`TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Yury Malyshev 184599IASM

SOFTWARE DEVELOPMENT FOR THREE-DIMENSIONAL CALCULATION OF LAGRANGIAN COHERENT STRUCTURES FROM FLUID FLOWS

Master's thesis

Supervisor: Jeffrey A. Tuhtan
Dr.-Eng.

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Yury Malyshev 184599IASM

TARKVARAARENDUS LAGRANGIANI SIDUSATE STRUKTUURIDE KOLMEMÕÕTMELISEKS ARVUTAMISEKS VEDELIKUVOOGUDE PÕHJAL

Magistritöö

Juhendaja: Jeffrey A. Tuhtan

Dr.-Eng.

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Yury Malyshev

[17.05.2020]

Abstract

Fluid motion consists of macroscopic regions of flow across which mass is not exchanged. The boundaries between these regions form an invisible “fluid skeleton”, and have been found to play important roles in biological flows such as arteries as well as large-scale flows such as ocean currents. In order to detect and track these boundaries, advanced numerical models use Lagrangian Coherent Structures (LCS) to define internal fluid boundaries. To study these structures, it is not only important to be able to calculate them, but also to visualize them. Largely due to the mathematical complexity of how LCS are defined and the corresponding high computational cost, there is a lack of applications for the calculation and visualization of LCS.

The main objective of this thesis is to research and develop an efficient and user-friendly calculation method as well as an easy-to-use application capable of LCS visualization. Aspects such as interpolation, data storage, and mesh creation are discussed in detail. The major findings are related to the optimization of the application and efficient use of hardware resources. The first, heavy parallelization of the code; each independent computation task is parallelized. The second, optimization of computation through the use of voxel-based space-partitioning data structure, which allowed to significantly improve the performance of interpolation. One novel aspect of this work is complex visualization tasks are performed using the Unity3D game engine, and its capabilities and drawbacks in comparison with standard tools are explored.

The final solution is divided into two independent solutions: calculation application and visualization application. The division allowed to make each solution more stable and reliable. In the future the applications can be further optimized; extensive user-testing will provide a better overview on which parts should be improved further.

This thesis is written in English and is 42 pages long, including 7 chapters, 25 figures and 8 tables.

List of abbreviations and terms

CFD	Computational fluid dynamics
LCS	Lagrangian Coherent Structures
FTLE	Finite-Time Lyapunov Exponent
UX	User experience
UI	User interface
PC	Personal computer
NN	Nearest neighbour
IDW	Inverse distance weight
GPU	Graphics processing unit
GUI	Graphical user interface
STL	Stereolithography (file format)
CSV	Comma-separated values (file format)
JSON	JavaScript Object Notation (file format)

Table of contents

List of figures

List of tables

1 Introduction

Fluid mechanics is the study of the thermodynamic state, motion, forces and composition of fluids and their interaction with external and internal bodies. One of the subfields of fluid mechanics is fluid dynamics, which studies the effects of both the motion and forces in fluids. Fluid dynamics is a branch of macroscopic physics which provides methods to study such a broad range of phenomena including currents in oceans and rivers, the spread of oil spills [1][2], weather patterns [3] and blood circulation [4]. From the obtained knowledge, it is possible to make better predictions, improve efficiency of hydropower plants, and detect and treat circulatory system illnesses. The state of the art is to create numerical simulations of these systems using computational fluid dynamics (CFD). Computers are now commonly used to simulate the flow of fluid and its interaction with complex geometries. The main applications of CFD are aerodynamics, aerospace, automotive engineering, heat transfer, geosciences, and fluid flows in biological systems (e.g. blood flow in hearts, lungs, and the brain) [5].

One of the complex tasks in post-processing CFD models is the calculation of Lagrangian Coherent Structures (LCS), which are plots which show discrete, internal regions of the flow field which exist due to the stretching and compression of the fluid as is travels through a given boundary geometry. Although undetectable to the human eye, LCS are physical structures which are formed as part of a dynamical system; and are readily observable in CFD models of fluid flows [6]. LCS can be used for better understanding of motion patterns, which results in better predictions and the possibility to optimize the system. However, because LCS are created from complex CFD models, there currently lack specific tools for their calculation and visualization in 2D and 3D space.

A leading field where high-resolution visualization in 2D and 3D space is needed in real-time is the game industry. According to [7], the game industry has a great influence on modern hardware and software development. Computer games try to achieve very high simulation and graphics fidelity while still remaining highly accessible to the public. In order to achieve this goal, it is necessary to have a cost-efficient approach to the development. The common method is to reuse the same base solution for a family of games using a so-called “game engine”. This advancement of game engines provides low-cost and time-efficient ways to develop high-level software systems from scratch, which includes the development of tools for scientific research [8].

Modern game engines, such as Unreal Engine [9] or Unity3D [10], are carefully constructed frameworks with a set of libraries. This allows to simplify the process of virtual world generation by utilizing world-building tools. It is also possible to create simple real-life physics simulations using in-built libraries. By utilizing graphics API, great visual fidelity can be enabled. Today, these tools have reached the quality of being used for the development of scientific research tools, while significantly reducing the cost [7].

The main objective of this work is to create, test and implement a user-friendly, easily accessible application for LCS calculation and visualization based on a game engine. The application should be able to perform the complex and computationally expensive calculations and display the results in a clear and structured form. One of the more challenging aspects of this work is to create an application which can quickly calculate and visualize three-dimensional model result to make LCS available to a wider range of fluid mechanics researchers.

1.1 Background

Fluid motion at the macroscale is not random, and fluid flows often contain include fine-scale structure such as shear zones, eddies and turbulent filaments. According to [11] some marine predators can use changes in local flow structure to detect and track their prey. Inside of a fluid, “ridges” are formed as the zero mass flux boundaries between these internal flow structures. Across these boundaries material transport does not occur, and this in turn influences diffusion, dispersion and advection processes. Internal flow structures can be detected and tracked as LCS, where they are defined as ridges of the Finite-Time Lyapunov Exponent (FTLE) fields [6]. Today, LCS are used in several different applications and spatial and temporal scales. At a very large scale, such as ocean flows [12], the structures can be used to determine where and how oil spills will propagate, which can help to prevent these disasters. The applications also cover very small scales, such as blood flows in the human body [4]. One of the interesting examples of large-scale structures is Great Red Spot on Jupiter (figure 1), where the structure is so large, it is visible with the naked eye (when using a telescope):

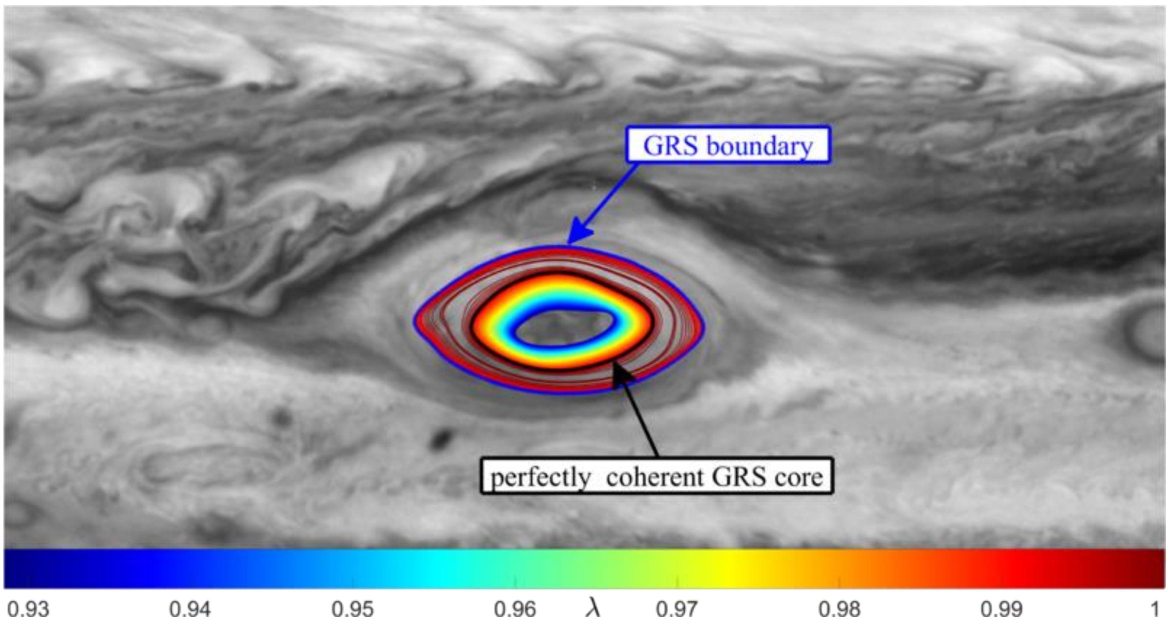


Figure 1. Great Red Spot on Jupiter [13]

According to [14], in order to analyse flows, traditionally other two observational perspectives are used, Eulerian and Lagrangian. Eulerian analysis is based on a fixed observer, what kind of change has happened by comparing the amount and properties of the matter which has entered and then left specified volume. The second, Lagrangian approach follows a single particle along its path. It helps to create a mathematical framework from which it is possible to see an interaction between particles. A hybrid perspective is created using LCS, which combines the two perspectives. This is achieved by observing at least one set of Lagrangian particles as they are advected through the flow field and the change in their trajectories is calculated as the FTLE and is then plotted based on their initial positions using an Eulerian reference frame.

Calculations of LCS have certain challenges. Firstly, it is not enough to calculate the field itself. The result is usually so complex that it is impossible to comprehend and understand what this result means by looking at the numbers alone. That is why it is needed to have an additional tool which would create a visual representation of the field. Flows are often 3D and the resulting structures are 3D as well. Thus, a tool which can easily work with 3D is needed for more complex and real-world problems. Secondly, the calculations require heavy processing. Even though it is possible to utilize super-computers for those calculations, the software plays a great role in how efficiently the hardware is used. Once the software is efficient and utilizes most of the hardware resources, it is possible to solve even complex problems on hardware easily accessible to an average user.

There are many challenges in visualization of 3D data. The first, it is needed to create an environment which would allow to easily change the rotation of the object and its position. The second challenge is to make the application easily accessible by introducing intuitive user interface (UI). The third, the application should be able to perform all the calculations and object manipulations in real-time. By utilizing game engines, it is possible to avoid some of the challenges, since they have already been solved by the game engine developers. There are, however, other challenges connected to the application itself which are parts of the user experience (UX). The product should be not only accessible and useful but also easy to find and be tested to establish credibility, as shown in Figure 2.

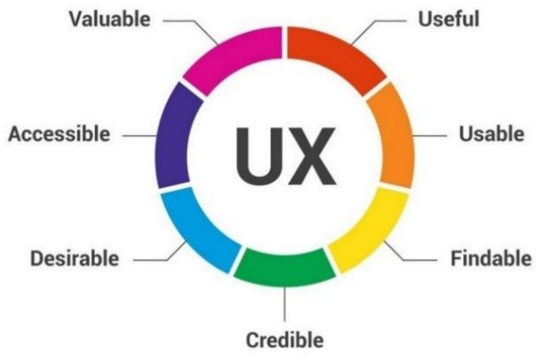


Figure 2. Elements of UX [15]

Value and usefulness are generated from the problem itself because there are not enough applications which are able to perform the task. By making open-source code, the software will be more findable and credible, because the users will be able to easily evaluate it. This paper will reinforce the creditability and desirability of the application.

1.2 State of the Art

There are multiple applications which can perform the calculations and visualization, such as, ManGen [16], LCS MATLAB Kit [17], cuda_ftle [18], Newman [19], and the MATLAB repository jtuhtan/lcs [20]. However, each of these has been hand-crafted for a specific flow and purpose, and there are certain limitations and inconveniences related to those projects. First, it is difficult to obtain the out-of-the-box version of the program or code. In several cases, some additional software, such as compiler, is required to run the application, and in other cases, it is difficult to obtain the executable file itself. Secondly, a lack of detailed documentation and working examples makes it difficult to understand the exact purpose of the application. Thirdly, some of the applications are written in MATLAB, which is generally much slower than other common programming languages. The slower the application, less user-friendly it is and harder it is to use when the input dataset is bigger. Additionally, a set of the applications is capable of processing 2D data only, which is a major drawback for real-life 3D flow scenarios.

This application aims to provide a clear and simple solution for the calculation and visualization of FTLE fields. User-friendly interface will help to work with the solution, thus reducing the time which is needed to get a solution for a problem. Documentation and independent executable files will make it easier to evaluate and understand the program.

1.3 Task definition

It is required to build an application, which can calculate FTLE field and visualize it in the form of a colour gradient in 3D space. It should be possible to run the application on a PC with average characteristics without sacrificing speed and quality. The task can be divided into six parts as described in table 1.

Table 1. Task specification

Task	**Definition**	**Estimated time**
Preparation	Selection of a programming language and visualization environment. The selection process should consider previous knowledge of the developer in order to reduce the time needed to start development. Visualization environment should support 3D. Language should support parallelization.	3 weeks
Algorithm implementation	Implementation of the existing algorithms in the selected programming language. Selection of preferred algorithm if applicable.	6 weeks

	Testing of the algorithms.	
Optimization of the code	The application should utilize all available resources in order to improve the time needed for calculations. Heavy parallelization of the application. The application code should be easily modifiable.	2 weeks
Usage of the algorithm	User should be able to easily access the algorithm, input the data into the application and get the output. Application should be able to handle different sets of data through either user input or automatically based on the data set.	4 weeks
Visualization of the results	Visualization of the resulting output data in the selected environment. The data should be colour-coded depending on the values. The user should be able to select which parts of the data are visible.	5 weeks
Optimization of the visualization	Time delay from switching between different views should be negligible from a user perspective. The movement of the camera should have a reasonable speed in order to improve user satisfaction.	2 weeks

The core of the data processing flow can be described using the following diagram (figure 3). First, a new position of a point is calculated (section 3.3). After that, the value at the new point is found using interpolation (section 3.2). Once the calculations are complete, the output dataset is put into an ordered grid and saved into a file. The results can be later visualized (section 4.4).

