D4.6 Visualizations for Global Challenges

Date: May 31st, 2024

Document Identification			
Status	Final	Due Date	31/05/2024
Version	1.0	Submission Date	31/05/2024

Related WP	WP4	Document Reference	D4.6
Related Deliverable(s)	D2.1, D4.7, D4.8	Dissemination Level (*)	PU
Lead Participant	MTG	Lead Author	David Caballero (MTG)
Contributors	SZE, PSNC, USTUTT, UNISTRA, FAU	Reviewers	Wojciech Szeliga (PSNC) Harald Koestler (FAU)

Keywords:
Visualization, Visual Data Analysis, CFD, VR

Document Information

List of Contributors	
Name	Partner
David Caballero	MTG
Luis Torres	MTG
Kilian Türk	USTUTT
Zoltán Horváth	SZE
Mátyás Constans	SZE
Christophe Prud'homme	UNISTRA
Harald Koestler	FAU
Wojciech Szeliga	PSNC

Document History			
Version	Date	Change editors	Changes
0.1	14/05/2024	David Caballero (MTG) Luis Torres (MTG)	First draft of the document
0.15	15/05/2024	Marcin Lawenda (PSNC) Harald Koestler (FAU)	ToC, timeline and responsibilities approved
0.2	17/05/2024	David Caballero (MTG) Mátyás Constans (SZE) Harald Koestler (FAU)	Integration of SZE and FAU contributions
0.25	21/05/2024	David Caballero (MTG) Christophe Prud'homme (UNISTRA)	Integration of UNISTRA contributions Preparation of the version for peer review
0.3	24/05/2024	David Caballero (MTG) Harald Koestler (FAU)	Changes in format, according to general template Integration of FAU contributions and corrections

0.35	27/05/2024	David Caballero (MTG) Kilian Türk (USTUTT) Wojciech Szeliga (PSNC)	Integration of new USTUTT contributions Integration of PSNC contributions
0.98	30/05/2024	David Caballero (MTG)	Integration of H. Koestler's review comments Integration of W. Szeliga's review comments Review and edition of the pre-final version for submission
0.99	30/05/2024	Harald Koestler (FAU)	Quality assurance check
1.0	31/05/2024	Marcin Lawenda (PSNC)	Final check and corrections

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	David Caballero (MTG)	30/05/2024
Quality manager	Harald Koestler (FAU)	30/05/2024
Project Coordinator	Marcin Lawenda (PSNC)	31/05/2024

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
ASCII	American Standard Code for Information Interchange
APK	Android Application Package
AR	Augmented Reality

BP	Blueprint
EC	European Commission
CAVE	Cave-Assisted Virtual Environment
CEEC	Center of Excellence for Exascale CFD
CFD	Computer Fluid Dynamics
CFDR	Computer Fluid Dynamics Rendering
COVISE	Collaborative Visualization and Simulation Environment
CPU	Central Processing Unit
DTM	Digital Terrain Model
Dx.y	Deliverable number y belonging to WP x
ESRI	Environmental Systems Research Institute
FBX	FilmBoX
FGA	Fluid Grid ASCII
FOAM	Field Operation And Manipulation
fps	Frames per second (Hz)
GIS	Geographical Information System
GPU	Graphics Processing Unit
GS	Gaussian Splatting
GUI	Graphical User Interface
HLRS	High-Performance Computing Center Stuttgart (**H**öchst**L**eistungs**R**echenzentrum **S**tuttgart)
HPC	High Performance Computing
IO	Input-Output
LUMI	Large Unified Modern Infrastructure
MTW	Material Transport in Water
NeRF	Neural Radiance Field
RES	Renewable Energy Sources
SHP	Shapefile
SVT	Sparse Volumetric Texture
UAP	Urban Air Project
UBM	Urban Building Model

UE	Unreal Engine
UEAV	Unreal Engine Advanced Visualization
VDB	Volume Dynamic B+Tree
VR	Virtual Reality
VTK	Visualization Toolkit
WF	Wildfires
WP	Work Package
WRF	Weather Research and Forecasting
WUI	Wildland-Urban Interface
XR	Extended Reality

Executive Summary

The main objective of the HiDALGO2 project is to explore, develop, and implement methods and techniques that utilize exascale HPC resources to address current environmental challenges such as air pollution, the use of renewable energy, sustainable building practices, and the prevention and mitigation of wildfire effects amongst others. The physical processes involved in these phenomena are complex, dynamic, volumetric, non-linear, and intricately interrelated, especially those occurring in or interacting with the atmosphere. Computational Fluid Dynamics (CFD) models are extensively used to simulate these processes and the evolution of governing factors, thereby enhancing our understanding of their behaviour and impact, and enabling the implementation of corrective or mitigating measures.

In the HiDALGO2 project, five pilot studies have been proposed, namely:

1. Urban Air Project (UAP),
2. Urban Building Model (UBM),
3. Renewable Energy Sources (RES),
4. Wildfires (WF), and
5. Material Transport in Water (MTW).

All of these pilots utilize CFD solutions to a varying degree on real-world scenarios interpreted as digital twins. These pilots are developed in WP05 and are related to data exploration and visualization methods in WP04 and parallel computing techniques in WP02 and WP03 of the project.

The models used in the pilots return time sequences of the evolution of volumetric fields of scalar values (temperature, pressure, pollutant concentration, etc.) and vector fields (wind, fluid flows etc.), which are sometimes challenging to analyse visually. Today, there are tools available to enhance visual analysis capabilities, especially in scientific and technical fields.

This deliverable, first, brings together the visualization requirements of each pilot, in line with the ultimate goal of analysing and utilizing the results while considering the specificities of each use case, as initially outlined in deliverable D2.1. It also describes the framework within which the visualization solutions are embedded, as part of the results, prototypes, and services that will be delivered as a dashboard at the project's conclusion. Additionally, the deliverable includes, based on the analysis of

requirements, the description of four complementary technological solutions for visualization, each with specific capabilities:

1. The first solution (**CFDR**) efficiently packages the CFD simulation data and sends it to web-based graphical clients for lightweight, real-time interactive visualization.
2. The second solution (**VISTLE-COVISE**) allows for interactive and immersive visualization in CAVE environments for collective analysis of simulation results.
3. The third solution (**UEAV**) proposes the compilation of photorealistic VR experiences, including simulation data and the integration of geographic components, objects, and special effects into highly portable packages for training.
4. The fourth solution (**KTIRIO-GUI**) enables the extraction of spatial data from the selected geographic region, the integration of simulation data, and interactive visualization using commonly used generic applications.

While each solution is originally designed to operate in specific pilots (CFDR for the UAP pilot, VISTLE-COVISE for the RES pilot, UEAV for the WF pilot, and KTIRIO-GUI for the UBM pilot), all offer sufficient flexibility to be used jointly or cross-cutting among different pilots. Therefore, the exchange formats used in each section are specified, and workflows are proposed to maximize their benefit within the project. Despite being based on mature and extensively tested technologies, the application of these solutions to these specific pilots involves a certain degree of innovation, thus extending their application in the short and medium term to other domains of environmental challenges. This deliverable explores immediate future developments that could enhance and extend their functionality. It also provides insights into emerging visualization technologies that could be applied in the coming years, such as the use of NeRFs or Gaussian Splatting, presenting some preliminary examples.

Although this document has a closed structured format, its content is intended to be an open document, primarily because the technologies and tools it involves are in constant and rapid development. Therefore, future updates to the deliverable can be expected as new approaches and solutions are tested and incorporated into the project. In this way, this deliverable D4.6 positions itself as an observatory of new technologies and applications for the visualization of HPC simulation data for environmental challenges.

1 Introduction

Many of the processes and phenomena observed in the environment occur dynamically, non-linearly, and in volumetric spaces, involving components that are difficult to represent for visual analysis. The environmental challenges addressed by the HiDALGO2 project are not exempt from this complexity. Atmospheric pollution, renewable energies, sustainable building practices, and forest fires involve complex modelling and results in the form of vector fields, scalar fields, evolution over time, and their relationships. Moreover, according to the requirements of end-users, it is necessary to obtain representation methods that are intuitive and clear but do not limit or reduce the accuracy obtained in numerical simulation.

In the scientific and technical domain, specific visualization programs have been used to input data files in the most common exchange formats. These programs, widely popular, have added capabilities such as temporal evolution and animations, shading methods with complex colour palettes, coupling of visualization strategies (isosurfaces, contour lines, wireframes, slicing, point clouds, flow lines, 3D glyphs, ray-marched volumes, etc.), and immersion using VR or XR devices.

Besides, in recent years, sophisticated visualization techniques supported by parallel hardware development with impressive capabilities have been developed, largely driven by the video game industry. These new developments, which include real-time ray tracing, virtual reality projection, or augmented reality scenarios, etc., also require visualization to be agile, interactive, and preferably device-independent.

Visualizing complex and costly simulations performed on HPC infrastructure requires the rationalized use of these techniques and methods to ensure an intuitive, fluid, yet meaningful representation. Thus, in the immediate future, these solutions for visual data analysis must be **I**^{3}, i.e., **I**ntuitive, **I**nteractive, and **I**mmersive, allowing also the interaction between multiple users in the same visualization environment.

This document explores the development and implementation of advanced data visualization strategies and technologies and their application to pilot studies in the HiDALGO2 project. With a clear focus on scalability, these methods can also be integrated into the workflow of other challenges and application domains.

1.1 Purpose of the document

This document provides the initial strategies to enrich pilot applications by meaningful visualization techniques such as virtual reality or scientific visualization approaches. It outlines data, requirements and methods that must be used/ developed to deliver efficient and productive visual analysis tools. To do that the deliverable translates prerequisites from D2.1 into functional software definitions in the visualisation domain. This is a living document that summarizes most recent findings and will be advanced in D4.7 and D4.8.

1.2 Relation to other project work

As can be understood, the visualization domain has many connections with other parts of the HiDALGO2 project, namely:

- On one hand, the proposed visualization solutions address the requirements obtained in the requirements analysis of **WP2** and documented in deliverable D2.1.
- Similarly, the visualization solutions are directly related to the development of GUIs and the services offered in the final dashboard of services, also in WP2.
- The most direct relationship is with other activities within **WP4**, to which this task belongs, particularly its relation with data analysis.
- Finally, each proposed visualization solution is directly related to the developments carried out in each of the pilots, in **WP5**, from which it draws.

1.3 Structure of the document

This document is structured into 8 major chapters, which are briefly described below:

- **Chapter 1 Introduction**. This chapter briefly describes the need addressed by the document, its purpose, structure, and relationship with other works within the project.
- **Chapter 2 Rationale**. This chapter establishes the logical framework of the document and the relationship between requirements and proposed solutions.
- **Chapter 3 Technology solutions**. This chapter describes the technological solutions adopted for the development of visualization methods in pilot studies.

- **Chapter 4 Application to pilots**. This chapter describes the workflows and implementation of technologies to address the specific requirements of each pilot.
- **Chapter 5 Looking into the future**. This chapter provides a brief overview of the future applications of the adopted methods and technologies, as well as a vision of emerging visualization technologies and how they could enhance or improve capabilities.
- **Chapter 6 Conclusions**. This chapter summarizes the main points developed in the document, projecting them towards future work.
- Additionally, a **Chapter 7** of **References** and a **Chapter 8** with the **Annexes** are included.

2 Rationale

In this document, first a review is done of the visualization **requirements** for each of the pilots, according to the general requirements established in deliverable D2.1 and subsequently, including the specifics of each pilot. Subsequently, a description of the **technologies** and visualization strategies adopted to meet the requirements is provided. These technologies include both existing solutions and bespoke developments aimed at facilitating the representation of HPC simulation results. The next stage is the description of the **workflows** that utilize the described technologies to meet the usage requirements established in the first point. These workflows can be integrated into a single application or result in data and models encapsulated in highly portable experiences. Although initially each workflow is associated with a pilot, the potential cross-application of technologies to other pilots is also explored. This aspect adds scalability and usability to the proposed solutions within the project and thus increases their potential for future exploitation. Finally, **emerging technologies** and alternative workflows or future applications are also identified, both in the pilot studies of the project and in their application within the framework of other projects or developments.

2.1 Objectives

The main objective of this document is to describe the workflows, formats, and tools used in the advanced visualization of calculation results in HPC infrastructures, with a strong focus on volumetric thermo-fluid dynamics solutions applied to environmental challenges (such as atmospheric pollution in cities, renewable energies, forest fires, etc.). These workflows are automated to varying degrees for potential application not only in the selected pilots but also more generally in other use cases within and outside the project. The visualization solutions themselves will be final products of the project applicable to general-purpose simulations.

Since the results obtained in fluid dynamics simulation and other physical phenomena considered in the selected environmental challenges represent volumetric, nonlinear, and dynamic datasets with complex relationships between them, the proposed visualization solutions aim to use 3D environments where interactivity and navigation are agile and intuitive, and, as far as possible, immersive. Additionally, for awareness and training applications, visualization solutions with a high degree of realism and georeferenced in specific locations at different scales are provided.

In line with the aforementioned general objectives, three lines of visualization development are outlined in the HiDALGO2 project, all three are complementary:

1. Development of a solution for advanced visualization of CFD simulations, for general use, accessible and operable through web services with an interactive and intuitive GUI.
2. Development of an immersive and cooperative solution based on CAVE technology for collective visualization of large territorial extensions and exploration of simulation results in HPC infrastructure.
3. Development of a solution for photorealistic, immersive, and interactive visualization of CFD and forest fire simulations, packaged in virtual reality experiences for training and awareness.

Additionally, other minor applications for visualization and result analysis based on ParaView and other generic solutions are also considered.

