\boldsymbol{u}$ in the form of a velocity field with which we can be able to simulate movement. Additionally, it will be necessary to define the *boundary conditions* that establish how it will behave near solid obstacles.

[2] A statement of Newton's Second Law and relates the sum of the forces acting on an element of fluid to its acceleration or rate of change of momentum.

To approach the representation of the fluid, a scheme based on the one in section 2.1 but modified to a 3D representation can be applied. A way to represent the simulation is to consider a rectilinear volume containing the space the fluid occupies. It can be subdivided into a grid of cells with a cubic form as shown in figure 2.2.2. This way it's easier on the GPU as the mapping between grid cells and voxels[3] in 3D is straight forward.

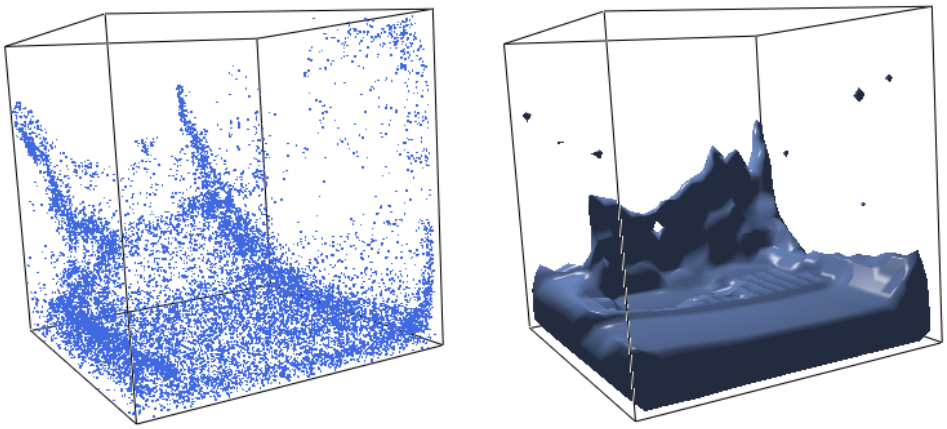


Figure 2.2.2: Example of a 3D grid-based real-time fluid simulation

Each grid cell stores both scalar quantities (such as pressure, temperature, and so on) and vector quantities (such as velocity). At each simulation step, we update these values by running computational kernels over the grid. A kernel is implemented as a pixel shader program that executes on every cell in the grid and writes the results to an output *texture*. A 2D or 3D texture is a bitmapped image where the values of the color, brightness, saturation and such of every pixel is stored. 3D textures usually have sliced-up parts of a 3D volume. Because GPUs are designed to render into 2D buffers, we must run kernels once for each slice of a 3D volume. Rather than solve the momentum equation all at once, we split it into a set of simpler operations that can be computed in succession.

2.2.1 Solid-Fluid Interaction

An advantage of real-time simulation over offline or pre-rendered, is that it allows fluids to interact with the environment. Movement, vector fields, terrain shape and any object in the simulated environment.

To calculate the effects and changes in a velocity field in real-time simulation normally

[3]Represents a single sample, or data point, on a regularly spaced, three-dimensional grid

external forces are applied. The result is normally approximated by giving the solid object or obstacle a surrounding shape like a box or a sphere and add the obstacle's average velocity to that region of the velocity field. These can be represented in a *f(x,y,z)* equation that can be evaluated by the pixel shader[4] at each grid cell.

A normal practise to generate motion in a solid fluid like smoke is to add a simple upward velocity and a smoke density. More advanced methods provide interaction with animations and moving obstacles (see figure 2.2.3). These require a volumetric representation of the obstacle's interior and of the velocity at its boundary. A more intricate and complex approach relative to the scope of this project's thesis but a step forward as future work.

Figure 2.2.3: A reproduction of the *Animated Gargoyle pushing smoke around by flapping its wings*. Adapted from GPU Gems 3(Figure 30-4), by Nguyen, H. and NVIDIA Corporation

2.3 Volume Rendering

It is a general term that refers to any method of taking a 3D volume of data[5] and projecting it to 2D. The process entails approximating the integration of rays casted[6] into the volume.

3D Volume Data can be found on similarly-structured formats such as a block or regular grid of 3D voxel. These are a value on a regular grid in three-dimensional space. Other times it is considered as an implicit 3D function, e.g. f(x, y, z).

[4]Also known as a *fragment shader*, is a program that dictates the color, brightness, contrast, and other characteristics of a single pixel (fragment)

[5]Data that is stored as a 3D image, such as medical or scientific data

[6]Ray casting is a rendering technique capable of transforming a limited form of data into a three-dimensional projection with the help of tracing rays from the view point into the viewing volume

Each voxel (or function value) is often a scalar, but gradient values may be stored alongside it if they are precomputed as well. This value may represent some kind of property of the underlying system. For instance, proton density in an MRI in the case of medical/scientific imaging. In order to visualize it more clearly we need to map it to a color and an opacity using a transfer function.

The transfer function highlights interesting features in the range of possible values. For example, bone or certain tissues distinguishable by scalar value could be mapped to high opacity, while air would typically be given zero opacity. Once color and opacity values are measured the volume is ready to be rendered.

Volume rendering is an important graphics and visualization technique. A volume renderer can be used for displaying not only surfaces of a model but also the intricate detail contained within. It has been around for over a decade and is still an active area of graphics and visualization research.

There are many techniques for Volume rendering data from using 3D textures to faking self shadowing. Techniques that some software environments similar to ours have available already like the Unreal Engine 4 [3].

One of those approaches is called direct volume rendering, where the volume data itself is used without transforming it into a different form. Others more complex are:

- **Ray marching:** This is an image-based method, where each pixel corresponds to a ray from the viewer into the scene containing the volume. It is the most common technique applied today and the rendering consists of stepping along the ray and accumulating the color/opacity contribution from each point. The algorithm consists of tracing a light ray backwards from each pixel in the "camera plane" or view, and sampling the scene at specific locations along the ray onto where it intersects in the scene.
- **Texture slicing:** This method uses proxy geometry in the form of slices through the volume as a way to optimize the process of composition. Mostly on modern graphics hardware tuned for blending large amounts of texture-mapped geometry.
- Another approach is transforming the volume into a **polygonal/mesh representation**, often using the marching cubes algorithm. This is seen in a number of voxel-based games with 'smooth' geometry and can be done on the fly

in modern hardware.

In NVIDIA's GPU Gems 3 approach [8] The result of the simulation is a collection of values stored in a 3D texture. However, there is no mechanism in Direct3D or OpenGL for displaying this texture directly. The volumetric is rendered using a ray-marching pixel shader.

Six quads (or faces of a cube) represent the faces of the simulated volume. These quads are drawn into a *deferred shading buffer* to determine where and how rays should be cast. We then render the fluid by marching rays through the volume and accumulating densities from the 3D texture, as shown in Figure 2.3.1.

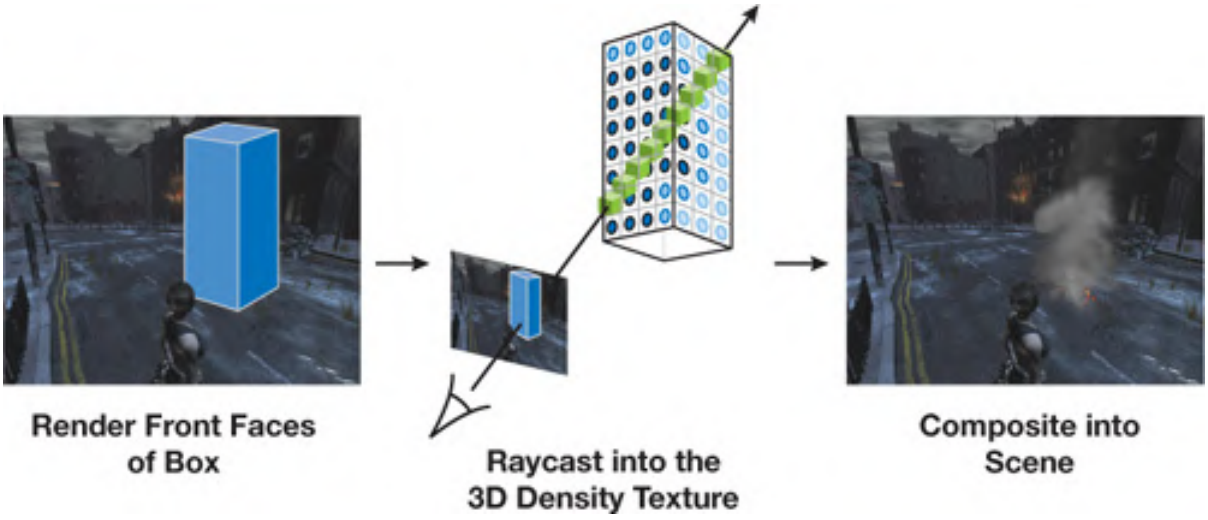


Figure 2.3.1: A reproduction of the *Conceptual Overview of Ray Casting*. Adapted from GPU Gems 3(Figure 30-17), by Nguyen, H. and NVIDIA Corporation

The process of ray casting is complex and takes many steps to render. Steps that could provide enough volume of information for a thesis of their own. These notions are not covered in this thesis not because they were discarded but because the volume rendering is done by the company's framework. This thesis is not trying to explain how volume rendering works but rather offer a study on a particular real-time simulation and the notions and knowledge needed to fully comprehend the process. Similarly to the way Rikard Olajos approached it in his real-time rendering of volumetric clouds' master thesis [9]. Although not with the same final goal.

2.3.1 Real-Time Volumetric Ray Marching and Ordered Dithering

Volumetric Lighting in computer graphics has been proven to be computationally costly. Specially when calculating it to have a more accurate and efficient simulation. Within computer graphics the aim is to calculate what a scene looks like given an abstract representation of it. Subsequently producing a displayable image in the screen.

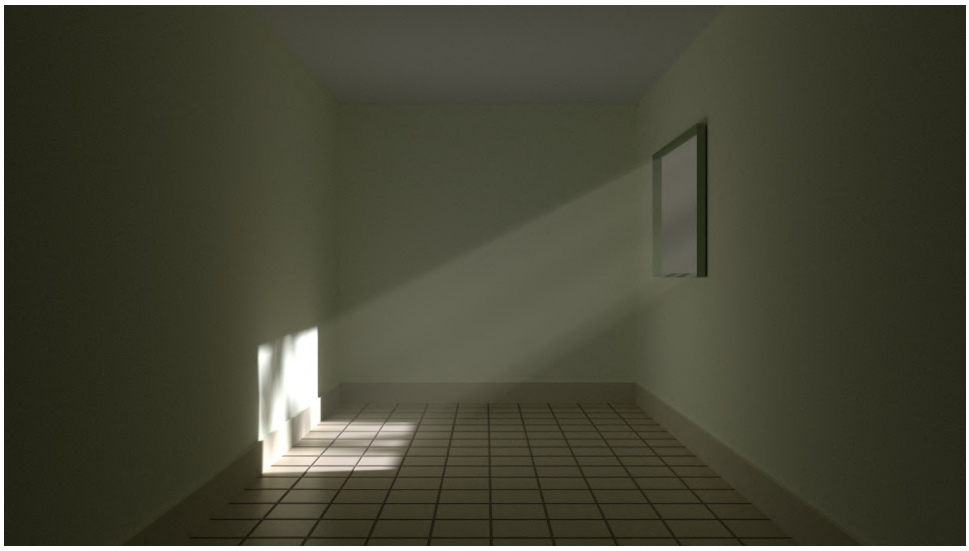


Figure 2.3.2: Volumetric Lighting in a simple rendered scene

Although this report explores a practical optimization (called *ordered dithering for real time GPU based ray marching*) of a volumetric lighting algorithm, these aspects are not present in this thesis. However, due to the similar technology used for this thesis, some of his insight can be considered background knowledge to build upon.

In computer graphics, the aim is to calculate the colour of each pixel on the screen and produce an image. A scene can be rendered in real-time, which entails that the calculations are fast enough to be used in interactive applications. Offline rendering is most commonly used in the movie industry where the rendering can be enhanced before releasing a final result. And because the calculations for these are not needed instantly and don't need to be updated depending on the camera view, they don't suppose as GPU-consuming and costly as real-time rendering.

Like Philip Sköld [17] so clearly puts it:

> *"With real time graphics, interactivity is key instead; each frame in a computer game needs to be calculated on the fly and in the end have enough performance to be calculated and presented on a screen in 30, 60 or even 90 frames per second."*

Due to volumetric lighting being computationally costly, real time applications often assume that light is travelling in a vacuum. Instead, they have been rendered using specialized algorithms and methods. These attempt to imitate the effect and look of certain volumetric phenomena without necessarily being physically based. However, in my thesis the notions of physically based fluid mechanics are part of the solution.

The inclusion of volumetric lighting is not only visually pleasing but can also enhance the experience and physical perception of the simulated environment. For instance, by giving a sense of depth perception (see figure 2.3.3). Even simpler, non-volumetric, models for fog greatly improve the depth perception in a scene.

Figure 2.3.3: Example of Depth perception by using Volumetric Lighting in the Unreal Engine

Philip Sköld's paper digs deeper into the realm of ray marching and dithering[7] which, while interesting, do not participate actively in the implementation of the real-time simulation in our case and would only confuse the reader. However, it is worth mentioning that there are several methods for calculating volumetric lighting and *ray*

[7]Dither is an intentionally applied form of noise used to randomize quantization error, preventing large-scale patterns such as color banding in images

marching is one of them. This is very appropriate for offline rendering because there is no definite constraint for the rendering time. However, for real-time simulation the algorithm has only recently been adapted, with suitable constraints and optimization, so that it is approaching real time rendering speeds (see results in figure 2.3.4).

Figure 2.3.4: Unreal Engine simulation with and without Volumetric Lighting

The gaming industry standard for real-time rendering has been 30fps for the past years. Nevertheless, lately it is being increased up to 60fps as GPU and CPUs become more powerful and capable of handling such computations. And with the incursion of new technologies like Virtual Reality, the standard is closer to 90fps. For the most part, the main goal is to imitate the sense of physical accuracy of atmospheric phenomena, such as volumetric fog, and not really attempt to capture it scientifically accurately. This means that in principal, the more one can cheat in order to produce more efficient

rendering algorithms the better as long as it is not resulting in perceivable visual artifacts.

And because of this notion, one can be driven to ask why so many physically-based methods and algorithms are being developed. The answer lies in the simplicity of producing effects based in physics concepts and principles as it is easier to translate mathematically. Non-physically based methods provide more artistic and visually mesmerizing results but the latest aim of games has been to capture reality as close as possible. However, there are games for everyone and the preferences vary and so does the games being developed. But ultimately, the issue with physically based volumetric rendering in games is primarily the computational aspect.

Chapter 3

Research and Engineering Methods

Due to the empirical nature of the research for this thesis, the use of experience, observation and study of previous research and experiments related take center focus. By following the different solutions to common issues and practises, a body of well-formed theories can be collected and analysed. These will be later used to evaluate and reveal relationships between them by means of experimentation. The experiment corresponds to the second part of the methodology of research. It consists in the implementation of a real-time simulation with volumetrically rendered data using the software environment provided by the company. The data that is both collected and created can be then analysed with either quantitative or qualitative methods.

All while compiling the procedural steps and different issues encountered along the way when carrying out such a computation. The research approaches to these are at times inductive and other times deductive. For instance, when the issue or step at hand is not been understood, a deductive approach to its hypothesis is taken using the previously acquired knowledge to solve it. However, if it is understood but its reason to exist is not clearly explained, an inductive approach is applied. Through which, an inference or an analysis of the procedure taken is carried out to clarify it.

The research strategies and designs are the guidelines, or the methodologies, for carrying out the research which includes organizing, planning, designing and conducting research.

The research strategies and designs for quantitative research commonly are ***Experimental Research***, which verifies or falsifies hypotheses and provides cause-and-effect relationships between variables; ***Ex post facto*** Research, which generally

means that it searches back in time to find plausible causal factors, it is similar to experimental research in that it also verifies or falsifies hypotheses and provides cause-and-effect relationships between variables but cannot provide safeguards to make strong inferences; ***Surveys***, *cross-sectional* and *longitudinal*, assess attitudes and characteristics of a wide range of subjects and it is a descriptive research method, which examines the frequency and relationships between variables.

And finally ***Case study***, an empirical study that investigates a phenomenon in a real life context where boundaries between phenomenon and context are not obvious. The case study is a strategy that we will be using fundamentally in this thesis. It involves an empirical investigation of a particular phenomenon, in our case, a simulation, using multiple sources of evidence.

3.1 Engineering-Related And Scientific Content

For qualitative research, the strategies and designs commonly used are *Surveys*, *Case Study*, *Action Research*, *Exploratory Research*, etc. However, the most relevant that the research carried out in this thesis are the *Exploratory research* and, in a way, the *Creative Research*.

The first provides a basis for general findings by exploring the possibility to obtain as many relationships between different variables as possible. It uses surveys to get an insight in the problem. It rarely provides definite answers to specific issues. Instead, it identifies key issues and variables to define objectives, using qualitative data collection.

The second, involves developing new theories, procedures and inventions, such as, artifacts or systems. In our case, the implementation of a simulation. Also this may not be the most suiting option as a similar method called *Design science* might be a better wording of it. It is a framework for developing artifacts in the area of information systems and not a scientific research method so in some situations it isn't considered valid. However, it should be noticed that it includes parts of software engineering methods used when designing systems which relates to the knowledge and notions applied to this particular project from my particular software engineering background.

Software Engineering notions such as product development planification, division

of tasks, problem-solving work ethic and trial and error experimentation, among others.

Chapter 4

Implementing a Real-time simulation of Volumetrically rendered fog

4.1 Shaders

The *book of shaders* [11] is targeted to those who possess coding experience, a basic knowledge of linear algebra and trigonometry. It teaches how to use and integrate computer programs commonly called *shaders* to improve their performance and graphical quality when manipulating and generating graphic data. The most common are the OpenGL Shading Language (GLSL) shaders which can be compiled and run on a variety of platforms and environments that use *OpenGL*[1] or *WebGL*[2].

Shader Languages have a single main function that returns a color at the end. This color is calculated for every single pixel in a case by case basis. This is mainly because the GPU's parallel processing intends to dedicate threads to each pixel when treating with graphical or real-time data processing.

Some nomenclature that may appear later in use to keep in mind is:

- ***vec*** (*vec2*, *vec3* and *vec4*) stands for a *n* dimensional vector of floating (*float*) point precision.
- **preprocessor macros** are part of a pre-compilation step. With them it is

[1] A cross-language, cross-platform application programming interface for rendering 2D and 3D vector graphics.

[2] A JavaScript API for rendering interactive 2D and 3D graphics within any compatible web browser without the use of plug-ins.

possible to *#define* global variables and do some basic conditional operation (with #ifdef and #endif). All the macro comands begin with a hashtag (#)

- **Vertex Shader (VS)** and **Pixel Shader (PS)**, e.g. *VS_input or PS_input* refer to Vertex Shader and Pixel Shader, two kinds of shaders. The data that is identified with one of these are processed by that certain shader or type.

vec4 is specially important because the four arguments correspond to the RED, GREEN, BLUE and ALPHA channels of the color. Also we can see that these values are normalized, which means they go from 0.0 to 1.0. Later, we will learn how normalizing values makes it easier to map values between variables.

Float types are vital in shaders, so the level of precision is crucial. Lower precision means faster rendering, but at the cost of quality. You can be picky and specify the precision of each variable that uses floating point.

4.2 The Software Environment

4.2.1 The Stingray Engine and the Rendering Pipeline

The way 3D and 2D data are rendered in the Stingray/Bitsquid game engine renderer is well described in the renderer.render_config script (in the manual it is described that there are three main render-pipelines but these either are not relevant or they are outdated as informed by the team).

This script is divided in different building blocks, these will be glanced over in order to not violate the signed NDA agreement:

- **Basic set-up:** settings that control the effects depending on user settings that could adjust to high or low-end systems, shader-related resources loading,
- **Global resource set:** specification of the GUI resources to be rendered (most commonly render targets[3])
- **Layer Configurations:** establishes the order of draw calls and state changes in the render back-end

[3]Render Targets (RT) is a feature of modern graphics processing units (GPUs) that allows a 3D scene to be rendered to an intermediate memory buffer (Render Target Texture (RTT)) instead of the frame buffer or back buffer. This RTT can then be manipulated by pixel shaders in order to apply additional effects to the final image before displaying it.

- **Resource Generators:** framework for manipulating GPU resources. Used for post processing, lightning, shadow rendering and basically any GPU-driven simulation, like probably environment fog and smoke rendering (the section I'd work mostly on)
- **Viewports:** viewports that are instanced from gameplay.
- **Editor Integration:** render pipe needs to support wireframe rendering[4]. Also it is implemented to assist artists debugging their work.

The Shading Environment is a UI with controls of the rendering pipeline that both artists and developers can use.

The system requirements for Autodesk Stingray engine can be found on its support page [19] (see table below)

[4]The most basic representation of a three-dimensional scene or object. It only includes vertices and lines. They are often used as the starting point in 3D modeling since they create a ”frame” for 3D structures

System requirements for Autodesk Stingray Engine

Operating Systems

Microsoft® Windows® 10

Microsoft Windows 8.1

Microsoft Windows 7 SP1

CPU Type

64-bit Intel® or AMD® multi-core processor; Intel Core i5 or higher recommended

Memory

8 GB or higher

Display Card

Microsoft® DirectX® 11 compatibile video card

Disk Space

4 GB of free space for install

Pointing Device

Three-button mouse

Browser

Supplemental content is available online. Autodesk recommends using a recent version of a modern web browser to access and view the content.

Deployment Platforms

Stingray enables developers to deploy projects to Microsoft Windows 7, Windows 8, Windows 10, Apple® iOS, Google® Android®, Sony® PlayStation® 4, and Microsoft Xbox One®.