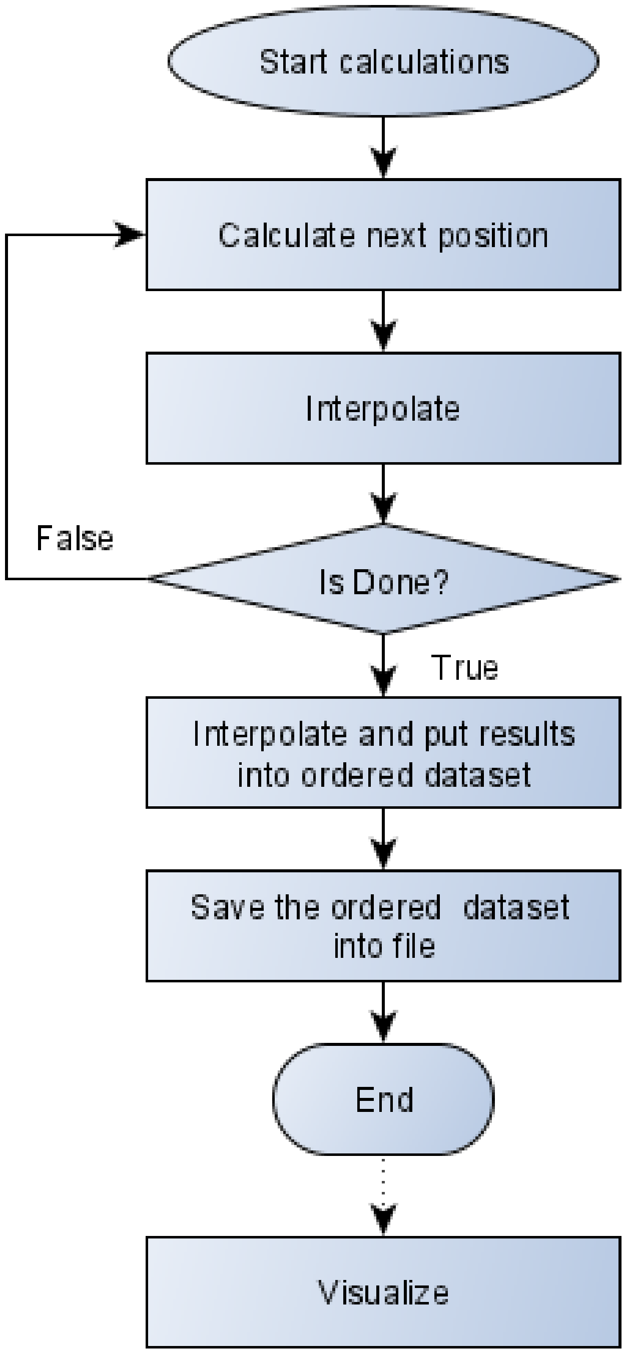

Start calculations
Calculate next position
Interpolate
False
Is Done?
True
Interpolate and put results into ordered dataset
Save the ordered dataset into file
End
Visualize

Figure 3. General workflow of the calculations

2 Project setup

In order to proceed with the development preparatory work needs to be done. First, it was needed to select tools which will be used for development. Later, it was needed to select the exact algorithms which will be used in the development, section 3.

2.1 Tool selection

Both the environment and the programming language have a strong impact on the development of the solution. The main restriction in this thesis was the author's existing knowledge of different languages and environments; therefore, language selection was limited to Java, C++, C# (table 2); environment selection was limited to Unity3D, Unreal Engine 4, libGDX, and jMonkeyEngine (table 3). Each item in the selection has its advantages and disadvantages.

Table 2. Advantages and disadvantages of different proposed languages.

Languages		
	Advantages	**Disadvantages**
Java	Easy to use. Can be easily moved to another platform without any modification. Excessive previous experience. In the top 2 most popular programming languages [21].	Relatively slow. Might not be able to access GPU acceleration.
C++	Reliable. Extremely powerful while remaining high-level object-oriented language. Can access low-level OS functionality.	Hard to use. Low previous experience.
C#	Easy to use. Similar to Java in syntaxis and structure. Possibility to access some OS functionality.	Partial garbage collection. Low previous experience.

Table 3. Advantages and disadvantages of different proposed game engines

Environments		
	Advantages	**Disadvantages**
Unity3D (C#)	Easy to use game engine designed for the creation of 3D games. Free. Extensive knowledge base and documentation. Active forums help to find relevant information.	Low previous experience. Forced structure.
Unreal Engine (C++)	Easy to use game engine designed for the creation of 3D games. Free. Possible to develop an application using graphical language.	No previous experience. Steep learning curve.
libGDX (Java)	Moderate difficulty game engine for game development. Free. Moderate previous experience.	Primarily designed for 2D environments. Lack of tutorials and information. Only basic documentation is available Steep learning curve.
jMonkeyEngine (Java)	Moderate difficulty game engine for the development of 3D games. Free.	Low previous experience. Only basic tutorials are available. Steep learning curve.

Conclusion. Even though previous knowledge of C# is extremely low, it is a combination of Java and C++; therefore, it should be quite easy to write and understand it. Unity3D is the only engine which fully supports 3D, while being known to some extent to the developer. Extensive knowledge base and arrive community ease process of development and provide the necessary help. Therefore, the final decision is to use Unity3D and C#.

3 Algorithms

Three different basic algorithms were used in the visualization tool. The most important is the algorithm used to calculate FTLE value. However, it is using two other algorithms in the background, interpolation and ordinary differential equation solver.

3.1 Double gyre

An analytical model used for testing LCS is the double gyre, which is described in [22]. The 2D flow field is based on the following stream function (1):

$$\psi(x, y) = \sin(\pi x) \sin(\pi y) \tag{1}$$

which is taken over the domain $D = [0, 2]$ x $[0, 1]$. This results in the velocity field shown in Figure 4.

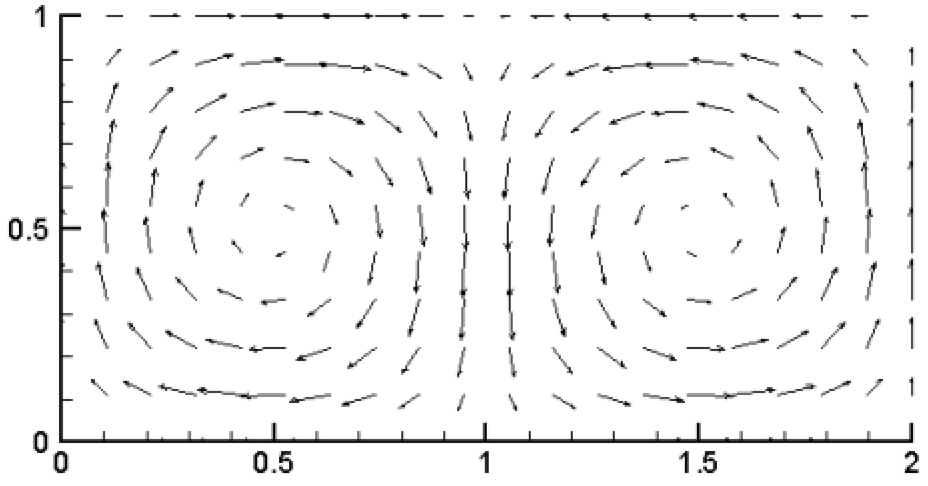


Figure 4. Velocity field of double gyre [22]

The resulting FTLE field for the double gyre is shown in Figure 5. This visual representation is then used as a standard for testing the developed application. Red regions indicate material lines, or boundaries across which mass is not exchanged. Blue areas are those in which particles remain “trapped” in trajectories which do not allow neighbouring particles to move away from each other over time and space.

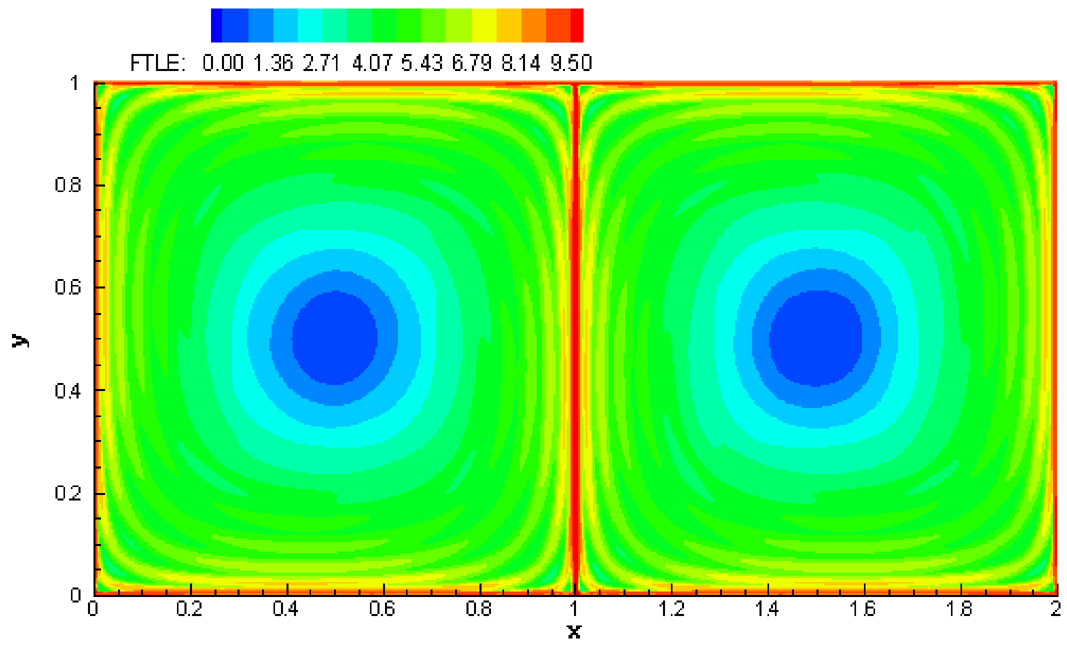


Figure 5. FTLE field of double gyre [22]

3.2 Interpolation

Interpolation is determination or estimation of a value from a certain known values [23]. It is commonly used for graphics scaling, in statistics and time-series forecasting. Correspondingly, an interpolation algorithm is the first step to implement in this work and to test since it is the keystone to the rest of the program. Interpolation algorithms can be classified into two main categories: regular grid [24] and irregular grid, also known as scatter data. Regular grid algorithms, such as bilinear interpolation, work only with structured data which have predetermined spacing. Irregular grid algorithms are more versatile and are able to work with scatter data. Regular grid algorithms are usually faster than irregular grid algorithms due to their deterministic nature. In this project, irregular grid algorithms are applicable because the data is not necessarily pre-arranged. The irregular grid algorithms which will be discussed further are inverse distance weighting and nearest neighbour.

Performance of interpolation algorithms can be optimized using different search optimization techniques. One of the methods is space partitioning; by dividing the complete dataset into smaller independent sections, parts of the initial dataset can be discarded immediately. Two of the common methods are k-d tree [25] and R-tree [26]. The optimization method used in this project is partially derived from R-tree, where the space is divided into sections and the search is performed within the section. This results in sections of equal volume, and creates a grid of voxels. Before the interpolation is performed, the most relevant voxels corresponding to known data points are found. The interpolation dataset then contains information only from these found voxels.

3.2.1 Nearest neighbour

Euclidean distance nearest neighbour is one of the simplest and fastest algorithms used to interpolate a value because it requires a low number of calculations. The value of an interpolated point is calculated by simply taking it from the nearest Euclidean neighbour of the selected point. Therefore, only the Euclidean distance (L2 norm) needs to be calculated between points. The pseudocode for the algorithm is:

```
Nearest_Neighbour (Coordinate)
   Minimum_Distance = Infinity
   Value = NULL
      FOR each Point in Dataset
         D = Euclidean_distance_between (Coordinate, Point.Coordinate)
         IF D < Minimum_Distance THEN
            Value = Point.Value
            Minimum_Distance = D
         ENDIF
   return Value
```

It should be noted that the accuracy of results using this method can be reduced dramatically, as they depend wholly on the input data. However, if the output data does not have to have high precision, then the increase in speed can be useful.

The visual representation of the results using nearest neighbour interpolation is shown in Figure 6. As can be seen, the general picture is recognizable, but the precision is too low for useful analysis.

3.2.2 Inverse distance weighting

The inverse distance weighting algorithm used in this project is fundamentally based on Shepard’s method [27], and modifications were introduced during the development in order to improve the program structure for increased computational efficiency.

Shepard’s method is expressed by the following formula (2):

$$u(x) = \begin{cases} \frac{\sum_{i=1}^{N} w_i(x) u_i}{\sum_{i=1}^{N} w_i(x)} \; if\; d(x, x_i) \neq 0\; for\; all\; i \\ u_i\; if\; d(x, x_i) = 0\; for\; some\; i \end{cases} \tag{2}$$

Where $w_i(x) = \frac{1}{d(x, x_i)^p}$

Where *u* is value; *w* is weight; *d* is distance; *x* is coordinates of some point.

The modification applied in the project is in the form of a distance constraint. This is done by setting a threshold for all distances less than *r* (3):

$$d(x, x_i) < r \tag{3}$$

A visual representation of the modified IDW method is shown in Figure 6. The precision is much higher than of the nearest neighbour, but it is not perfect. From the picture, the usage of the limited boundary can be clearly seen.