2.2 Requirements

General requirements on visualization are stated in deliverable D2.1 Requirements Analysis and Scenario Definition, namely:

- REQ-POR-027: Web-based 3D CFD-visualization. A web-based post-processing tool which can visualize ParaView compatible datasets
- REQ-POR-028: Visual components for time-series data. Visual components for presentation of collection of observations (behaviour) for a single subject (entity) at different time intervals
- REQ-POR-029: Visual component for geo-temporal data. Provide visual components for visual exploration of geo-temporal data, and for annotating patterns. Geo-visualization can display distributions of geo-referenced events. Users can annotate patterns. Web-based component running in a browser
- REQ-POR-030: Access to data storage for visualization. Define an API access data for visualization and further processing. Data source is accessible and the data can be retrieved quickly in the format and resolution required for visual exploration.
- REQ-POR-031: Visualize 3D spatial data. Visualize 3D spatial variables, both scalar and vector valued functions in a moving time frame (with videos), cross sections or in innovative ways efficiently, be able to visualize from each module of the workflow

- REQ-POR-032: Requirements of the visualization of intermediate process data are specified. The requirements for the visualization of intermediate process data needs to be specified based on the needs of implementation partners
- REQ-POR-033: Implementation of a visualization system for data of intermediate workflow steps. The visualizations are readily available for the respective partners
- REQ-POR-034: Visual guidance component. Unobtrusive process-visualization that displays predicted steps of an analytical workflow. A workflow concludes with a simple preview of the result. The user can view and select suggested analytical steps. The user is guided through the workflow towards the desired analytical result.

2.3 Pilots requirements for visualization

In Task 4.4 of the HiDALGO2 project, a High-Performance CFD Data visualization and statistics generation tool will be developed, accessible via the project's portal interface. This tool will support all pilots within the HiDALGO2 portal, especially the current use cases. The task leader, SZE, developed the basic solver for this visualization, known as CFDR (CFD Rendering) software, by M12. Initially, CFDR supported an efficient internal file format. During the project's initial planning stages, SZE conducted a survey to gather various pilot requirements, which were analysed and discussed in project meetings. Based on this analysis, the development goals were formulated.

Below is a summary of the survey results and the main development goals (conclusions).

I. Data Dimensionality, Data Type

- UAP: 1D, 2D, 3D CFD Urban Air Pollution
- UBM: 0D, 3D CFD, Heat Transfer
- RES: 3D CFD
- WF: 3D CFD, Wildfires
- MTW: 3D CFD, particle data, heat and concentration data
- Implementation Goal: Support for 1D, 2D and 3D CFD data.

II. File Format & Structure

- UAP: ExodusII, SEACAS, netCDF, VTK, Ensight Gold, CSV, binary.
- UBM: CSV, Ensight Gold, HDF5, VTK
- RES: NetCDF
- WF: NetCDF, GRIB format, FGA, TIFF

- MTW: VTK, binary
- Implementation Goal: Support for NetCDF, CSV, Ensight Gold

III. Visualization type

- UAP: 3D volume, surface field data of scalars, vectors, streamlines, geometry, volumetric field, 2D line network, 2D points, labels, time and space dependency.
- UBM: Heat maps, isosurfaces, slices, streamlines, vector fields.
- RES: All of the things mentioned above.
- WF: 3D Vector fields, 3D scalar fields visualised using slicing planes, volumetric clouds or iso-surfaces. 3D Point clouds. geometry, large amounts of polygons, such as tree stands. Raymarching 3D particle systems
- MTW: 3D Vector fields, 3D scalar fields, 3D particle systems
- Implementation Goal: Support for general geometry visualisation, slices, 3D volumetric data, vector fields, streamlines, points in 3D space with labels attached to them.

IV. Size of unprocessed data ("one task"), Compression

- UAP: ASCII or Binary data from 1kb to 10TB.
- UBM: from KB (CSV) to tens of GB (Ensight Gold) depending on the simulation.
- RES: Tens of MB per timestep.
- WF: Hundreds of MB. Data is uncompressed.
- MTW: uncompressed data varies from several GB to several TB depending on domain size and number of time steps
- Implementation Goal: Apply some form of generic compression (LZ4) on the processed data by CFDR to significantly reduce the size of the output compared to the original data.

V. Visualization "task" count

- UAP: ~10 000
- RES: 48h-72h, visualize snapshots in 1h interval.
- WF: 180 to 360 minutes with 1 to 10 minute time steps.
- MTW: a full simulation run can contain several million time steps
- Implementation Goal: The UI for time stepping should be able to handle a lot of timesteps. All the timesteps for a dataset cannot be immediately loaded in as one chunk, since the data would be significantly way too large to download reasonably in one go.

VI. "Are there any time steps, if so, what's the time duration?"

- UAP: Yes, parameterized by timesteps. 1 hour to 1 year timespan.
- UBM: Yes, from minutes to years.
- RES: Yes, 48-72 hours.
- WF: Yes, 1 minute.

- MTW: in case of explicit solver (LBM) time steps are short (seconds or less) in case of implicit solver (e.g. multigrid) time steps can be much larger
- Implementation Goal: The handling of the UI for time stepping should be very flexible, since the timestamp ranges can vary from seconds to years.

VII. "Does your data have any statistical distribution? Is it compressible?"

- UAP: Raw binary data, uncompressed. Lots of zero entries at times, potential for compression
- RES: No, but not sure.
- UBM: 3D: planning for compression, 0D: experimenting with compression currently.
- WF: Strong spatial topology associated. Doesn't know if data is compressible.
- MTW: raw binary data is uncompressed, there is potential for compression
- Implementation Goal: SZE shall explore further how to optimally compress data from all the different pilots (not by a case-by-case basis, but generically).

2.3.1 Urban Air Project (UAP)

As seen from the summary of Section 2.2, support of several input file formats are requested, because UAP has 3 different CFD solvers, naturally operated with different formats. The UAP solvers need visualizations for the urban air and related physical variables, among others:

i. **Scalar variables**, like pressure, energy, temperature, concentrations, wind speed.

ii. **Vector variables**, like wind velocity.

2.3.2 Urban Building Model (UBM)

The goal of the urban building pilot is to automatically generate a report on the city's energy performance based on the simulation data produced by the models. In addition to the report, two types of visualizations are created:

1. **Scientific visualization**, of the city 3D mesh and the simulated fields
   - Integration of building meshes (geometry)
   - Integration of scalar fields
     - Solar shading coefficients
     - Ambient temperature
     - Interior temperature of buildings
     - Temperature of wall surfaces
     - Concentration of CO_2 and NO_X
     - Comfort
   - Integration of vector fields

- Heat flux
- Concentration flux

2. **Information visualization**, which essentially involves displaying statistics overlaid on the 3D mesh of the city to analyse and understand their spatial and temporal evolution.

2.3.3 Renewable Energy Sources (RES)

The visualization in RES pilot is focused on **unsteady surface** and **volumetric weather**-related fields being an output of two forecasting applications executed in the backend of the RES framework: WRF and EULAG. The results of both are saved in NetCDF format. Although the output files contain numerous variables, the ones being the most relevant for renewable energy sources pilot are:

- Wind velocity (3D vector field)
- Pressure (3D scalar)
- Temperature (3D scalar)
- Precipitation (2D scalar)
- Immersed boundaries (3D scalar - EULAG only)

2.3.4 Wildfires (WF)

The goal of visualization in the wildfires pilot is to achieve a **realistic** representation of the **fire front** and **smoke** production and dispersion over the landscape. To achieve this, the following is necessary:

1. Volumetric, animated, and realistic representation of smoke and flames.
2. Integration of GIS layers of geographic information, such as roads, cadastre, contour lines, other infrastructures, etc.
3. Integration of 3D landscape, including topography, buildings, trees, and other components.
4. Integration of vector fields, particularly wind, with two objectives:
    a. To intuitively and realistically show air movement, using flying vectors or tracers.
    b. To use vector fields to govern tracer simulation through particle systems, specifically for simulating the movement of flying embers around structures and houses.
5. Integration of other supporting elements to enhance understanding of the landscape and risks posed by wildfires.
6. Development of tools and methods for immersive navigation in the virtual scenario.
7. Development of tools and methods for interaction with menu options and objects in the virtual scene.
8. Regarding the final representation, there are two lines of requirements:

a. Development of visual analysis and interaction experiences on desktop computer screens.
b. Development of immersive VR experiences. While the former requires fewer graphics computing resources, the VR experiences designed must ensure smooth interaction, visualization, and navigation, with frame rates not less than 70 fps.VR experiences can be PC-VR, meaning with VR display devices tethered to a desktop computer.

2.3.5 Material in Water (MTW)

Similar to UAP and following the summary of Section 2.2, MTW involves different CFD data related to physical variables. In addition to that also particle data is required.

1. **Scalar variables**, like pressure, temperature, concentrations,
2. **Vector variables**, like fluid velocity,
3. **Particle lists**, e.g. containing particle properties, positions, velocities.

3 Technology Solutions

As part of the development of the HiDALGO2 activities, some visualization technological solutions that best suit the requirements of each of the pilots have been explored, developed, and implemented. Although these technologies have a marked cross-cutting character, they can be applied in a more generic way. Given that in their selection, design, and development, it has been sought for these solutions to use formats and methods that are universally used, their use in other applications and thematic domains is ensured, as well as their future expansion with new functionalities. These technological solutions are briefly explained below.

3.1 CFDR WEB Visualization

3.1.1 Overview

The Computer Fluid Dynamics Rendering (CFDR) is a visualization system which architecture is summarized in Figure 1.

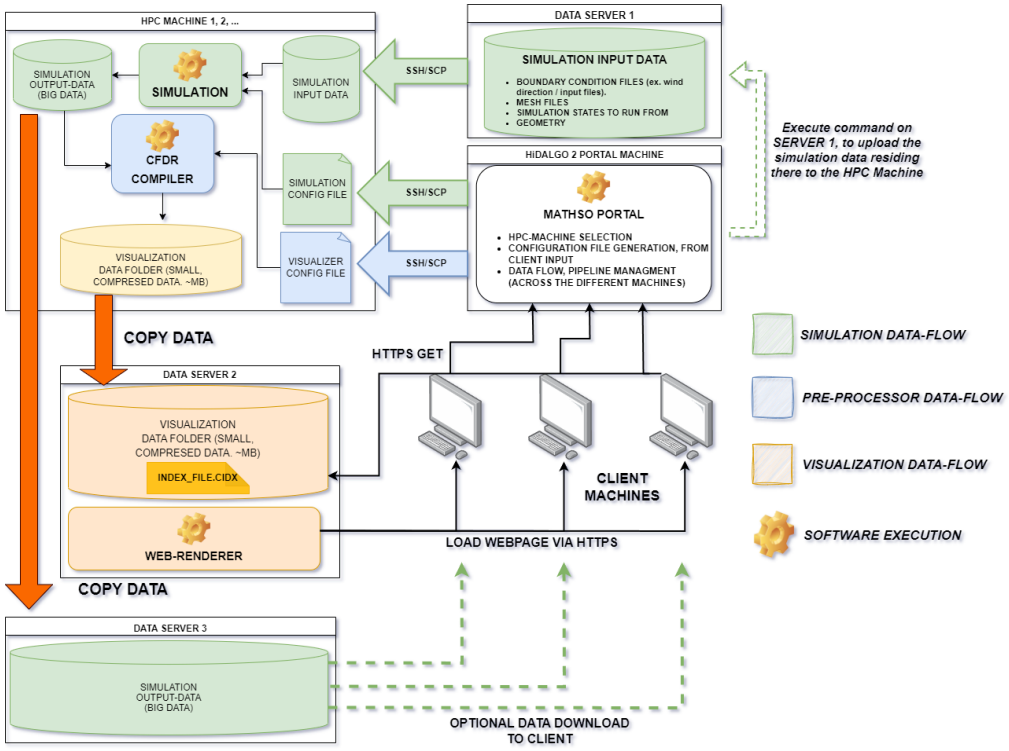

Figure 1. Scheme of the CFDR architecture and data flow

In a nutshell, CFDR consist of two major components:

1. **CFDR Compile**, which runs on the HPC machines where the simulation is running and extracts the data from the simulation raw data (tens of terabytes) for the visualization configured in the configuration files and compress them (resulting data is of tens of megabytes),
2. **CFDR Web Renderer**, which processes the extracted data file on another computing machine and provides data for the clients web browser.

The associated infrastructure scheme is as follows:

1. Dedicated HiDALGO2 machines
    a. HiDALGO2 portal
    b. Data server
2. HPC machines
    a. EuroHPC
    b. Local clusters (for development)
    c. Clients, users local machines

For specific use cases, the infrastructure has to be defined and applied.

3.1.2 The CFDR Compile component

In this subsection the core of the CFDR visualization, the CFDR Compile is introduced.

Execution

The CFDR compiler can be executed using the following syntax:

```
./cfdr_compile [LUA CONFIGURATION FILE]
```

CFDR uses Lua as its scripting language (~configuration files), since it allows for flexibility in numerical expressions, and ways to express the data in multiple ways.

Scripting API

Before any calls are made to the CFDR api, **the API must be initialized first**.
In order to do so, the following function call must be made:

```
cfdr_workspace(path, title)
```

Where:

- path is a **string** representing an **absolute or relative** path to a directory. Cannot exceed 512 characters.
- title is a **string**. Cannot exceed 512 characters.

The call to this function accomplishes multiple things. It defines what title to use for the visualization, and more importantly thanks to the directory path provided, CFDR will create the appropriate directory, and create all the appropriate files and directory hierarchy for the visualization to work. **It is very important not to modify or move these generated files** and to keep the directory clean of other files, since any modification of these will fundamentally break the way the visualization tool works. Fundamentally, any data generated by CFDR will adhere to the following directory structure:

```
workspace
    | index.cidx
    | geo
    | sli
    | ...
```

It is relevant to emphasize the existence of the file **index.cidx** here. This is the so-called **index file** and its purpose is to map out the entire workspace directory, by pointing to the location of all resources residing in the folder, all starting with a three letter acronym (geo, sli, str, …). It also stores some (relatively) lightweight information like the title, or **POI** (Point of Interest) data.