4.2.2 The Render Configuration

The rendering program or *renderer*, is the one in charged of representing all the data in the screen. It does so by carrying out the calculations needed to transform the 3D data into 2D. The previous information defined the overall structure of the renderer configuration script establishing the different parts. But more importantly, it also establishes the order some shaders are executed in each *pass*.

A *pass* is known as the process of passing the data to the fragment function (a.k.a fragment shader or pixel shader). It runs once for each pixel on the screen each polygon covers and ultimately returns data for each pixel, usually a color. It does so one at a time to be stored and usually eventually displayed. This corresponds to one single pass. Nevertheless, in order to update the data dynamically for each change like the change of camera view, new passes need to occur.

The building blocks of the renderer are:

- **Global includes** array of other shader_source files to include in current file
- **Render and sampler states** sets up a group of states to get compiled efficiently for the rendering back-end to execute
- **Shader code** includes compiler options for shaders, insertion of scopes, association of samplers, instancing, etc.
- **Shader declarations** tie together the shader code authoring and state blocks and creates a final shader template that can be mapped to a material. An important element to mention is the contexts as sometimes there's a need to switch between different shader passes and configs depending on a specific scenario. These can contain passes defined and which can be enabled or disabled if needed.
- **Static compiles**. The static_compile array allows building the declared shader permutations without the need to reference the shader from a material file.

When creating a shader that deals with subtle or even nonessential immersive effects, it should be written in the ***post_processing*** script. Including any effect that might not be a priority in the ***stingray_renderer*** (*shader_libraries*).It is something we will see later when showing the first shader program created as our thesis implements a post processing effect.

The last step when creating a shader is to add a static compile directive. We need to tell *Stingray* when it boots up to take the code that we just provided and compile it into a binary shader representation that the graphics card can run.

4.2.3 *My First Shader*

Three files are modified when a shader is implemented:

- the renderer configuration file (named: *render.render_config*)
- the shader we are going to implement (named: SHADER_NAME.shader_source)
- in our case of working in the post processing phase of the graphics pipeline, the post processing shader (named: post_processing.shader_source)

We started by declaring the *full_screen pass* using our first shader *shader_test*:

```
{type="fullscreen_pass" shader="test_0" input=["hdr0_rgb" "linear_depth"]
    output=["hdr1_rgb"] profiling_scope="taa_history_copy"}
{type="resource_swap" swaps = {"hdr0_rgb"="hdr1_rgb"}}
```

This takes as input **hdr0** which would correspond to the output of previous pass (the one above this in the script). With **_rgb** (i.e. *hdr0_rgb*) and **linear_depth** in meters, that corresponds to the screen depth and also **hdr1** with _rgb. In **resource_swap**, we prepare the input of the next pass, swapping *hdr0* with *hdr1*, meaning *hdr1* is now *hdr0* and now the next pass can use *hdr0* as input.

You can have more than one pass in the same shader when trying to apply different changes or effects to the pixels. When doing so, the output becomes the input of the next pass. Usually that is not done for a full screen pass. For these it is used extensive masks, that is, when representing things through the stencil process[5] then any sort of change can be applied to the sky box[6] or everything that is in front of it. This is done by filtering the sky box green and the foreground elements with a blue filter.

With this configuration, you can have two passes and different render states that

[5] A *stencil buffer* is an extra data buffer, in addition to the color buffer and Z-buffer, found on modern graphics hardware. In the simplest case, the stencil buffer is used to limit the area of rendering (stenciling)

[6] A method of creating backgrounds to make a video game level look bigger than it really is. It has the shape of a cuboid

discard pixels near camera and far from camera as it doesn't process pixels that are not going to appear on camera. This grants the whole process efficiency and speed. Now the shader is moved to development by creating in the **post_processing.shader_source**, the shader itself is implemented (**test_1**)

4.3 Volumetric Fog On The Stingray Engine

Firstly we will study the integrated volumetric fog provided by the Stingray engine. The interest of improving this aspect of a simulation is born in the fact that it is a very common and actual effect that is highly seeked lately. There has been an increase on the development of atmospheric effects as they are believed to enhance the sense of immersion within the simulated environment. Additionally, they provide a sense of realism to the landscapes. Mist, smoke and fog are, with water and other fluids, amongst the most seeked out effects to be improved. The main reason is the closer these are to their realistic counterparts the higher the sense of improvement results compared to any other potential improvement. This is mainly due to the fact that fluids and atmospheric elements are hardly ever static. They remain in constant movement or reacting to many elements like light, wind, obstacles, image reflection or refraction and changes in perspective when you move around any setting.

The main issue or rather potentially improving aspect of the volumetric fog is that it lacks that sense of realistic movement and interaction as showcased in both figures 4.3.1 and 4.3.2. Fog and mist particularly are affected by external forces like wind or obstacles. For instance, if smoke enters a room without an exit or way out the smoke would gather and become more condensed, increasing the density of it. This particular functionality is hard to implement particularly because it depends so highly on specific use cases and scenarios. These are varied and include the aforementioned enclosed room or others like its interaction with slopes in a terrain, or trees or walls where it should clash into and change its path.

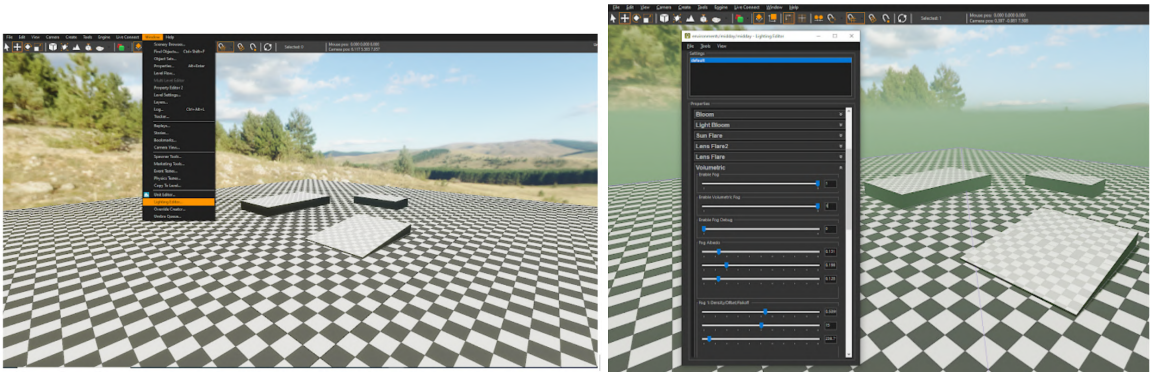


Figure 4.3.1: Screenshot of Stingray's Level Editor (Fatshark's release) without and with volumetric fog activated

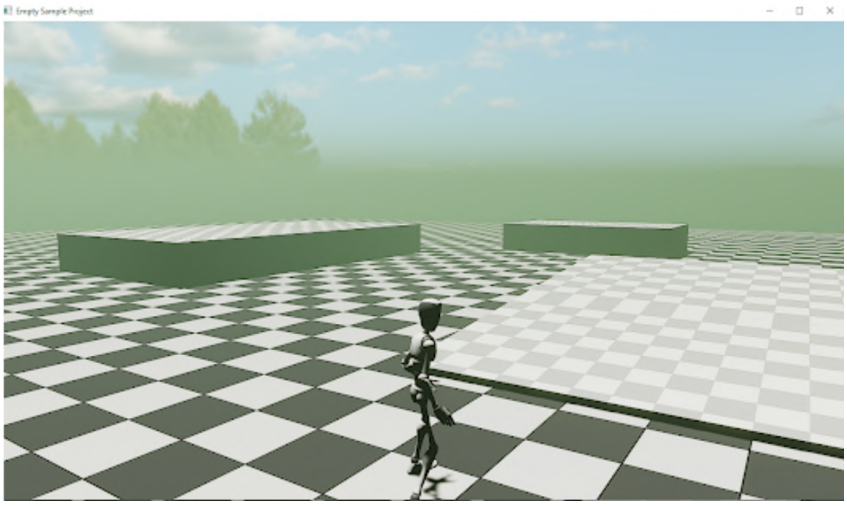


Figure 4.3.2: Screenshot of the ran simulation of volumetric fog activated

The approach now is to generate a simulation of fog or mist that could be less static and, hopefully, more realistic. A way to do this is to allow it to interact with the environment in someway. The best way to do so is by basing our approach in the dynamism of fluids and the creation of wind. Wind is generated by using vector fields in 3D which uses delta time to calculate the swifts in the vectors of each grid/texel to generate a flow. Fluids' structure is interesting as it follows a function to change its volume and shape to create waves and other properties of fluids. These are two aspects that could be integrated in the volumetric fog in the system to make it more interactive.

Considering fog like a fluid would require we base its behavior and rendering similarly to that of the rendering of fluids like water. Aspects like current or a vector field influencing the pixels. In a way it can be considered as a "solid-fluid" given its density.

Fluid flows are everywhere from rising smoke and mist to the flow of rivers and oceans.

Mathematically, the state of a fluid at a given instant of time is modeled as a velocity vector field: a function that assigns a velocity vector to every point in space.

A velocity field on its own is not really visually interesting until it starts moving objects such as smoke particles, dust or leaves. The motion of these objects is computed by converting the velocities surrounding the object into body forces. Light objects such as dust are usually just carried along with the velocity field: they simply follow the velocity. In the case of smoke, it is inefficiently expensive to model every particle. Hence in this case the smoke particles are replaced by a smoke density: a continuous function which for every point in space tells us the amount of dust particles present.

The density usually takes values between zero and one: where there is no smoke the density is zero, and elsewhere it indicates the amount of particles present. The evolution of the density field through the velocity field of the fluid can also be described by a precise mathematical equation.

4.3.1 First 2D Vector Field

When setting up a 2D scenario for creating a vector field that would affect certain pixels in *Stingray* you proceed as follows:

- Generate 2 *render_target*s and their respective *_copy* (for the update of the input value in the next iteration of the simulation) the *render_config*

```
    // Density (color shader for vector field study)
{name="color_target" type="render_target" image_type="image_2d"
    width=312 height=312 format="R16G16B16A16F"}
{name="vector_field" type="render_target" image_type="image_2d"
    width=312 height=312 format="R32G32F"}

//copies as "temp" targets to update pixel for simulation
{name="color_target_copy" type="render_target" image_type="image_2d"
    width=312 height=312 format="R16G16B16A16F"}
{name="vector_field_copy" type="render_target" image_type="image_2d"
    width=312 height=312 format="R32G32F"}
```

- Define where the back_buffer is cleared so it doesn't contain the pre-simulated

pixel image and where we define the call to the simulation:

```
default_vector_field = [
    //Clears back buffer
    { render_targets=["back_buffer"]
        depth_stencil_target="depth_stencil_buffer"
        clear_flags=["SURFACE", "DEPTH", "STENCIL"]
        profiling_scope="clear_render_targets" }
    { resource_generator="vector_field_simulation" }
]
```

- We define the ***vector_field_simulation*** where the inputs are defined in the first two lines, first is the white "dot" or **color_target** and the second is the "noise" or **vector_field target** and are placed accordingly in the screen with first parameter being coordinate u the other v and the other two width and height. The following line performs and defines the simulation getting both targets as input and saving the outputs on the temporary variables _copy. The last two take the output of the simulation (received in _copy targets) and updates the non-_copy versions to prepare them for the next simulation.:

```
vector_field_simulation = {
    modifiers = [
        { type = "fullscreen_pass" shader="copy" input="color_target"
            output="back_buffer"
            profiling_scope="copy_color_target_to_back_buffer" dest_rect =
            [ 0.5 0.0 0.5 0.5] }
        { type = "fullscreen_pass" shader="copy" input="vector_field"
            output="back_buffer"
            profiling_scope="copy_color_target_to_back_buffer" dest_rect =
            [ 0.5 0.5 0.5 0.5] }
        { type = "fullscreen_pass" shader="simulate"
            input=["vector_field" "color_target"]
            output=["vector_field_copy" "color_target_copy"]
            profiling_scope="copy_color_target_to_back_buffer"}
        { type = "fullscreen_pass" shader="copy"
            input="color_target_copy" output="color_target"
            profiling_scope="copy_color_target_to_back_buffer" }
```

```
        { type = "fullscreen_pass" shader="copy"
            input="vector_field_copy" output="vector_field"
            profiling_scope="copy_color_target_to_back_buffer" }
    ]
}
```

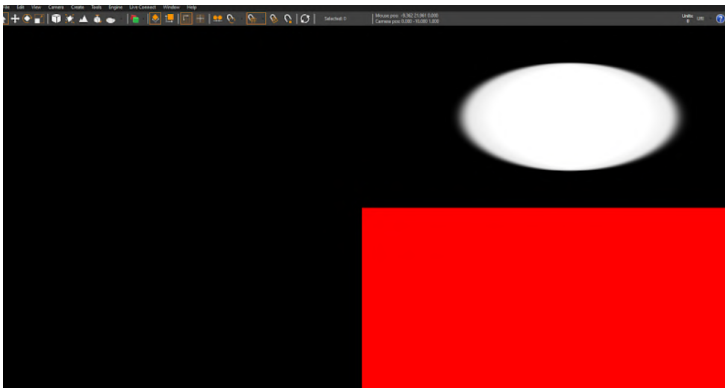


Figure 4.3.3: outputs the result of the mentioned shader on the shader_test code into the output targets

- Under viewports we implemented or added the default code which calls upon the vector field and makes the input of the screen pitch black with the representations according to the simulate. It basically sets up the buffers and all that display the result generated in the simulation (figure 4.3.3)

```
default = {
        // resources instanced once per viewport
        resources = [ ]

        layer_config = "default_vector_field"
        output_rt = "output_target"
        output_dst = "depth_stencil_target"
}
```

Now we proceed to implement the actual shader that will apply the vector field to the colors of the pixels represented, both the white circle and the red color underneath.

The steps are:

- (We insert the reference to the noise shader on the includes at the top of the

script)

- In hlsl_shaders we declare and define the shaders color_init, vector_field_init and simulate:

```
color_init = {
    includes = [ "common" ]
    samplers = {
    }

    code="""
        struct VS_INPUT {
            float4 position : POSITION;
            float2 uv : TEXCOORD0;
        };

        struct PS_INPUT {
            float4 position : SV_POSITION;
            float2 uv : TEXCOORD0;
        };

        CBUFFER_START(c0)
            float4x4 world_view_proj;
        CBUFFER_END

        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        PS_INPUT vs_main(VS_INPUT input) {
            PS_INPUT o;
            o.position = mul(input.position, world_view_proj);
            o.uv = input.uv;
            return o;
        }

        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        float4 ps_main(PS_INPUT input) : SV_TARGET0 {
            float dist = distance(input.uv, 0.5);
            float r = smoothstep(0.3, 0.29, dist);
            return float4(max(float3(r, 0, 0),0), 1);
```

```
        }
    """
}
```

float dist = distance(input.uv, 0.5); calculates the distance of the input pixel coordinates given by ***input.uv*** and calculates it with respect to the origin of coordinates in the screen (which is the top left corner in this case). So in order for it to be represented in the top right corner it has to be 0.5.

float r = smoothstep(0.3, 0.29, dist); takes the previously defined distance and applies the smoothstep effect on it which is making it smoother and making an interpolation as seen in different value assignments in figure 4.3.4. ***return***

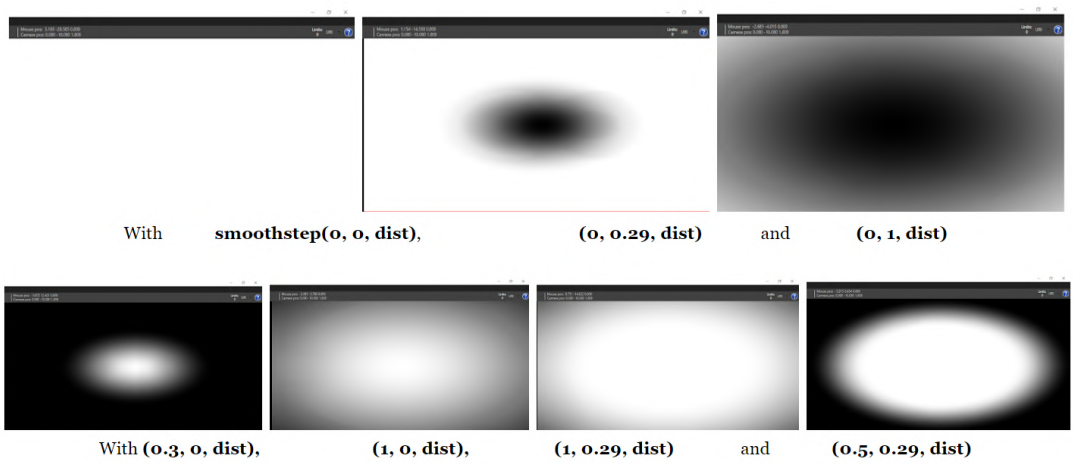


Figure 4.3.4: The different results of the smoothstep function effect over the pixels depending on the input values

float4(max(float3(r, 0, 0),0), 1); where if any number is written in the first two zeros don't affect the *r* because those channels are unused. The other 0 and 1 value don't affect them.

- The above part was only the first section of the shader. We proceed with the following part:

```
vector_field_init = {
    includes = [ "common" "noise_functions"]
    samplers = {
    }

    code="""
```

```
            struct VS_INPUT {
                float4 position : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct PS_INPUT {
                float4 position : SV_POSITION;
                float2 uv : TEXCOORD0;
            };

            CBUFFER_START(c0)
                float4x4 world_view_proj;
            CBUFFER_END

            DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
            PS_INPUT vs_main(VS_INPUT input) {
                PS_INPUT o;
                o.position = mul(input.position, world_view_proj);
                o.uv = input.uv;
                return o;
            }

            DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
            float4 ps_main(PS_INPUT input) : SV_TARGET0 {
                /*float a = Perlin2D(input.uv * 20) * PI;//adding noise
                float x = cos(a);
                float y = sin(a); */

                float x = 1;
                float y = 0;

                float dist = distance(input.uv, 0.5);
                float r = smoothstep(0.3, 0.29, dist);
                return float4(max(float3(x, y, 0),0), 1);
            }
        """
    }
```

The vector_field_init has a similar structure to the color_init. It has the same *distance* and *smoothstep* functions which produce the red square and the white dot. But if we change the values of the input variables we see the following results:

- Incrementing or decrementing the ***x*** variable increases or decreases the velocity field to the right. That is, the velocity with which the white spot moves to the right.
- Incrementing or decrementing, but never under negative values, the ***y*** variable changes the color of the box underneath to yellow and augmenting to 1 adds a y-axis force making it go diagonally. But if *y* is increased to 10 the angle of the y-directional force is different (third picture in figure below) as well as the velocity but the color does not change. Fourth picture has y increased to 50 so the velocity is increased as well as the angle up to a vertical movement. See figure 4.3.5 to contemplate the results.

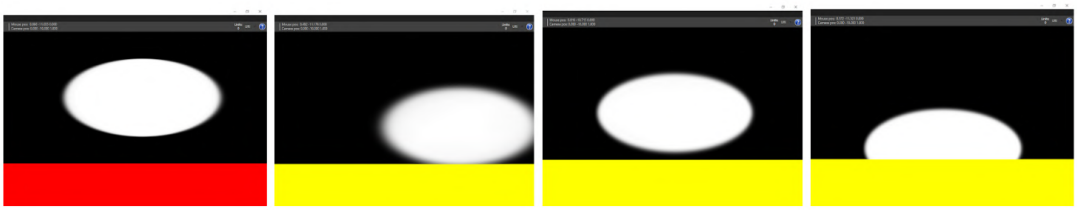


Figure 4.3.5: The different results based on the value of the *y* variable. The first picture being the standard result

- ***float4(max(float3(x, y, 0),0), 1);*** If the *y* is taken out and replaced by any number it will not affect the outcome unless *y* has a number higher than 1 (we get the results seen in the figure above) so, in a way, it only adds on what was actually being affected through the *x*. To see the results of replacing *x* and *y* by *r* over time see figure 4.3.6.