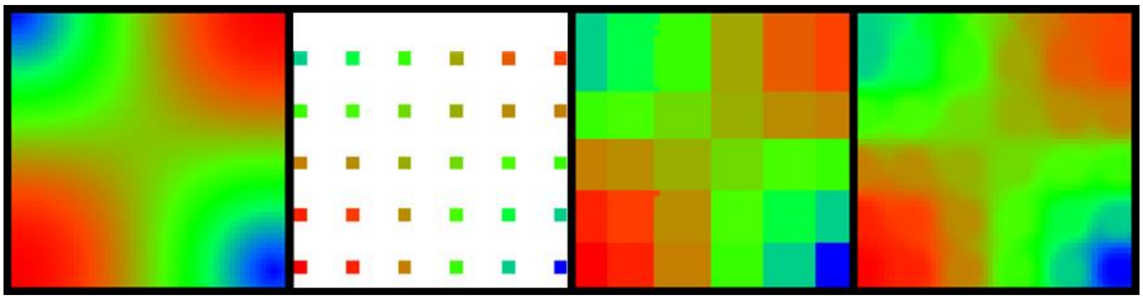


Figure 6. Visual representation of interpolation. Picture: left to right are: expected field, input dataset, nearest neighbour interpolation, inverse distance weight interpolation. Input dataset spacing is 0.1 unit; interpolation spacing is 0.01 unit; voxel size is 0.25 units.

3.2.3 Algorithm comparison

The evaluation of interpolation algorithms was done using the following procedure: a randomly chosen, known point is chosen and removed from input data set; the point is interpolated 1,000 times and total time is measured; the resulting point is compared with the original. Input data set is an equally spaced grid 1 by 2 units. The spatial resolution of the grid is 0.01 units (Table 4, and Table 5 provide an overview of the influence of the chosen diameter on the voxel count and point density per voxel).

Table 4. Interpolation algorithms evaluation. Dataset has 200 points arranged into 2x1 units rectangle.

Measurement	**Diameter [unit]**	**NN**	**IWD**	**NN with voxels**	**IWD with voxels**

Error [10^{-3}*unit]	0.1	0.66	0.05	0.66	0.05
Time [ms]		1.99	2.05	1.51	1.51
Error [10^{-3}*unit]	0.25	0.66	0.16	0.66	0.16
Time [ms]		1.97	1.85	0.16	0.20
Error [10^{-3}*unit]	0.5	0.66	0.31	0.66	0.31
Time [ms]		1.97	1.88	0.31	0.49
Error [10^{-3}*unit]	1	0.64	0.64	0.66	0.64
Time [ms]		1.86	2.05	1.14	1.72
Error [10^{-3}*unit]	5	0.66	0.85	0.66	0.85
Time [ms]		1.60	5.77	1.22	5.63

Table 5. Number of voxels and average number of points depending on the diameter of a voxel for the dataset with 20 000 data points arranged into 2x1 units rectangle

Diameter [unit]	**Number of voxels**	**Average number of points per voxel**
0.1	1557	13
0.25	45	444
0.5	15	1 333
1	6	3 333
5	1	20 000

As it can be seen (Figure 7), nearest neighbour interpolation has a constant error, due to the nature of the algorithm. This is because the inverse weighted distance error is changing depending on the diameter of the search, where larger diameters increase the error. This outcome is logical, because even though points which are distant from the interpolated point and have very low weight are still accounted for and create a persistent contribution to the error.

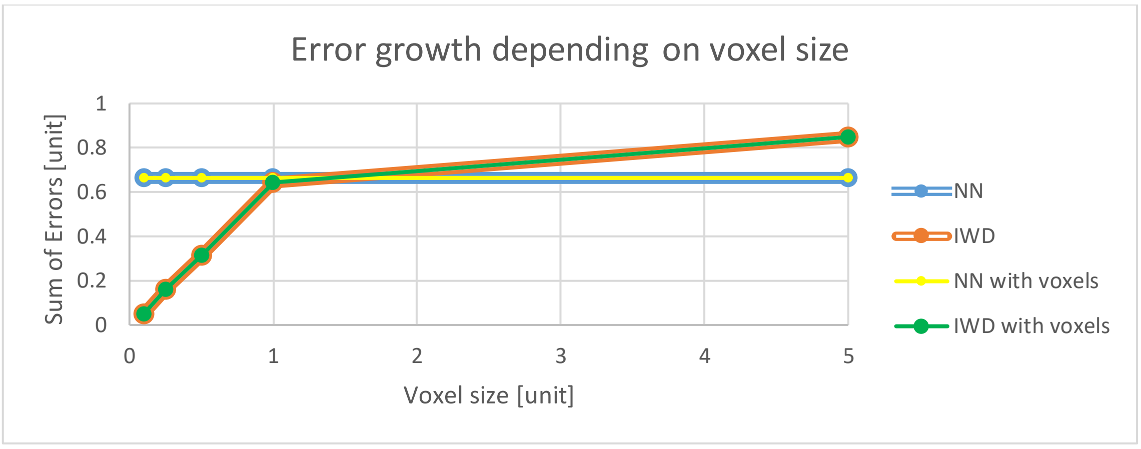


Figure 7. Error growth depending on voxel size.

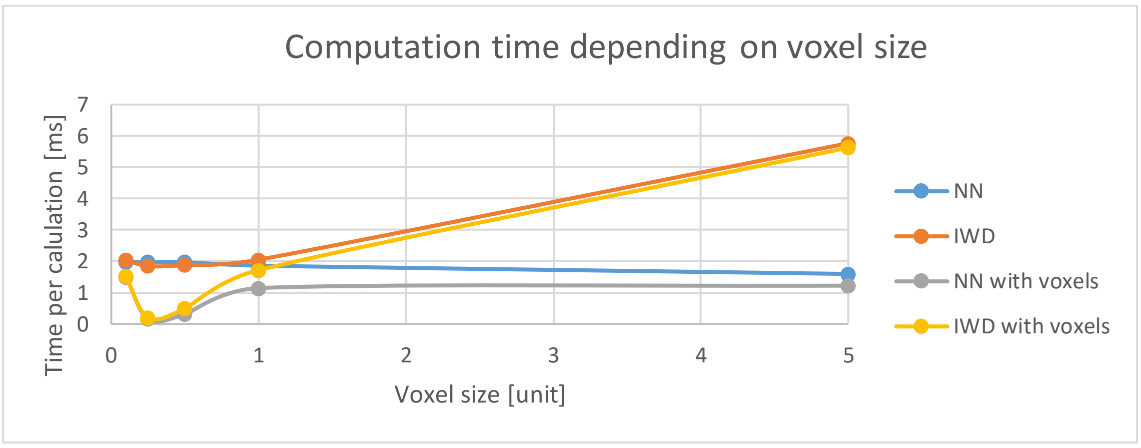


Figure 8. Dependence of computation time on voxel size.

Computation time (Figure 8) is highly dependent on the voxel size and consequently the number of voxels. A large number of small volume voxels negatively affects the performance, due to the increase in computational effort. However, large volume voxels affect IWD interpolation due to the number of additional calculations. It should be noted that loading time is not included in these values, since it can be considered negligible if compared to accumulated saved time. However, smaller voxel size does affect loading time and if viewed independently from calculations the difference is significant (table 6). Data was loaded once per dataset.

Table 6. Loading time compared to the number of voxels. Dataset includes 20 000 points arranged into 2x1 units rectangle. Dataset is loaded once.

Voxel size	Number of voxels	Loading time [ms]
0.1	1557	1325
0.25	45	119
5	1	77

In conclusion, IWD interpolation using voxels as an input performs the best in most cases. Additionally, NN is used for interpolation of data which is outside of predefined diameter but is required, and IWD with the whole data set is used when the input data is not structured (e.g. scatter data vs. grid data).

3.3 Solving ordinary differential equations

In order to calculate FTLE value, it is required to know how a particle propagates through the velocity field. To solve this problem, an ordinary differential equation has to be solved first, which is done using an iterative approach. The simplest method is the Euler one-step method, which assumes that the movement is linear in time and does not consider previous behaviour (e.g. no inertial or additional external forces). The result of the procedure is a set of straight tangent lines which follow the pathlines of the CFD model velocity field. Due to the absence of memory, the error grows continuously, and growth depends on the step size. Equation which describes the Euler method is the following (4):

$$c_{next} = c_{current} + v_{c_{current}} * \Delta t, \text{where } c \text{ is a coordinate } (\mathrm{x}, \mathrm{y}, \text{or z}) \quad (4)$$

3.4 FTLE field creation

The FTLE is calculated as a change in the size of the right Cauchy-Green deformation tensor. This change can be seen in Figure 9. Even though the initial particles, green and red, are the same, their dimensions change as they are advected through the velocity field. The higher the difference, the higher the FTLE value.

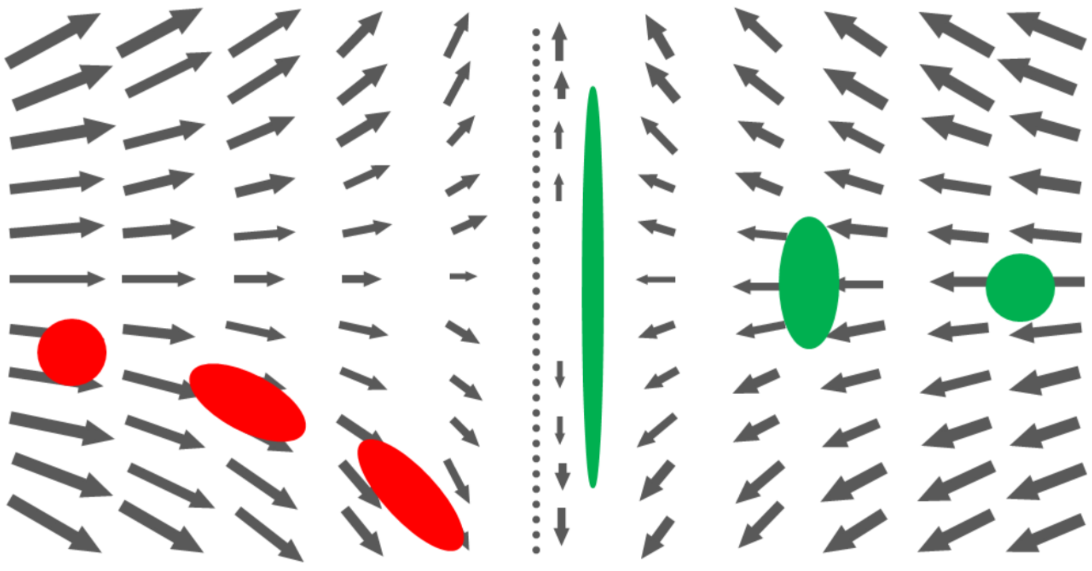


Figure 9. Visualization of a particle advected through velocity field. Left: the flow field stretches the initial red blob because of local differences in the velocity field. Right: the green blob expands as the velocity field diverges into the vertical direction [14]

In order to make a particle which has a volume a set of dimensionless points was taken instead (Figure 10). Then the difference in dimensions is the difference in coordinates of the points. This method is known as the right Cauchy–Green deformation tensor.

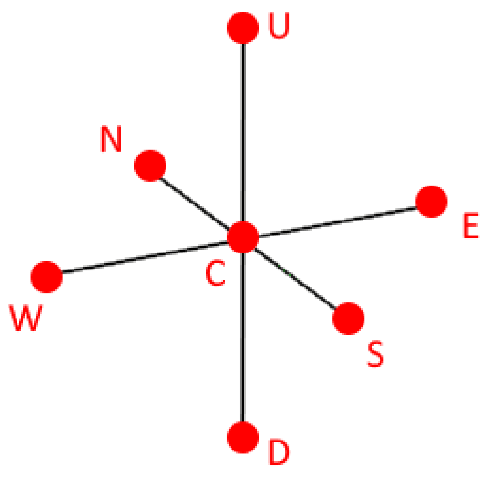


Figure 10. Pseudo particle structure.

To calculate the right Cauchy–Green deformation tensor it needed to know the Jacobian of the flow map, which is calculated using the following formula (5):

$$J(\Phi_0^T) = \frac{\partial \Phi_0^T}{\partial x} = \\ = \begin{bmatrix} \frac{x_{(i+1)J}(T) - x_{(i-1)J}(T)}{x_{(i+1)J}(0) - x_{(i-1)J}(0)} & \frac{x_{(j+1)J}(T) - x_{(j-1)J}(T)}{x_{(j+1)J}(0) - x_{(j-1)J}(0)} & \frac{x_{(k+1)J}(T) - x_{(k-1)J}(T)}{x_{(k+1)J}(0) - x_{(k-1)J}(0)} \\ \frac{y_{(i+1)J}(T) - y_{(i-1)J}(T)}{y_{(i+1)J}(0) - y_{(i-1)J}(0)} & \frac{y_{(j+1)J}(T) - y_{(j-1)J}(T)}{y_{(j+1)J}(0) - y_{(j-1)J}(0)} & \frac{y_{(k+1)J}(T) - y_{(k-1)J}(T)}{y_{(k+1)J}(0) - y_{(k-1)J}(0)} \\ \frac{z_{(i+1)J}(T) - z_{(i-1)J}(T)}{z_{(i+1)J}(0) - z_{(i-1)J}(0)} & \frac{z_{(j+1)J}(T) - z_{(j-1)J}(T)}{z_{(j+1)J}(0) - z_{(j-1)J}(0)} & \frac{z_{(k+1)J}(T) - z_{(k-1)J}(T)}{z_{(k+1)J}(0) - z_{(k-1)J}(0)} \end{bmatrix} \quad (5)$$

Where the index corresponds to the spatial Cartesian dimension (x,y,z).