PUSH API

Once the CFDR API has been initialized, the user can finally start providing some of their own data to be processed. Each type of data handed in by the user is characterized by a 3 letter acronym, for example: **GEO** (= Geometry), **SLI** (= Slice), **STR** (= Streamline), **VOL** = (Volumetric), **POI** = (Point Of Interest). If the user would like to submit data to be visualized, they can use the Push API of CFDR:

```
cfdr_push(data, type)
```

Where:

- data is a **table,** containing the appropriate entries (as detailed below)
- type is a **string**. This is the aforementioned 3 letter acronym, describing how to interpret the data table. It has to have one of the following values:

Type Value	Description
GEO	Geometry
POI	Point Of Interest

CFDR's push API is used for submitting ready-to-go data; that is, the data has already been processed, and no heavy computations are needed. If the user would like to do computationally heavy processing, for example, generating slices, streamlines or volumetric representation of CFD results, please refer to the other sections.

In the following sections, we provide specific details on how to submit data via the push API, for all types.

GEO – Geometry API

For submitting geometry. The data table must be of the following form:

```
geo_table_example = {
{ id = [string],
        visible = [boolean],
        path = [string],
        r = [double],
        g = [double],
        b = [double],
        a = [double]
},
{ id = [string],
:
},
:
}
```

Essentially, the data table is comprised of multiple **sub-tables**, where the following entries can be assigned value:

Table 1. Geometry API data sub-tables entries

Table Entry	Description
id [string]	identifier (name) for the entry. This is how the element will be referred to as in the visualization. Must be unique
visible [boolean]	Specifies whether the object is hidden or visible by default
path [string]	Path to the geometry file to use . Can be one of the following formats: **OBJ, STL**
r [double]	red channel for colouring
g [double]	green channel for colouring

b [double]	blue channel for colouring
a [double]	alpha channel for colouring

As mentioned in the overview section, after declaring the data table, it must be submitted via the following API call:

```
cfdr_push(geo_table_example, "GEO")
```

Of course, multiple calls to the push api for different tables to push is totally valid and encouraged, to make the script files cleaner:

```
cfdr_push(geo_table_example_1, "GEO")
cfdr_push(geo_table_example_2, "GEO")
```

POI – Point of Interest API

For submitting Points Of Interests. These are essentially probe points, with a label attached to them. They're useful for identifying different sections of simulations, or the geometries. The data table must be of the following form:

```
poi = {
   { id = [string],
         visible = [boolean],
         x = [double],
         y = [double],
         z = [double],
         r = [double]
   },
   { id = [string],
   :
   },
   :
   }
```

The data table is comprised of multiple **sub-tables**, where the following entries can be assigned value:

Table 2. Points of interest API data sub-tables entries

Table Entry	Description
id	identifier (name) for the entry. This is how the element will be referred to as in the visualization. Must be unique.
visible	Specifies whether the object is hidden or visible by default.
x	x coordinate of the probe.
y	y coordinate of the probe.
z	z coordinate of the probe.
r	radius to use for the probe (every probe is represented as a sphere).

SLI – Slice API

For **submitting** slice data. Slices are described as surface meshes, with associated variables and timestamps. For each variable and time-stamp, there is a datafile, containing a vector of data associated with each vertex of the mesh.

The formats describing these are detailed thoroughly in the format annex section. The data table must be of the following form:

```
slice = {
{ id = [string],
        preload_series = [boolean],
        visible = [boolean],
        path = [string]
},
{ id = [string],
:
},
:
}
```

The data table is comprised of multiple **sub-tables**, where the following entries can be assigned value:

Table 3. Slice submitting API data sub-tables entries

Table Entry	Description
id	Identifier (name) for the entry. This is how the element will be

	referred to as in the visualization. Must be unique.
visible	Specifies whether the object is hidden or visible by default.
preload_series	Pre-load time series when slice is active in visualizer
path	Path to a directory containing a **slice.lua** file. The format for this file is documented in Annex A.

For **generating** slice data. Slices are described as surface meshes, with associated variables and timestamps. For each variable and time-stamp, there is a datafile, containing a vector of data associated with each vertex of the mesh.

The formats describing these are detailed thoroughly in the format annex section. The data table must be of the following form:

```
generate_slice = {
{
        base_id = [string],
        mesh    = [string],
        meta    = [string],
        sampler_list = {
        [[string]]  = [string],
:
},

        variable_list = {
```

The data table is comprised of multiple **sub-tables**, where the following entries can be assigned value:

Table 4. Slice generating API data sub-tables entries

Table Entry	Description
base_id	base identifier name to user. For each sampler provided, base_id + "_" + sampler_id will be the generated identifier. (refer to sampler_list, for sampler_id)
mesh	Path to a .CMSH file (see Annex A)
meta	Path to a .CMET file

	(see Annex A)
sampler_list	A table, where each entry must be as follows: [sampler_id] = sampler_path, Where sampler_id is a string id to assign for the sampler, and sampler_path is a string path to an .OBJ file (see Annex A)
variable_list	A table, where each entry must be as follows: [variable_name] = variable_path. Where variable_name is a string describing the name of the variable, and variable_path is a path to a .CDAT file (see Annex A)

GEN – Generate API

While the PUSH API is meant to submit already processed data to CFDR, the GEN API's goal is to generate data (i.e. slices, streamlines, volumetric data). Before using this API, the user must convert their data into one of CFDR's formats. In order to do so, please refer to the ***CFDR Converter*** section of the document.

The overall usage of the API is exactly the same as the PUSH API. This time, the user must call the follow function, with the appropriate table and string:

```
cfdr_push(data, type)
```

Where:

- data is a **table,** containing the appropriate entries (as detailed below).
- type is a **string**. A 3 letter acronym, describing how to interpret the data table. It has to have one of the following values:

Type Value	Description
SLI	Slice Generation

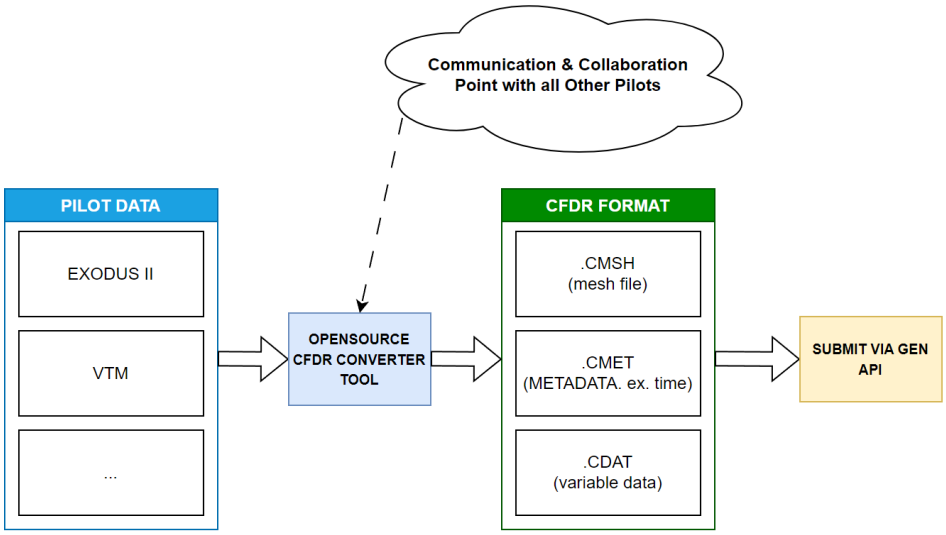

Figure 2. Data conversion and use through GEN API into CFDR

3.1.3 Workflow on a local machine

The goal of this section is to present the average workflow of using the CFDR toolkit, in order to help the user visualize their own data. We start with a relatively simple example with only geometry processing and no simulation data. The average run-workflow can be represented with the following diagram.

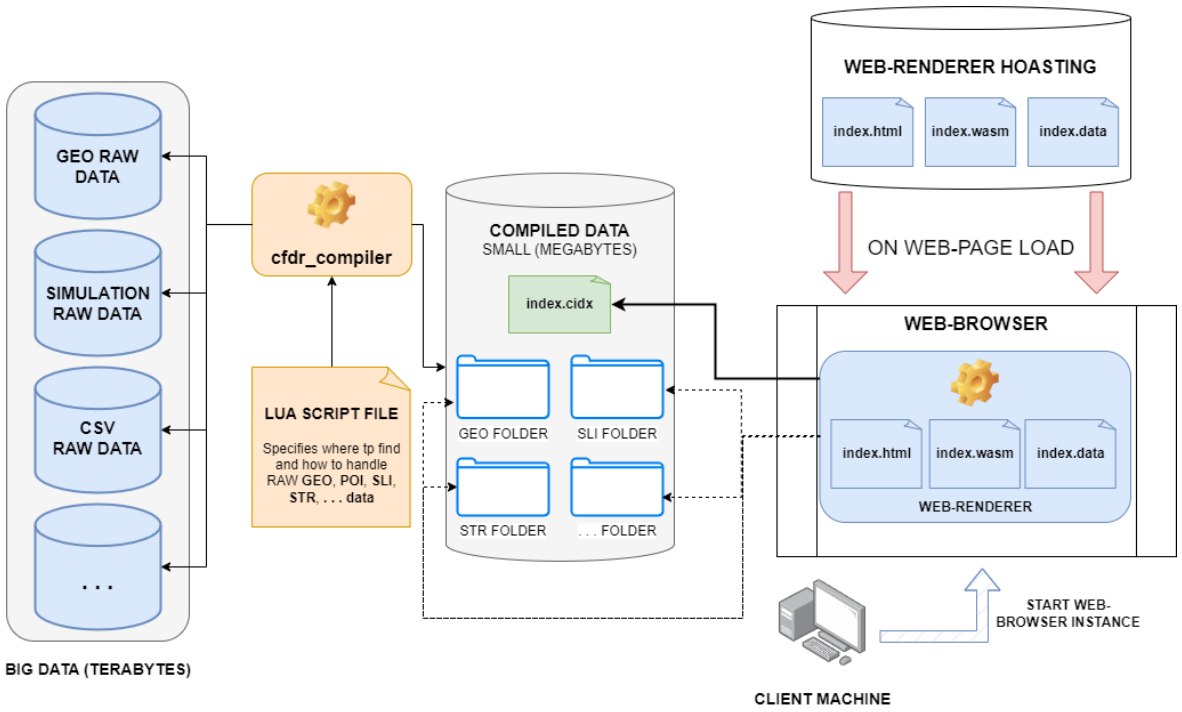

Figure 3. General scheme of the workflow for the use of CFDR

I. A **Lua Script File** must be written, as described in the CFDR Scripting section, that will be used as the input for the CFDR compiler. We provide an example of such a script below, that we ran on a local windows machine. Of course, the same applies to any other operating system like windows; the user just has to make sure to set the appropriate paths pointing to their data. We would also like that, while relative paths are totally fine and accepted by the system, we urge the user to use absolute paths instead so things are less likely to break overtime:

~/CFD/CFDR/gyor.lua

```
- First, we declare a workspace directory, with a title. This is where the data will be packaged.
cfdr_workspace("~/CFD/CFDR/gyor", "Gyor3b - 17/09/2023")

-- Declare a table containing all the 3D models to use.
city = {
   { id = "buildings",     visible = true,   path = "~/CFD/CFDR/data/geo/buildings.obj",  r = 0.9, g = 0.9, b = 0.9, a = 1.0 },
   { id = "forest",                visible = true,   path = "~/CFD/CFDR/data/geo/forest.obj",     r = 0.3, g = 0.8, b = 0.3, a = 0.6 },
   { id = "ground",                visible = true,   path = "~/CFD/CFDR/data/geo/ground.obj",     r = 0.4, g = 0.4, b = 0.4, a = 1.0 },
   { id = "paths",                 visible = true,   path = "~/CFD/CFDR/data/geo/paths.obj",      r = 0.8, g = 0.7, b = 0.2, a = 1.0 },
   { id = "railways",              visible = true,   path = "~/CFD/CFDR/data/geo/railways.obj",   r = 0.4, g = 0.4, b = 0.4, a = 1.0 },
   { id = "roads",                 visible = true,   path = "~/CFD/CFDR/data/geo/roads.obj",      r = 0.6, g = 0.6, b = 0.6, a = 1.0 },
   { id = "vegetation",   visible = true,   path = "~/CFD/CFDR/data/geo/vegetation.obj", r = 0.1, g = 0.9, b = 0.1, a = 0.3 },
   { id = "water",                 visible = true,   path = "~/CFD/CFDR/data/geo/water.obj",      r = 0.3, g = 0.3, b = 0.8, a = 0.6 },
}

-- Add geometry.
cfdr_push(city, "GEO")
```

```
-- Declare a table containing all points of interest.
poi = {
   { id = "Probe 1", visible = true, x = 1000.0, y = 2000.0, z = 50.0,  r = 25.0 },
   { id = "Probe 2", visible = true, x = 0.0,    y = 0.0,    z = 50.0,  r = 25.0 },
}

-- Add points of interest.
cfdr_push(poi, "POI")
```

II. The **CFDR compiler** must be called in order to generate the compiled workspace directory.

./cfdr_compile ~/CFD/CFDR/gyor.lua

III. Once the data has been generated in the specified folder (in our case *~/CFD/CFDR/gyor*), we must make this folder accessible via HTTP GET on the web. In order to so, we can either open up an http port on a server and share the file publicly, or we can do the following to test things on a local machine:

A. Move the generated folder, *~/CFD/CFDR/gyor* in our case, into the CFDR web renderer's hosting directory (The directory where index.html, index.wasm and index.data are present).
B. Issue the following python command to emulate a http server on our local machine:

python -m http.server 8080

IV. Once the data is available via the web (either by HTTP server emulation on the local machine like specified prior, or traditionally), we should be able to visualize the data, by opening up the web-renderer Web page, and specifying with the following syntax where our data folder is.