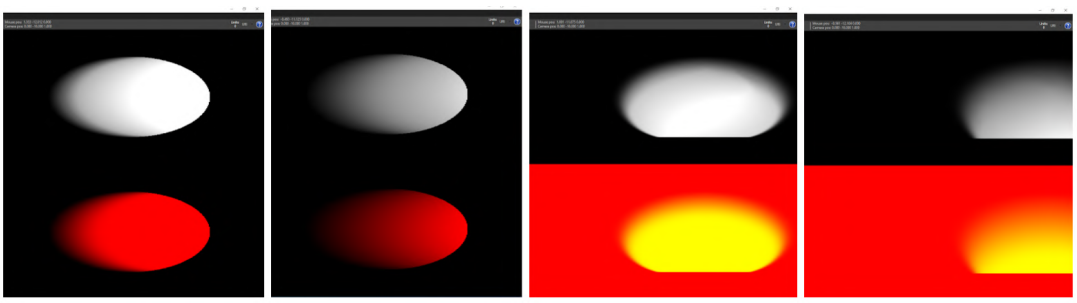


Figure 4.3.6: Time-lapsed results if we replace x with r (first two results) or if y is replaced by r (last two results)

But with the previous implementation which appears commented in the script and which corresponds to the addition of noise. And by commenting the "uncommented" x and y variables, we obtain the result in figure 4.3.7

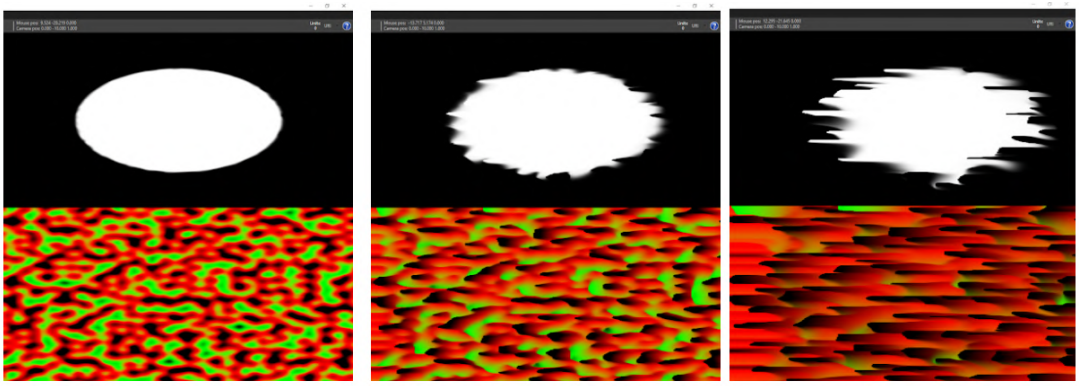


Figure 4.3.7: Result of running the commented text in the script which adds noise to the pixels aside from the vector field

- Now we proceed to see the **simulation shader** which will make use of the *color_init* and *vector_field_init* shaders. We declared two input 2D textures for linear interpolation (*clamp_linear*). And the difference between the structures of other shaders like *test_1* is the *PS_OUTPUT* struct we declared for hosting the velocity and color for which will hold the result processed from the respective *vector_field_init* and *color_init* as target 0 and 1 respectively. That change is also reflected in the *ps_main* method which will be discussed below the code.

```
simulate = {
    includes = [ "common" ]
    samplers = {
        input_texture0 = { sampler_states = "clamp_linear" } //LINEAR
            INTERPOLATION!
```

```
        input_texture1 = { sampler_states = "clamp_linear" }
    }

    code="""
        DECLARE_SAMPLER_2D(input_texture0);
        DECLARE_SAMPLER_2D(input_texture1);

        struct VS_INPUT {
            float4 position : POSITION;
            float2 uv : TEXCOORD0;
        };

        struct PS_INPUT {
            float4 position : SV_POSITION;
            float2 uv : TEXCOORD0;
        };

        CBUFFER_START(c0)
            float4x4 world_view_proj;
        CBUFFER_END

        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        PS_INPUT vs_main(VS_INPUT input) {
            PS_INPUT o;
            o.position = mul(input.position, world_view_proj);
            o.uv = input.uv;
            return o;
        }

        struct PS_OUTPUT {
            float2 velocity : SV_TARGET0;
            float4 color : SV_TARGET1;
        };

        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        PS_OUTPUT ps_main(PS_INPUT input) {
            float2 dt = TEX2D(input_texture0, input.uv).rg * 0.01 *
```

```
                delta_time;
            float3 color = TEX2D(input_texture1, input.uv - dt).r;

            float2 new_velocity = TEX2D(input_texture0, input.uv - dt).rg;

            PS_OUTPUT o;
            o.color = float4(color, 1);
            o.velocity = new_velocity;
            return o;
        }
    """
}
```

- ***VS_main*** receives the *VS_input* which contains *position* and *uv* texture coordinate of a pixel of the screen as we have previously been using as input in shader functions. This generates and output whose format is the input of the *PS_main* function we will see later on. The function itself simply reassigns values in the new outputted format, the *o.position* (which is the *PS_input* position) is the multiplication of the input position (*VS_input* pos) by the *world_view_projection* and the *o.uv* with *input.uv*.
- ***PS_main*** receives as input the previous processed *PS_input* containing *position* and *uv* to which the delta time *dt* is calculated and used for calculating the *color* and the *new_velocity*. *Dt* is calculated using the *vertex_field_init* as *texture0* and *input.uv*. Color needs *texture1* (*color_init*) and *uv - dt*.

The goal of this shader program was to show the behavior of pixels and how the math and the functions applied to them affect the end result. This step was fundamental for understanding the core functionality of vector fields and noise functions that we will later use. Now we will proceed to study the latter to add a randomized effect to a moving color produced by the vector field.

4.3.2 Apply Noise Function To A Moving Color

Before we begin to implement the code for the noise, it is important to understand what constitutes a noise function in our context. In other words, in regards to the

implementation of the renderer in the *HLSL*

float a = sin(input.uv.x - time)

Now ***a*** is the output, or "density" in the shader context, in the case of the *color* and *vector_field* implemented in shader_test. The goal is to try to maximize or keep above 0 this density as the *sin* function is normally between 1 and -1 with the mean at 0. In this particular example we are applying a *sin* function in the x-axis or *input.uv.x* being affected by *time*. And the vector_field defined before should be used as velocity vector or speed.

The way to see noise in 2D is to keep in mind that in the world of computer graphics it's equivalent to saying it is a 2D texture. In code, it is equivalent to a **float2(x,y)**.

Like in the previous step, this time we will modify some of the code done above to apply a sin function with a given range of amplitude, frequency and noise that is affected by a time variable. This exercise was inspired and referenced by *The Book of Shaders: Fractal Brownian motion* [11] and Inigo Quilez's *Fractals Brownian Motion* article [12]

The steps carried out are the following:

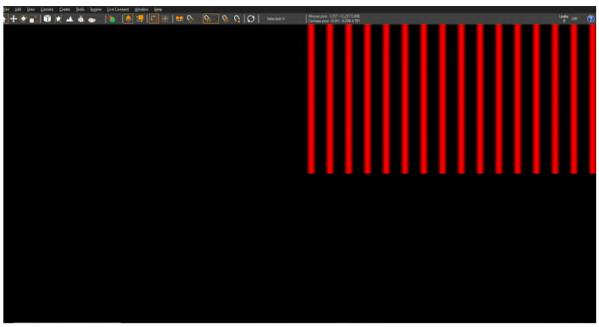
- In the simulate hlsl_shader we modified certain section of the code like such:

```
simulate = {
    includes = [ "common"]
    /*********** PREVIOUS FROM THIS POINT IS THE SAME ***********/
        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        PS_OUTPUT ps_main(PS_INPUT input) {
            //This function is called for every frame repeatedly so we
                placed the
            //sin function here so that the "time" vble is updated at
                every frame
            //We added a "noise_functions"

            /*float2 dt = TEX2D(input_texture0, input.uv).rg * 0.01 *
                delta_time;
            float3 color = TEX2D(input_texture1, input.uv - dt).r;
            float2 new_velocity = TEX2D(input_texture0, input.uv - dt).rg;
            PS_OUTPUT o;
```

```
            o.color = float4(color, 1); //float4 (float3, float1): float3
                + float1 = float4!
            o.velocity = new_velocity;
            return o;*/

            float amplitude = 1.;
            float frequency = 100.;
            float x = input.uv.x;
            float velocity = 1.;
            float y = amplitude * sin(x* frequency - time*velocity);
            PS_OUTPUT o;
            o.color = float4(max(y, 0), 0, 0, 1);//max(y, 0) prevents from
                obtaining negative values
            o.velocity = 0;
            return o;
        }
    }
```

The changes are the addition of the *sin* function described in the above referenced book of shaders which needs an *amplitude*, *frequency* and *x* variable. The x given it is the input that will be changed over time as in every sin function. In our shader's case it will be the x coordinate of the *uv* of the input pixel ***input.uv.x***. And we will also add a parameter *velocity* as we have a vector_field that is adding speed to the color. This we multiply by the time variable and we subtract it from the ***input.uv.x*frequency*** inside the sin function.

For the output we generate the color by limiting the values of the sin function to be above 0 (non negative) using the ***max()*** function and supplying it to the red channel. The rest are left at 0 except the last one which, like in the last implementation, corresponds to the *alpha channel* and it has a constant value of 1. The output velocity is 0 because it does not affect the result, the velocity has already been implemented previously. We obtain the following result:

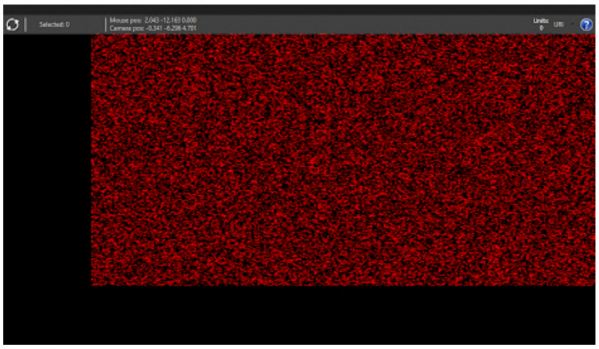


- Now we add the noise. We import the noise from the library of the company as they are generally a default or standard asset. And in the body we add the *Perlin2D* noise we used before:

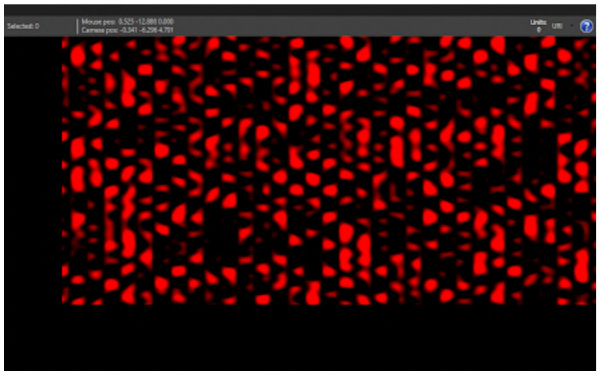
```
simulate = {
    includes = [ "common" "noise_functions"]
    /*********** PREVIOUS FROM THIS POINT IS THE SAME **********/
        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        PS_OUTPUT ps_main(PS_INPUT input) {
            float amplitude = 1.;
            float frequency = 100.;
            float x = input.uv.x;
            float velocity = 1.;
            float noise = Perlin2D(input.uv * 20) * PI;//adding noise
          //PERLIN NOISE: By default values oscillates from -1 to 1.
            //By *PI it would increase to 3.14 to -3.14
            float y = amplitude * sin((x* frequency -
                time*velocity)*noise);
            PS_OUTPUT o;
            o.color = float4(max(y, 0), 0, 0, 1);//max(y, 0) prevents from
                obtaining negative values
            o.velocity = 0;
            return o;
        }
}
```

And we can add the noise in the sin function in two ways:

- Inside it:
  float y = amplitude * sin((x* frequency - time*velocity)*noise);



– Outside it: ***float y = amplitude * sin(x* frequency - time*velocity)*noise;***



There's a mistake however, we multiplied the noise function in the *y* instead of replacing it. So it should have been like this:

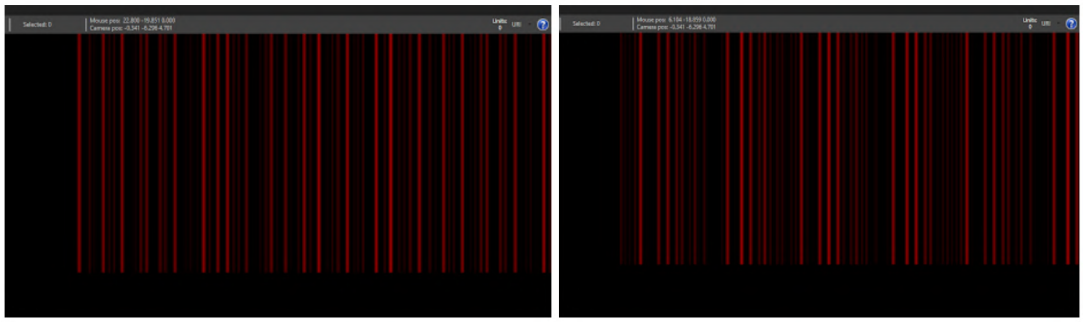
```
float y = amplitude * Perlin2D(x* frequency - time*velocity);
```

What this does is apply the noise directly. Nevertheless, this is in 1D, that is, it is only affecting the x axis (not y) that is why the time and velocity it is only affecting the x direction. We can transform it to 2D by opening the 2 channels with ***float2(x,y)*** like so:

```
float amplitude = 1.;
       float frequency = 100.;
       float x = input.uv.x;
       float velocity = 1.;
       float noise = Perlin2D(input.uv * 20) * PI;//adding noise
       float y = amplitude * Perlin2D(float2(x,0) * frequency -
          time*velocity); //float2(x,0) is transforming x to 2D
```

```
    PS_OUTPUT o;
    o.color = float4(max(y, 0), 0, 0, 1);//max(y, 0) prevents from
        obtaining negative values
    o.velocity = 0;
    return o;
```

The results in all the executions of the code are hard to show in a document or report through screenshots as the modifications change in time, however the following two screenshots correspond to a time-lapsed execution.

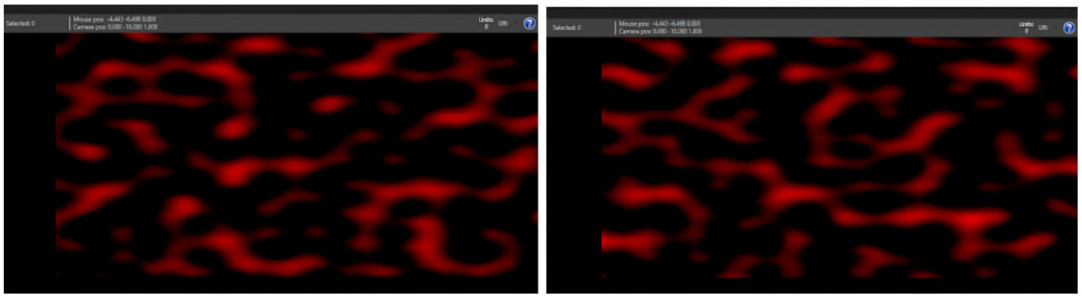


It can not be appreciated at first glance but the effect of the code above gives a wavy and smooth transition similar to a watery-like frequency.

There are other ways of transforming the output to 2D but it won't have the same output because other variables are affected by the channel-opening not only x (frequency, time etc).

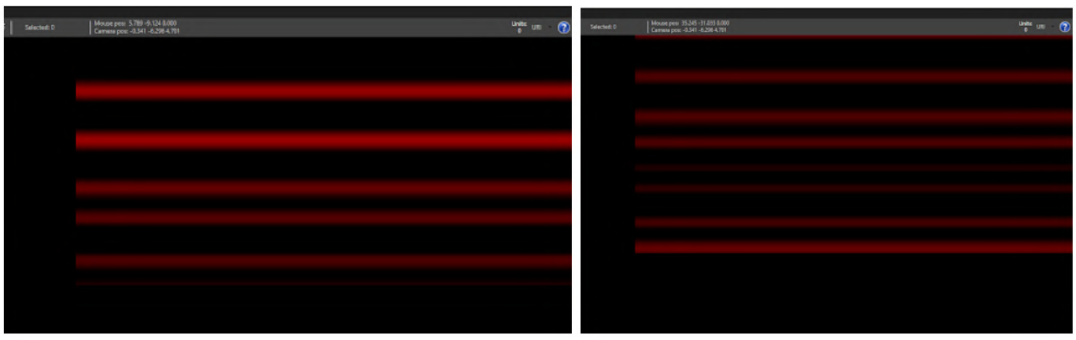
- We can also make it so different variables are affected or have their channels opened, so they can affect the 2 dimensions x and y differently if needed, like so:

```
float amplitude = 1.;
float2 frequency = float2(10., 10.);
float2 x = input.uv.xy;
float2 velocity = float2(1., 0);
float y = amplitude * Perlin2D(x * frequency - time*velocity);
//float2(x,0) is transforming x to 2D
PS_OUTPUT o;
o.color = float4(max(y, 0), 0, 0, 1);//max(y, 0) prevents from obtaining
    negative values
o.velocity = 0;
return o;
```



In this case we have frequency and velocity being applied to the x and y channels and we can choose different values for each depending on what we want. So in velocity for instance, if we set it to ***float2(1., 0)*** the noise would move in the x direction. If you inverse it (***float2(-1., 0)***) it would go the other direction. Also we have made input x 2D by ***input.uv.xy***.There are other ways of doing this and that is by multiplying it directly in the function like so:

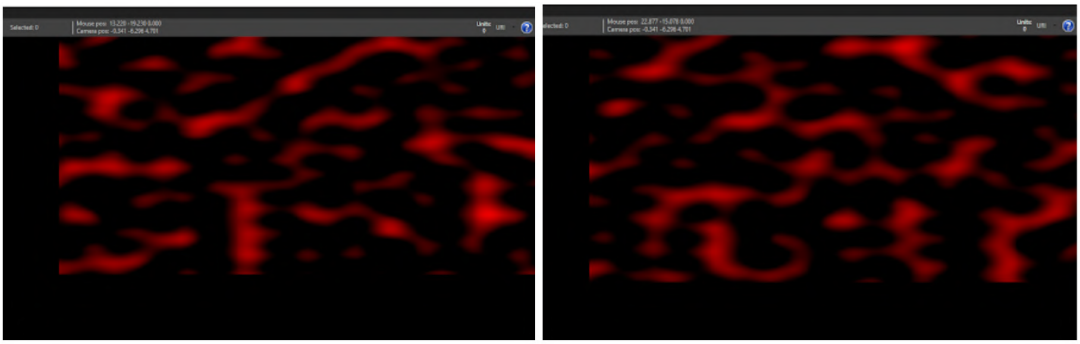
```
float amplitude = 1.;
float frequency = 10.;
float2 x = input.uv.xy;
float velocity = 1.;
float noise = Perlin2D(input.uv * 20) * PI;//adding noise
float y = amplitude * Perlin2D((x * frequency - time*velocity*float2(1,
    -1))*float2(0, 1));
PS_OUTPUT o;
o.color = float4(max(y, 0), 0, 0, 1);//max(y, 0) prevents from obtaining
    negative values
o.velocity = 0;
return o;
```



As we can see, the channel x is deactivated (or constant/unaffected) and there's a

velocity applied in the y direction and it is inverted so it goes up. If it is changed for a 1, it would flow upwards just with the same output view.

Notice that the other variable that has been modified for the output is the *x* to a *float2* with *input.uv.xy*. If the outer float2 is changed to *float2(1,1)* the x channel would be activated and we would have the following result moving in the x axis: If you have this:



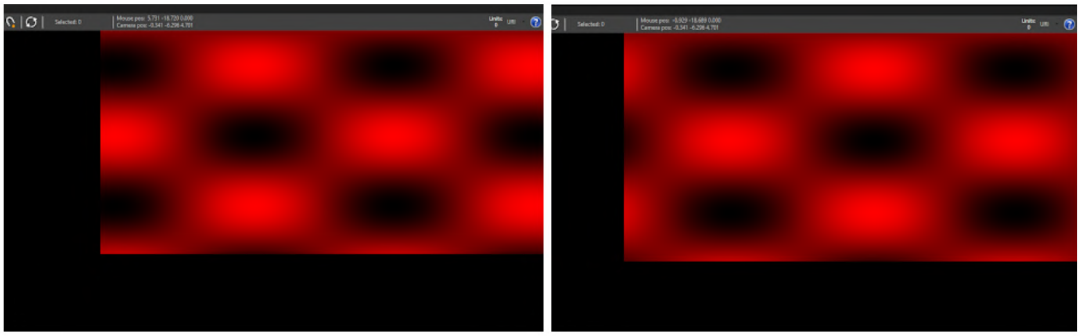
```
    float y = amplitude * Perlin2D((x * frequency -
        time*velocity*float2(1, 1))*float2(1, 1));
```

We are applying velocity and time to x and y direction as well as enabling the operation, as it behaves exactly like an operation. That is, the multiplication with velocity also affects the multiplication of time, just like with mathematical properties. It moves diagonally from top left corner to bottom right corner.

- Also, there's the option of creating your own noise like so (creating a function in a shader):

```
//Noise function without any noise
float sin_noise (float2 input){
    float2 x = sin(input);
    return x.x*x.y;
}

DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
PS_OUTPUT ps_main(PS_INPUT input) {
    float amplitude = 1.;
    float2 frequency = float2(10., 10.);
    float2 x = input.uv.xy;
```

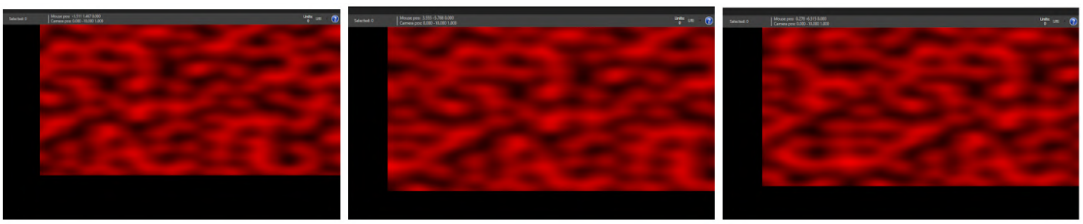
```
    float2 velocity = float2(1., 0);
    float noise = Perlin2D(input.uv * 20) * PI;//adding noise
    float y = amplitude * (sin_noise(x * frequency - time*velocity) + 1)
        * 0.5;…

}
```



What is being done with the **(sin_noise(x * frequency - time*velocity) + 1) * 0.5** is moving the amplitudes and mean of the wave so with the +1 it moves between 2 and -1 instead of 1 and -1, and the *0.5 makes the mean go higher.