Then the tensor is calculated as (6):

$$\Delta = J(\Phi_0^T)^{-1} \cdot J(\Phi_0^T) \quad (6)$$

And the FTLE at T is the natural logarithm of maximum eigenvalue of the tensor (7):

$$FTLE(T) = \ln\left(\max\left(\lambda(\Delta)\right)\right) \quad (7)$$

To find the structures it is required to find the maximum value within the pathline. Therefore, each element value in a pathline is assigned to the maximum value.

As a base for this application, an existing algorithm written in MATLAB was taken and translated into C# and adjusted to the application structure [14][20].

4 Development

Development of any product can be done using different models, such as waterfall, incremental, V, spiral, and others [28]. Each model has its advantages and disadvantages; for example, simple waterfall model works well with straightforward development systems, but can lead to undesired results if the expectations are not clear at the beginning of the development. In the development of the visualization software, the spiral development model was used. It allowed to make small incremental steps, assess the risks and shortcomings of the project. Spiral model usually follows the following structure: planning, development, and assessment of the outcomes [28].

Development of the visualization tool can be divided into four major parts. Selection of the tools: environment and language; implementation of algorithms in code; development of a prototype, proof of concept; and finally, development of the complete application. However, the total number of steps was undefined at the beginning of the project, due to many unknowns.

4.1 Data structure

Structure of the data plays an important role in any project and can become especially important when the size of the data set grows to a larger scale. Input data in the visualization tool is usually not ordered and can manifest itself in non-regular shapes when plotted. This makes it difficult to store it in containers such as predefined matrix, due to the possible waste of space, or a list, because search could become significantly more time-consuming. A possible solution is to use a combination of the two, store data points in a list, while the lists have defined boundaries – voxels. In the simplest terms, a voxel is an element compromising a 3D entity and it is usually used to describe a colour of a point of a 3D object [29]. Which means that voxel has a value or set of values, position, and a unit volume. In this work, values are points with coordinate and velocity, volume is defined by the user.

Graphical representation of the most used classes is shown in Figure 11. A single input data point is stored in the application as a “Point”, and contains its position, velocity, and other input variables. An extension of a point is called “SeedPoint” and additionally stores calculated values; it is used as an output data point.

A voxel stores unordered list of data points as well as positions of its corners. This allows to easily search for required voxel and add new voxels which would follow the grid.

Results of the solution of a differential equation is stored in a “Pathline” container. It allows to easily differentiate between multiple passes, and it simplifies the process of parallel computing.

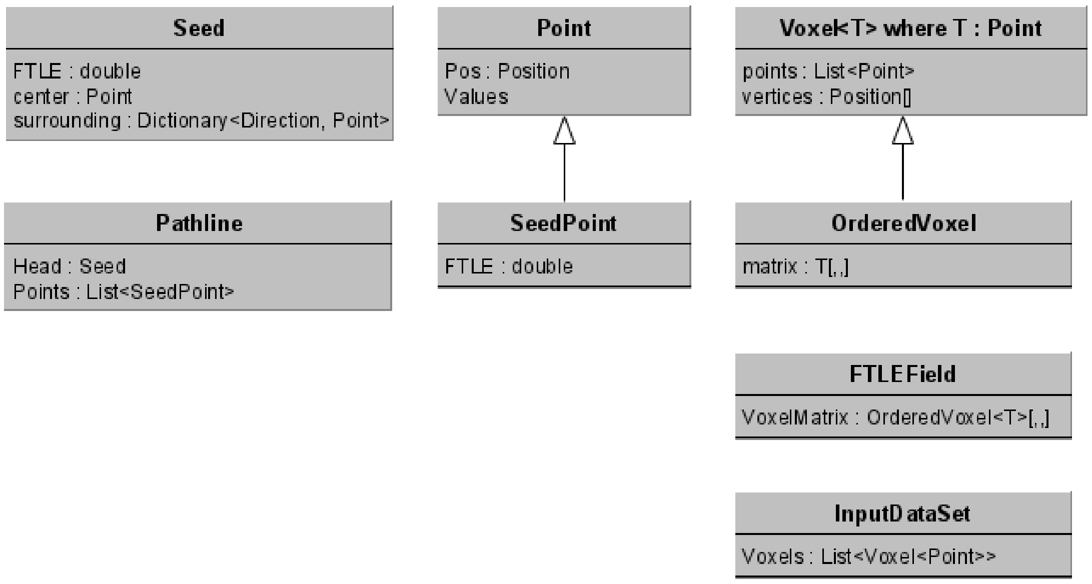


Figure 11. Class structure of important elements

4.2 Development of proof of concept

The proof of concept during the development stage can be divided into three parts: development of data loading procedure, implementation of necessary algorithms, and simple data visualization.

4.2.1 Loading of input data

Test data was stored in a text file and contained the position of a point and velocity at the point. Since the order and structure of the file are not defined, it is impossible to make it fully automatic. Depending on the end application, it may be needed to extend classes, modify the interpolation function, and change the parser accordingly.

One of the initial major problems with loading of data is speed. The reading of the file is performed relatively quickly; however, the process of object creation takes significant time, ranging from a couple of seconds to a minute, which could be mistaken for an error by an unaware user.

4.2.2 Calculations

In the first iteration of the development, all calculations were done in a single thread. By taking the estimated result from table 4 and the fact that several thousand calculations are likely needed, the resulting time can easily turn into hours of calculations. By making the calculations in parallel it is possible to dramatically reduce the computational time. However, after performing several repeating tests, it appeared that the game engine sometimes is unable to handle a massive amount of threads. On one side, it is connected to the hardware and number of real threads is highly limited. However, the game engine was specialised for creating graphics and simple physics calculations-Data visualization

Unity3D provides the Prefab system, which allows to create and store a GameObject, which can be later easily reused. It was the simplest method to visualize data. A point would be represented using a cube, which is coloured according to its value. This creates a large number of scattered points making it simply a 3D-scatter graph.

There are three major drawbacks with this method. First, the more GameObject instances there are in a given scene, the lower the performance is. It can be partially compensated by making objects static, or non-moving. Second, even if the points have equal spacing, it is still needed to calculate and change the scale of all the cubes in order to make them most visible. However, mostly input data is not arranged, making it hard to decide on the scale. Last, the speed of the instancing of objects is relatively slow. The problem was partially solved by utilizing Unity’s GPU Instancing with addition to the instancing of a smaller number of objects per frame. Additionally, there are several minor possible issues; for example, it might be difficult to search for a certain cube in order to change its colour or modify it in any other way.

4.2.3 Summary

To make the application more user-friendly it was needed to put all heavy processing and generally time-consuming functionality into parallel threads, decoupling GUI and processing. Usage of progress bars, loading messages can help to distinguish regular behaviour of the program from an erroneous state.

Since Unity3D has a number of its processes working in the background, and it is mainly a visualization tool, it is better to decouple calculations and visualization completely. By creating a lightweight 2D applications used only for calculations it is possible to improve performance and ease the process of debugging.

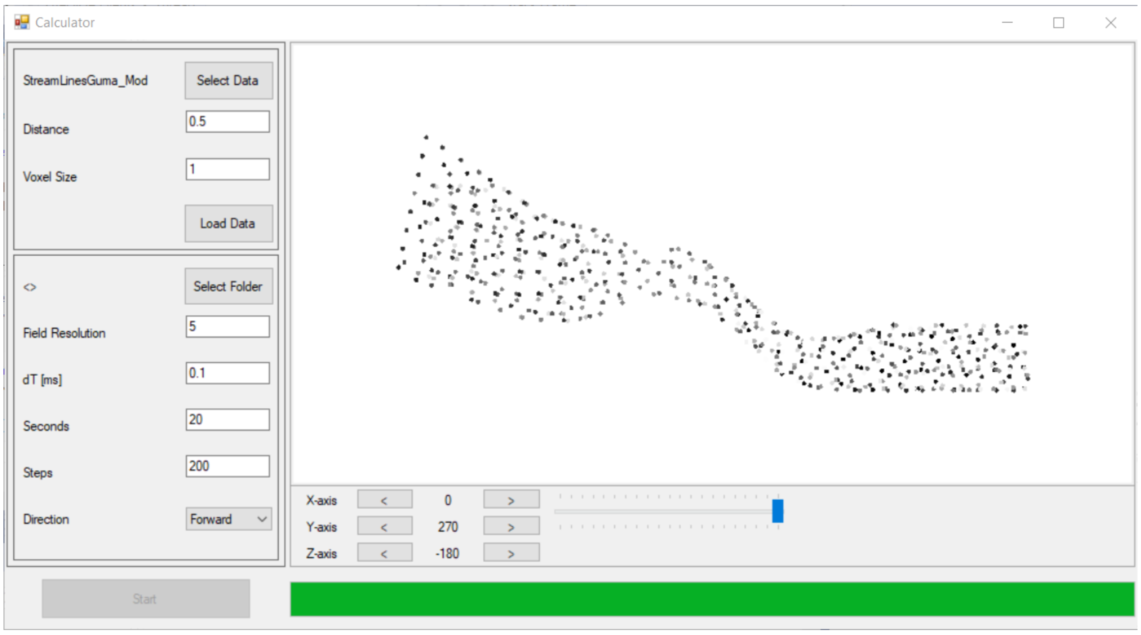
Instead of creating a 3D scatter plot, it is possible to create one single object by utilizing mesh functionality of Unity. It would allow the creation of custom shapes and could be useful for displaying layers of data.

4.3 Calculations application

The calculations application has to be lightweight and easy to use, which includes the development of GUI. Microsoft provides a free graphical framework called Windows Forms. It allows to easily build a simple application graphically using Visual Studio. The main reasons for its selection in this thesis were the ease of use; instant availability, since it is part of Visual Studio; and unification in the programming language, C#. There are no matrix manipulation tools provided in C# natively. Therefore, a third-party library – Accord [30], was used.

4.3.1 GUI

The development of the GUI can be divided into several parts. First, assembly of the interface itself, deciding how it looks and what functionality it should perform. Second, the addition of action listeners to the buttons, so the application can perform a necessary function. Last, different fool-proof functionality should be implemented in order to simplify the flow of usage and prevent errors. This includes disabling certain buttons until they can be used, creating filters for text fields, and adding help messages. Figure 12 shows the final variant of the user interface. It has all the functionality, but some features might be reworked in the future to improve UX.


Calculator

StreamLinesGuma_Mod — Select Data

Distance: 0.5

Voxel Size: 1

Load Data

<> — Select Folder

Field Resolution: 5

dT [ms]: 0.1

Seconds: 20

Steps: 200

Direction: Forward

X-axis < 0 >

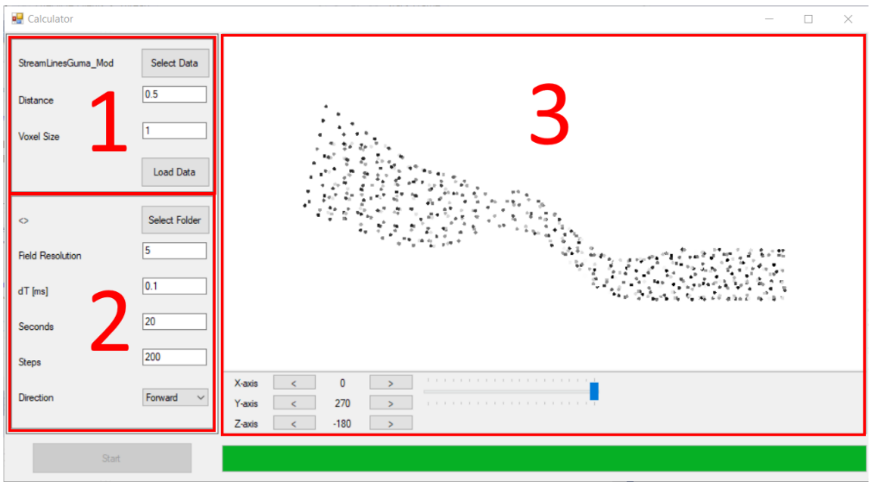
Y-axis < 270 >

Z-axis < -180 >

Start

Figure 12. Semi-final GUI

The GUI is divided into three main parts. Selection and configuration of input data, configuration of calculations, and a graphical representation of the input data (see Figure 13). Description of each element is provided in table 7.



Calculator

StreamLinesGuma_Mod — Select Data

Distance: 0.5

Voxel Size: 1

Load Data

<> — Select Folder

Field Resolution: 5

dT [ms]: 0.1

Seconds: 20

Steps: 200

Direction: Forward

X-axis < 0 >

Y-axis < 270 >

Z-axis < -180 >

Start

Figure 13. GUI. Parts.

Table 7. GUI elements description.

Element	**Description and function**
Section 1	
Select Data	Opens a file selection dialog. The selected file is expected to be in a predefined format and should contain all input data. Disables Section 2 and enables the rest of Section 1.
Distance	Average estimated distance between points. Used later to reduce the amount of painted points on the canvas.
Voxel Size	Desired size of a voxel. Measured in the same unit as input coordinates.
Load Data	Loads data into memory from the file. Loaded data is then painted on Picture Box.
Section 2	
Select Folder	Opens a folder selection dialog. The selected folder will be used to save the output files.
Field Resolution	Number of points per dimension of a voxel in the resulting field. Resolution of N will result in a voxel filled with N^3 points.
dT	Size of a time step. Measured in 10^{-3} of the original unit. Usually it is in milliseconds.
Seconds	Number of time unit steps. Since the time usually is in seconds, it would represent the number of seconds of the simulation.
Steps	Number of dT steps.
Direction	Direction of advection. Can be forward, backward or both. If both are selected, the number of steps is divided equally into the forward and the backward direction.
Section 3	
Picture Box	Used to draw an image of input data. Additionally, it is used as an input method to select the initial set of points. Point opacity depends on the distance from the user, further points appear whiter. The picture is 2D but creates a weak illusion of 3D.
Axis	Axis of rotation of the input data set. Allows to rotate the picture by 90*n degrees.