[URL TO THE WEB-RENDERER]/#?location=[URL TO THE GENERATED GYOR FOLDER]

If the python method was used to open up the http server, we can simply visualize our data by copy-pasting the following URL into a web-browser:

http://localhost:8080/#?location=gyor

3.2 Vistle Covise - CAVE

COVISE (**Co**llaborative **Vi**sualization and **S**imulation **E**nvironment) is an open-source extendible distributed software environment integrating post-processing and

visualization functionalities focusing on collaborative and interactive use in VR environments.

In COVISE, a visualization is created by assembling a **visualization pipeline**, which is divided into **modules**. These modules organize the complex data and control flow into smaller, manageable parts. Figure 4 illustrates a typical visualization pipeline, which includes a reader module, a geometry cutting module, and modules for creating cutting surfaces, isosurfaces, and streamlines, along with a renderer module. The specific behaviour of these modules can be defined through settings in the visualization pipeline or adjusted interactively during rendering.

Modules can be distributed across various heterogeneous machine platforms, particularly utilizing HPC infrastructure. COVISE's rendering modules support diverse VR environments, including powerwalls, curved screens, head-mounted displays, full domes, and CAVEs. This capability allows users to interactively explore data within fully immersive, interactive and collaborative spaces.

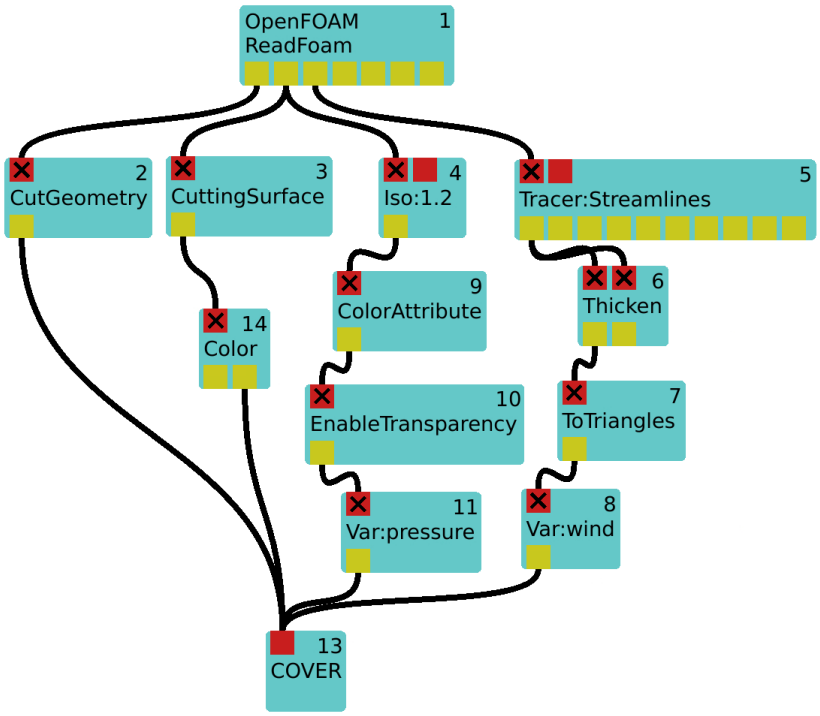

Figure 4. Example of a visualization pipeline in Vistle

Figure 5 shows a visualization output from COVISE or Vistle, which can be used in various VR environments and standard 2D displays. This includes the integration of multiple heterogeneous environments, enabling collaborative exploration of the same

visualizations without requiring physical proximity. The rendering modules can be extended with plug-ins, allowing for specialized functionality tailored to different use cases.

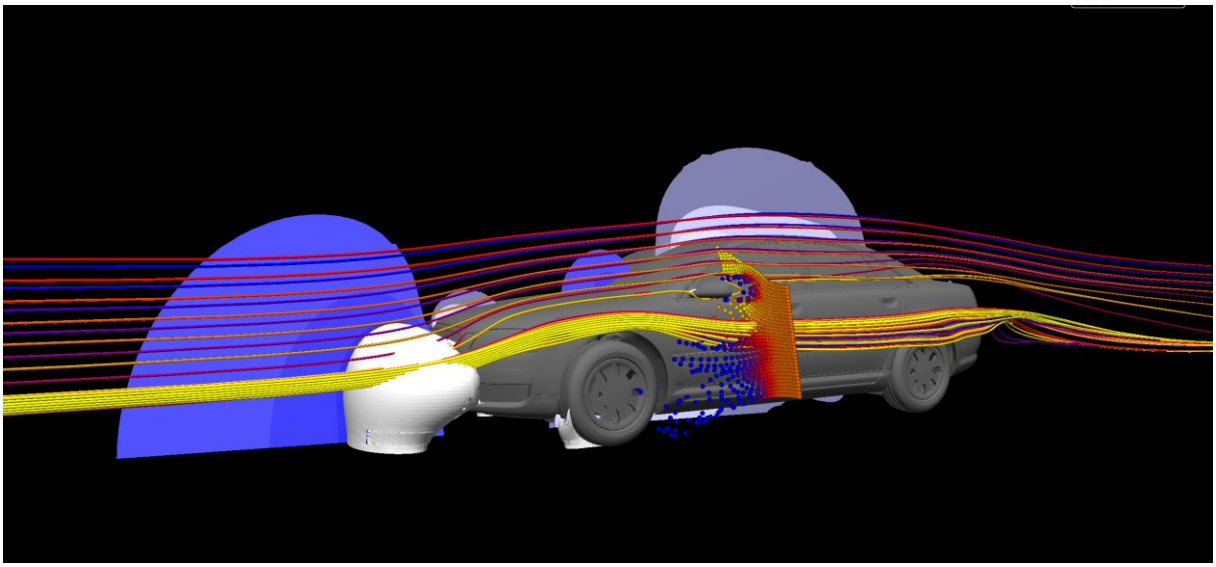

Figure 5. Example of a visualization generated using Vistle

Vistle, the successor to COVISE, introduces module parallelization, enabling the creation of interactive visualizations for larger datasets. Its features include remote rendering and an interface for in-situ visualization of simulations. The five-sided CAVE at HLRS in Stuttgart facilitates collaborative data exploration and visualization in an immersive VR environment, as shown in Figure 6.

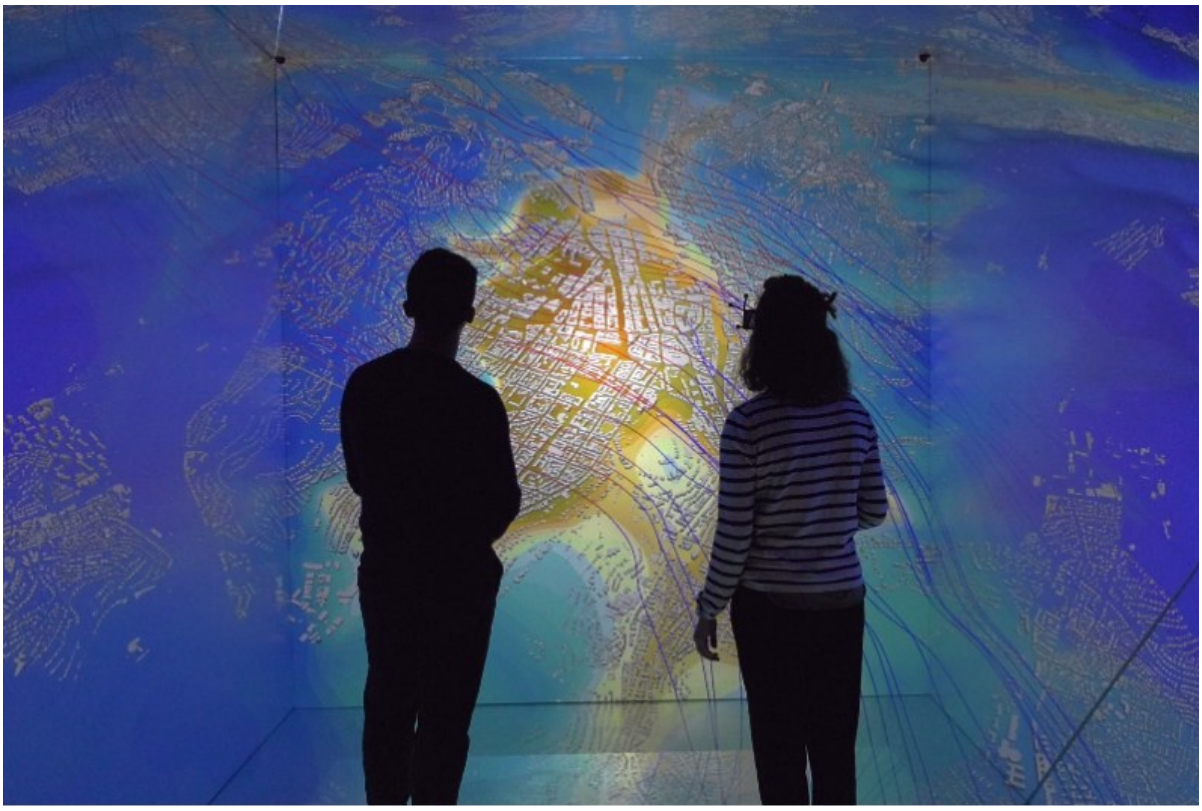

Figure 6. Urban digital twin of Stuttgart city displayed in a CAVE environment at HLRS

3.3 Unreal Engine Advanced Visualization (UEAV)

Unreal Engine (UE)

Unreal **E**ngine (https://www.unrealengine.com/) is a widely used digital game development platform in the publishing of some of the most significant titles in this industry. Additionally, UE has been adopted by other domains of technical and scientific development as a powerful visualization tool, such as in the fields of architecture, construction, smart buildings, creation of digital twins, or territorial planning among others [1]. Furthermore, UE has been used as a solution for the development of serious games with broad implementation in the military or emergency management and civil protection fields.

UE is built modularly, including a powerful volumetric lighting engine and other sub-engines with specific functionalities, such as particle dynamics (Niagara), procedural landscape and vegetation creation (Foliage), world partitioning, AI, destruction (Chaos), physics engine, character animation, facial animation (MetaHuman), hair and clothing dynamics, and volumetric special effects (exponential fog), among others.

The UE paradigm is constructed from scenes where assets (also generically called actors) are placed. These assets can be any elements described within the engine, such as objects, materials, particle systems, light sources, animated characters, cinematics, colliders, interaction regions, etc. These scenes can utilize local or global variables that govern game mechanics and the behaviour of actors in the scene. UE allows for programming in C++ or, alternatively, through a visual scripting language called Blueprints (BP) that enables developers to conceptualize and design game functionalities and actor behaviours more intuitively and quickly. Programming these BP is the cornerstone that allows UE experiences to achieve a high degree of interactivity and dynamism.

Another interesting aspect of UE is that any actor deployed in scenes can have associated functionality, thanks to BP programming. For example, a Blueprint can include a geometry component of an object (what we see in the scene), a camera component (which acts as the user's eyes), and other components that enable interaction with other objects, the user, or factors or processes occurring in the scene. This feature makes UE a very compelling platform for developing digital twins and complex simulation scenarios with nested or interrelated models.

UE is designed for the development of digital games, which implies a wide range of optimization strategies for real-time visualization without compromising the final result's realism. One of the most relevant technologies is Nanite, an algorithm that pre-processes the complex geometry of objects with thousands of facets, such as a high-resolution digital terrain model or a forest of trees. Nanite applies a sophisticated system of culling, compression, and data streaming, allowing highly complex objects to be rendered in real time. Additionally, UE features an advanced lighting engine (Lumen) that incorporates common real-world effects like reflections, refractions, radiosity, and volumetric effects (fog, light shafts, etc.). The final scenes are compiled as executables for different operating systems (Windows, iOS, Android, etc.) and platforms (PC screen, VR, XR, etc.), making it ideal for multi-platform development. Furthermore, UE provides the source code, allowing for modifications, compilations, or the development of plug-ins as needed for each case, significantly enhancing its flexibility.

MeteoGrid has developed specific methods for Unreal Engine to enhance the functionalities necessary for the visualization methodology in the wildfire pilot. One of these tools is the ability to import GIS layers in SHP format (ESRI/ArcGIS) and render them as three-dimensional curves in the scene [1]. Additionally, it is possible to use

each segment of each line as independent objects, allowing interaction and functionality through BP. Since these objects can be converted to Static Meshes, it is also beneficial to apply Nanite to the resulting geometry (consisting of thousands of polygons) to improve performance.

As can be seen, this methodology is perfectly applicable in a generic manner to other pilots, particularly those resulting in scalar volumetric fields and vector volumetric fields. The joint application of wildfire propagation visualization and atmospheric pollution modelling in cities (UAP) is especially interesting, as it involves working conceptually in similar temporal and spatial spaces.

Interchange Data Formats

For the forest fire pilot in HiDALGO2 project, the use of Unreal Engine has been proposed as an integrating platform for scenarios and visual simulation of forest fires at various scales. This is possible because UE natively implements reading and writing of some of the most commonly used exchange formats in fire simulation and CFD, such as FGA, VDB, or more generally ASCII and RAW binary formats. The way to integrate the results of the numerical simulation of the fire-atmosphere interaction and the production and dispersion of smoke, both at the landscape scale (WRF-SFIRE) and the local urbanization scale (OpenFOAM-fireFOAM), is summarized in Figure 7.