- The final version however, is different because certain input variables like frequency, the x variable and velocity are precomputed in 2D and we use the previously described calculations to move up the amplitudes. But finally the *Perlin2D* function is the chosen one to be used as noise:

```
float amplitude = 1.;
float2 frequency = float2(10., 10.);
float2 x = input.uv.xy;
float2 velocity = float2(1., 0);
float noise = Perlin2D(input.uv * 20) * PI;//adding noise
float y = amplitude * (Perlin2D(x * frequency - time*velocity) + 1) *
    0.5;
```



- I proceeded to implement the Octaves loop to add additional noise and did so by modifying the code like this:

```
float amplitude = 1.;
float2 frequency = float2(10., 10.);
float2 x = input.uv.xy;
float2 velocity = float2(1., 0);
float noise = Perlin2D(input.uv * 20) * PI;//adding noise
float y = amplitude * (Perlin2D(x * frequency - time*velocity) + 1) *
    0.5; //float2(x,0) is transforming x to 2D
PS_OUTPUT o;
o.color = float4(max(y, 0), 0, 0, 1);//max(y, 0) prevents from obtaining
    negative values
//o.color += fbm(input.uv.xy*3.0);//calling the octaves' function
float amp = .50;
for (int i = 0; i < OCTAVES; i++) {
    o.color += amp * Perlin2D(x * frequency - time*velocity);
    x *= 2.; //Modifies the "granularity"
    amp *= .5;//Modifies the gain
}
o.velocity = 0;
return o;
```

Finally the result is shown in figure 4.3.8 but as previously mentioned, it is hard to show the actual final effect created, there are two noise functions moving in the x direction at different velocities making it look like “mist” or different volumes of clouds moving. In the next section we will apply this knowledge and new-found concepts to the volumetric fog introduced at the beginning of section 4.3

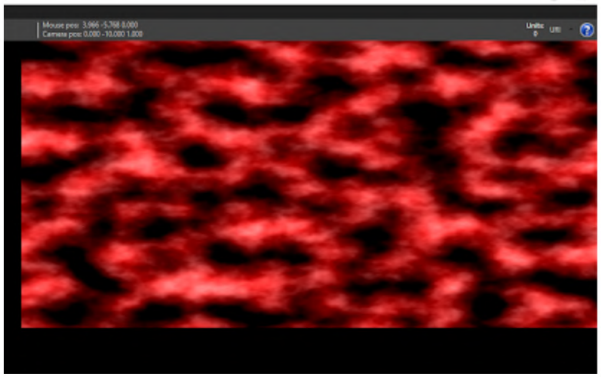


Figure 4.3.8: Final result of the vector field force and noise applied to a 2D texture to create a moving fog-like or cloud-like effect

4.3.3 Adding Noise to 3D Volumetric Fog

Firstly to obtain the 3D rendered scene we change the viewport of the rendering engine environment so that we can access a sample project similar to the one seen at the beginning of section 4.3.

Firstly, in the script that renders all the volumetric data where we will contribute with our implementation we add the implemented section of the code:

```
float3 wp = view_to_world(ss_to_view(ss_pos, 1.0), 1.0);
/*** Adding noise function to the volumetric fog ***/
float noise = Perlin3D(wp + time*float3(1., 0., 0.)); //-1 to 1. If *PI it
    would inrease to 3.14 to -3.14
//If you take + 1.0 away the amplitude is decreased
float extinction = max(global_extinction(wp)*noise,0);
```

Where ***wp*** corresponds to the *world position* of the fog and the ***extinction*** and ***global_extinction*** is a variable that controls the level of extinction in terms of visual representation and condensation of the volume in case a more or less dense is needed.

This piece affects the overall fog volume, however, there are particular samples of volumetric data that can make the effect easier to visualize. If we want to test vector fields or flow maps in a single unit of fog, the effect should be applied to the ***local_extinction***. This one is the equivalent to the *global_extinction* but for individual fog units.

This volumetric data will be commonly used throughout the thesis to showcase the results of the advancements in the effect over the overall fog. Mainly because the volumetric fog is normally deactivated for its costly and performance-heavy impact on the GPU. Therefore, when testing new changes or effects the individual smaller fog unit is used instead. Nevertheless they are both rendered the same so the effects will be the same. We can see these in figure 4.3.9

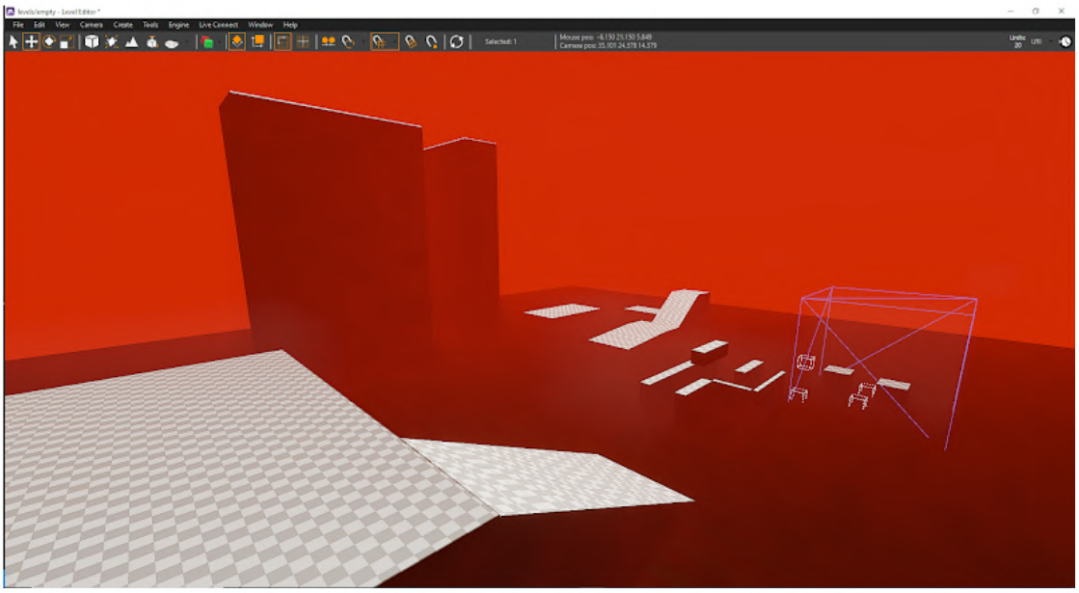


Figure 4.3.9: The selected box is a fog unit, the red mist is the volumetric fog which is being affected by a noise function

4.4 Creating a Flow Map

We have broken the uniformity on the distribution of the basic volumetric fog introduced in section 4.3. By adding a noise function and vector field we hoped to break the static behavior of the fog. Now, we will try to bring it to life by using a *flow map*.

Flow maps are in cartography a mixture of maps and flow charts, that represent the movement of objects from one location to another. Such as the number of people in a migration, the amount of goods being traded, or the number of packets in a network. This concept is applied to computer graphics as a way of creating a pattern using color interpolation to dictate how the pixels should behave.

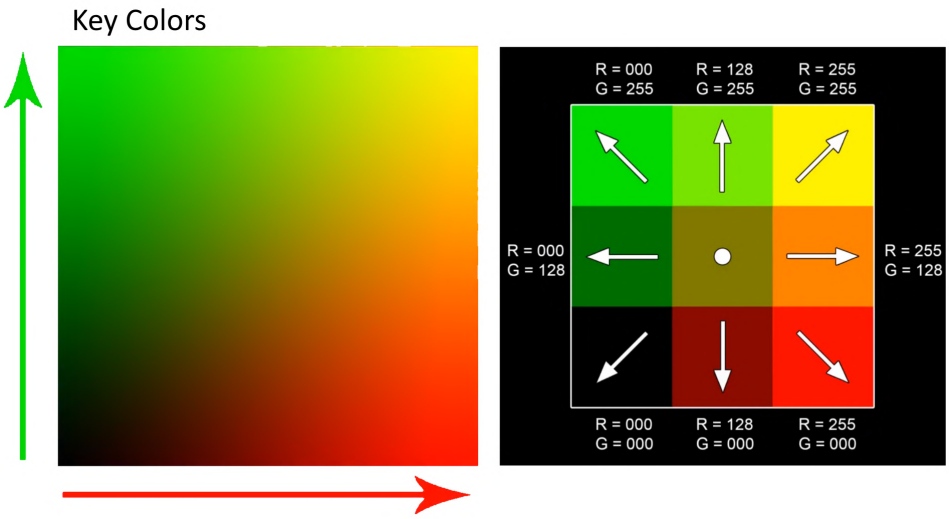


Figure 4.4.1: Simplification of the different values the pixel can have in the flow map which dictates its speed direction

Basically it uses 2 channels of a texture (the red and the green) to move texture coordinates along the x and y axes. Each channel controls one axis, that is, red controls the x-axis and green the y (see figure 4.4.1).

A channel color value of 0.5 means no flow, a value greater than 0.5 moves the coordinates along one direction of an axis and a value smaller than 0.5 moves it to the opposite direction. For instance, a value of 0 in the red channel will move the pixel fully to the left, 0.5 will not move at all and 1 will move all the way to the right. How much the pixel is moved is scaled based on the flow speed and time value.

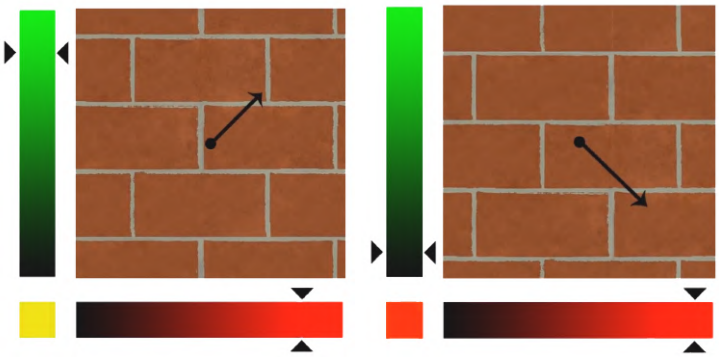


Figure 4.4.2: Examples of Flow Map values to establish the direction or speed

So with the right values, textures can be moved in all directions and that for each location individually. And because the way speed or velocity is assigned to each color, the value of the color interpolation will be used to measure the amount of speed being applied to a certain pixel. So if more value is added to yellow, which is the addition of

red and green, then the faster it travels diagonally (see figure 4.4.2 for a reference)

Additionally, to minimize the pulsing effect a noise texture can be used to add some random offset. We can see examples of these in figure 4.4.3.

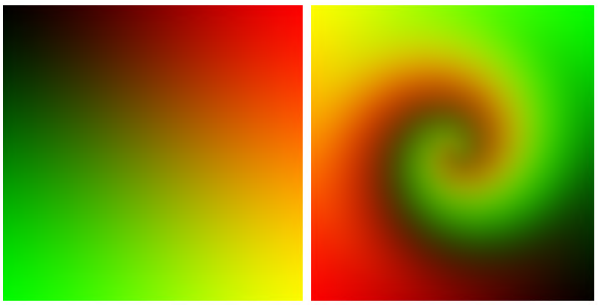


Figure 4.4.3: Examples of Flow Map stored in a 2D texture in the Computer Graphics context

The use we are going to make of it resembles the one used and showcased in Alex Vlachos's SIGGraph proceeding of *Water Flow* [20]. In it, a flow map was designed to affect the water dynamic so it follows a pattern designed so it looks like it hits trees and other obstacles in its path like in figure 4.4.4.

In the Stingray engine there's an add-on functionality implemented where you can import a 2D texture file with which we can apply it to the volumetric data. We use this functionality to add the noise texture in the simulation like this:

```
simulate = {
    includes = [ "common" "noise_functions"]
    samplers = {
        input_texture0 = { sampler_states = "clamp_linear" } //LINEAR
            INTERPOLATION!
        input_texture1 = { sampler_states = "clamp_linear" }
        noise = { sampler_states = "clamp_linear" }
    }

    code="""
        DECLARE_SAMPLER_2D(input_texture0);
        DECLARE_SAMPLER_2D(input_texture1);
        DECLARE_SAMPLER_2D(noise);
    ..."""
}
```

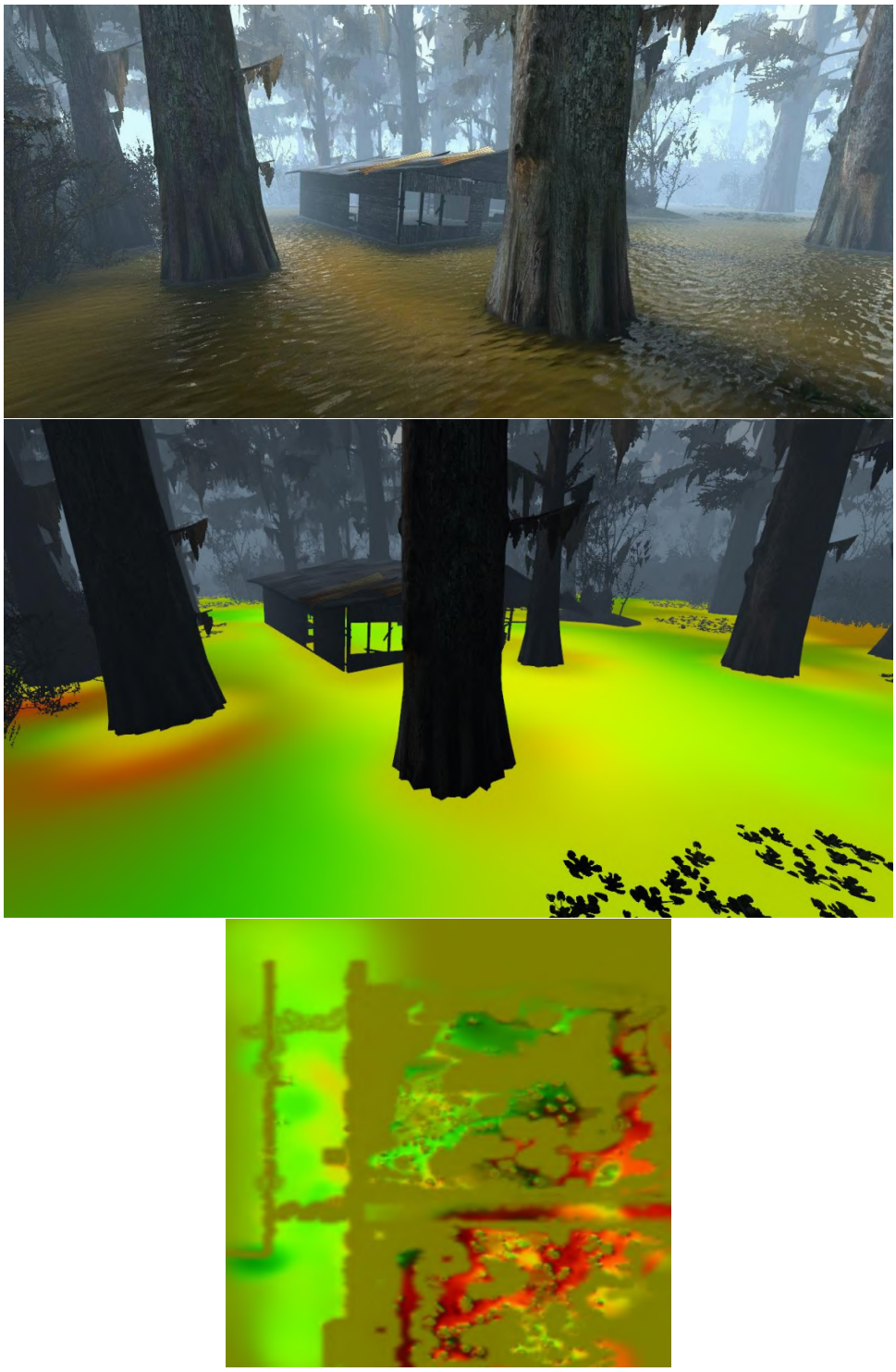


Figure 4.4.4: Reproductions of the Water Flow simulation by from *Valve*'s proceeding. Adapted from *SIGGRAPH 2010 Advanced Real-Time Rendering Cours: Water Flow* (Slides 8-13), by Alex Vlachos

The basic idea of flow maps is that you create a 2D texture that you will map to your fluid. And this map will contain the flow directions that you want the fluid to flow, with each pixel in the flow map representing a flow vector. This allows you to have varying velocity (based on length of the flow vector), and varying flow directions (based on the color of the flow vector). You then use this flow map to alter the texture coordinates of the normal maps instead of scrolling them. Now the goal at hand: using flow maps to establish the direction and limitations of the fog by treating it like a fluid.

4.4.1 Applying A Flow Map To A 2D Texture

The implementation takes a PNG texture and through the texture manager it changes into a format that the renderer can read and manipulate. Once that's done and the pertinent lines of code have been added to import the texture in both the render_config and the shader, we implement the following code:

```
/* float intensity = 3.;
   float timescale = .09;

   float2 uv = input.uv.xy;
   float2 distortion = (uv - .5) * intensity;
   // calculated normal map (sinkhole towards center)

   float scaletime = (time + TEX2D(noise, uv).r) * timescale;
   // but this looks cooler with my calculated normals

   float flow_t0 = frac(scaletime);
   float flow_t1 = frac(scaletime + .5);
   float alternate = abs((flow_t0 -.5) * 2.);

   float4 samp0 = TEX2D(noise, uv + distortion * flow_t0);
   float4 samp1 = TEX2D(noise, uv + distortion * flow_t1);

   PS_OUTPUT o;
   o.color = lerp(samp0, samp1, alternate);
   o.velocity = 0;

   return o;*/
```

```
float2 uv = input.uv.xy;//vec3 uv = fragCoord.xy / iResolution.xy;
float2 flowDirection = TEX2D(flowmap_twirl, uv).xy * -0.002;
//vec2 flowDirection = texture(DisplayField,uv).xy * -0.002;

PS_OUTPUT o;
o.color = float4(flowDirection, 0.0, 1.0); //fragColor = vec4( flowDirection,
    0.0, 1.0 );
o.velocity = 0;

const float cycleTime = 10.0;
const float flowSpeed = 0.5;

// Use two cycles, offset by a half so we can blend between them
float t1 = time/ cycleTime; //iTime / cycleTime;
float t2 = t1 + 0.5;
float cycleTime1 = t1 - floor(t1);
float cycleTime2 = t2 - floor(t2);
float2 flowDirection1 = flowDirection * cycleTime1 * flowSpeed;
float2 flowDirection2 = flowDirection * cycleTime2 * flowSpeed;
float2 uv1 = uv + flowDirection1;
float2 uv2 = uv + flowDirection2;
float4 color1 = TEX2D( weird_texture, uv1 );
float4 color2 = TEX2D( input_texture1, uv2 );//velocity vector = vector field

// Ping pong between the two flows, showing the least distorted and allowing
    uv resets on both.
o.color = lerp( color1, color2, abs(cycleTime1-0.5)*2.0 );
//fragColor = mix( color1, color2, abs(cycleTime1-0.5)*2.0 );

return o;
```

What this code does is create a color with the imported flow map (***flowDirection***) and calculates it for the x and y channel, that is, the red and green.

Now initially the ***o.color*** (output color) has been assigned by the flow map defined before in *flowDirection*. And it is set up omitting the blue channel all together (third

argument in the *float4* is 0.0, the fourth is the alpha value which, as seen previously, it's always 1.0).

Now for what was inside that if, we defined a *cycleTime* and *flowSpeed* variables. These will be used for defining the two cycles differentiated by an offset. The floor function returns the largest integer number that is smaller or equal to *t1* or *t2* respectively. Consequently, two new flow directions are created for the two offseted cycles and are used for defining the changes (addition) to the uv coordinates in two new coordinates respectively.

Now these two uv coordinates are transformed into ***colors 1*** and ***2*** respectively. One is assigned the texture to be affected by the flow map and the other is the ***input_texture1*** which would correspond to the added speed or velocity generated from the velocity_field created previously. Finally we ***lerp*** (or mix in GLSL) both colors with an added time variable and the result is a switching from static to spiraling. Unfortunately, I don't know if this was done correctly or if the flow map provided (the spiral seen previously) is well done but either way the result is similar to what we were trying to accomplish.