Element	**Description and function**
Slider	Slider is used to hide all points above a certain level. It is used to access lower layers when selecting an input point.

4.3.2 Input data graphical manipulation

The simplest method to visualize input data in Windows Forms is to use 2D graphics. Therefore, a 3D object is simply a 2D projection on a canvas, where x and y coordinates are used directly, and the z coordinate is represented using a colour. It provides a basic understanding of geometry for a user but to make it complete the object must be rotatable. Rotation can be performed manually using rotation matrices [31][32] (8):

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \qquad (8)$$

$$R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & \sin(\gamma) & 0 \\ -\sin(\gamma) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

And the complete rotation is a multiplication of three matrices (9):

$$R = R_x(\alpha) * R_y(\beta) * R_z(\gamma) \qquad (9)$$

New position of a point is a multiplication of its original position and rotation matrix (10):

$$P_{new} = P_{orig} * R \qquad (10)$$

To make the right projection it is needed to know the size of the screen, the allowed portion of the screen, and extremes of the boundary. Additionally, it is required to display the complete picture. Therefore, a scaling multiplier can be calculated the following formula (11):

$$Multiplier = Min\left(\frac{w * p}{\Delta x}, \frac{\mathrm{h} * \mathrm{p}}{\Delta y}\right), \tag{11}$$

where w is the width of the screen, h is the height of the screen, p is the ratio of allowed area to complete screen, Δx is maximum span across x-axis, and Δy is maximum span across y-axis.

To position the final image to the screen centre, it is needed to apply offsets:

$$\begin{aligned} Offset_x &= \frac{w}{2} - \frac{x_{min} + x_{max}}{2} * Multiplier \\ Offset_y &= \frac{h}{2} - \frac{y_{min} + y_{max}}{2} * Multiplier \end{aligned} \tag{12}$$

Finally, the coordinate on the screen is found using:

$$\begin{aligned} x_{screen} &= x_{rotated} * Multiplier + Offset_x \\ y_{screen} &= y_{rotated} * Multiplier + Offset_y \end{aligned} \tag{13}$$

The set of entry points is selected by a user. The selection is performed by pointing at a location on the screen. Therefore, it is needed to convert the mouse location into the object coordinate space. This process is the inverse of the visualization process.

4.3.3 Numerical calculation of LCS

The core of the application can be described using a flowchart shown in Figure 14. The process can be divided into four major parts: preparation of the data, which is done as a part of GUI; calculation of the pathlines, see sections 3.2, 3.3 and Figure 15; creation of the FTLE field, section 3.4; and writing the results into the file.

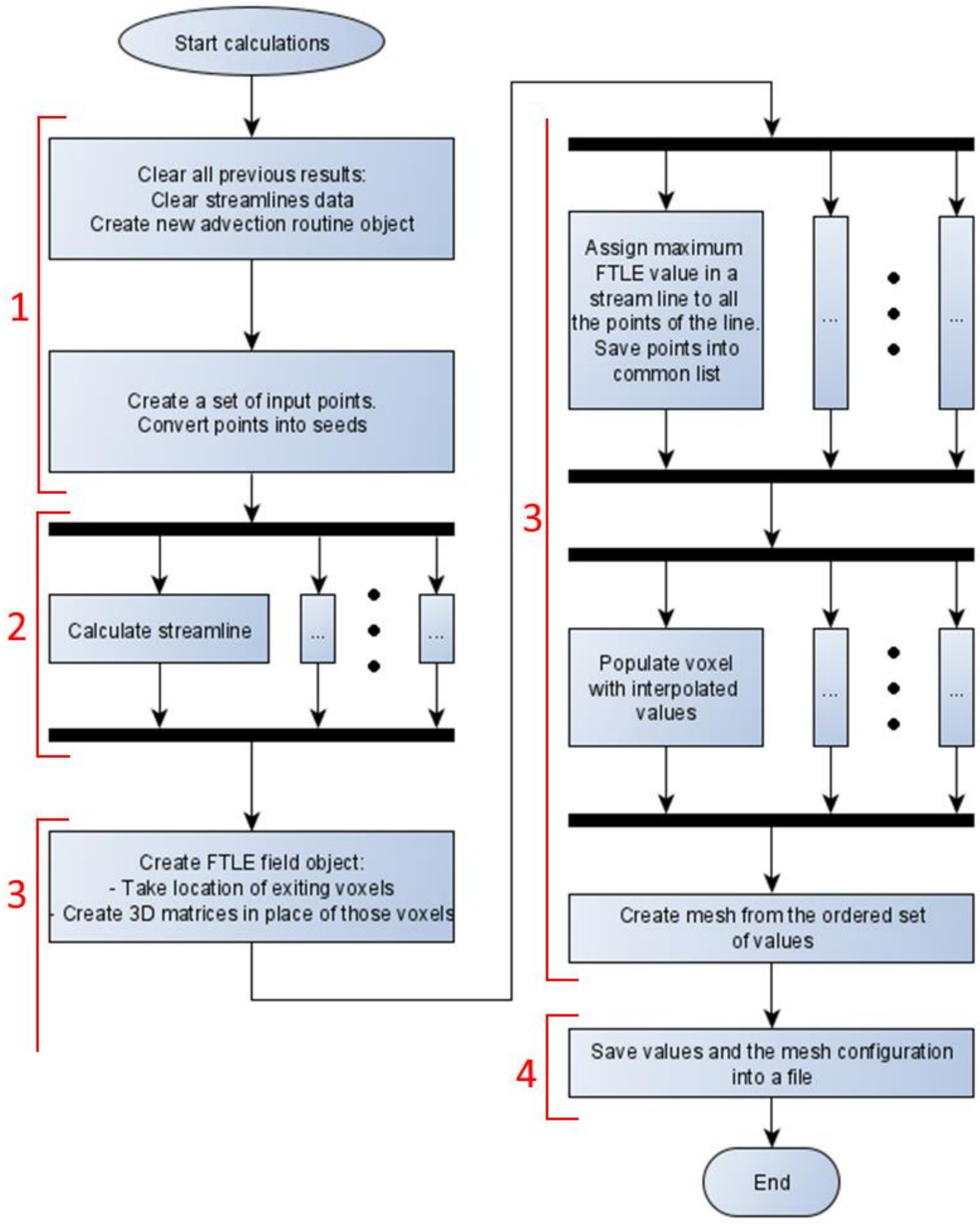

Start calculations
Clear all previous results:
Clear streamlines data
Create new advection routine object
1
Create a set of input points.
Convert points into seeds
2
Calculate streamline
...
...
3
Create FTLE field object:
- Take location of exiting voxels
- Create 3D matrices in place of those voxels
3
Assign maximum FTLE value in a stream line to all the points of the line.
Save points into common list
...
...
Populate voxel with interpolated values
...
...
Create mesh from the ordered set of values
4
Save values and the mesh configuration into a file
End

Figure 14. Program flow of calculation application

Calculation of the pathlines is another complex process, which is visually represented in Figure 15. Due to a high number of possible threads, the number of active threads is limited using a semaphore. This allows to reduce the time taken by context switching and increase the speed of calculations. Every particle calculation can be further parallelized if there is a need. Due to the simplicity of calculations, it was decided to leave this process sequential.

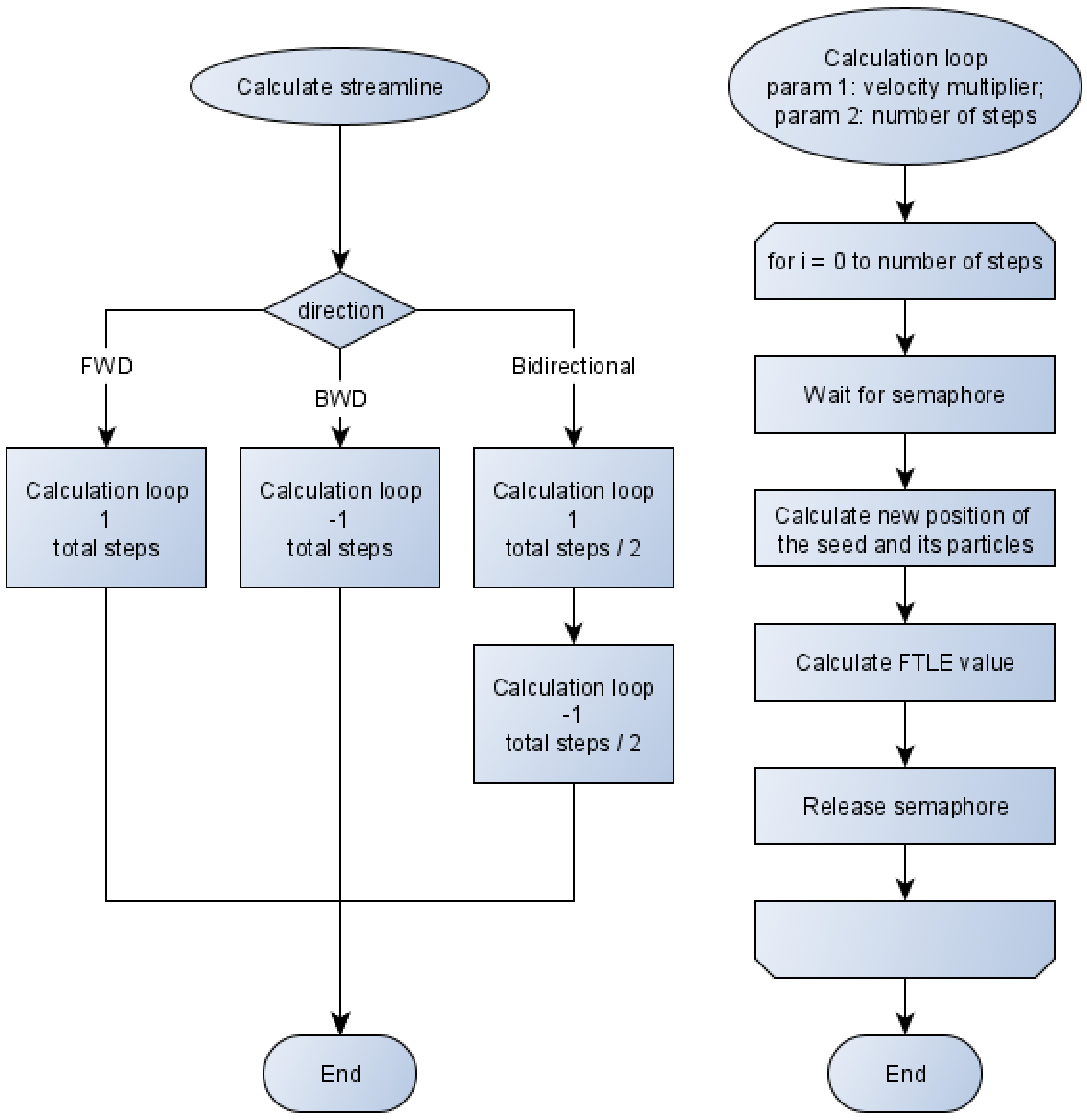

Calculate streamline
direction
FWD
BWD
Bidirectional
Calculation loop 1 total steps
Calculation loop -1 total steps
Calculation loop 1 total steps / 2
Calculation loop -1 total steps / 2
End
Calculation loop param 1: velocity multiplier; param 2: number of steps
for i = 0 to number of steps
Wait for semaphore
Calculate new position of the seed and its particles
Calculate FTLE value
Release semaphore
End

Figure 15. Program flow of pathline calculations

4.4 Visualization application

The main purpose of the visualization application is to create a 3D model from the calculated values. However, it is also needed to make the application easily accessible, with a simple GUI, and a certain minimum of possible interactions.

Unity has only one thread which can be used for manipulations with the graphical objects. Therefore, any function call which takes longer than one frame to process creates a disturbance for a user. A solution to this problem is to execute only functions which are directly connected to manipulation of the world in the main thread and perform all other calculations in independent threads. To make the code cleaner a state machine can be used (Figure 16), where each state either manipulates Unity related objects or waits for a certain thread to finish.