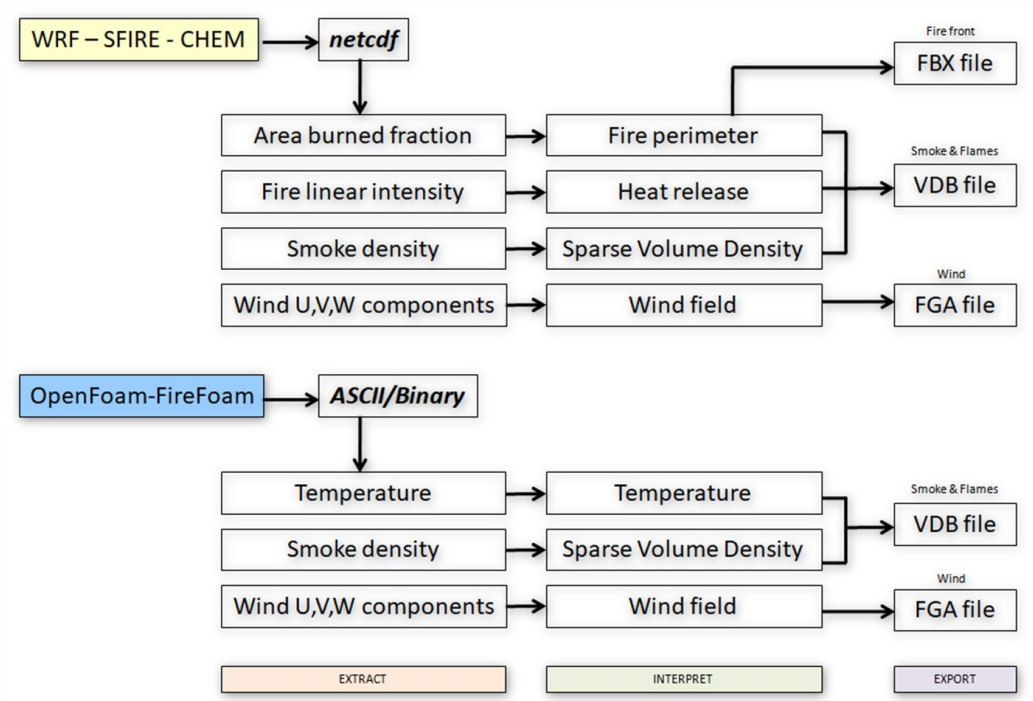

Figure 7. Scheme for utilizing standard exchange formats and workflow from HPC simulations, highlighting the three implementation phases: extraction, interpretation, and exportation. As can be observed, the majority of the results are represented as vector fields and scalar fields

In the development of the visualization solution using UE, two exchange formats are specifically utilized: the **FGA** format, for incorporating volumetric vector fields, and the **VDB** format, for integrating scattered volumetric scalar fields, such as smoke plume simulations (see details in Annex 1).

The **FGA** files are integrated into UE as data structures (*Struct*) or directly as global vector fields, as UE natively reads these files. There is a limitation of 128 cells on each axis of the calculation domain, but UE allows coupling several of these global vector fields, filling the desired space. Specifically, one of the developments tailored for the forest fire pilot in HiDALGO2 is the bridge software that allows sampling the scalar fields resulting from WRF-SFIRE simulations (*netcdf* format) or OpenFOAM-fireFOAM simulations (RAW binary format) and converting them into VDB sparse data files. This solution is already present in some of the simplified CFD packages used in digital games, such as Houdini Pyro (SideFX), Embergen (JangaFX), or Niagara Fluids (Epic Games), all of which have been conveniently explored and utilized in this pilot as a simpler alternative for volumetric visual generation of smoke and flames.

Sparse volumetric data refers to data that does not fill the entire three-dimensional space; instead, only some voxels contain data, as is the case with smoke density in forest fire propagation simulations. The **VDB** format provides an efficient hierarchical approach based on trees (such as octree or B+Tree), storing only the voxels with data, called leaf nodes [2]. Additionally, for the development of applications using this format, OpenVDB is a C++ library for reading and writing in VDB format, with a Python version also available.

For the generation of these VDB files that collect data from the numerical simulation (volumetric flames and smoke), multiple channels (scalars) are stored in respective RGBA channels, particularly:

- Smoke Density (R)
- Temperature (G)
- Flames (B)
- Velocity (A)

These data essentially describe the geometry of the flame at each point (length, height, angle with the vertical, width, etc.) and the density and dispersion of the smoke, as well as their variation over time in specific time loops. In the simulations conducted, the temporal sampling performed is at 60 frames per second (60 Hz), where each frame corresponds to a moment on the time axis and is stored in the corresponding volumetric matrices within the VDB files. The default time loops used are 240 frames (about 4 seconds).

UEAV Visualization Workflow

MeteoGrid has proposed a specific workflow to leverage the capabilities of Unreal Engine within the framework of the HiDALGO2 project, particularly for the wildfire pilot, although this workflow can be adapted and used for the other pilots. This workflow, in general, is as follows:

1. Simulation of wildfire propagation, fire-atmosphere interaction, and smoke generation and dispersion across the landscape, on HPC facilities.
2. Interpretation of the results, including the creation of sparse volumetric data VDB files for smoke and flames and FGA for wind vector fields. Additionally, incorporation of simulated local vorticity using multiple nested layers of Perlin noise to enhance visual appearance.
3. Integration of resulting VDB files as sparse volumetric textures, creation of corresponding volumetric materials, and assignment to heterogeneous volumes.

4. Placement of resulting heterogeneous volumes in the scene at exact positions and scales. Animation of smoke and flame evolution loops.
5. Integration of vector fields exported as FGA files, particularly wind vectors with coupled flame front effects. These files are imported into Unreal Engine and interpreted as global vector fields.
6. Generation of particle systems to represent airflow according to the imported vector fields.
7. Incorporation of reference geographical elements as objects (Static Mesh), particularly digital terrain models, GIS layers (roads, cadastral, contours, points of interest, etc.).
8. Alternatively, integration of the CESIUM-Google Tile Service for immediate creation of digital twins with this geometry.
9. Programming and incorporation of other components typical of UE environments, particularly lighting and volumetric effects.
10. Programming of actor functionality, particularly the development of Blueprints associated with simulation objects.
11. User navigation programming through virtual environments, with the corresponding BPs.
12. User interaction programming with scene objects, development of corresponding BPs.
13. Development of experience mechanics, interactive menus, and level mechanics.
14. Development and integration of cinematics, which are pre-calculated animations or time-evolving processes dynamics.
15. Compilation of the experience into highly portable executable packages.

As can be seen, the aim is to encapsulate the outcome of multiple forest fire simulations, resulting from all possible variations in wind intensity and direction, fuel moisture configuration, or ignition point location, into immersive, interactive virtual reality experiences. The integration scheme is summarized in Fig 8.

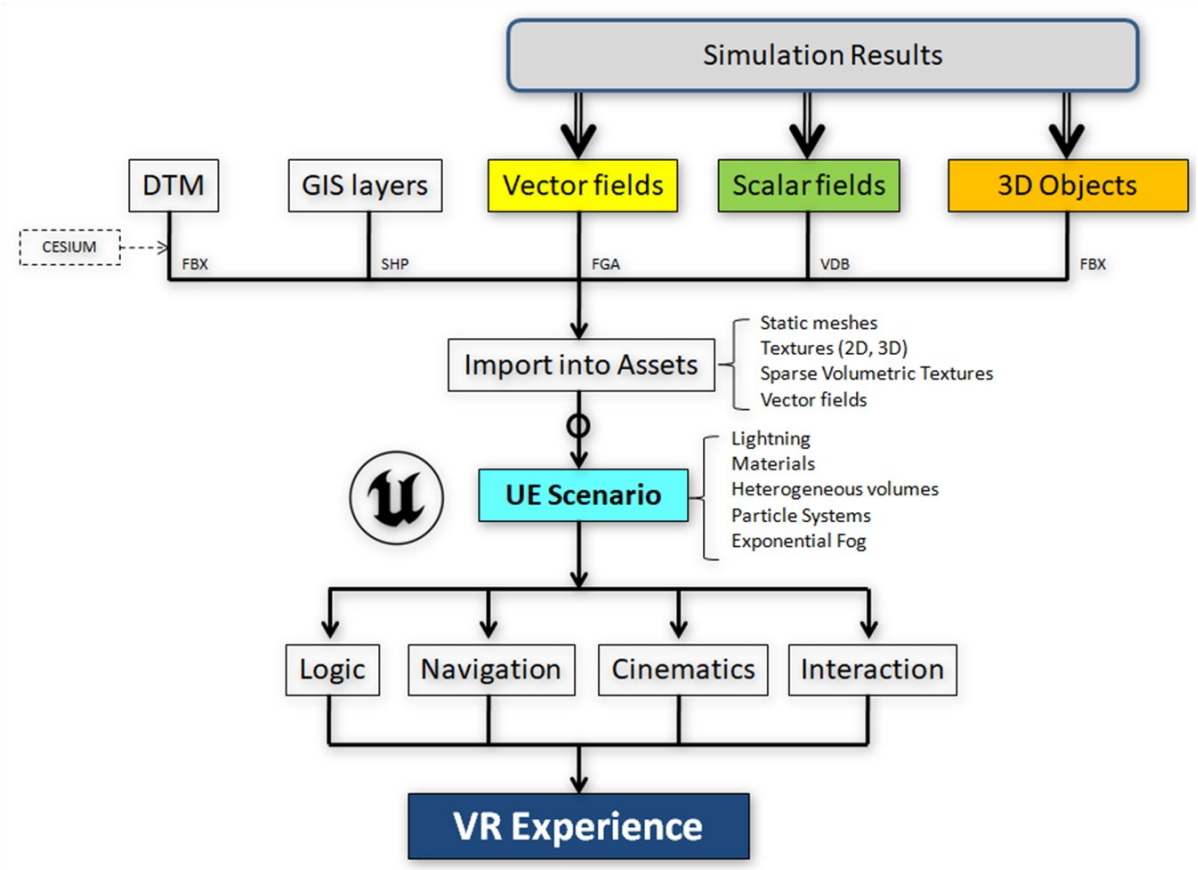

Figure 8. Scheme for integrating exchange files along with other objects into Unreal Engine environments. The digital terrain model (DTM) can be replaced by the CESIUM service, which injects 3D geometry from the Google Tiling Service. All data is interpreted as assets in UE. Experiences include descriptions of gameplay logic, navigation and interaction mechanics, and cinematic sequences for explanation and context. Everything is encapsulated in highly portable executable files (.EXE)

The real contribution in this project is the creation of bridge software that allows for the extraction, interpretation, and exporting of HPC simulations into universal exchange files (ASCII, Binary, FGA, FBX, VDB, etc.) that can subsequently be integrated into experiences in UE. This bridge software systematically samples the arrays of vector and scalar fields resulting from simulation in HPC infrastructure and converts them into animated sequences in the aforementioned exchange formats. Additionally, since the ultimate destination is visual simulation, some visual enhancements are added to the numerical results, such as smoke micro-vorticity or flame shredding, among others. This increases the visual aspect from numerical results of CFD simulations, although it does not increase the information.

As an example, a 6-hour simulation of the propagation of a real fire, performed on HPC facilities like LUMI or VEGA using WRF-SFIRE software, generates a total of 360

frames, corresponding to the state of the flame front and the volumetric dispersion of smoke every minute. In the playback of the simulation in UE, imported as VDB into an animated SVT volumetric texture, 24 fps are used, corresponding to a 15-second loop. This way, the 6-hour simulation is compressed into a VDB file that is visualized in a 15-second loop in the UE experience. Additionally, as many simulations as there are configurations of the boundary conditions can be incorporated.

This approach provides notable flexibility to the proposed method, as it is independent of the CFD models used initially. Indeed, MeteoGrid has bridged this with other CFD solutions, particularly FDS or ANSYS Fluent, widely used in fire protection engineering. In these specific cases, the sampled fields have been vectorial (Fluent) and scalar (FDS), derived from the volumetric matrices generated (particularly in FDS, the results obtained by Plot3D tool).

3.4 Ktirio-GUI visualisation

Ktirio-GUI is a graphical user interface written in Qt (C++) that allows preparing the data for the urban building pilot.

In particular, it allows to:

- Generate the GIS dataset
- Generate the mesh dataset
- Partition the GIS and mesh dataset
- Visualize the mesh
- Upload the dataset to a data management platform
- Submit execution on Euro-HPC or local supercomputers

A workflow based on Ktirio-GUI, that generates the data and submits the HPC job on EuroHPC or local systems, has been developed. The results of the jobs are finally uploaded to a storage facility for post-processing.