The goal now is to try to make a consistent and continuous effect through a different flowmap. Make some sort of fluid simulation in 2D so we can understand what "makes it tick" and then craft a possible way of creating it in 3D and given a height map.

```
float2 flowDir = TEX2D(internet_flowmap, input.uv.xy).xy * 2.0 - 1.0;
//flowDir *= 0.05; //flowSpeed

float phase0 = frac(time * 0.5 + 0.5);
float phase1 = frac(time * 0.5 + 1.0);

float4 tex0 = TEX2D(weird_texture, input.uv.xy + flowDir.xy * phase0);
float4 tex1 = TEX2D(input_texture1, input.uv.xy + flowDir.xy * phase1);

float flowLerp = abs((0.5 - phase0) / 0.5);
PS_OUTPUT o;
o.color = lerp(tex0, tex1, flowLerp);
o.velocity = 0;
return o;
```

This code is similar in many ways to the one coded previously only in this one it is simplified and the effect is much better. It's more fluid and constant as seen in figure 4.4.5

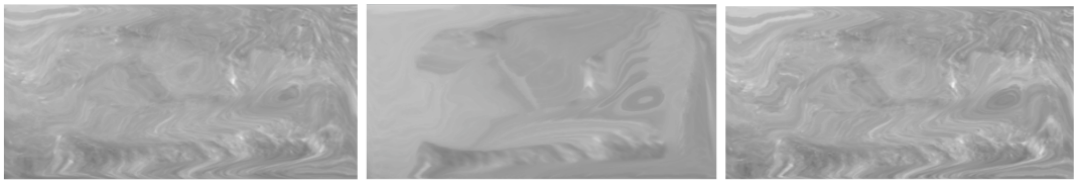


Figure 4.4.5: Examples of Flow Map stored in a 2D texture in the Computer Graphics context

4.5 Applying the Flow Map to the Volumetric Fog

We now proceed to copy the shader code with slight modifications and we try to make it work in a sampled fog unit using the procedure used by Taras Osiris [10]. Before that we will depart from where we last included code for creating a noise function in the global_extinction like the noise:

```
cs_fog_material_data = {
    includes = [...]
    stage_conditions = {...}

    samplers = {
        sun_shadow_map = { sampler_states = "shadow_map" }
        static_sun_shadow_map = { sampler_states = "shadow_map" }
        global_diffuse_map = { sampler_states = "clamp_linear"}

        internet_flowmap = { sampler_states = "wrap_linear" }
        noise = { sampler_states = "wrap_linear" }
    }

    code = """
        RWTexture3D<float4> input_texture0;
        DECLARE_SAMPLER_2D(internet_flowmap);
        DECLARE_SAMPLER_2D(noise);
```

```
        ...
        void get_data(uint i, out float4x4 inv_world, out float3
            bound_volume_min, out float3 bound_volume_max, out float3 albedo,
            out float extinction, out float phase, out float4 local_falloff)
        {...}
        ...
        void cs_main(uint3 Gid : SV_GroupID, uint3 DTId : SV_DispatchThreadID,
            uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex )
        { ...

            //NEW: Adding noise function to the volumetric fog
            //float noise = Perlin3D(wp + time*float3(1., 0., 0.));
            //-------------------------------------
            float2 flowDir = TEX2D(internet_flowmap, wp.xy * 10).xy * 2.0 - 1.0;
            float phase0 = frac(time * 0.05 + 0.5);
            float phase1 = frac(time * 0.05 + 1.0);
            float4 tex0 = TEX2D(noise, wp.xy * 10 + flowDir.xy * phase0);
            float4 tex1 = TEX2D(noise, wp.xy * 10 + flowDir.xy * phase1);//We
                use the same input in both!
            float flowMap = 1;//lerp(tex0, tex1, abs((0.5 - phase0) / 0.5)).r;
            //-------------------------------------
            //float extinction = max(global_extinction(wp)*flowMap,0);
            float extinction = global_extinction(wp);
            ...
        }
    """
}
```

But the result was not visible for some reason, it did not apply well the flow map to the fog so we made some modifications. For a smaller victory, we applied it to a single fog unit to see it clearly. If it succeeded, it could be then applied to the general volumetric fog.

```
cs_fog_material_data = {
    includes = [ ... ]
    stage_conditions = {...}
    samplers = {
```

```
        sun_shadow_map = { sampler_states = "shadow_map" }
        static_sun_shadow_map = { sampler_states = "shadow_map" }
        global_diffuse_map = { sampler_states = "clamp_linear"}
        internet_flowmap = { sampler_states = "wrap_linear" }
        noise = { sampler_states = "wrap_linear" }
    }

    code = """
        RWTexture3D<float4> input_texture0;
        DECLARE_SAMPLER_2D(internet_flowmap);
        DECLARE_SAMPLER_2D(noise);
        ...
  void cs_main(uint3 Gid : SV_GroupID, uint3 DTId : SV_DispatchThreadID,
      uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex )
        {
            float3 sample_pos = sample_position(DTId,
                inv_input_texture0_size.xy);
            float2 uv = sample_pos.xy * inv_input_texture0_size.xy;
float depth = froxel_to_linear_depth(sample_pos.z, inv_input_texture0_size.z,
    volumetric_distance, uv);
            float non_linear_depth = linear_to_clip_depth(depth);
            float3 ss_pos = float3(uv, non_linear_depth);
            float3 wp = view_to_world(ss_to_view(ss_pos, 1.0), 1.0);

            //NEW: Adding noise function to the volumetric fog
            //float noise = Perlin3D(wp + time*float3(1., 0., 0.));
            //-------------------------------------------------------------------------
            /*float2 flowDir = TEX2D(internet_flowmap, wp.xy * 10).xy * 2.0 -
                1.0;
            float phase0 = frac(time * 0.05 + 0.5);
            float phase1 = frac(time * 0.05 + 1.0);
            float4 tex0 = TEX2D(noise, wp.xy * 10 + flowDir.xy * phase0);
            float4 tex1 = TEX2D(noise, wp.xy * 10 + flowDir.xy * phase1);
            float flowMap = lerp(tex0, tex1, abs((0.5 - phase0) / 0.5)).r;*/
            //-------------------------------------------------------------------------

            //float extinction = max(global_extinction(wp)*flowMap,0);
```

```
            float extinction = global_extinction(wp);
            float3 scattering = fog_color * extinction;
            float3 emissive = 0;

            float total_weight = extinction;
            #if defined(VOLUMES_ENABLED)
    ...
      //-----------------------------------------------------------------
        float2 uv = op.xy/50;
        float2 flowDir = TEX2DLOD(internet_flowmap, uv, 0).xy * 2.0 - 1.0;
        float phase0 = frac(time * 0.1 + 0.5);
        float phase1 = frac(time * 0.1 + 1.0);
        float tex0 = TEX2DLOD(noise, uv + flowDir.xy * phase0, 0).r;
                        //The added *10 is for the tiling
        float tex1 = TEX2DLOD(noise, uv + flowDir.xy * phase1, 0).r;
                       //We use the same input in both!

        float tex2 = TEX2DLOD(noise, float2(0, op.z/50), 0).r;
        //this texture is for the x axis to have a noise function applied to it

        float flowMap = lerp(tex0, tex1, abs((0.5 - phase0) / 0.5)).r*tex2;
        local_extinction *= flowMap; //float 2 * float 1 = float 2
        //-----------------------------------------------------------------

        //Apply perlin noise function to local fog unit
        //float nise = Perlin3D(op.xyz + time*float3(1., 0., 0.));
        //local_extinction *= nise;

        scattering += local_albedo * local_extinction;
        extinction += local_extinction;
        //phase += local_phase * local_extinction;
        total_weight += local_extinction;
    }
}
#endif
```

The principal differences between both this and the previous code is that instead of calling the ***TEX2D*** macros in both textures ***tex0*** and ***tex1*** we call ***TEX2DLOD***. The ***wp*** (world position) coordinate was first substituted by ***op*** but it was wrong to use the general form without specifying the channels/axis, so we corrected it using the ***op.xy***, x and y channels as *float2* as the textures were in *float2*, the flow map and all.

And then in order to not have static columns of fog because the z axis, the height, was not affected by the flow map. In other words, in 3D you could only see the flow from a top view and from the side it was quite uniform. We created a new texture, ***tex2***, which added noise in the ***op.z*** axis.

The velocities or speeds in the ***phase0*** and ***phase1*** were increased from 0.05 to 0.1. Before the *lerp* was applied the previous implementation used only one *tex* and it was easier to visualize as it was clearer than with two offsets:

```
float flowMap = tex0.r*tex2;//lerp(tex0, tex1, abs((0.5 - phase0) / 0.5)).r;
local_extinction *= flowMap;
```

Before, uv was declared divided by 100 instead:

```
float2 uv = op.xy/100;
```

Before *tex0* and *tex1* were declared as *float4* but they are not because we only measure the ***.r*** channel:

```
float4 tex0 = TEX2DLOD(noise, uv + flowDir.xy * phase0, 0).r;
float4 tex1 = TEX2DLOD(noise, uv + flowDir.xy * phase1, 0).r;
```

Also, at some point the ***op*** coordinates were used multiplying it like this which was a big mistake:

```
float4 tex0 = TEX2DLOD(noise, op * 10 + flowDir.xy * phase0, 0).r;
float4 tex1 = TEX2DLOD(noise, op * 10 + flowDir.xy * phase1, 0).r;
```

Unfortunately the result could not be perceived properly because the density was too high or perhaps the speed was too fast. To remedy this, the *op* coordinates (now renamed *uv*) were divided by a number to make it smaller and reduce the frequency. Finally, we lerped the values multiplying lastly by the z-axis texture noise

and *local_extinction* *= *flowMap*. The results can be appreciated in figure 4.5.1.

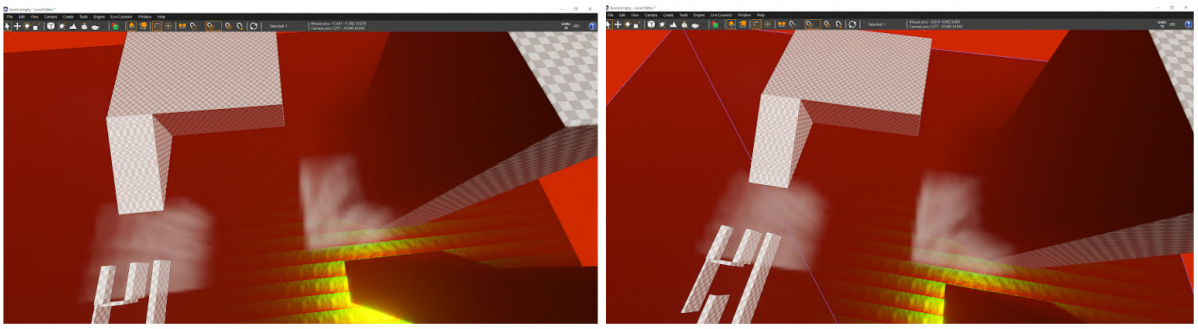


Figure 4.5.1: Time-lapsed results of the flow map applied to volumetric fog units

4.6 Conclusions So Far

When applying the same concept to the overall volumetric fog, we realized the end result was not as perceivable. The result successfully provided a randomized interactive effect to the fog data but was too unclear or translucent to admire. We proceeded to adjust some of the variables that control its other properties like increasing the density and the global extinction as well as the offset and Distance values. The reason why the effect was more perceivable in the individual samples of fog was because the density and extinction of these were higher.

The result provides a more interactive and non-stationary behavior to the fog seen at the beginning of section 4.3 (figure 4.6.1) so it entails a step on the right direction.

We reached the point where we have more or less an idea of what needs to be done. We have checked that flow maps do work with 3D textures but these can only be 2D so it only affects the x and y axis leaving no interactivity in the z axis. Some alternatives and solutions, but mainly ideas, for this were drawn like:

- Researching for the possibility of applying 3D noise in the Z axis
- Research more on the so-called *Flow shaders* or on how to apply flow maps to noise functions and similar.
- Or, how to calculate the height map on a 3D model or terrain because this implementation lends for working with the z axis which generally is the dimension that height maps control better.

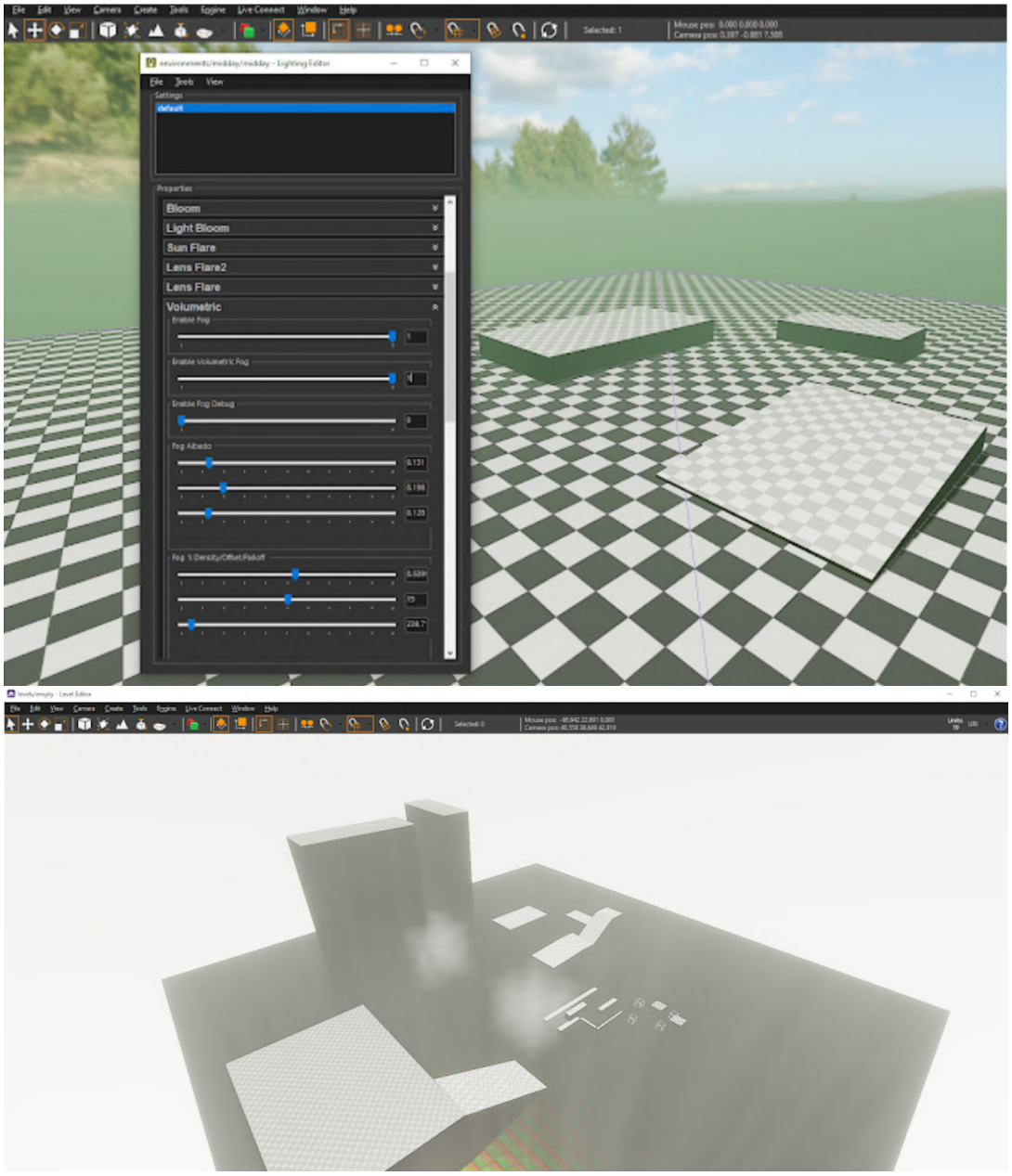


Figure 4.6.1: Initial behavior of the volumetric fog data in the *Stingray Engine*(top) compared to the time-lapsed results of the flow map applied to volumetric fog units(bottom)

4.7 Corona Virus Outbreak

Unfortunately this project was greatly affected by the *COVID-19* outbreak at the beginning of March 2020. Its progress haulted heavily due to the difficulties in communication and lack of resources on the first weeks. Alternative measures were provided to continue the work like a remote connection and, later on, an office computer. However, the first weeks turned out to be overflowed with additional measures of course lessons and grading techniques at KTH. Added to the already existing issues with remote connections and communication to the now *work-from-home* environment, efficiency decreased.

Supervisors and supporting employees were, in a way, less accessible although always available through video and message chats. Mobility was reduced so the attendance to an office as a distinct working environment from that of a distracting home was non-existent. Viewed mostly as a commodity factor that evolved into an uncomfortable environment that mildly affected the psychological toll of the staying-at-home work conditions.

Moreover, due to the complexity of the resources and advanced graphics software environment, the efficiency of the new changes and additions to the program was reduced. Mainly due to the lagging execution and time-consuming set-up of the simulation.

When the outbreak of the pandemic had calmed down, I was permitted to go to the office to pick up a work computer. It provided a faster work dynamic but the exceptional case of having to set-up the software programs and frameworks of the work computer took some time. Mostly due to their complexity and heavy memory space and complex set up instructions.

4.8 New Additions

It was hard to continue the train thought of the last version of the project. Between final examinations and the adjustment to new working conditions, the work suffered a loss of integrity which had to be picked up.

After the software environment and all the consequent tools were properly installed. We figured a way to address the current issue at hand: providing interactivity in the Z-axis as the X and Y were vessels for the effects of a flow map integration implementation.

A new method was thought of which involved the use of an integrated tool in the Level Editor of the *Stingray* engine. It simply generated a texture containing the specification of the height map information pertaining to the simulation terrain. This idea was, in part, born from the new found goal of providing the fog of the possibility of detecting obstacles if only in this particular scenario.

4.8.1 Height Maps

In computer graphics, a *heightmap* or *heightfield* is generally a 2D texture used mainly as Discrete Global Grid in secondary elevation modeling. Each pixel store values, such as surface elevation data, for display in 3D computer graphics. A height map can be used in bump mapping to calculate where this 3D data would create shadow in a material. In displacement mapping, it can be used to displace the actual geometric position of points over the textured surface. For terrain mapping it converts the height map into a 3D mesh[7] as seen in figure 4.8.1

White represents the highest height and the black the lowest height. In figure 4.8.1 we can see a river in the left picture being at sea level, which is considered the lowest height in the terrain context; and a volcano in an island on the right picture with a gradual color change from dark gray to white signifying its peak.

[7]The structural build of a 3D model consisting of polygons. 3D meshes use reference points in X, Y and Z axes to define shapes with height, width and depth.

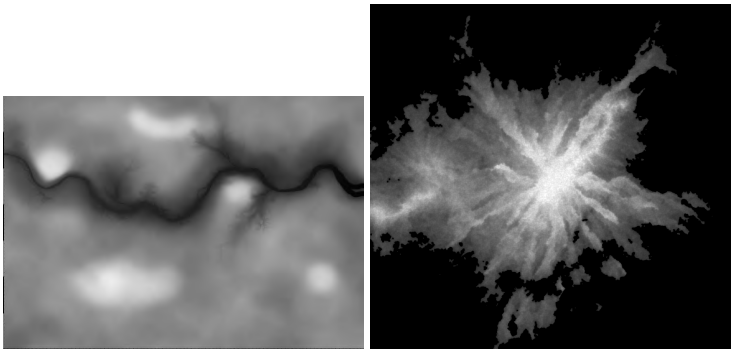


Figure 4.8.1: Examples of terrain topography height maps, left hosting a river and right representing an island volcano

Therefore, we proceeded to generate a terrain with the built-in function of the *Stingray* engine's level editor. When a hill, mountain or other mount or lump is created it is reflected in a 2D texture it generated and modified dynamically. One of the first iterations of this functionality's outcome can be seen in figure 4.8.2.

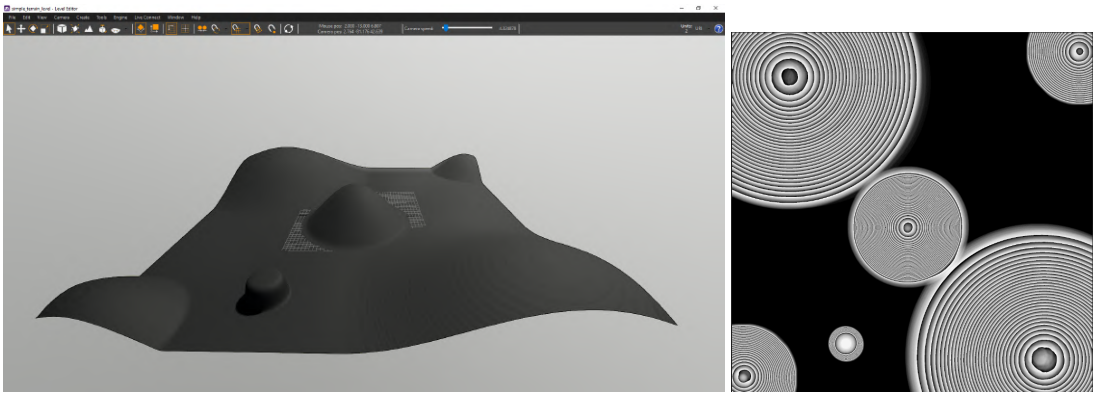


Figure 4.8.2: Initial behavior of the volumetric fog data in the *Stingray Engine* compared to the terrain

The result may appear inconsistent compared to that of figure 4.8.1. However the peculiar result was a choice of design implementation of the built-in functionality as it regarded every modification of every lump and mount created manually with an almost pixel-wise precision.

Nevertheless, the result was enough to provide the intended effect we subsequently will implement. Another built-in functionality the engine had, very closely related to the previous, was the importing and manipulation of these 2D terrain height maps. It generally treated them as the previously manipulated textures. This made it easy to use a particular concept related to the simulated environment which was the height of the terrain.

The control of the heights of this would help devise the slopes and obstacles the terrain has an aid at the time of using its structure as a *flow map* to move the fog in the x and y axis in accordance to the particular terrain.

The solution was replacing the flow map texture with the automatically generated height map texture. Configured at the beginning with 100 meters as maximum height. However, in order to study the results of this built-in texture generator, the height map was applied to a 2D texture or color as previously done with the flow map study.

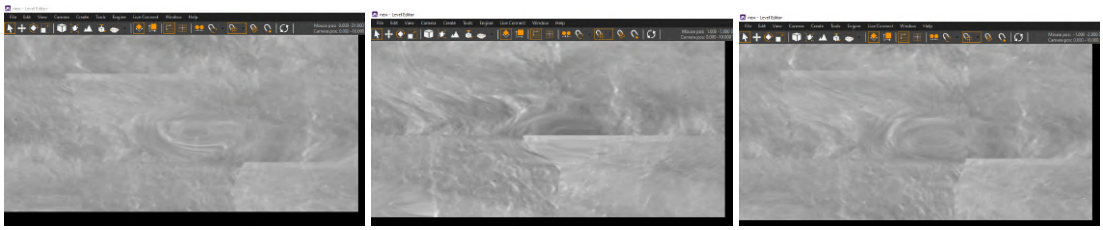


Figure 4.8.3: Result of the applied terrain height map to the previously seen sample texture

The movement can be appreciated well from the sample texture in 4.8.3. Consequently, we proceeded to apply a noise texture instead which is much more uniform and easier to study. With it we can see the sort of movement it generates.