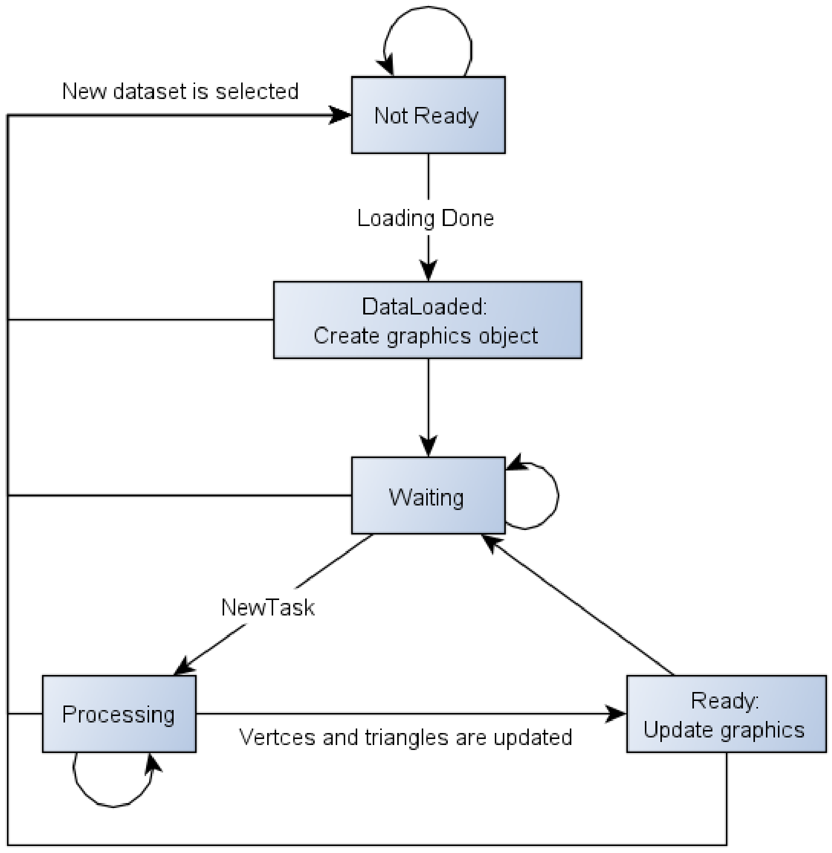


Figure 16. Visualization state machine

4.4.1 Tessellation

After the calculation's application has finished, the FTLE field creation it was still not possible to visualize the data efficiently, as described in section 4.2. Commonly, it is required to divide some physical shape into small stable shapes. One of the methods is to divide the surface into triangles [33] and, for example, build an STL model. The triangle is selected as a base element because it is the simplest and the most stable of 2D shapes, three points will always lie on the same plane. Therefore, dividing a surface into a set of adjacent triangles with defined normal vector is a simple and elegant solution. However, this solution is only perfect for surfaces and modification of the surface might result in a new complex problem. Another possibility is to divide the complete volume into a set of simple 3D shapes, such as tetrahedrons. This would allow to change the visibility of each tetrahedron, without major disturbance in the resulting surface. By using overlapping tetrahedrons it is possible to improve the quality of the resulting surface. There are different procedures on how to perform the division; for example, Voronoi diagram [34] or Delaunay refinement [35]. However, both of the methods require some initial shape, surface, boundary, which is not available in the developed application.

It is required to build a 3D object from scatter points without any boundary such that visibility of selected volumes can be changed during runtime. Additionally, this process should be performed within a certain time limit, to avoid the creation of a disturbance for a user. There are two general approaches for the problem, static and dynamic. Completely static approach would allow to change the visibility of a pre-defined set of points, and it would work as a look-up table. Which might not be very useful for a researcher. However, the performance of such approach would be very good. A completely dynamic approach, on the other hand, could allow complete freedom for selection. Additionally, it would allow for a very high-quality resulting surface, since it is recalculated on every change. Unfortunately, the dynamic approach would require considerably higher processing capabilities and more time. Thus, reducing the performance on average PC.

Due to performance requirements, the dynamic approach cannot be used, and the static approach does not provide enough freedom. Therefore, a combination of the two should be used. The first attempt to solve the problem mostly used the dynamic approach. From the regular grid of points, points with specified value would be selected. Based on the selected set, a mesh would be created such that the surface is as smooth as the grid allows. However, the solution has created major performance issues and was declined.

The solution used in the project mostly relies on the static approach. After the field of values is generated and arranged into a regular grid it can be viewed as a set of cubes, Figure 17 shows one cube from the grid. Every cube can be divided into four tetrahedrons: BCAF, DACH, EHFA, GFHC, where first three vertices are one of the bases arranged clockwise when viewed from the inside, and the last vertex is located on the same axis with the first vertex. This allows to have a relatively small set of pre-defined tringles. In the visualization application, this set is loaded into memory. Once a new minimum or maximum value is selected, each tetrahedron is tested if all four points lie within the required range. If at least one point lies outside of the range, the tetrahedron is removed from the mesh. However, since the set is stored in the memory changing visibility back does not require any additional processing.

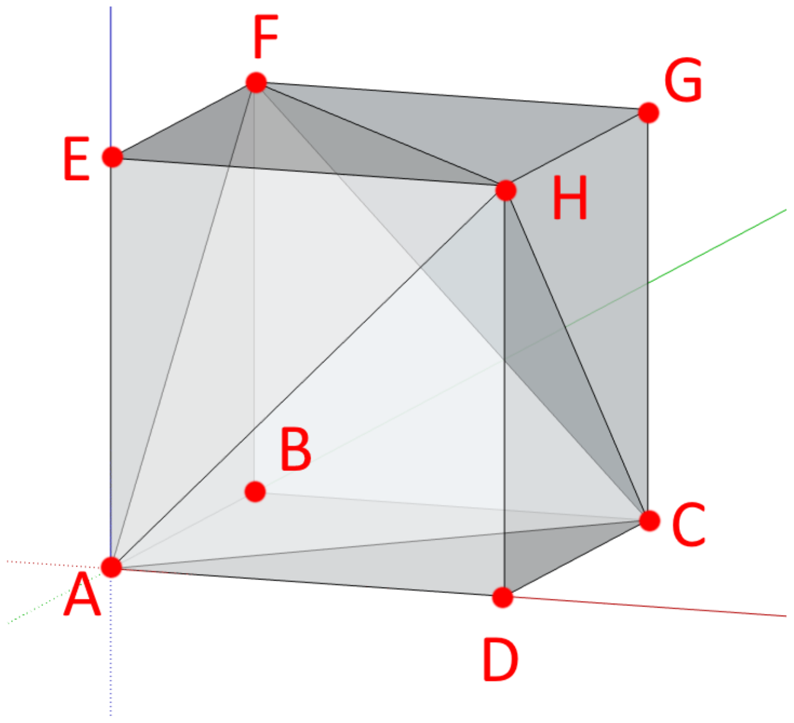


Figure 17. A single cuboid with equidistant node spacing is divided into 4 tetrahedrons.

4.4.2 GUI

The GUI can be divided into three main part (Figure 18). The first part is the canvas itself. Canvas is used to draw the object (examples can be seen in Figure 19), rotate and move the camera around it, and change zoom of the camera. All camera operations are performed using a mouse and follow some of the standard schemes, such as Unity3D scene editor control scheme. The second integral part is control of the visible layers of the object. To change the visibility of the layers simple sliders are used, which helps to associate the change in object geometry with the values. The sliders are divided into three categories: middle layer selection, width of a layer, and cut-off boundary. Middle layer is the value which is the central of the object. Width of the layer represents a value around the middle value past which the visibility changes. Cut-off values allow to create a contrast of values to make the picture clearer. Different settings can be seen in Figure 20.

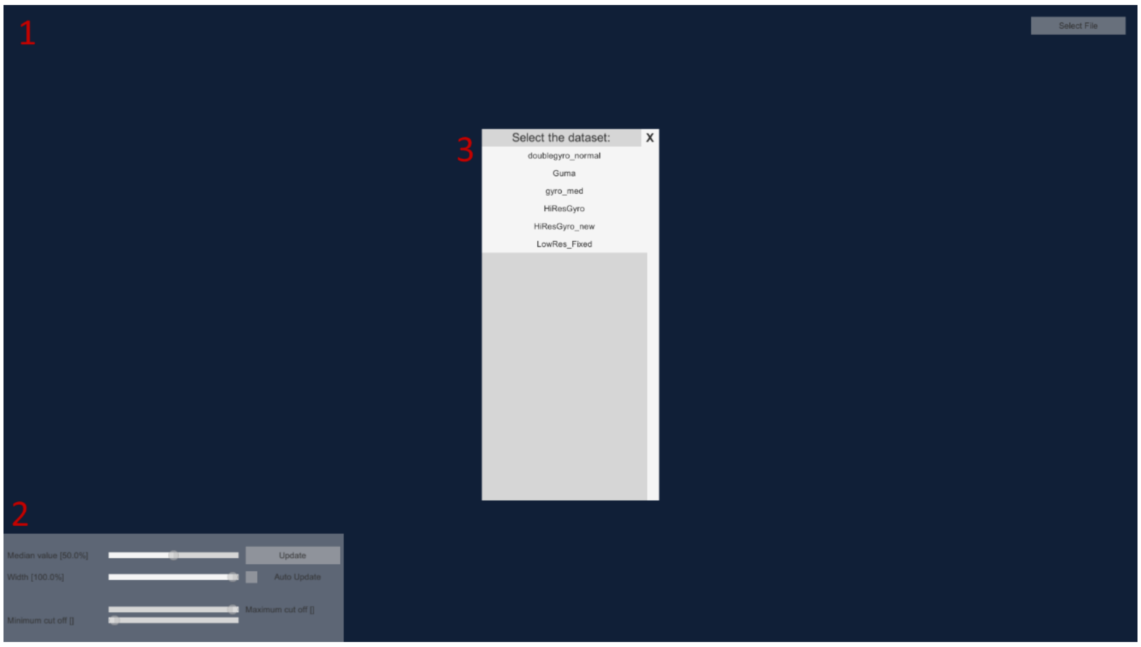


Figure 18. Visualization application GUI

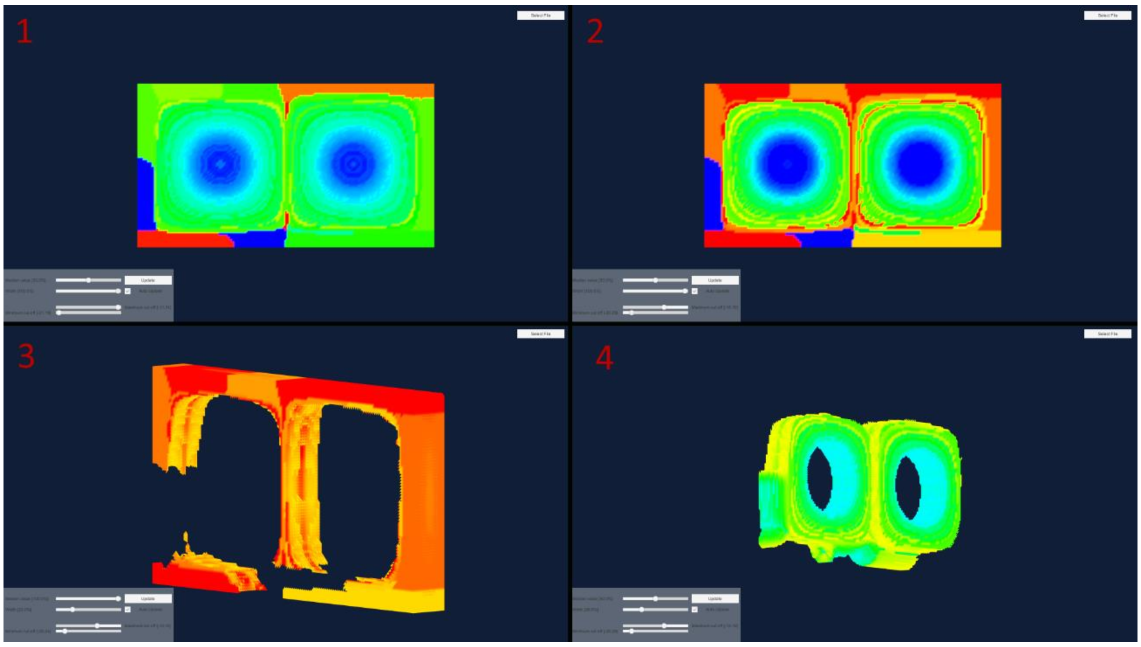
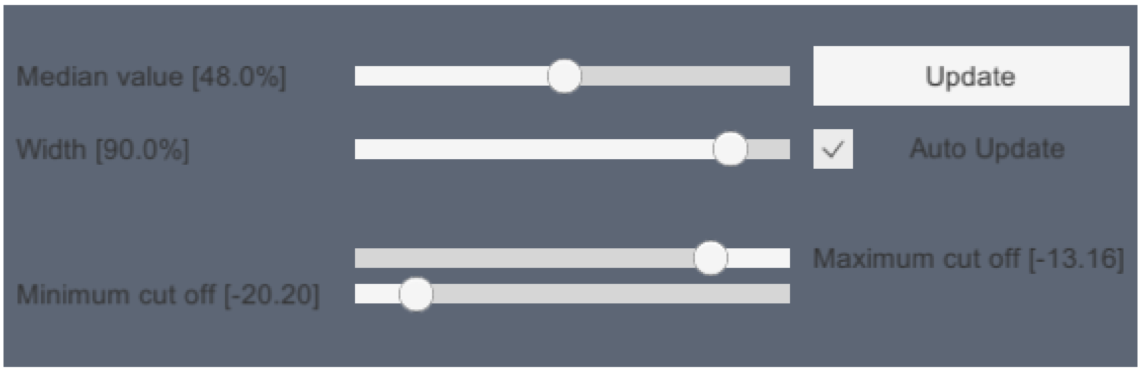


Figure 19. Different views and settings for a double gyre. Input dataset is planar, values share the depth value.

Maximum value: -11.74. Minimum value: -21.19.

1) Original, unmodified object. Outer boundaries are outside of the input dataset and create a visual disturbance. 2) Using cut-off values the contrast is increased. Boundary is more visible. 3) By setting width lower and median value higher only points with the highest values are visible. 4) The results of interpolation are more visible. Since the initial dataset is planar, depth is interpolated. This interpolation makes the boundary more continuous.

It is interesting that the boundary in Figure 19 is not uniform. The main reason is that the actual model is slightly smaller; therefore, the boundary is simply extrapolated from known data. Additionally, the model is physically restricted and the data outside the initial boundary is irrelevant. In order to make it more clear for the viewer it would be needed to have the physical model which would be used as a mask.


Median value [48.0%]

Width [90.0%]

Auto Update ✓

Maximum cut off [-13.16]

Minimum cut off [-20.20]

Figure 20. Visualization GUI. Settings panel.