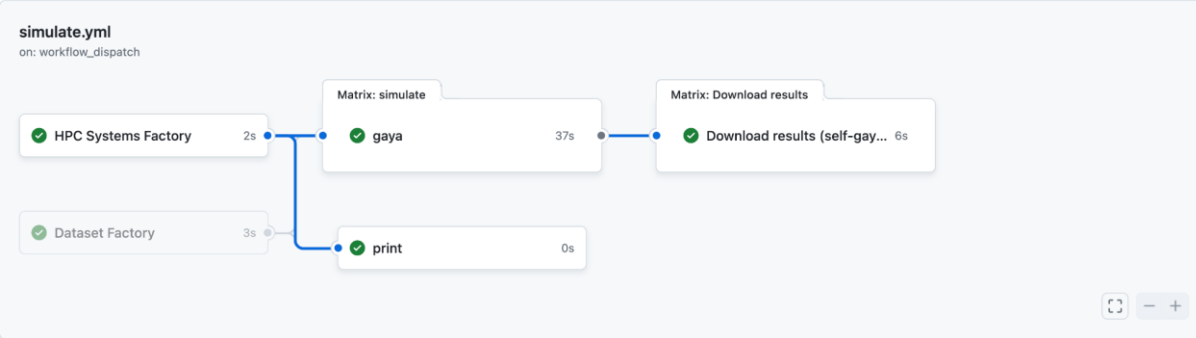

Figure 9. Simulation workflow triggered by Ktirio-GUI

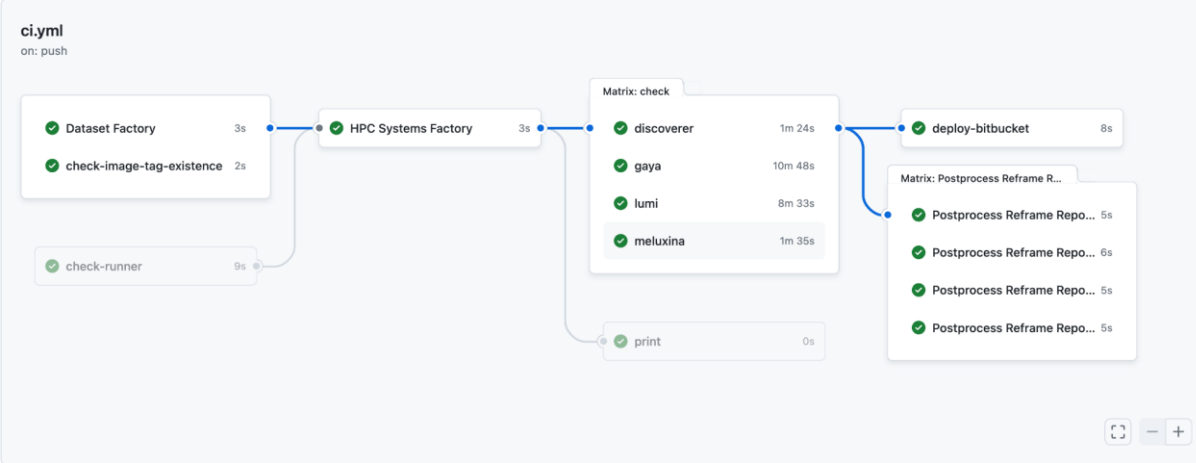

Figure 10. Benchmarking workflow with the upload of the results of the job

4 Application to Pilots

The applications of the aforementioned technologies to visualization in the HiDALGO2 pilots are summarized below.

4.1 Urban Air Project (UAP)

SZE has applied the **CFDR visualizer** for the CFD calculations relative to the city of Győr (Hungary), with the following selection of machines:

1. Development
   a. HPC: **Solyom** (SZE)
   b. Portal: **Solyom** (SZE)
   c. Data server: **Solyom** (SZE)
2. Testing

a. HPC: **Altair** (PSNC), **LUMI**
b. Portal: **Solyom** (SZE) / **PSNC VM** (PSNC)
c. Data server: **Solyom** (SZE) / **PSNC VM** (PSNC)

3. Production
   a. HPC: **LUMI**
   b. Portal: *(TBD)*
   c. Data server: *(TBD)*

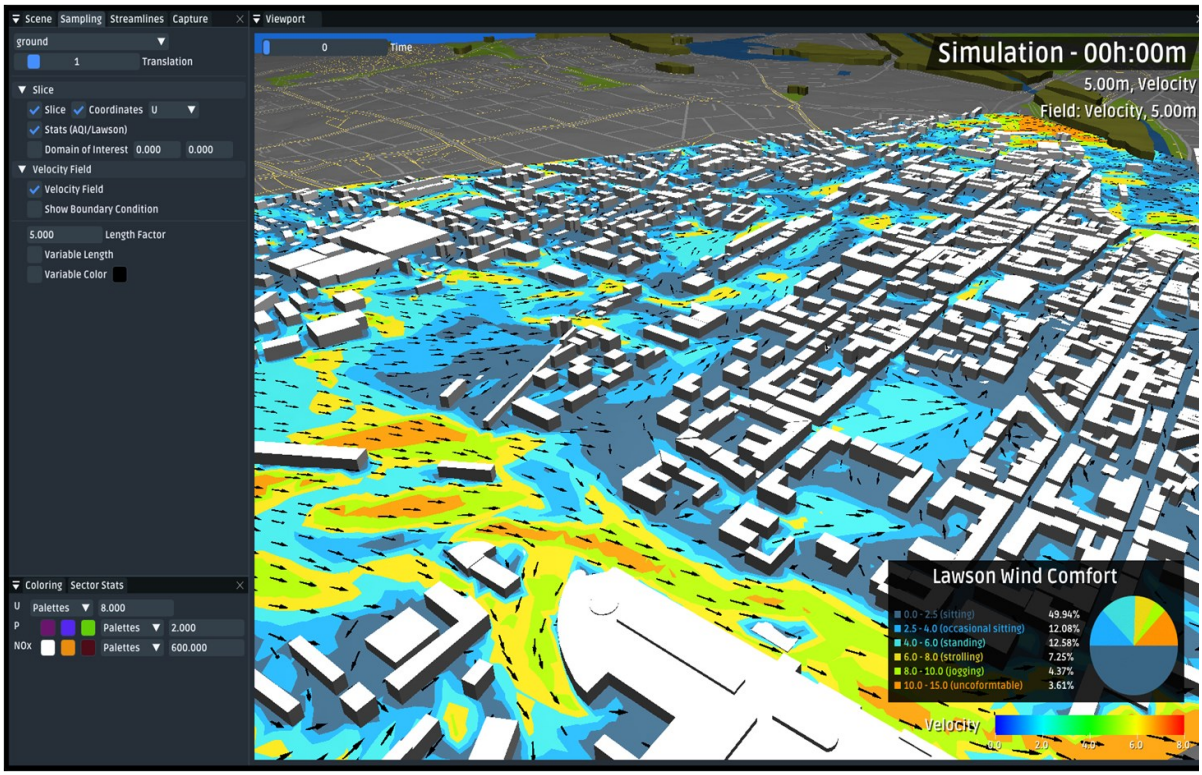

Figure 11. Example of CFDR web-based visualization on HPC and visualization on the web, using the automatically compiled data from simulations, corresponding to the city of Győr, Hungary.

4.2 Urban Building Model (UBM)

UNISTRA has applied the visualisation achieved with **Ktirio-GUI** and ParaView to the city of Strasbourg (France), for the Urban Building Model pilot.

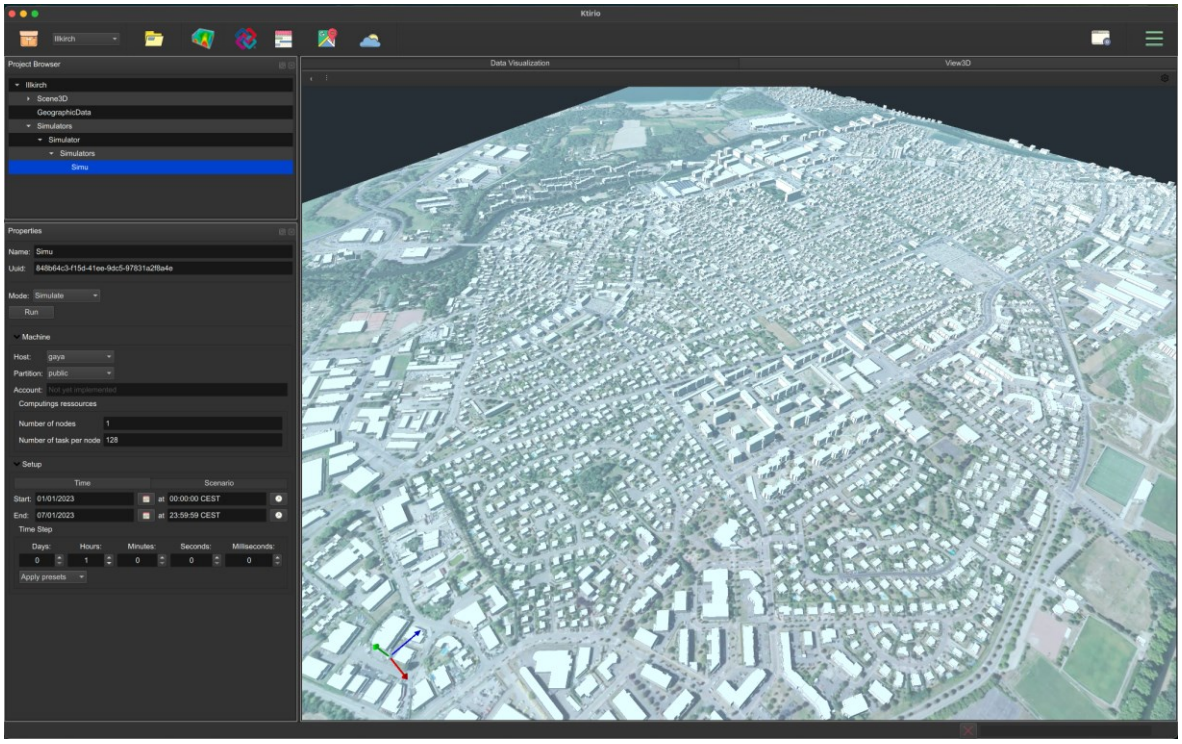

Figure 12. Example of the Ktirio-GUI displaying a city building set over an ortophoto, corresponding to the commune of Illkirch-Graffenstaden, in Strasbourg, France

4.3 Renewable Energy Sources (RES)

The RES framework is equipped with visualization module based on dedicated Python libraries which generates both static PNG images representing each timestep and GIF animations of top-view of the simulated domain(s) for all the relevant variables mentioned in section 2.3.3. Each picture represents a cross section including both the visualized variable and the geometry of buildings modelled with immersed boundary method.

Apart from the built-in module, for a more enhanced images including topography, power lines and more informative legend, QGIS toolkit is used for static images. Three-dimensional animations including possible camera movement are prepared in ParaView software. An example of static image made with QGIS is presented in Figure 13.

In the future it is planned to couple RES toolkit with CFDR and UEAV visualization tools described in sections 3.1 and 3.3 respectively. It is estimated that the integration process with UEAV will be relatively easy, as the RES tooklit uses in the backend the WRF software which is also used in WF pilot.

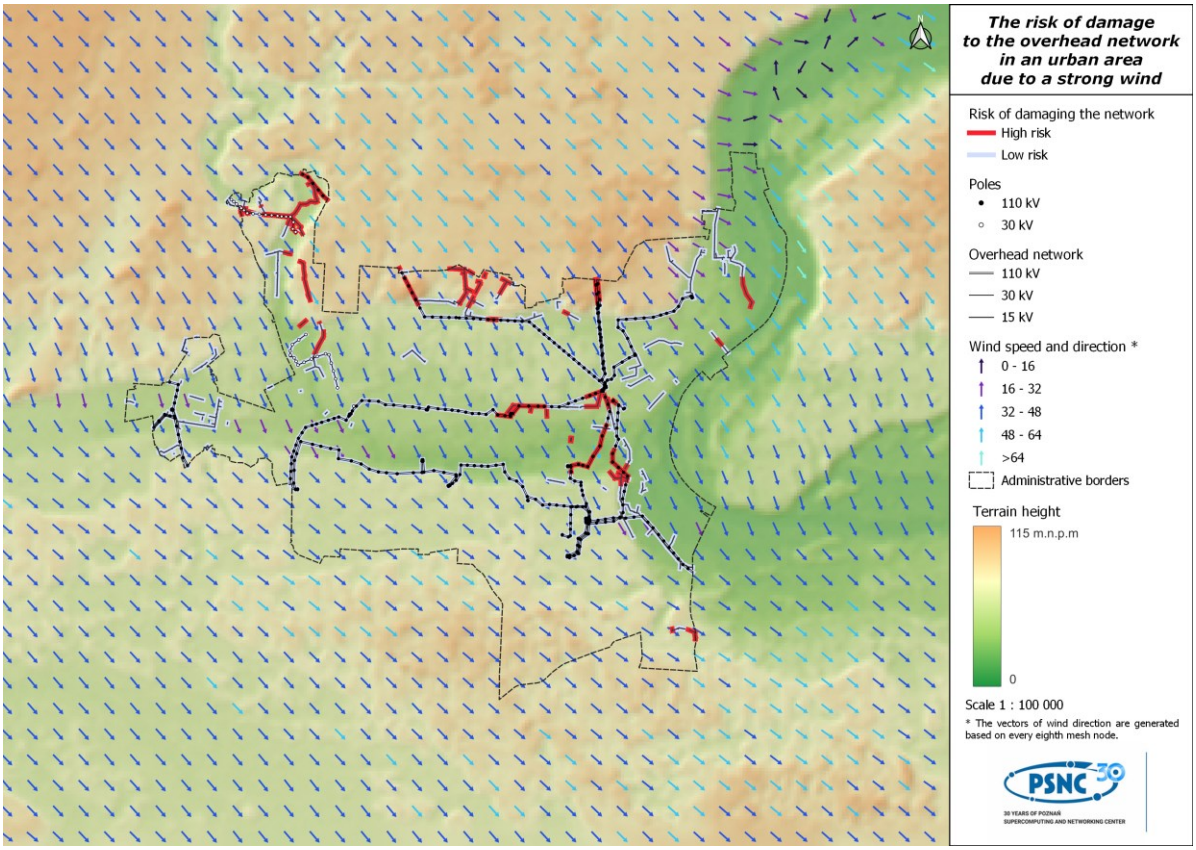

Figure 13. The exemplary visualization of the power lines damage risk analysis due to strong winds in one of the major Polish cities

4.4 Wildfires (WF)

MeteoGrid has firstly applied the advanced Unreal Engine Advanced Visualization **UEAV** to the area of Rectoret-Les Planes, in the outskirts of Barcelona (Spain) for the visualization testing in the wildfires pilot. This area was selected due to its historical risk of forest fires, its nature as a wildland-urban interface (WUI) where urban developments intermingle with forested areas, and the existence of prior planning efforts that have incorporated fire modelling and the extraction of 3D territorial information. These data have served as the starting point for simulations and the verification of simulations conducted in HiDALGO2.