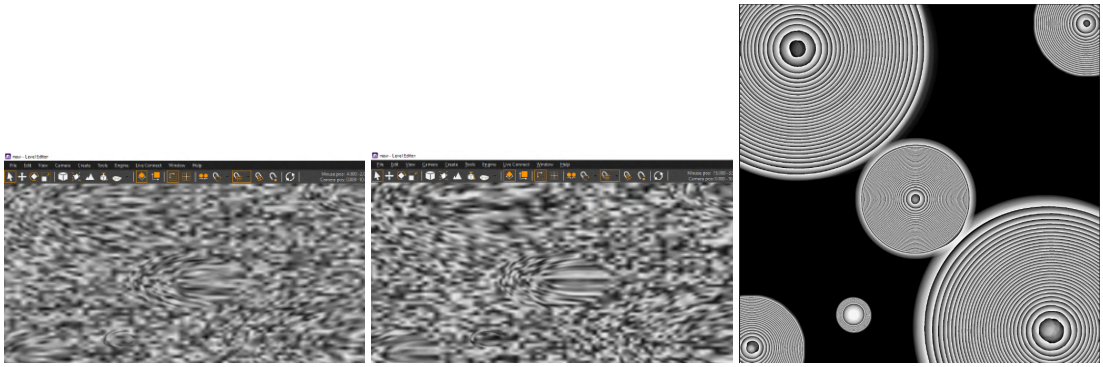


Figure 4.8.4: Result of the applied terrain height map to a noise texture. Compared with the height map

The result was a success. The texture would circle and move according to the different hills and valleys of the terrain. now the tricky part was to add it to the overall volumetric fog of the bigger terrain.

But when it was applied to the fog the results were not clear. So we decided to concentrate on a certain unit of fog and realised it was tiled. Tiling is a technique that augments the frequency or dissects the surface it is going to affect in as many pieces

as specified. In my case I had previously tiled the flow map by multiplying by 10, what that did is dissect in x and y by 10 and apply on that new scaled down section the flow map. It is a technique to add extra noise and therefore a sense of randomness or unpredictability.

```
float4 tex0 = TEX2D(noise, input.uv.xy * 10 + flowDir.xy * phase0); //The
    added *10 is for the tiling
float4 tex1 = TEX2D(noise, input.uv.xy * 10 + flowDir.xy * phase1);
```

After fixing the tiling issue, we finally implemented the effect for the fog data unit. A sampled singular representation of volumetric fog in a 3D object. Additionally, we also switched the function that applies the texture from a 2D to a 3D function as we are operating with 3 dimensions.

```
float2 uv = op.xy/50;
float2 flowDir = (TEX2DLOD(simple_terrain_hmap, uv, 0).rg - 0.5)*2.0;
float phase0 = frac(time * 0.1 + 0.5);
float phase1 = frac(time * 0.1 + 1.0);
float tex0 = TEX3DLOD(noise, uv + flowDir.xy * phase0, 0).r;
float tex1 = TEX3DLOD(noise, uv + flowDir.xy * phase1, 0).r;

float tex2 = TEX3DLOD(noise, float2(0, op.z/50), 0).r;
//this texture is for the z axis to have a noise function applied to it

float flowMap = lerp(tex0, tex1, abs((0.5 - phase0) / 0.5)).r*tex2;
local_extinction *= flowMap; //float 2 * float 1 = float 2*/
```

The result is as eye-grabbing as it is interactive (see figure 4.8.5)

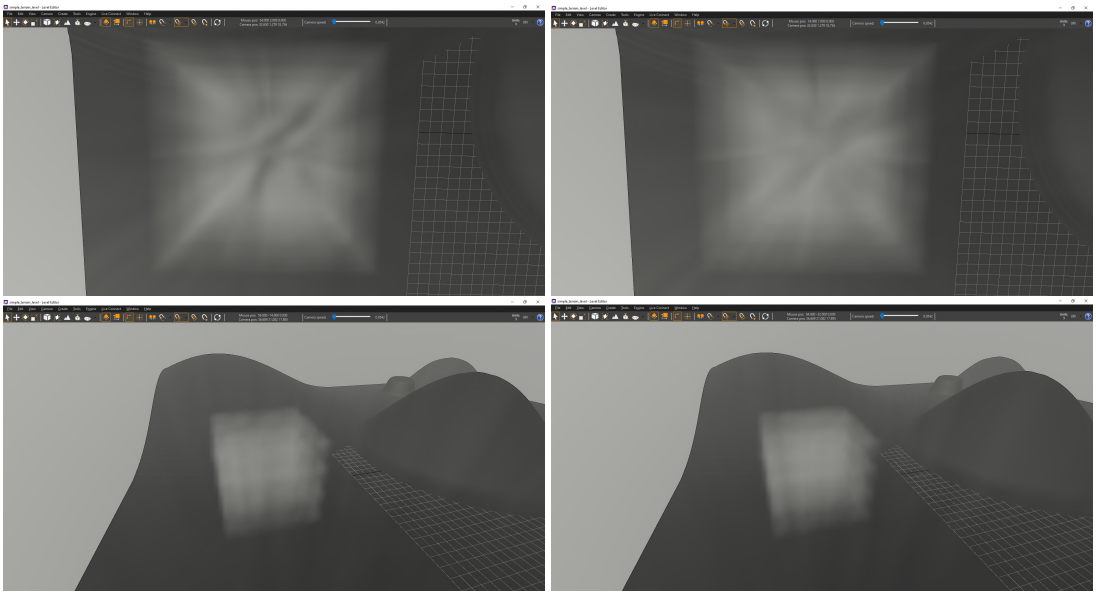


Figure 4.8.5: Result of the applied terrain height map to a sampled fog unit

This new implementation was adapted to the overall volumetric fog by instead of using the local position of a sampled fog unit (*op*) it was the world position (*wp*). The results, while they wouldn't seem much different from the last iteration, had the additional feature of following the shape of the terrain height map applied to it. See figure 4.8.6

```
float2 uv1 = wp.xy/50;
float2 flowDir1 = (TEX3DLOD(simple_terrain_hmap, uv1, 0).xy - 0.5)*2.0;
float phase01 = frac(time * 0.1 + 0.5);
float phase11 = frac(time * 0.1 + 1.0);
float tex01 = TEX3DLOD(noise, uv1 + flowDir1.xy * phase01, 0).r;
float tex11 = TEX3DLOD(noise, uv1 + flowDir1.xy * phase11, 0).r;

float tex21 = TEX2DLOD(noise, float2(0, wp.z/50), 0).r;
//this texture is for the z axis to have a noise function applied to it
float flowMap1 = lerp(tex01, tex11, abs((0.5 - phase01) / 0.5)).r;//.r*tex21

float extinction = max(global_extinction(wp)*flowMap1,0);
//float extinction = global_extinction(wp);
float3 scattering = fog_color * extinction;
float3 emissive = 0;
```

The results will be contemplated further in depth in the following chapter.

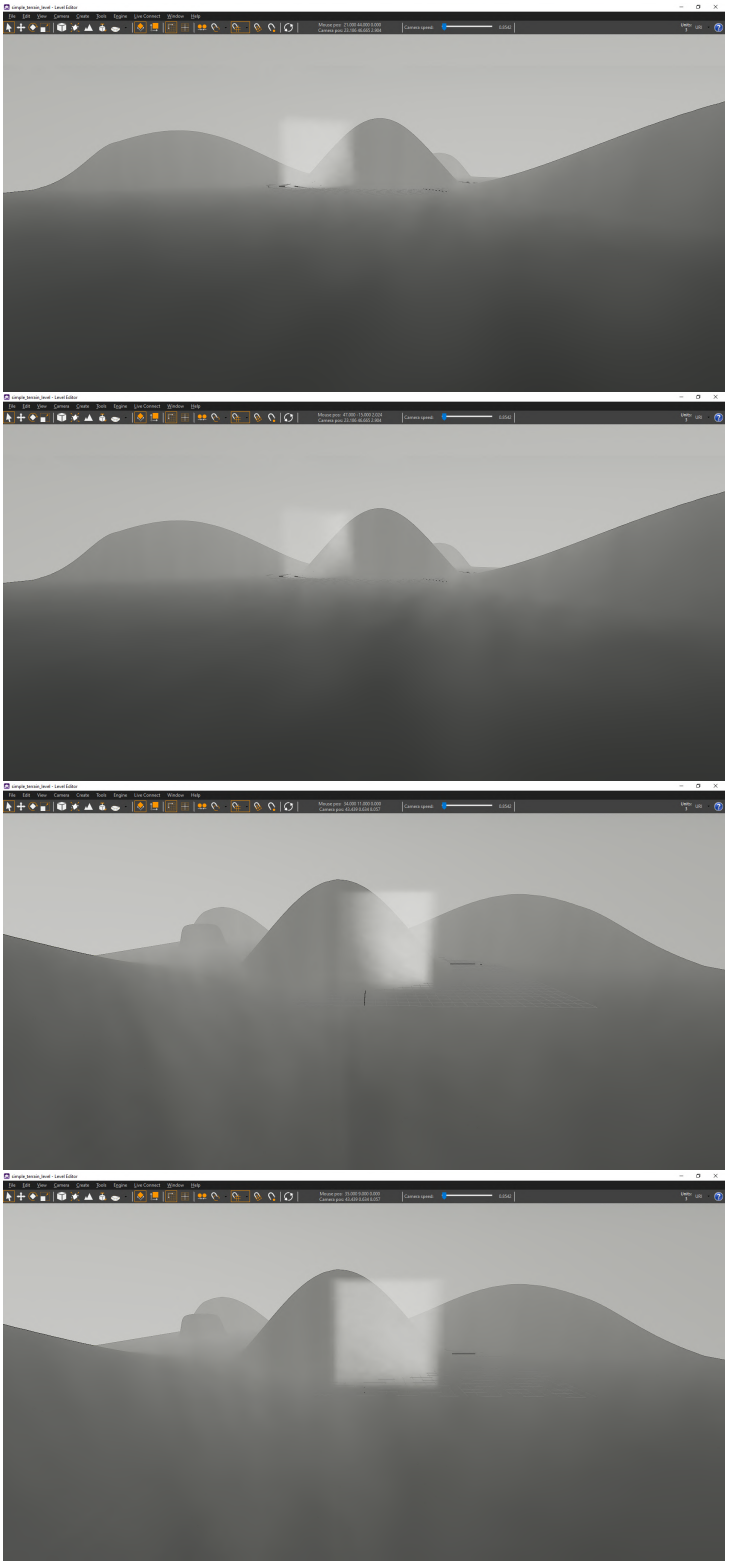


Figure 4.8.6: Result of the applied terrain height map to the entire volumetric fog data

Chapter 5

Results

As we can see in the initial state of the volumetric fog data provided by the company (see figure 5.0.1), the fog showcased a uniformity and static behavior deprived of realism. All the code and outcomes previously presented in the Work section (section 4) are part of the end result of the project. It took many small steps starting with the mathematical foundations and then exploiting the potential of computer graphics.

The toll the COVID-19 outbreak put on the biggest was the testing and debugging after every modification to the code. Mainly due to the vast number of calculations and computational power that was needed to run the program. But moreover, the lack of a powerful work station that had become inaccessible because of it. Of course, this can not be attributed to the hosting company which had provided additional methods and assistance. Nevertheless, it is undenying the set back it meant for the final stages of the project.

As an empirical research, it relied heavily on trial and error, changing parameters to scrutinize the results and infer the reason why. Therefore, in its final stages when the effect was rendered successful and effective, the final feature implementation had to be dropped. Added instead to the future work section of the report.

As final result we have the following screenshots of the different effects accomplished in the implementation of the shader code that we have seen previously:

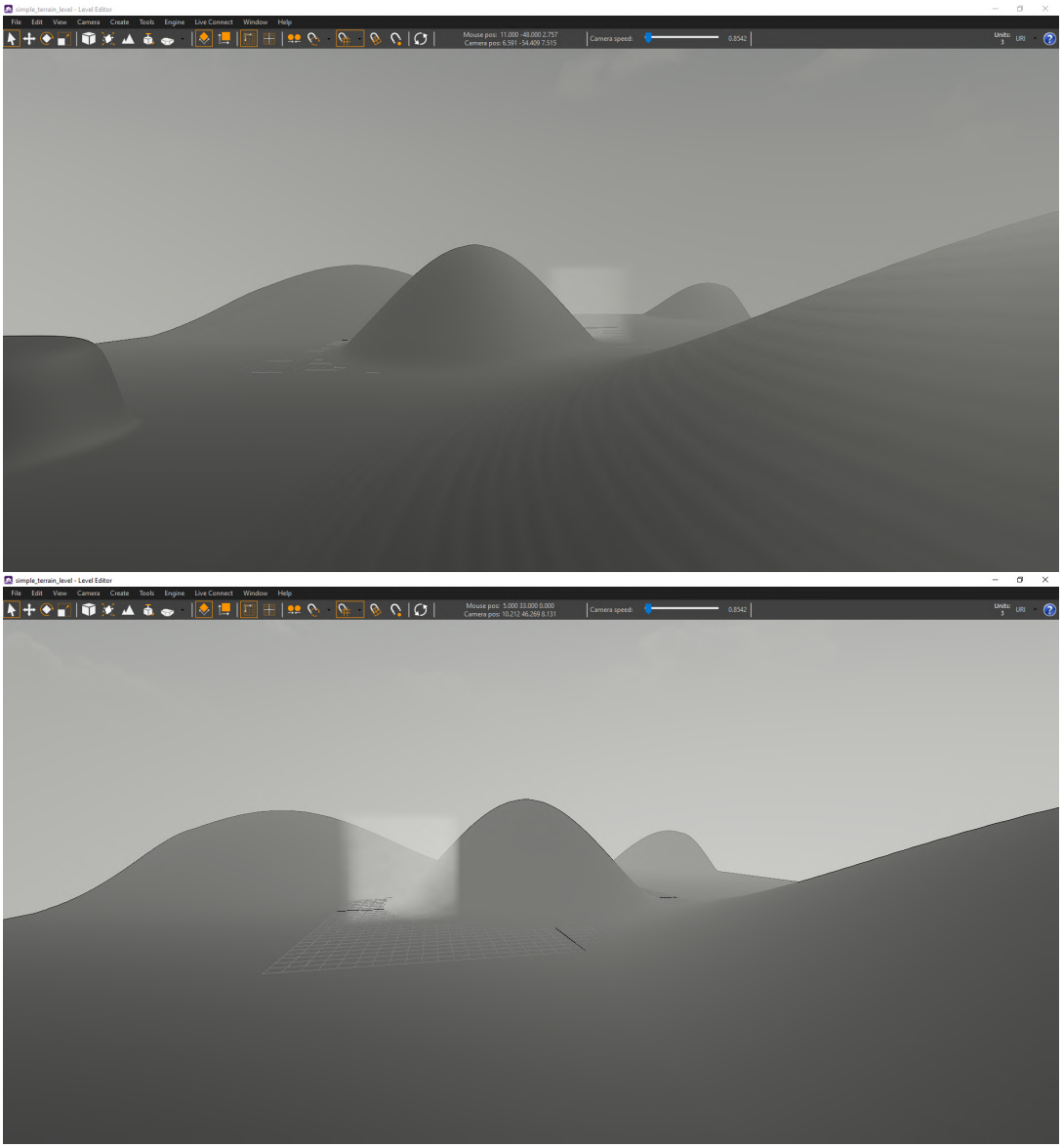


Figure 5.0.1: Snapshots of the initial state of the Volumetric Fog and Fog Unit in Fatshark's Stingray engine implementation

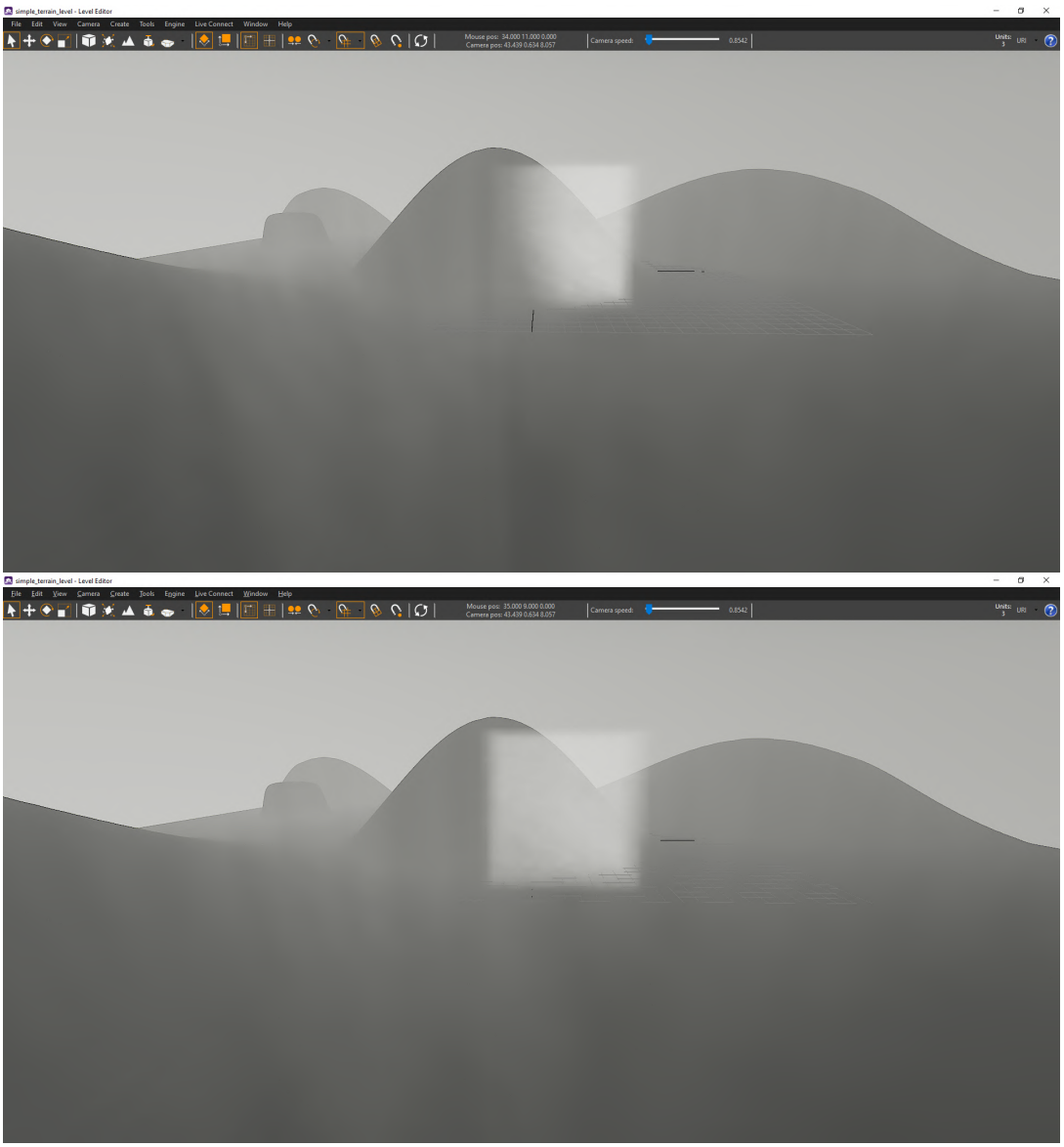


Figure 5.0.2: Snapshots of the execution of the final implementation of the real-time simulation

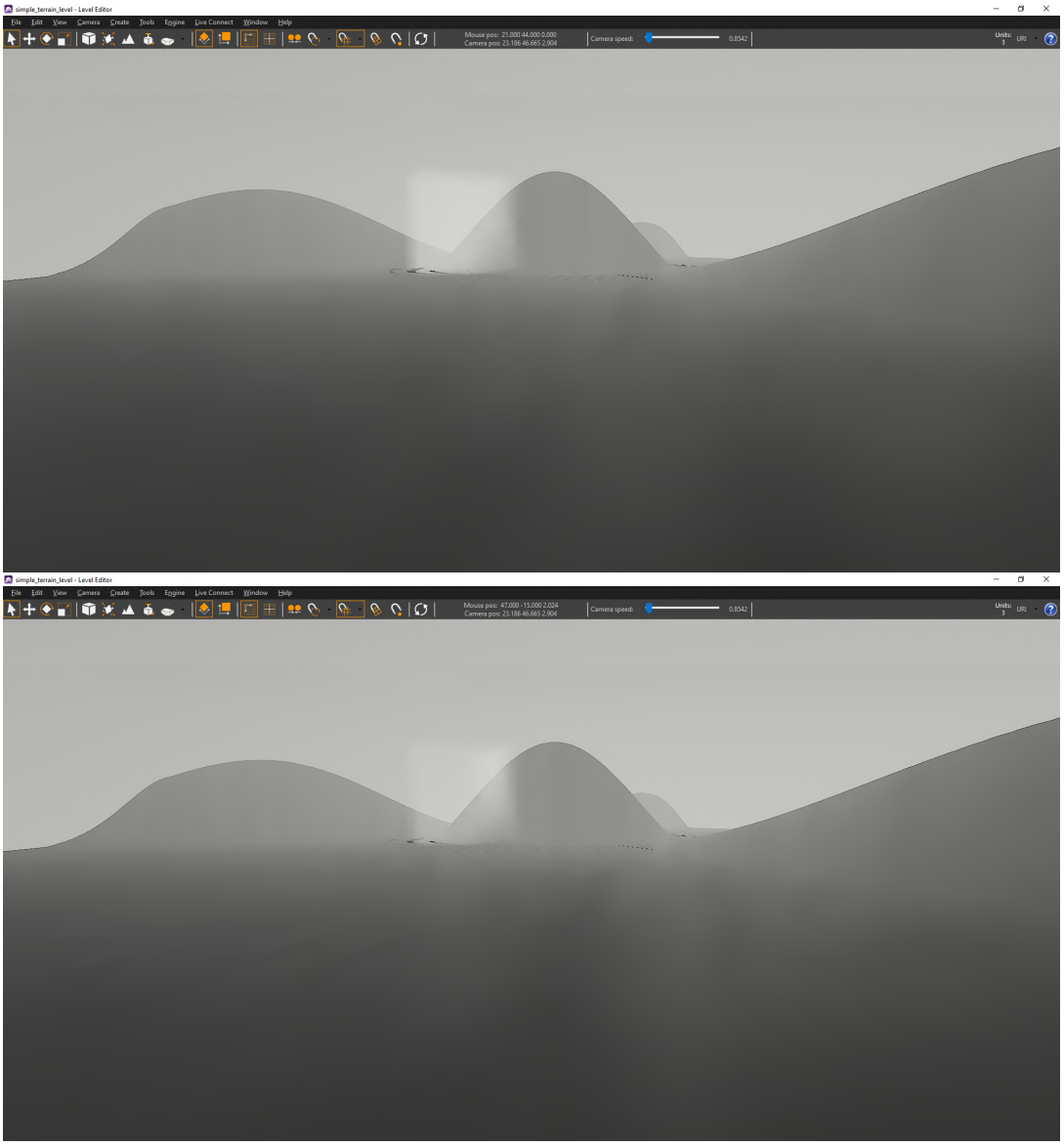


Figure 5.0.3: More Snapshots of the execution of the final implementation of the real-time simulation (different angle)

The success of the correct implementation of these fundamental concepts allowed for the effects shown in figure 5.0.2. Although hard to distinguish, we can appreciate the sort of dynamism and immersive atmospheric effect we have accomplished.