4.4.3 Output data format

Since the applications are no longer connected it is needed to transfer data from one to another. It is also advised to have a local copy of the output, if it will ever be needed later.

Due to the large number of points, every unnecessary byte of data can take a considerable amount of space later. Therefore, storing data in any string format, such as CSV or JSON, is not efficient. The more efficient method is to convert output directly into bytes and store it as a binary file.

There are two significant datasets: FTLE field and a set of predefined tetrahedrons. From FTLE field it is needed to know the position of every point and its values. Additionally, in case of wrong order, numeric ID is added. Therefore, the format is the following:

```
ID X Y Z Vx Vy Vz FTLE ⇔ int float float float float float float double
```

Which results in 36 bytes per point. With a relatively small dataset of 40 000 points, the total volume of the file is 1.44 * 10^6 bytes. By utilizing csv file format, the size could be more than 3 times larger. Additionally, it could potentially create a risk of the wrong conversion, since many in-built converters use culture information of how the number is represented directly from the operating system.

A tetrahedron set is based on the points listed in FTLE field. Therefore, in order to reduce data repletion, only IDs of the points are used. The resulting format of the file is then the following:

```
ID_A ID_B ID_C ID_D ⇔ int int int int
```

Which requires 16 bytes per single tetrahedron. The set of 40 000 points will approximately result in more than 130 000 tetrahedrons with total size of 2.1 * 10^6 bytes.

5 Results

To evaluate the system and ensure that it performs as expected several tests were performed. The main metrics are the speed of calculations, accuracy of calculations, and performance of the visualization application. One of the concepts to described performance is time complexity, usually denoted as $O()$. Time complexity shows the worst-case scenario of how the number of elements affects the computation time. Due to the complexity of the algorithms, it was decided to find $O()$ empirically, by measuring run-time of multiple tests.

The first test case is the performance of the application when the number of calculation steps is different. It can be seen from Figure 21 the speed is linearly dependent on the number of steps. This is caused by the sequential nature of pathline calculations and it is only possible to improve the performance using different measures. Field creation in Figure 21, 22, and 23 represents the time needed to create a regular grid of FTLE values. This process depends on the speed of interpolation. Field creation has a linear dependence on the number of steps; however, the reason for the increase is connected to the interpolation algorithms. Since the total number of points has increased, the interpolation algorithm is working with the higher numbers, which results in the longer calculation time.

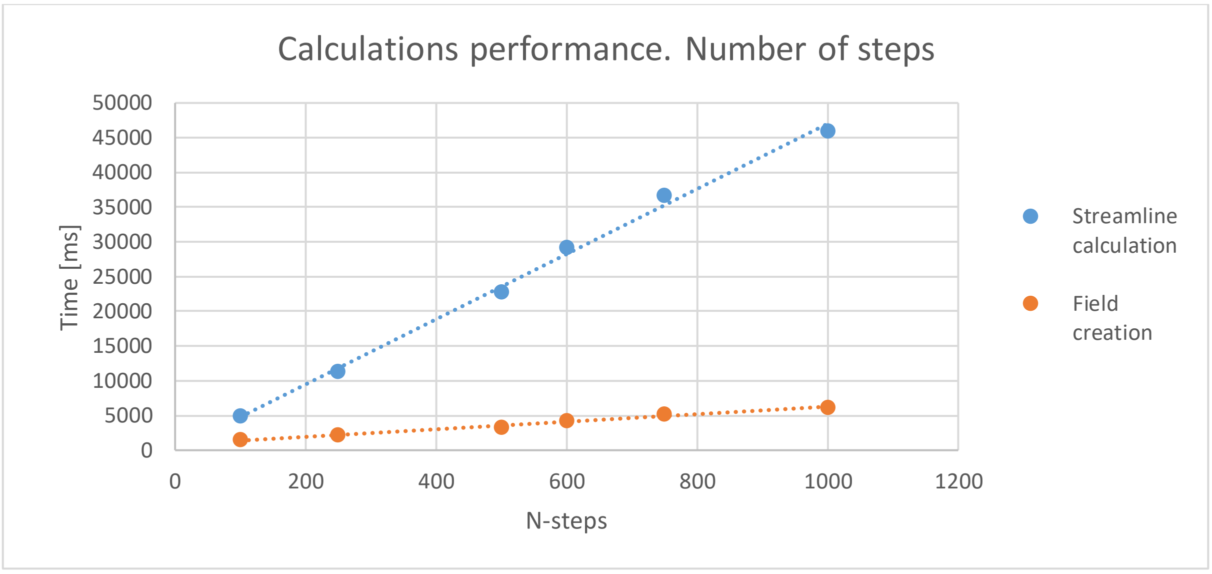


Figure 21. Performance of calculation process depending on the number of steps in a pathline.

The second test case was used to show the relation between the number of entry points and the computation time. Field preparation is a process of setting each value within a pathline to the maximum FTLE value of the pathline. Additionally, a new set of voxels is created, which will be filled with new values during field creation. As it can be seen from Figure 22, the number of entry points affects not only calculation time and field creation but also field preparation. It happens because the number of pathlines has increased, and it is needed to analyse each pathline in order to create the field. The relation is linear, except for the beginning. It is related to the parallelization of the program. Since the number of threads is limited, the growth in time is linear after the limit is reached. However, before the limit is reached the performance increase is substantial.

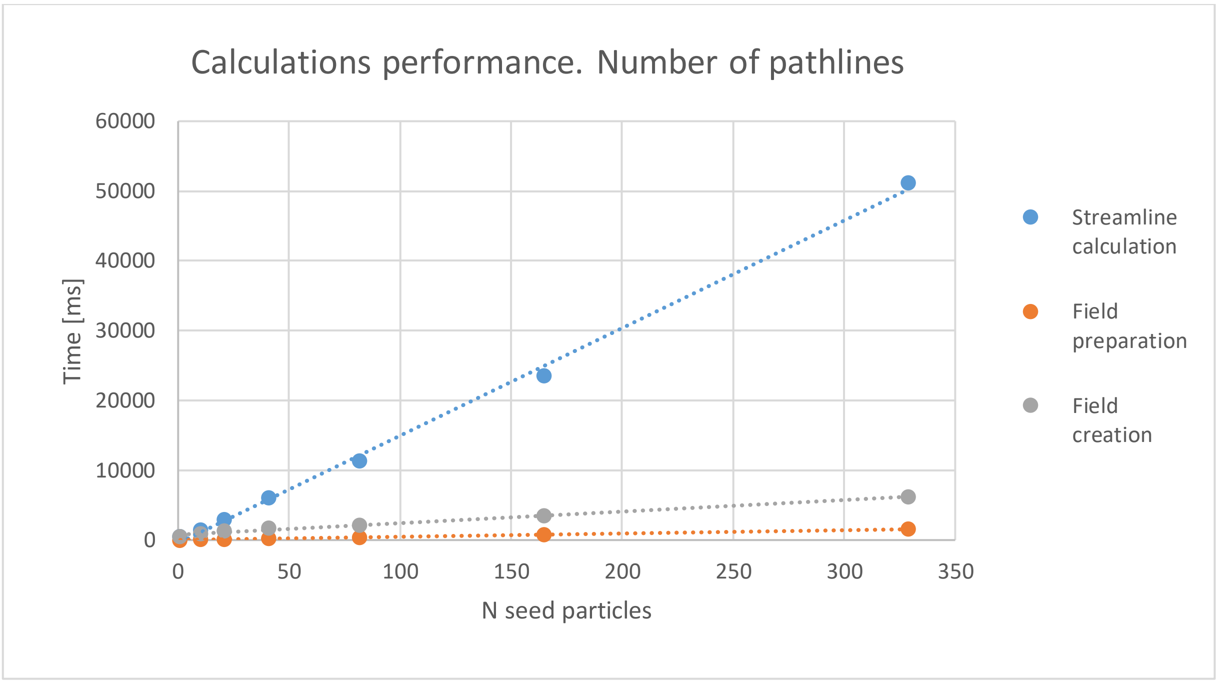


Figure 22. Performance of calculation process depending on the number of pathlines

Increasing the resolution of the field has a great impact on performance (Figure 23). First, the amount of points is increasing in $v * r^3$, where v is the number of voxels, r is points per voxel side. Field creation is then linearly dependent on the total amount of points. However, serialization of the field and the connections has $O(n^2)$ which reduces the performance further.

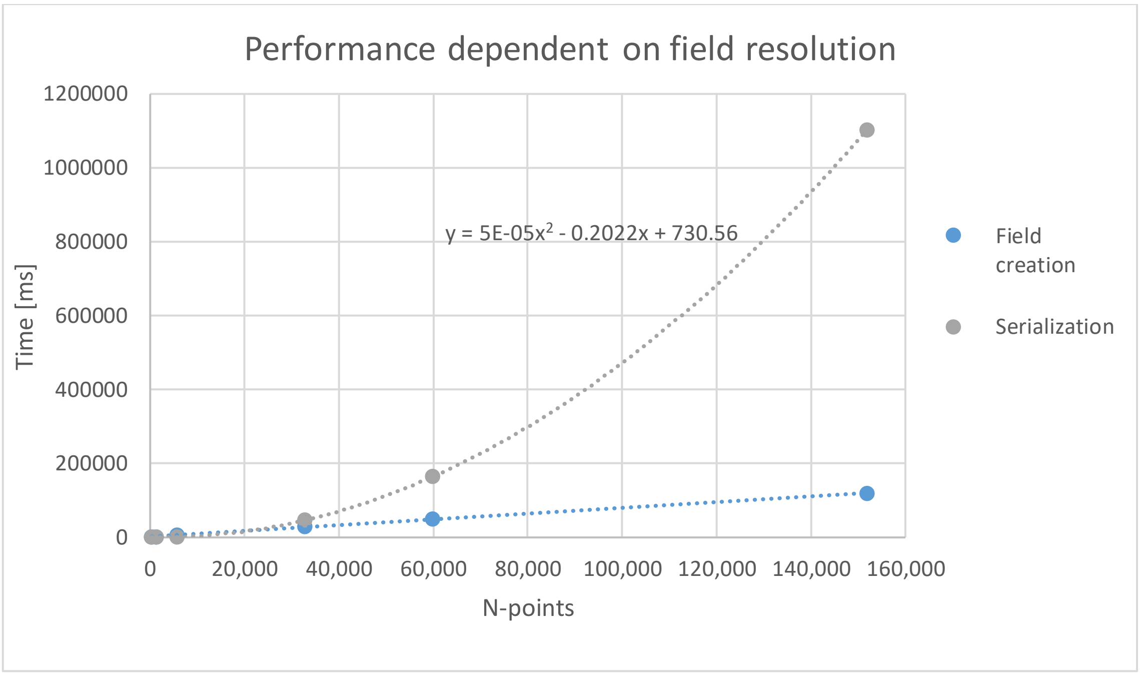


Figure 23. Performance results depending on field resolution/total number of points. In this example, 150 000 points are equivalent to the resolution of ~216 000 points per unit cube

Several tests have shown that the total number of points does not affect the performance of the visualization software. Tested numbers varied from 360 to 360 000 data points or resolution of two points per voxel side to 20. Larger amounts still might create a disturbance. However, due to limitations of serialization time, it is not a major issue.

The accuracy of the results depends largely on the size of a time step. Figure 24 and 25 shows three different steps: 0.05, 0.1, 0.4. Since the gyre is supposed to have a closed loop, the error can be clearly visible. As it can be seen from Figure 24, the further the point from the centre, the higher the error. In order to calculate the rate of error growth depending on the timestep, a series of tests were performed. Initial point was fixed to a certain position and error between first and a point on the same Y-level was calculated (Table 8). The percentage is calculated as (14). With the linear increase in computation time, it is advised to reduce the size of a step until the desired computation time limit is reached.

$$Error\ (\%) = \frac{Error\ (\Delta)}{|x_{centre} - x|} * 100\% \qquad (14)$$

Size of a voxel does not seem to significantly improve accuracy in the case of a gyre. Therefore, the computational time should be used as the main guideline for voxel size selection.

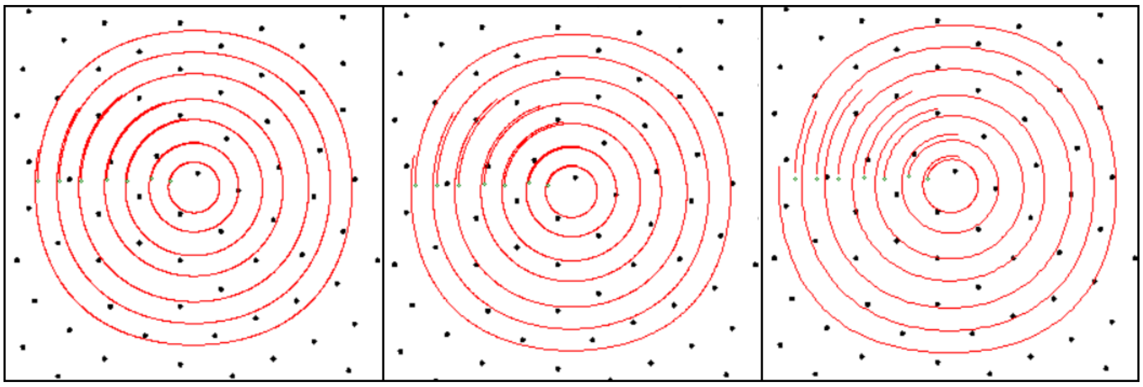


Figure 24. Accuracy of results depending on the length of a timestep. The error is represented graphically as a difference between start and end points of a pathline along x-axis. The start and end points are given by the green dots.