Simultaneously, MeteoGrid has performed simulations on several historical fires in central Spain, particularly in the Madrid Autonomous Region, where detailed information is available that has been used to calibrate and validate the simulations performed on HPC facilities. One of the fires that has received the most attention is the

one that occurred on June 28, 2019, in Almorox-Cadalso de los Vidrios, the largest forest fire in the history of the Madrid Autonomous Region.

Given that Unreal Engine, starting from version 5.3, natively reads VDB files and converts them into Sparse Volumetric Textures (SVT), these have been used in volumetric materials with ray marching functions for very realistic smoke representation and its interaction with light (Figure 14).

Figure 14. An example of integrating a fluid dynamics HPC simulation of wildfire propagation into an immersive 3D visual scenario in Unreal Engine, including 3D representation of the terrain via the CESIUM service, volumetric clouds, and photo-realistic lighting according to the month, day, and time of the simulation. Graphics are computed in real-time, making extensive use of the GPU. The area corresponds to Les Planes-Rectoret, in the outskirts of Barcelona, Spain

4.5 Material Transport in Water (MTW)

For 3D CFD data waLBerla has already started to explore **Vistle** for visualization of particulate flows within the CEEC project (https://ceec-coe.eu/). The simulations conducted till now in waLBerla involve fluid-particle coupled physics for which there are already some visualisations generated with the help of VTK files and ParaView (Figure 15).

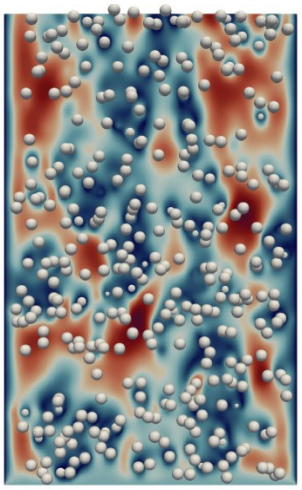

Figure 15. The VTK visualisation of velocity field with solid spherical particles in a fluidized bed simulation with fluid-particle coupled physics (without material concentration coupling).

Additionally for MTW, e.g. the concentration or temperature field has to be included in the visualization. It is planned to use CFDR and Unreal Engine advanced visualisation as potential integrating platforms for visualisation of MTW simulations, since they complement the features of Vistle. We also plan to couple to the Wildfire simulations and thus a visualization in UE is important to ensure a continuous workflow. CFDR can be beneficial for visualization of simulations results e.g. when physical tests are performed and it could be directly included in the CI/CB pipeline.

5 Looking into the future

The technologies and methodologies presented have both immediate and long-term prospects, as outlined below.

5.1 Future applications

COVISE and Vistle will be used to create a **urban planning** case for Stuttgart. The urban planning case will be focusing on the applications of methods and tools developed in the Urban Air Project. The foundation of the urban planning case is a very detailed traffic simulations for different scenarios representing different potential traffic interventions.

These traffic simulations will be coupled with the simulations tools of the Urban Air Project to create air quality predictions for the scenarios. This results will be integrated in a **digital twin** of Stuttgart which is build up with Vistle. The digital twin enables city planners and political decision makers to explore the results of the Urban Air Project in a collaborative manner and enables an evidence based decision.

In regards to the advanced visualization in Unreal for wildfires, Immediate developments, in line with those proposed in the forest fire pilot, first involve incorporating various flame and smoke emitters resulting from different combustion typologies, as occurs, for example, in wildland-urban interface (WUI) environments, where residential fuels and burning vegetation can be found, both with very different flame and smoke productions.

This involves developing techniques for true blending of heterogeneous volumes in UE; currently, this coexistence is not possible as UE draws either one or the other heterogeneous volume depending on the distance from the observer, resulting in flickering that can be confusing. It is also necessary to represent the evolution of the flame front on large surfaces (propagation) in a continuous manner; currently, the proposed methodology uses a heterogeneous volume for each of the moments represented in the flame front propagation, typically every 10 minutes. Such continuous propagations would help assess more accurately the effect of factors (wind, topography, etc.) on the evolution of the flame front.

Additionally, the proposed methodology of representing scalar and vector fields in a specific geographical environment can be extended to other pilots in HiDALGO2,

particularly UAP, where pollutant concentrations such as volumetric fogs or heterogeneous volumes (similar to how smoke is represented in forest fires) can be represented (Figs.16 and 17), and vector fields as flying tracers indicating flow intuitively.

Figure 16. An example of applying a scalar field to modulate an exponential volumetric fog over CESIUM-Google geometry over Madrid city (Spain)

Figure 17. Parametrization of the optical response of aerosol composition in wildfire smoke, particularly the proportion of black carbon and its effect on light absorption

The use of extended reality techniques that allow seeing the simulation results in specific geographical locations, blending real images with simulated fire and smoke images, is also very promising. In this pilot, much emphasis has been placed on the importance of the optical response of smoke, according to its composition and density, in order to provide the most realistic representation possible in fire-fighter and civil protection training scenarios. Therefore, further improvement of volumetric materials and ray marching routines used in Unreal Engine is necessary. Representation of the scattering-to-absorption ratio of light, based on the composition and density of smoke, is particularly relevant and has only been superficially explored so far. Unreal Engine also allows realistically incorporating other atmospheric phenomena, especially rain and clouds, as well as their interaction with light. Cloud cover and its effect on terrain shading can be very useful for calculating and representing the efficiency of solar energy generation, as proposed in the energy pilot.

5.2 Future developments and technologies

COVISE and Vistle modular architecture allows for expansion by developing new modules for data input/output, data manipulation, or rendering plug-ins to meet the

evolving needs of pilot owners throughout the project. For the urban planning case, a visualization of the traffic simulation is planned. Depending on the traffic simulation dataset specifications, a new rendering plug-in may be necessary to integrate traffic into the visualization, forming part of a digital twin. This digital twin combines data from multiple sources and analysis methods to generate new insights. Figure 18 shows an example of an urban digital twin, integrating simulation data, spatial analysis, and sensor information into a 3D city model.

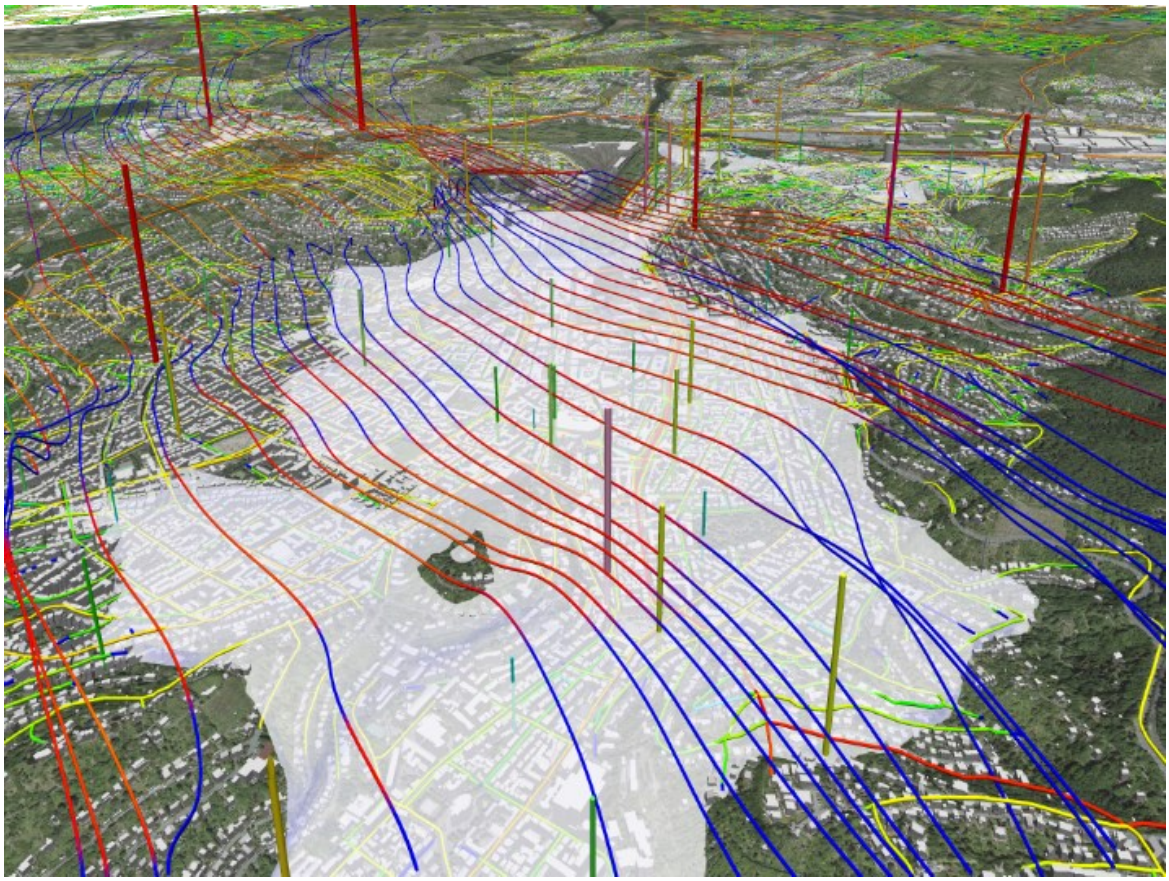

Figure 18. Urban digital twin integrating multiple data sources

Advanced visualization in Unreal Engine will also be able to draw upon new technologies in the very near future. In fact, the maturity level of technologies like **NeRFs** and **Gaussian Splatting** (Figure 19) already allows for capturing real-world scenarios with a high level of detail from a set of photographs, particularly those obtained from a drone, and generating an exact volumetric visual environment that users can navigate. The exploration of this technology is planned in the forest fire pilot, applied at a very local scale, within building and facility environments, where volumetric smoke and flame simulations are integrated as previously described. This approach is particularly promising for improving risk perception among home-owners and facility

owners, as they can experience what a forest fire threat would be like exactly in their real environment.

Figure 19. First trials by MeteoGrid integrating volumetric smoke and flame simulation with a Gaussian Splatting training, generated from drone-acquired photographs in Gandullas (Spain). The scenario is calculated in real-time and has been compiled in Unreal Engine for immersive training experiences

Additionally, and in connection with this technology, the application of **Extended Reality** (XR) techniques, where volumetric smoke and fire simulations are blended with the image of the real world, can offer training scenarios and contextual perception in operational situations, which is very useful for fire-fighter training. Another technology whose development has already begun to be explored in the forest fire pilot is the development of **hybrid immersive full stereo 360º video** experiences with three-dimensional object scenarios. This technology would allow for encapsulating complex and costly simulations and representations of combustion, flame production, and smoke in stereoscopic 360º videos projected onto the virtual reality device, but mixed with three-dimensional object scenarios in the foreground. Thus, the user would have a visually complex and realistic environment without an extraordinary burden on graphics computing, while still having objects to interact with. These promising technologies, already under exploration, could be integrated into training experiences and applications for fire-fighters and home-owners, encapsulated in **APK applications**

for Android VR devices, such as Meta Quest Pro, thus avoiding dependence on expensive and cumbersome equipment with powerful CPUs and GPUs.

6 Conclusions

The phenomena associated with environmental challenges, especially those occurring in or related to the atmosphere, are dynamic, non-linear, volumetric, and interrelated. The simulation of these processes heavily relies on fluid dynamic models running on HPC facilities, resulting in volumetric fields of scalars and vectors and their temporal evolution. Additionally, these phenomena occur in specific geographical locations, requiring numerical and visual alignment with geographic references.

The requirement analysis reveals the needs for designing and implementing visualization solutions. These include, on one hand, the need for an agile and lightweight method for packaging simulation results and subsequently visualizing them interactively in 3D on Web clients. On the other hand, there is a need for the collective analysis of simulation results over large geographic areas, allowing multiple users to interact with large datasets. Additionally, there is a need for photo-realistic scenarios incorporating simulation results in an immersive, interactive, and intuitive environment that includes landscape elements. Finally, there is a need for using general visualization tools that can automatically be fed with simulation results, including the extraction of geographic data from the area of interest.

In response to these needs, the HiDALGO2 project addresses the challenge of visualizing simulation results from HPC facilities through **four methodological approaches**, each with its capabilities and specificities, yet flexible enough to be used generically. These methodological proposals are:

- **CFDR**, which efficiently packages the CFD simulation data and sends it to web-based graphical clients for lightweight, real-time interactive visualization.
- **VISTLE-COVISE**, which allows for interactive and immersive visualization in CAVE environments for collective analysis of simulation results.
- **UEAV**, which proposes the compilation of photo-realistic VR experiences, including simulation data and the integration of geographic components, objects, and special effects into highly portable packages for training.
- **KTIRIO-GUI**, which enables the extraction of spatial data from the selected geographic region, the integration of simulation data, and interactive visualization using commonly used generic applications.

The proposed solutions are designed to be applied to specific pilots, particularly CFDR for the UAP pilot, VISTLE-COVISE for the RES pilot, UEAV for the WF pilot, and

KTIRIO-GUI for the UBM pilot. However, their use will be explored crosswise among pilots and for other future applications, based on the specifications of the exchange files.

The technical details of the operation and functionalities of the selected or developed visualization solutions have been presented, with a special focus on the data input formats and the technical requirements for their use. This will provide the proposed technologies with greater flexibility for more generic use. Workflows that utilize the described tools have been presented to meet the identified requirements and needs of each case study, but in a way that is sufficiently flexible to be applied to other domains.

There is a clear need to continue exploring and expanding future developments on the proposed tools. Therefore, this document remains open-ended, given the rapid evolution of the technologies associated with visualization, with updates expected throughout the project's lifespan.

References

1 Caballero, D. (2024) A method for importing and drawing SHP files in Unreal Engine. Technical Notes Series (preprint). DOI: 10.13140/RG.2.2.26576.48648/1.