The effect is subtle and the noise applied to it prevents the defined distinction that is clearly visualized in the 2D version. However, this may be due to the lack of research on the powerful and complex volume data provided by the company (the fog system). Should more time was available the more it could have been improved.

On figure 5.0.3 we can see the results on the real-time simulation of a sample fog unit volume data. However this one is an additional implementation of a speed applied in the Z-axis. It wasn't mentioned in the work because it was a last minute implementation that didn't work together with the prior code. Nonetheless, it has a soothing effect that applies a speed function to it. But it didn't comply with the noise or the height map, so it was scrapped.

And in figure 5.0.4 are some more snapshots of the same simulation from a different angle.

The final implementation of the sample fog results in figure 5.0.6. It successfully shows the desired noisy effect and flow. And, although hard to appreciate, it provided the sample fog of a realistic cloud-like behavior.

The results are rendered successful. At the beginning the aim was to generate an add-on functionality that could provide an interactive behavior to the volume data. To make it affect and be affected by the design and obstacles of any kind of simulated environment. However, the scope was found too big. The level of expertise required not only in the area of study but in the tools used, was too great.

However, the experience and preparation granted by the software engineering graduate and postgraduate studies in different tier-leveled software project development provided assistance and ease. Especially when adapting to the requirements of this new area of research. At first the planning for the project was a bit loose and dependant on the complexity of the subareas to be researched. When dedicating the first months for accustoming to the software developing tools and the background notions mentioned in section 2.1

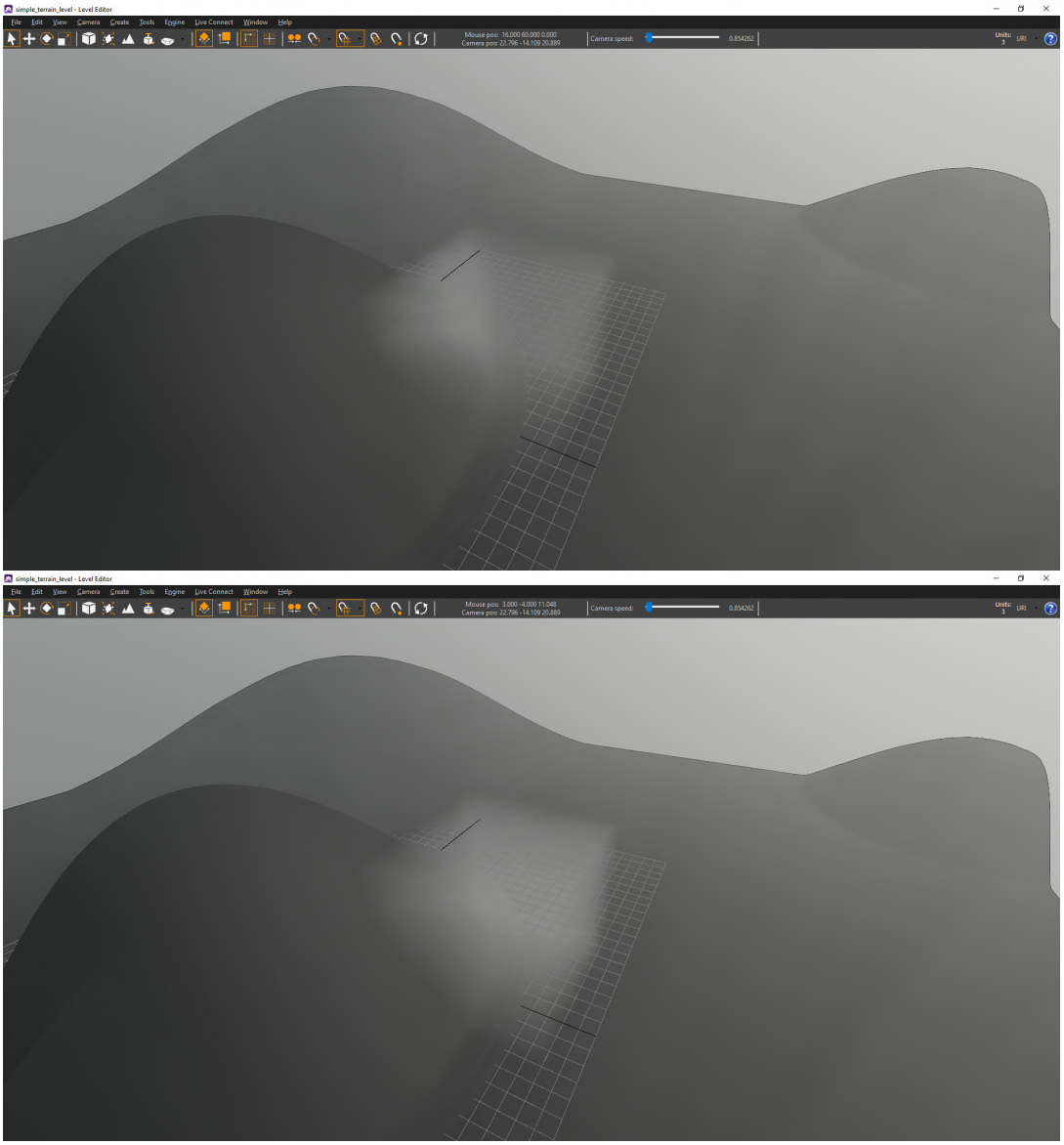


Figure 5.0.4: Snapshots of the execution of the scrapped feature in the real-time simulation of sample volume data

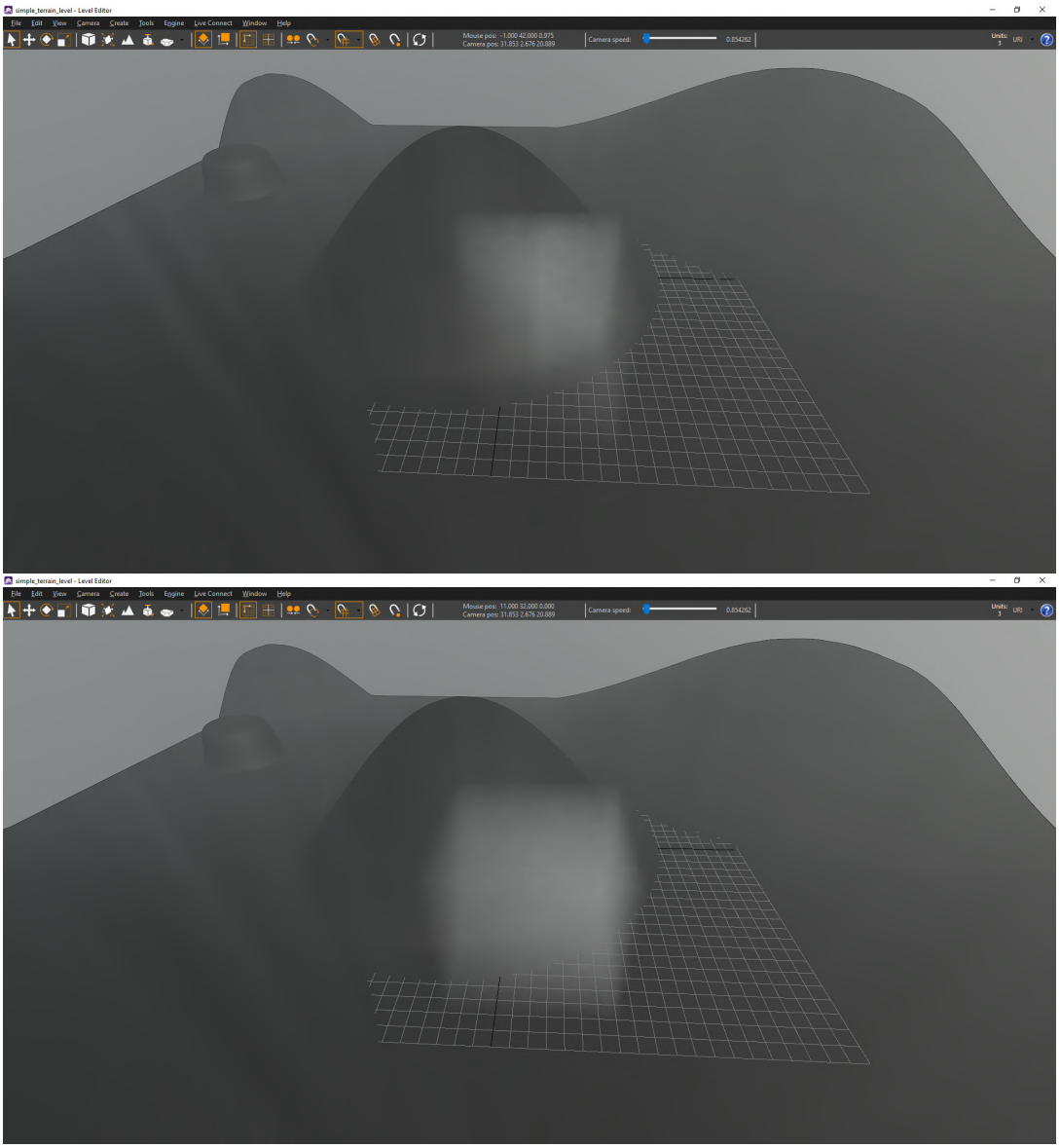


Figure 5.0.5: Snapshots of the execution of the real-time simulation of compacted volume data from a different angle

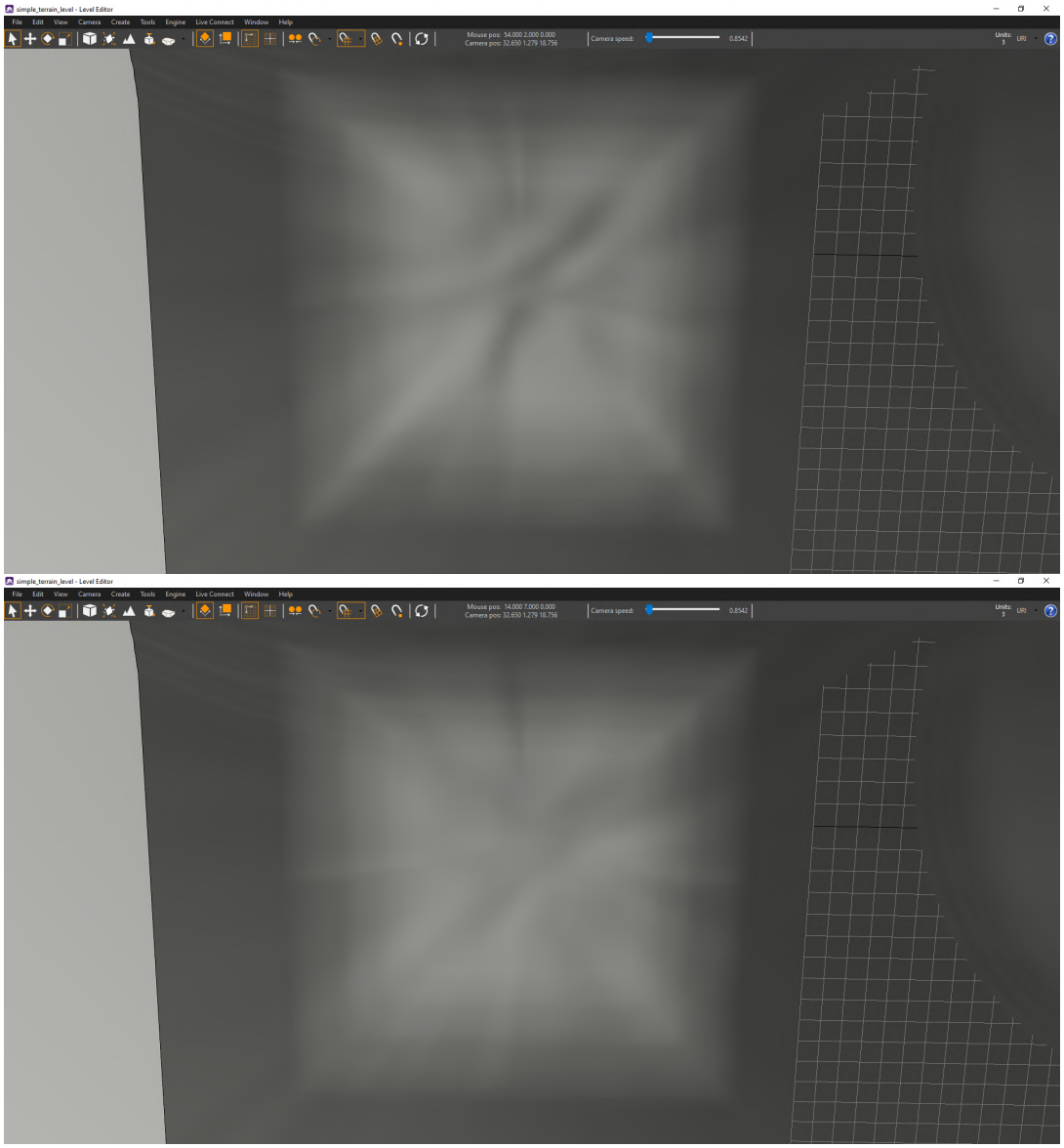


Figure 5.0.6: Snapshots of the final execution of the real-time simulation of sampled volume data

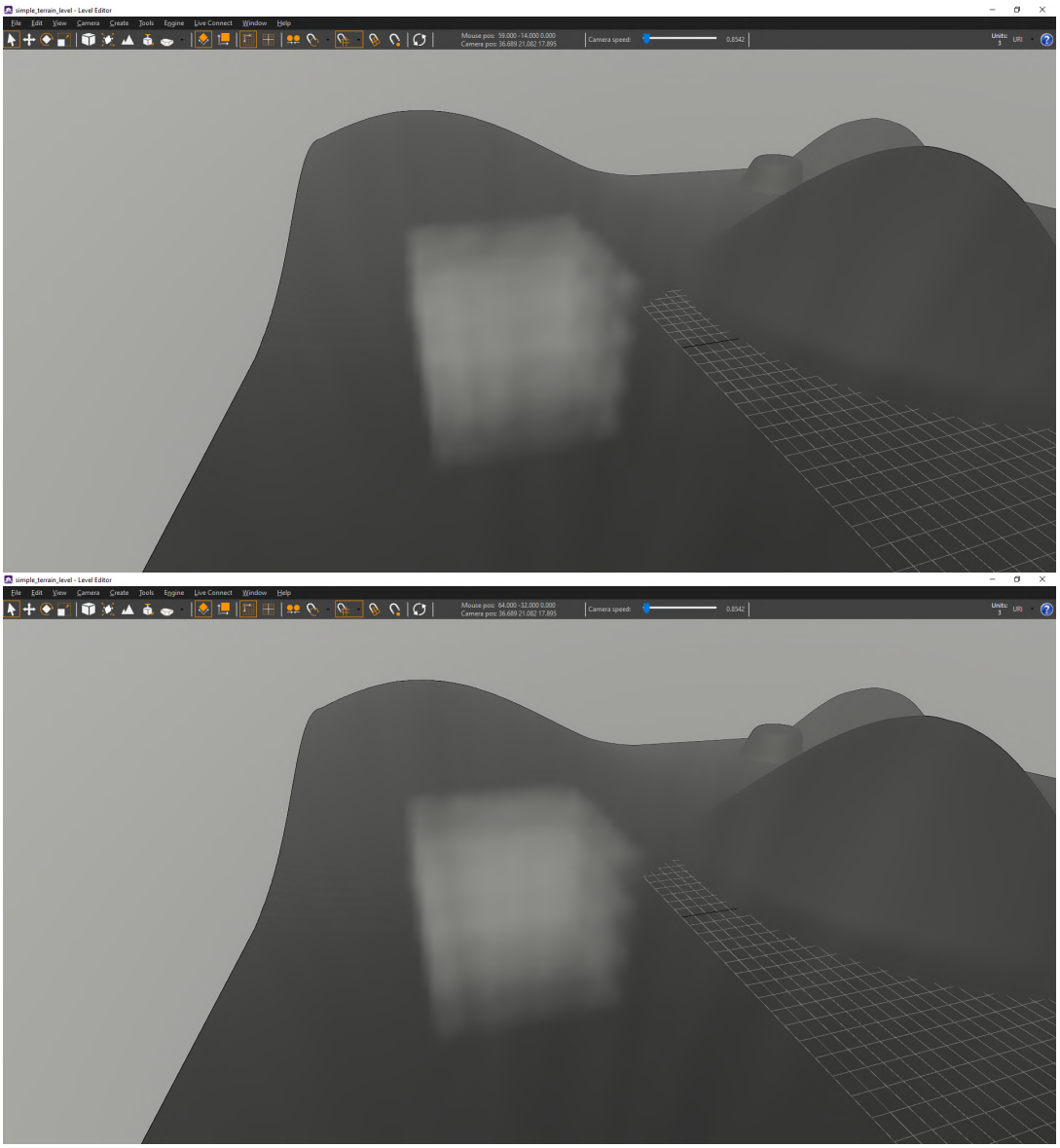


Figure 5.0.7: More snapshots of the final execution of the real-time simulation of sampled volume data, different angle

Much was learned from the background researched, enough to understand the scope of any kind of project of this magnitude. The assistance of company coworkers and the immersion in an environment dedicated to the development of the highest quality of computer graphics was encouraging. It provided an extra layer of work dynamic and quality to the project. Mostly thanks to the inclusion in many meetings and peeks at the life-cycle of the projects that were being developed there. It served as an insightful look at the planning dynamic, the outcomes of reaching or not the set milestones and the contributions every member provided to the end product.

Chapter 6

Conclusions

The project served its overall purpose: to provide an insightful introduction to the knowledge needed to implement a real-time volume rendering simulation.

My personal interest in the subarea and the immersive working environment at the company encouraged this impactful journey. It provided an additional excitement to learn more about an area that was too vast to learn in a single project. At times it felt like more time was needed and wished upon to dig more into it. Bigger volumes of knowlege had to be scrapped in order to focus on the matter at hand. Books were left unread, methods were scratched the surface off and technology was only slightly glanced upon. But like with any area of research, unless it is pioneering in nature, it draws foundation and basis on similar areas from where they are born. And in that way, created a cycle of knowledge that would take more than one semester to be able to fully grasp.

As previously mentioned in the Result chapter, the biggest issue came with the initial scope of the project. Such was the engagement and advancing of the project that in an effort to prove my commitment worth, a scope was promised too impossible to grasp. It was too far from the knowledge and expertise owned and could have plausibly learned in such short time. But then the pieces fell together quite nicely without drifting away from the initial concept of the thesis. A new scope was born, one that was much more centered and therefore straight and direct.

Unfortunately so the second biggest drawback arrived by the name of COVID-19. Not to dwell on it too much, but the impact of stripping the project away from such healthy and exciting work environment also stripped away some of the inspiration. However,

the biggest toll was not really on that matter but rather the pile of issues that arose with it. Alternative measures were imparted in the teaching at KTH, the set-up issues and reconfigurations of the borrowed office laptop and a personal tragedy in my family that lived in Spain. The constant worrying and the stress of not succeeding at setting up properly (on my behalf, not the company's) the tools needed made for the loss of the *train of thought*. And the freshness of the new scope and ideas that were discussed were not recaptured the same.

The progression of the thesis has halted heavily due to the outbreak of the Corona virus. Some advancements in the organization of the documentation of the thesis was made. Given I had visited my family in Spain the weekend before the outbreak in Spain I was highly recommended to stay and work from home. Now, because of this new situation and the fact that it took me a while to set up the remote access to the work computer, I could not work much on the implementation. I counted with the help of my supervisor as he assured me his availability. Nevertheless I found it was a setback to the usual dynamic available in the office. Some of the most advanced software tools were in the work computer and when connecting remotely with it resulted slow and glitchy. This caused an intermittent and slow testing efficiency.

Moreover, it should be stated that it is not the company's fault the issues arising from the software pulled out and mercurial problems as the exceptional circumstances prevented for an efficient "work from home" environment.

6.1 Future Work

As previously mentioned, the project suffered a bit of a set back that tampered with the scope of it. Therefore, the future work includes the implementation of a more pixel-specific fog effect that could mold and adapt itself to any sort of terrain dynamically.

This thesis began as an implementation of a real-time volume rendered data program that could read the terrain and base the vector field and speed to it. However, due to the halting work dynamic and the seemingly out-of-scope thesis idea, it had to be rethought of and scoped in a way that it was plausible for someone without that much expertise.

More applications of the implemented code that have come up during the research of it is its use for volumetric clouds that behave similarly in high terrain environment, like the one seen in figure 6.1.1.

Figure 6.1.1: Example of low clouds coursing through the mountains in a fluid-like motion as studied earlier

A research that could be assisted by the works such as Fredrik Häggström's paper on Real-time rendering of volumetric clouds [6] or other *Stingray* similar projects [5]. But also to improve the end result of real-time simulations by improving the fluid dynamics used to simulate the behavior of the volume data of both the single fog units and overall volumetric fog like in Jos Tam's paper on Real-Time Fluid Dynamics for Games [18] or by improving the interactivity with objects dynamically like in Unreal Engine's study similar to the one in figure 6.1.2.