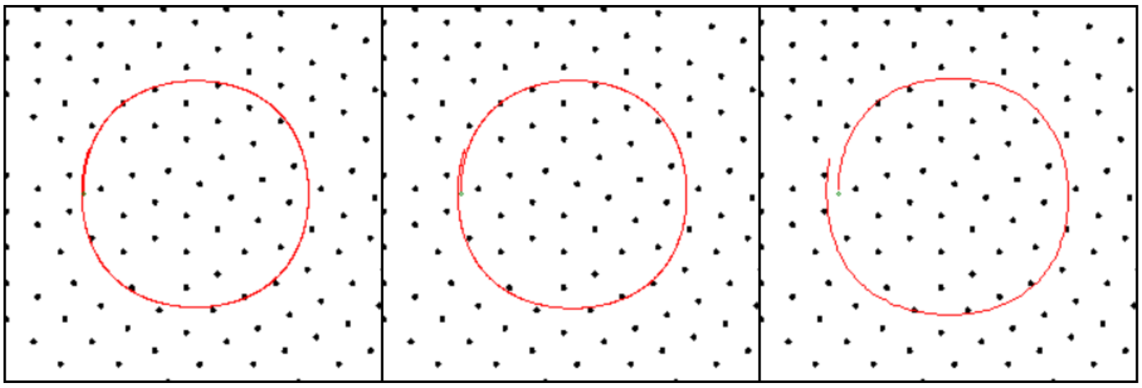


Figure 25. Fixed point at x = 0.15 is advected and the error is calculated

Table 8. Error growth depending on the time step. Centre is at 0.5

Timestep [unit]	**X = 0.15 (-0.35)**		**X = 0.25 (-0.25)**	
	Error [Δ]	**Error [%]**	**Error [Δ]**	**Error [%]**
0.05	0.0034	1.0	0.0032	1.3
0.1	0.0068	1.9	0.0064	2.6
0.4	0.0263	7.5	0.0257	10.3

Additional outputs are shown in Appendex 2.

6 Future development

Future development can be classified into several separate tasks. The first is to perform extensive user testing and improve the UX in the process. This includes the addition of help menus, overlay hints, improvement of textual context. In addition to those straightforward measures, it is also might be needed to implement new tools for input and output data manipulations in order to simplify the process and to fulfil all the needs. The second task is the optimization of the structure of the code. After the core of the program is done it is possible to look at it as a whole and to overhaul its structure. By making a more readable code it is possible to make it much easier to comprehend and later use it as a start point for further development. The third is the optimization of the calculations and further parallelization of the application. Even though complete parallelization won't be needed, it still might prove to be useful because it will be possible to scale the system up easier. The last task is to improve the performance by accessing the hardware through different means. Due to limited time and knowledge of the means, it wasn't possible to get into the details of this problem. With additional resources it is possible to address the task, as a result, it is possible to make further optimization and to make the tool even more accessible.

7 Conclusion

The selection of the language and environment had a great impact on the development. C# proved to be simple to use language. High-quality Unity3D and Microsoft documentation allowed to easily use all available features of the language and of the environment. The active community has provided answers to many problems during the development.

Through testing and analysis of different interpolation algorithms, it was possible to improve the accuracy and performance of calculations. Due to the heavy use of interpolation, even a small increase in performance has resulted in significantly better results. However, there are still unexplored algorithms, which could improve the quality and speed in the future.

Multiple iterations of development cycle provided a great increase in the quality of the final solution. Since many problems were resolved in the early stages, it was possible to concentrate on the performance optimization of the program. A good structure and encapsulation allowed to make the iterations easier without significant changes in the overall hierarchy.

By dividing the development into two separate branches, it was possible to simplify the process of development and to concentrate on each task separately. Heavily parallelized calculation application allowed to use hardware resources efficiently. This significantly increase the performance and the main limitation is hardware now.

Several challenges were faced during the development. Firstly, it was difficult to find the right code structure. This has led to a complete overhaul of the code at the beginning of the development and several minor re-implementations at the middle and end of the development. Secondly, it is challenging to parallelize the code. The synchronization issues sometimes are not visible, and it is difficult to debug the code. Additionally, it is not always possible to parallelize the code in its initial state and it is required to restructure the program flow. Finally, selection and implementation of algorithms can be difficult due to a lack of previous knowledge, since the number of possible algorithms and their implementations is high.

All in all, it was a very interesting project. It was exciting to learn a new language and master the understanding of Unity3D. Additionally, it became very clear why the performance of an algorithm is important in computer science. Once the algorithm is used intensively, every imperfection becomes more visible in terms of total computation time.

Link to the code repository and general instructions about usage is in Appendix 1.

References

[1] F. Lekien, C. Coulliette, and J. Marsden, "Lagrangian structures in very high-frequency radar data and optimal pollution timing," in *AIP Conference Proceedings*, Aug. 2003, vol. 676, no. 1, pp. 162–168, doi: 10.1063/1.1612209.

[2] M. J. Olascoaga and G. Haller, "Forecasting sudden changes in environmental pollution patterns," *Proc. Natl. Acad. Sci. U. S. A.*, vol. 109, no. 13, pp. 4738–4743, Mar. 2012, doi: 10.1073/pnas.1118574109.

[3] M. B. Mathur and M. B. Mathur, "The National Meteorological Center's Quasi-Lagrangian Model for Hurricane Prediction," *http://dx.doi.org/10.1175/1520-0493(1991)119<1419:TNMCQL>2.0.CO;2*, Jun. 1990, doi: 10.1175/1520-0493(1991)119<1419:TNMCQL>2.0.CO;2.

[4] J. Vétel, A. Garon, and D. Pelletier, "Lagrangian coherent structures in the human carotid artery bifurcation," *Exp. Fluids*, vol. 46, no. 6, pp. 1067–1079, Jun. 2009, doi: 10.1007/s00348-009-0615-8.

[5] J. Blazek, *Computational fluid dynamics: principles and applications*. Butterworth-Heinemann, 2015.

[6] S. C. Shadden, F. Lekien, and J. E. Marsden, "Definition and properties of Lagrangian coherent structures from finite-time Lyapunov exponents in two-dimensional aperiodic flows," *Phys. D Nonlinear Phenom.*, vol. 212, no. 3–4, pp. 271–304, Dec. 2005, doi: 10.1016/j.physd.2005.10.007.

[7] J. Jacobson and M. Lewis, "Game Engines in Scientific Research," *Commun. ACM*, vol. 45, no. 1, pp. 27–31, Jan. 2002.

[8] T. Bergmann *et al.*, "Inspiration from VR gaming technology: Deep immersion and realistic interaction for scientific visualization," in *VISIGRAPP 2017 - Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 2017, vol. 3, pp. 330–334, doi: 10.5220/0006262903300334.

[9] "Features - Unreal Engine." https://www.unrealengine.com/en-US/features (accessed May 07, 2020).

[10] "Game engines - how do they work? - Unity." https://unity3d.com/what-is-a-game-engine (accessed May 07, 2020).

[11] E. T. Kai *et al.*, "Top marine predators track Lagrangian coherent structures," *Proc. Natl. Acad. Sci. U. S. A.*, vol. 106, no. 20, pp. 8245–8250, May 2009, doi: 10.1073/pnas.0811034106.

[12] M. R. Allshouse and J. L. Thiffeault, "Detecting coherent structures using braids," *Phys. D Nonlinear Phenom.*, vol. 241, no. 2, pp. 95–105, Jan. 2012, doi: 10.1016/j.physd.2011.10.002.

[13] Wikimedia Commons, "File:Wikipedia image 2.png --- Wikimedia Commons{,} the free media repository." 2016, [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:Wikipedia_image_2.png&oldid=218520861.

[14] B. Flood, "Investigating Lagrangian Coherent Structures in Environmental Flows," no. December, 2013.

[15] "56e960696b7cf.jpg (1500×500)." https://public-media.interaction-design.org/images/ux-daily/56e960696b7cf.jpg (accessed May 02, 2020).

[16] "LCS Tutorial: MANGEN." https://shaddenlab.berkeley.edu/uploads/LCS-

tutorial/mangen.html (accessed Apr. 28, 2020).
[17] “Software — DABIRI LAB.” http://dabirilab.com/software (accessed Apr. 28, 2020).
[18] “GitHub - elehcim/cuda_ftle: My version of Jimenez’s FTLE parallel calculator written in CUDA.” https://github.com/elehcim/cuda_ftle (accessed Apr. 28, 2020).
[19] “GitHub - Shibabrat/Newman.” https://github.com/Shibabrat/Newman (accessed Apr. 28, 2020).
[20] “jtuhtan/lcs: MATLAB code for processing 2D Langrangian Coherent Structures from a stationary velocity vector field.” https://github.com/jtuhtan/lcs (accessed Apr. 08, 2020).
[21] “index | TIOBE - The Software Quality Company.” https://www.tiobe.com/tiobe-index/ (accessed May 02, 2020).
[22] “LCS Tutorial: How the FTLE is used.” https://shaddenlab.berkeley.edu/uploads/LCS-tutorial/FTLE-interp.html (accessed May 03, 2020).
[23] “Interpolation | mathematics | Britannica.” https://www.britannica.com/science/interpolation (accessed May 14, 2020).
[24] “Grid.” .
[25] J. L. Bentley, “Multidimensional Binary Search Trees Used for Associative Searching,” *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, doi: 10.1145/361002.361007.
[26] A. Guttman, “R-TREES. A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING.”
[27] SHEPARD D, “Two- dimensional interpolation function for irregularly- spaced data,” in *Proc 23rd Nat Conf*, 1968, pp. 517–524, doi: 10.1145/800186.810616.
[28] B. W. Boehm, “A Spiral Model of Software Development and Enhancement,” *Computer (Long. Beach. Calif).*, vol. 21, no. 5, pp. 61–72, 1988, doi: 10.1109/2.59.
[29] “Voxel | Definition of Voxel by Merriam-Webster.” https://www.merriam-webster.com/dictionary/voxel (accessed May 14, 2020).
[30] “Accord.NET Machine Learning Framework.” http://accord-framework.net/ (accessed May 10, 2020).
[31] E. W. Weisstein, “Rotation Matrix.”
[32] “Rotation Matrix -- from Wolfram MathWorld.” https://mathworld.wolfram.com/RotationMatrix.html (accessed Apr. 11, 2020).
[33] “Unity - Scripting API: Mesh.” https://docs.unity3d.com/ScriptReference/Mesh.html (accessed Apr. 03, 2020).
[34] D.-M. Yan, W. Wang, B. Lévy, and Y. Liu, “Efficient Computation of 3D Clipped Voronoi Diagram.”
[35] J. R. Shewchuk, “Tetrahedral Mesh Generation by Delaunay Refinement.”

Appendix 1 – code repository and instructions

The project can be found on GitHub - https://github.com/YuryMalyshev/unity3d-master

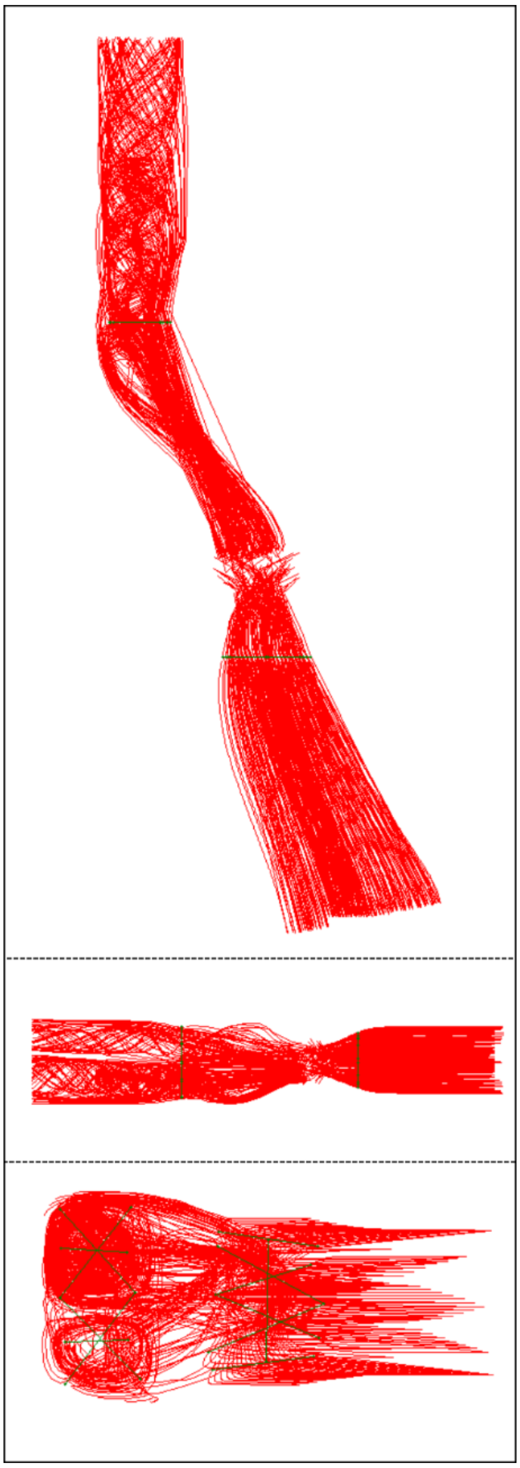
The project is divided into two applications: calculations and visualization. Calculation application can be found under 'AdvectionCalculationsGUI', which is a Visual Studio 2019 project. Visualization application is located under 'LCS', it is a standalone Unity3D 2019 project.

Compiled applications can be found 'Build' directory in the repository. No third-party software is required to launch the compiled application. The applications were built under Windows platform.

Additional and up-to-date information can be found in README of the git repository.

Appendix 2 – Visualization of hydropower plant

Additional resulting images. The first image shows pathlines which were generated during advection procedure from different views.



The second image is FTLE visualization performed in Unity3D.