Figure 6.1.2: Example of simulation of fog to interact and dynamically change with movement

And many other improvements pertaining the inclusion of more environmentally immersive additions such as a changing wind or speed being applied to the fog. Like the one presented at Game Developers Conference (GDC) 2019's Sony Santa Monica Studio proceeding of *Wind simulation in God of War* [15]. Or the inclusion of *cloud maps* or volumetric clouds using the same principle used for the volumetric fog like the method presented by Rockstar Games in SIGGRAPH 2019's Advances in Real-Time rendering conference [2]. See figure 6.1.3 for a look at the way they addressed it which can be accomplished with the concepts exposed here.

Furthermore, in 2017 SIGGRAPH convention, Guerrilla Games presented an approach to artistically authoring real-time volumetric cloudscapes for games [16]. By combining this method with the aforementioned one used by Rockstar, a realistic real-time cloud simulation can be accomplished.

Figure 6.1.3: Reproduction of Cloud Map method slides presented in SIGGRAPH 2019 by Rockstar Games

Bibliography

[1] Aaron Lefohn, Natalya Tatarchuk. *SIGGRAPH 2017: Open Problems in Real-Time*. URL: http://openproblems.realtimerendering.com/s2017/index.html. (accessed: 03.05.2020).

[2] Bauer, Fabian. *SIGGRAPH 2019 Advances in Real-Time rendering in 3D graphics and Games: Creating the Atmospheric World of Red Dead Redemption 2: A Complete and Integrated Solution*. URL: https://advances.realtimerendering.com/s2019/index.htm. (accessed: 8.06.2020).

[3] Brucks, Ryan. *UE4 Volumetric Fog Techniques*. URL: https://shaderbits.com/blog/ue4-volumetric-fog-techniques. (accessed: 10.06.2020).

[4] Fernando, R. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2005. URL: https://developer.nvidia.com/gpugems/gpugems/.

[5] Grenier, Jean-Philippe. *Volumetric Clouds in Stingray*. URL: http://bitsquid.blogspot.com/2016/07/volumetric-clouds.html. (accessed: 10.06.2020).

[6] Häggström, Fredrik. "Real-time rendering of volumetric clouds". In: 2018. URL: http://www.diva-portal.org/smash/get/diva2:1223894/FULLTEXT01.pdf.

[7] *Proc. The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013*. Lecture Notes in Computer Science. Las Vegas USA: CSREA Press U.S.A: Springer, 2013.

[8] Nguyen, H. and Corporation, NVIDIA. *GPU Gems 3*. Lab Companion Series v. 3. Addison-Wesley Professional, 2008. ISBN: 9780321515261. URL: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-30-real-time-simulation-and-rendering-3d-fluids.

[9] Olajos, Rikard. *Real-Time Rendering of Volumetric Clouds*. URL: http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8893256&fileOId=8893258. (accessed: 13.06.2020).

[10] Osiris, Taras. *Flow Map Shader for Unity Sprites*. URL: https://somecodingrecipes.wordpress.com/2015/03/24/flow-map-shader-for-unity-sprites/. (accessed: 21.05.2020).

[11] Patricio Gonzalez Vivo, Jen Lowe. *The Book of Shaders*. URL: https://thebookofshaders.com/. (accessed: 21.05.2020).

[12] Quilez, Inigo. *Fractal Brownian Motion*. URL: https://iquilezles.org/www/articles/fbm/fbm.htm. (accessed: 21.05.2020).

[13] Quilez, Inigo. *Normals for a SDF (Signed Distance Field)*. URL: https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm. (accessed: 21.05.2020).

[14] Raudsepp, Siim. *Siim Raudsepp Volumetric Fog Rendering*. 2018.

[15] Renard, Rupert. "Wind Simulation in God of War, Sony Santa Monica, GDC 2019". In: 2019.

[16] Shneider, Andrew. *SIGGRAPH 2017 Advances in Real-Time rendering: Nubis: Authoring Real-Time Volumetric Cloudscapes with the Decima Engine*. URL: https://advances.realtimerendering.com/s2017/index.html. (accessed: 8.06.2020).

[17] Sköld, Philip. "Philip Sköld Real Time Volumetric Ray Marching with Ordered Dithering: Reducing required samples for ray marched volumetric lighting on the GPU". In: 2019. URL: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%5C%3A1273389&dswid=8399.

[18] Stam, Jos. *Real-Time Fluid Dynamics for Games*. URL: https://pdfs.semanticscholar.org/847f/819a4ea14bd789aca8bc88e85e906cfc657c.pdf. (accessed: 10.06.2020).

[19] Team, Autodesk Support. *System requirements for Autodesk Stingray*. URL: https://knowledge.autodesk.com/search-result/caas/sfdcarticles/sfdcarticles/System-requirements-for-Autodesk-Stingray.html. (accessed: 15.06.2020).

[20] Vlachos, Alex. “Water Flow, Valve, SIGGRAPH 2010 Advanced Real-Time Rendering Course”. In: 2010.

[21] Wronski, Bartlomiej. *SIGGRAPH 2014: Volumetric Fog: Unified Computer shader based solution to atmospheric scattering*. URL: http://advances.realtimerendering.com/s2014/#_VOLUMETRIC_FOG:_UNIFIED. (accessed: 30.05.2020).

Appendix - Contents

Appendix A

Shader_Test

Code of the ***shader_test*** shader script for implementing the various effects for the 2D textures including Flow Map, Vector Field and Noise. Most of the structure is reducted as the content belonngs to *Fatshark AB*, however, they are added to the code below to serve as skeleton of the script and also because it is available to use in other versions of the *Stingray Engine*

```
includes = ["..."
/************** "REDACTED" **************/
...
"core/stingray_renderer/shader_libraries/common/math.shader_source",
        "core/stingray_renderer/shader_libraries/common/noise.shader_source"]

render_states = {
/************** "REDACTED" **************/
}

sampler_states = {
}

hlsl_shaders = {

   test_1 = {
      includes = [ "common" ]
      samplers = {
         input_texture0 = { sampler_states = "clamp_point" }
```

```
    input_texture1 = { sampler_states = "clamp_point" }
}

code="""
    DECLARE_SAMPLER_2D(input_texture0);//hdr0_rgb
    DECLARE_SAMPLER_2D(input_texture1);//linear depth

    struct VS_INPUT {
        float4 position : POSITION;
        float2 uv : TEXCOORD0;
    };

    struct PS_INPUT {
        float4 position : SV_POSITION;
        float2 uv : TEXCOORD0;
    };

    CBUFFER_START(c0)
        float4x4 world_view_proj;
    CBUFFER_END

    DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
    PS_INPUT vs_main(VS_INPUT input) {
        PS_INPUT o;
        o.position = mul(input.position, world_view_proj);
        o.uv = input.uv;
        return o;
    }

    DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
    float4 ps_main(PS_INPUT input) : SV_TARGET0 {
        float3 c = TEX2D(input_texture0,
            input.uv).rgb;//input_texture0.tex.Load(int3(input.position.xy
            / 8, 0)).rgb; Axel's alternative code
        float depth = TEX2D(input_texture1,
            input.uv).r;//input_texture0.tex.Load(int3(input.position.xy /
            8, 0)).rgb;
```

```
        float3 color = c * lerp(float3(1, 0, 0), float3(0, 0, 1),
            smoothstep(5, 10, depth));

        return float4(color, 1);
      }
    """
  }

  color_init = {
    includes = [ "common" ]
    samplers = {
    }

    code="""
      struct VS_INPUT {
        float4 position : POSITION;
        float2 uv : TEXCOORD0;
      };

      struct PS_INPUT {
        float4 position : SV_POSITION;
        float2 uv : TEXCOORD0;
      };

      CBUFFER_START(c0)
        float4x4 world_view_proj;
      CBUFFER_END

      DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
      PS_INPUT vs_main(VS_INPUT input) {
        PS_INPUT o;
        o.position = mul(input.position, world_view_proj);
        o.uv = input.uv;
        return o;
      }
```

```
        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        float4 ps_main(PS_INPUT input) : SV_TARGET0 {
              float dist = distance(input.uv, 0.5);
              float r = smoothstep(0.3, 0.29, dist);
           //SMOOTHSTEP FUNCTION: applies a formula of cubic function which
              takes the basic gradient that I have and control it so that I
              know where the black and white colors go and how is the
              transitions between white and blacks. The closer the numbers
              are to each other the neatier that distinction is: completely
              black circle (if the right one is bigger than the left one) or
              a completely white circle if viceversa.
              return float4(max(float3(r, 0, 0),0), 1); we create and return
                 the color here max function: the max function will return a
                 number between 0 and LEFT_HAND_ARGUMENT (float3(r, 0, 0) in
                 this case)
        }
     """
}

vector_field_init = {
   includes = [ "common" "noise_functions"]
   samplers = {
   }

   code="""
      struct VS_INPUT {
         float4 position : POSITION;
         float2 uv : TEXCOORD0;
      };

      struct PS_INPUT {
         float4 position : SV_POSITION;
         float2 uv : TEXCOORD0;
      };

      CBUFFER_START(c0)
         float4x4 world_view_proj;
```

```
      CBUFFER_END

      DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
      PS_INPUT vs_main(VS_INPUT input) {
         PS_INPUT o;
         o.position = mul(input.position, world_view_proj);
         o.uv = input.uv;
         return o;
      }

      DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
      float4 ps_main(PS_INPUT input) : SV_TARGET0 {
         /*float a = Perlin2D(input.uv * 20) * PI;//adding noise
         float x = cos(a);
         float y = sin(a); */

         float x = 1;
         float y = 0;

         float dist = distance(input.uv, 0.5);
         float r = smoothstep(0.3, 0.29, dist);
         return float4(max(float3(x, y, 0),0), 1);
      }
   """
}

simulate_NOISE = {
   includes = [ "common" "noise_functions"]
   samplers = {
      input_texture0 = { sampler_states = "clamp_linear" } //LINEAR
          INTERPOLATION!
      input_texture1 = { sampler_states = "clamp_linear" }
   }

   code="""
      DECLARE_SAMPLER_2D(input_texture0);
      DECLARE_SAMPLER_2D(input_texture1);
```

```
struct VS_INPUT {
   float4 position : POSITION;
   float3 uv : TEXCOORD0;
};

struct PS_INPUT {
   float4 position : SV_POSITION;
   float3 uv : TEXCOORD0;
};

CBUFFER_START(c0)
   float4x4 world_view_proj;
CBUFFER_END

DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
PS_INPUT vs_main(VS_INPUT input) {
   PS_INPUT o;
   o.position = mul(input.position, world_view_proj);
   o.uv = input.uv;
   return o;
}

struct PS_OUTPUT {
   float2 velocity : SV_TARGET0;
   float4 color : SV_TARGET1;
};

//Noise function
float sin_noise (float2 input){
   float2 x = sin(input);
   return x.x*x.y;
}

#define OCTAVES 6
float fbm (float2 st) {
   // Initial values
```

```
    float value = 0.0;
    float amplitude = .50;
    float frequency = 0.;
    //
    // Loop of octaves
    for (int i = 0; i < OCTAVES; i++) {
       value += amplitude * sin_noise(st);
       st *= 2.;
        amplitude *= .5;
    }
    return value;
}

DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
PS_OUTPUT ps_main(PS_INPUT input) {
    //This function is called for every frame repeatedly so we placed
        the
    //sin function here so that the "time" variable is updated at every
        frame
    //We added a "noise_functions"

    /*float2 dt = TEX2D(input_texture0, input.uv).rg * 0.01 *
        delta_time;
    float3 color = TEX2D(input_texture1, input.uv - dt).r;

    float2 new_velocity = TEX2D(input_texture0, input.uv - dt).rg;

    PS_OUTPUT o;
    o.color = float4(color, 1);
    o.velocity = new_velocity;
    return o;*/

    float amplitude = 1.;
    float2 frequency = float2(10., 10.);
    float2 x = input.uv.xy;
    float2 velocity = float2(1., 0.);
    float noise = Perlin2D(input.uv * 20) * PI;//adding noise.
```

```
        //PERLIN NOISE: By default values oscilates from -1 to 1. By *PI it
            would inrease to 3.14 to -3.14
            float y = amplitude * (Perlin2D(x * frequency - time*velocity)
                + 1) * 0.5; //float2(x,0) is transforming x to 2D
        PS_OUTPUT o;
        o.color = float4(max(y, 0), 0, 0, 1);//max(y, 0) prevents from
            obtaining negative values
        //o.color += fbm(input.uv.xy*3.0);//calling the octaves' function
        float amp = .50;
        for (int i = 0; i < OCTAVES; i++) {
           o.color += amp * Perlin2D(x * frequency - time*velocity);
           x *= 2.; //Modifies the "granularity"
            amp *= .5;//Modifies the gain
        }
        o.velocity = 0;
        return o;
      }
   """
}

simulate = {
   includes = [ "common" "noise_functions"]
   samplers = {
      input_texture0 = { sampler_states = "clamp_linear" } //LINEAR
          INTERPOLATION!
      input_texture1 = { sampler_states = "clamp_linear" }

      flowmap_twirl = { sampler_states = "wrap_linear" }
      noise = { sampler_states = "wrap_linear" }
      internet_flowmap = { sampler_states = "wrap_linear" }
      terrain_hmap = { sampler_states = "wrap_linear" }
      simple_terrain_hmap = { sampler_states = "wrap_linear" }
      weird_texture = { sampler_states = "wrap_linear" }
      grain_flowmap = { sampler_states = "wrap_linear" }
   }

   code="""
```

```
DECLARE_SAMPLER_2D(input_texture0);
DECLARE_SAMPLER_2D(input_texture1);
DECLARE_SAMPLER_2D(flowmap_twirl);
DECLARE_SAMPLER_2D(noise);
DECLARE_SAMPLER_2D(weird_texture);
DECLARE_SAMPLER_2D(internet_flowmap);
DECLARE_SAMPLER_2D(terrain_hmap);
DECLARE_SAMPLER_2D(simple_terrain_hmap);
DECLARE_SAMPLER_2D(grain_flowmap);


struct VS_INPUT {
   float4 position : POSITION;
   float3 uv : TEXCOORD0;
};

struct PS_INPUT {
   float4 position : SV_POSITION;
   float3 uv : TEXCOORD0;
};

CBUFFER_START(c0)
   float4x4 world_view_proj;
CBUFFER_END

DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
PS_INPUT vs_main(VS_INPUT input) {
   PS_INPUT o;
   o.position = mul(input.position, world_view_proj);
   o.uv = input.uv;
   return o;
}

struct PS_OUTPUT {
   float2 velocity : SV_TARGET0;
   float4 color : SV_TARGET1;
};
```

```
        DEFAULT_ROOT_SIGNATURE_ATTRIBUTE
        PS_OUTPUT ps_main(PS_INPUT input) {

          float2 flowDir = TEX2D(simple_terrain_hmap, input.uv.xy).xy * 2.0 -
              1.0;
          //float2 flowDir = TEX2D(terrain_hmap, input.uv.xy * 10).xy * 2.0 -
              1.0;
          //The *10 is the amount of tiling being done. Tiling means making
              the effect on a certain part or tile of the texture

              float phase0 = frac(time * 0.05 + 0.5);
              float phase1 = frac(time * 0.05 + 1.0);

              float4 tex0 = TEX2D(noise, input.uv.xy + flowDir.xy * phase0);
                  //The added *10 is for the tiling
              float4 tex1 = TEX2D(noise, input.uv.xy + flowDir.xy *
                  phase1);//We use the same input in both!

          //float4 tex0 = TEX2D(noise, input.uv.xy * 10 + flowDir.xy *
              phase0); //The added *10 is for the tiling
              //float4 tex1 = TEX2D(noise, input.uv.xy * 10 + flowDir.xy *
                  phase1);//We use the same input in both!

              float flowLerp = abs((0.5 - phase0) / 0.5);
          PS_OUTPUT o;
          o.color = lerp(tex0, tex1, flowLerp).r; //the .r makes the noise go
              grey scale so it isn't so psicodelic
          o.velocity = 0;
              return o;
        }
      """
    }

}

shaders = {
```

```
   test_0 = {
      editor_advanced_mode = true

      contexts = {...
      }

      compile = {
         default = [
            { defines=[""] }
         ]
      }
   }
   /*************** "REDACTED" **************/

    color_init = {...
   }

    vector_field_init = {...
   }

   simulate = {...
   }
}

static_compile= [
   { shader="test_0" }
   { shader="color_init" }
   { shader="vector_field_init" }
   { shader="simulate" }
]
```

Appendix B

Volumetric Rendering Script for the Real-Time simulation

The script for the real-time simulation of the volumetric fog data with the interactive height map shader is mostly redacted for privacy purposes but the structure is present as well as the more relevant code.

```
cs_fog_material_data = {
includes = [ "..." "..." ]
...

samplers = {
   ...
   //*************** Shader Test addition ***************
   internet_flowmap = { sampler_states = "wrap_linear" }//simple_terrain_hmap
   terrain_hmap = { sampler_states = "wrap_linear" }
   simple_terrain_hmap = { sampler_states = "wrap_linear" }
   noise = { sampler_states = "wrap_linear" }
}

code = """
   RWTexture3D<float4> input_texture0;
   //*************** Shader Test addition ***************
   DECLARE_SAMPLER_2D(internet_flowmap);
   DECLARE_SAMPLER_2D(terrain_hmap);
```

```
    DECLARE_SAMPLER_2D(simple_terrain_hmap);
    DECLARE_SAMPLER_2D(noise);

/********************* "REDACTED" *********************/
    ...
    float3 wp = view_to_world(ss_to_view(ss_pos, 1.0), 1.0);

    //Adding noise function to the volumetric fog
    //float noise = Perlin3D(wp + time*float3(1., 0., 0.)); //-1 to 1. If *PI
        it would inrease to 3.14 to -3.14
    //If you take + 1.0 away the amplitude is decreased
    //-------------------------------------------------------------------
    float2 uv1 = wp.xy/50;
    float2 flowDir1 = (TEX3DLOD(simple_terrain_hmap, uv1, 0).xy - 0.5)*2.0;
    float phase01 = frac(time * 0.1 + 0.5);
    float phase11 = frac(time * 0.1 + 1.0);
    float tex01 = TEX3DLOD(noise, uv1 + flowDir1.xy * phase01, 0).r;
    float tex11 = TEX3DLOD(noise, uv1 + flowDir1.xy * phase11, 0).r;

    float tex21 = TEX2DLOD(noise, float2(0, wp.z/50), 0).r;
    //this texture is for the z axis to have a noise function applied to it
    float flowMap1 = lerp(tex01, tex11, abs((0.5 - phase01) / 0.5)).r;//.r*tex21

    float extinction = max(global_extinction(wp)*flowMap1,0);
    //float extinction = global_extinction(wp);
    float3 scattering = fog_color * extinction;
    float3 emissive = 0;
    ...
    /********************* "REDACTED" *********************/
    #if defined(VOLUMES_ENABLED)
        ...
        if (is_inside) {
        ...
        //*************** Shader Test addition ***************
          float2 uv = op.xy/50;
          float2 flowDir = (TEX2DLOD(simple_terrain_hmap, uv, 0).rg - 0.5)*2.0;
          float phase0 = frac(time * 0.1 + 0.5);
```

```
  float phase1 = frac(time * 0.1 + 1.0);
  float tex0 = TEX3DLOD(simple_terrain_hmap, uv + flowDir.xy * phase0,
     0).r;
                 //The added *10 is for the tiling
  float tex1 = TEX3DLOD(simple_terrain_hmap, uv + flowDir.xy * phase1,
     0).r;
                //We use the same input in both!

   op.z = frac(op.z)*64.0;
   float iz = floor(op.z);
   float fz = frac(op.z);
   float2 a_off = float2(23.0, 29.0)*(iz)/64.0;
   float2 b_off = float2(23.0, 29.0)*(iz+1.0)/64.0;
   float a = TEX2DLOD(noise, op.xy + a_off, -999.0).r;
   float b = TEX2DLOD(noise, op.xy + b_off, -999.0).r;

  float tex2 = lerp(a, b, fz);//TEX3DLOD(simple_terrain_hmap, float2(0,
     op.z/50), 0).r;
  //this texture is for the z axis to have a noise function applied to it

  float flowMap = lerp(tex0, tex1, abs((0.5 - phase0) / 0.5)).r*tex2;
  local_extinction *= flowMap; //float 2 * float 1 = float 2   **/

//*************** Shader Test addition ***************
  float2 uv = op.xy/50;
  float2 flowDir = (TEX2DLOD(simple_terrain_hmap, uv, 0).rg - 0.5)*2.0;
  float phase0 = frac(time * 0.1 + 0.5);
  float phase1 = frac(time * 0.1 + 1.0);
  float tex0 = TEX3DLOD(noise, uv + flowDir.xy * phase0, 0).r;

  float tex1 = TEX3DLOD(noise, uv + flowDir.xy * phase1, 0).r;

  float tex2 = TEX3DLOD(noise, float2(0, op.z/50), 0).r;
  //this texture is for the z axis to have a noise function applied to it

  float flowMap = lerp(tex0, tex1, abs((0.5 - phase0) / 0.5)).r*tex2;
  local_extinction *= flowMap;
```

```
      //Apply perlin noise function to local fog unit
      //float noise = Perlin3D(op.xyz + time*float3(1., 0., 0.));
      //local_extinction *= noise;
    //*************************************************************

    scattering += local_albedo * local_extinction;
    extinction += local_extinction;
    //phase += local_phase * local_extinction;
    total_weight += local_extinction;
    }
  }...
}
"""
}
```