2 Museth, K. (2013) VDB: High-resolution sparse volumes with dynamic topology. ACM Trans. Graph. 32, 3, Article 27 (June 2013) 22 pages. DOI: http://dx.doi.org/10.1145/2487228.2487235

Annexes

Annex 1. File Formats

CDAT Format

CDAT is a binary file format, with the goal of storing only the scalar/vectorial values of a simulation, without the mesh, in a compact form. The file starts with the following header:

Table 5. CDAT format header entries

Type	Description
uint64: magic	MAGIC code, for version checking.
uint32: float_size	4: float is used to store the data 8: double is used to store the data
uint32: cell_count	Number of entries in the data
uint32: step_count	Number of time steps in the data
uint32: compressed	0: No compression used. 1: LZ4 compression used.
uint32: components	Number of components for each entry. 1 for scalar data series, 2 for storing 2D vectors, etc.

FGA Format

The FGA (**F**luid **G**rid **A**SCII) format is a simple ASCII structure widely used as an exchange file in fluid dynamics applications for volumetric vector fields. It consists of a three-line header:

- The first line specifies the number of cells along each axis X, Y, Z.
- The second line specifies the minimum coordinates of the bounding box.
- The third line specifies the maximum coordinates of the bounding box.

Following the header, the (u,v,w) components of the vectors in the volumetric field are written, corresponding to the centre of each voxel, with one vector per line until all cells

are covered. Each line ends with a comma, and the data are also separated by commas. An example is given below:

```
10.0, 10.0, 10.0,        # Cell count in X, Y and Z axis
0.0, 0.0, 0.0,           # Bounding box minimum X Y Z
100.0, 100.0, 100.0,     # Bounding box maximum X Y Z
0.0, 0.0, 0.0,           # Vector components in the first cell
0.0, 0.0, 0.0,           # Vector components in the second cell
:                              :
0.0, 0.0, 0.0,           # Vector components in the last cell
```

Note that the resolution of the cells is implicitly given by the length covered along each axis and the corresponding number of cells. The FGA format is natively read and written by many CFD applications. Additionally, Unreal Engine can read FGA files and interpret them as global vector fields for use in systems such as particle systems.

OBJ Format

The OBJ file format, originally developed by Wavefront Technologies for its Advanced Visualizer animation package, is an open geometry definition standard now widely adopted by various 3D graphics applications.

The OBJ format is stored as straightforward ASCII files, representing only 3D geometry. It includes the position of each vertex, the UV coordinates for texture mapping, vertex normals, and the faces that define each polygon, listed as vertices and texture vertices. By default, vertices are stored in a counter-clockwise order, eliminating the need for explicit face normal declarations. While OBJ coordinates are unit-less, scale information can be included in a human-readable comment line.

An OBJ file has a simple physical structure, consisting of lines that start with keywords. Following each keyword, appropriate options and values are specified. Long lines can be broken up, using a backslash (\) character at the end of lines to be continued. Anything following a hash character (#) is a comment.

A vertex is defined by a line starting with the letter 'v', followed by the coordinates (x, y, z[, w]). The 'w' coordinate is optional and defaults to 1.0. A right-hand coordinate system is used for specifying the vertex positions. Some applications support vertex colours by appending red, green, and blue values after x, y, and z, which precludes specifying 'w'. The colour values range from 0 to 1.

Faces are defined using lists of vertex, texture, and normal indices in the format vertex_index/texture_index/normal_index. Each index begins at 1 and increases according to the order in which the corresponding element was defined. Polygons, such as quadrilaterals, can be specified by using more than three indices.

A valid vertex index corresponds to the elements in a previously defined vertex list. If the index is positive, it refers to the position in the vertex list, starting at 1. If the index is negative, it refers to the position relative to the end of the list, with -1 indicating the last element. Each face can contain three or more vertices.

Materials that define the visual properties of the polygons are stored in external .mtl files. An OBJ file can reference multiple external MTL material files, each containing one or more named material definitions. The material name specified by this tag corresponds to a named material definition in an external .mtl file, indicating the material to be used for the subsequent element.

A complete description of the OBJ specification can be found at: https://paulbourke.net/dataformats/obj/

VDB Format

Introduction

Sparse volumetric data refers to data that does not fill the entire 3D space, meaning that only some voxels contain data. The VDB format offers a hierarchical tree approach (such as octree or B+Tree) that efficiently stores only the voxels with data (leaf nodes). **VDB** stands for **V**olume **D**ynamic **B**+Tree (but also Voxel Data Base or Volumetric Data Blocks). One of the main advantages of the VDB format is that it is natively read by Unreal Engine, where it is interpreted as sparse volumetric textures (SVT). Additionally, the efficiency in storage, operation, and rendering of sparse volumes makes it particularly attractive for representing smoke columns and flame fronts.

General Information about the Data Structure

VDB aims to represent volumetric unit (voxel) data within a grid structure that is axis-aligned and regularly spaced. To achieve this, a root node is established that

encompasses the entire space (Root). The space is then divided into several cubic regions of equal size, each a power of two along each dimension.

In the 5-4-3 variant, the upper-level nodes are 32x32x32. This means they have 32 * 32 * 32 = 32,768 children. Note that 1 << 5 = 1·2^5 = 32. This is where the "5" in the variant name comes from. One can infer what this means for the remaining numbers. Each child of a 32x32x32 node is itself a node with 16x16x16 children because 1 << 4 = 1·2^4 = 16. The children of each of these nodes are in turn 8x8x8 nodes. These are the Leaf Nodes in the tree and contain the voxel data itself, which can technically be of any arbitrary type, including user-defined types, but we will focus on simple floating-point data, specifically 16-bit floating-point numbers.

Each region has an origin, which places them in Cartesian 3D space. Let's call these regions the Upper-Level Nodes of our tree (Node-5). They are the direct children of the root node, and we can have as many as we want, but we want them not to overlap.

The dimension (dim) of each node is a power of two. Thus, a property can be defined for each node level:

32x32x32 nodes: log2dim = 5
16x16x16 nodes: log2dim = 4
8x8x8 nodes: log2dim = 3

- A Node-5 is a node that has 32x32x32 children. These are the upper-level nodes.
- A Node-4 is a node that has 16x16x16 children.
- A Node-3 is a node that has 8x8x8 children. The children in this case are the voxel data themselves and are thus also called Node-3 leaf nodes.

We could also refer to the voxels as Node-0 to be semantically consistent, in which case the voxels would technically be the leaf nodes. However, we will refer to the leaf nodes as Node-3 to better distinguish between nodes and voxels. Another reason is that voxel data is written "directly" in the VDB format and is not explicitly written as a node.

To cover as much space as possible, there can be an arbitrary number of Node-5s. They can be placed anywhere in the space by specifying their origin. Each Node-5, however, contains a fixed number (32x32x32) of children. Node-4 and Node-3 also have a fixed number of children. This differs from how trees are generally structured, where nodes can have a variable number of children at any level. However, this is an important feature of VDB, as it allows us to efficiently determine where to write voxel data given its position in Cartesian 3D space.

We know how many children each node has. Let's now find out how much 3D space each node covers. Node-3s are the simplest and most obvious; each Node-3 covers a cube of 8x8x8 voxels. A Node-4 contains 16x16x16 Node-3s, meaning it covers a cube of 128x128x128 voxels. Finally, a Node-5 covers a cube of 4096x4096x4096 voxels.

To get an idea of the scale, we can look at how many voxels are in each volume. For Node-3s, there are only 8*8*8 = 512 voxels. A Node-4 spans 128*128*128 = 2,097,152 voxels, around 2 million. Moving up to a Node-5, it spans 4096*4096*4096 = 68,719,476,736 voxels, or 68.7 billion! This means that if we were to store a 16-bit floating-point number for each voxel, we would need 128 GB of memory for the voxel data of a single Node-5!

Note that the word "spans" is used instead of "contains." This is because, although each Node-5 conceptually contains 32x32x32 Node-4s and each Node-4 conceptually contains 16x16x16 Node-3s, not all of these nodes contain voxel data and, therefore, do not need to be explicitly stored. On the other hand, the 8x8x8 voxels of a Node-3 are always stored fully, or densely if you will (unless compression is enabled).

As mentioned earlier, VDB allows us to represent these vast regions of space sparsely. Not all voxels in a 4096x4096x4096 region may contain data. These voxels are said to be inactive and assume a so-called background value, which only needs to be specified once. Therefore, we do not necessarily need 128 GB of data for a Node-5; it can be achieved with much less. Let's see how this is done using bit masks.

Bit Masks

The tree structure described so far would be useless if it were completely fixed in size, as we might as well store the data in a dense 3D array. Clearly, the tree structure must be useful in some way. Its value comes from the fact that we can skip some nodes, eliminating entire subtrees.

For example, consider a Node-4 (spans 128x128x128 voxels). Suppose we have some interesting voxel data in just one 8x8x8 region, meaning only one of the children is active. It would be advantageous to store only these 8x8x8 voxels without the remaining 128*128*128 - 8*8*8 = 2,096,640 voxels. To do this, we need to know the location of these 8x8x8 voxels within their parent Node-4.

In VDB, this is achieved with bit masks. The Node-4 will store a bit for each of its potential Node-3 children, i.e., 4096 bits in this case (16*16*16). These can be compactly stored in 64-bit integers, which we will use for our implementation. However,

it's worth noting that the underlying integer type is not really important and is an implementation detail, as long as the bits are written and read in the correct order. This is the formula to achieve it:

$$bit\_index = z + y * 16 + x * 256$$

Here, bit_index is the index in the bit array for a Node-3 with an offset of (x, y, z) relative to its parent Node-4. The order of traversal is: first, sweeping through the Z-axis, covering a total of 16 positions (from 0 to 15); then, moving to the next Y position and repeating the Z sweep; until all 16 Y positions (from 0 to 15) are covered, after which we move to the next X position. In other words:

For every X (For every Y (For every Z))

A Node-4 can have up to 16x16x16 children, so the range of values for (x, y, z) is [0, 16]. It can be verified that setting all of them to the minimum value of 0 yields an index of 0, and using the maximum value of 15 yields the index of the last bit in the 4096-bit array (i.e., the range is from 0 to 4095).

Node-5 and Node-3 also have these bit masks, but since they contain a different number of children, they require a different number of bits. A Node-5 requires 32,768 bits (512 64-bit integers), while a Node-3 requires 512 bits (8 64-bit integers). Even though Node-3 is a leaf node, it still has a bit mask indicating which of its voxels are active. The formula for a Node-3 would be:

$$bit\_index = z + y * 8 + x * 64$$

And for a Node-5:

$$bit\_index = z + y * 32 + x * 1024$$

The generalized formula, based on the definitions above, is:

$$bit\_index = z + y * dim + x * dim^2$$

Since dim = 1 << log2dim (from the previous definition), the multiplications can be converted into bit shifts:

$$bit\_index = z + (y << log2dim) + (x << (log2dim << 1))$$

Moreover, considering that the range of x, y and z is [0, dim], we see that the number of bits needed for each is log2dim bits. Notice how it shifts upwards by log2dim bits, which is exactly how many bits it occupies. For instance, X shifts by log2dim << 1 = log2dim * 2 bits, which is the number of bits occupied by Y and Z. For a Node-3, the final bit positions would look like this:

xxxyyyzzz

The additions (+) can be replaced with bitwise OR operators (|):

bit_index = z | (y << log2dim) | (x << log2dim << 1))

The formula can be written using multiply-add (*madd*) operations:

bit_index = madd(madd(x, (1 << logdim), y), (1 << logdim), z)

Where:

madd(a, b, c) = a * b + c

These masks are referred to as secondary masks because they encode the topology of the child elements of a node.

File Structure

In VDB, all integer and floating-point numbers are in little-endian format. The convention is to describe floating-point numbers (floats) with an "f" and integers with a "u" (unsigned) or an "i" (signed), followed by the number of bits.

The file begins with a **header** containing the following information in this order:

1. An 8-byte magic number, consisting of the bytes: {0x20, 0x42, 0x44, 0x56, 0x00, 0x00, 0x00, 0x00}, where the first byte is a space character, and the next three bytes spell "BDV." The remaining four bytes are set to zero.
2. A u32 indicating the file version, use version 224.
3. Two u32 values, where the first indicates the major version and the second indicates the minor version of the library. Set the major version to 8 and the

minor version to 1 to mimic the file being written by OpenVDB 8.1, used to generate the reference files.
4. A u8 indicating whether grid offsets will be specified. These are byte offsets pointing to the beginning and end of a grid. Initially set to 0, but as more grids are added, specifying these offsets becomes useful so that file readers can easily skip grids they are not interested in. Grid offsets are not stored in the header and are described later.
5. A string representing a 128-bit UUID. Use a valid but constant UUID for testing purposes. Eventually, you will want a valid one, which can be generated using a library. However, implementing this yourself is straightforward.
6. Metadata about the entire file. The number of entries is written first as a u32 followed by the metadata itself. Write a u32 zero to indicate no metadata. However, it is recommended to specify metadata per grid.
7. A u32 for the number of grids; usually, only one grid is written.

In VDB data is stored in so-called trees, as mentioned. Each tree and its data may be referenced by one or more grids. In this example, only one grid and one tree are created.

A grid not only references a tree, it also has an associated transform. This is a map (in the mathematical sense) that converts coordinates from index-space to world-space. Index-space is simply a continuous extension of the discrete (x, y, z) indices used to locate a specific voxel. In VDB, the coordinates (0, 0, 0) maps to the first voxel's centre.

The grid is written just after the header, with the following structure:

1. The name of the grid as a length-based string. In many applications the word 'density' is used.
2. The grid type as a length-based string. it determines how the data is to be interpreted. Use 16-bit floating point to store voxel data and a 5-4-3 tree structure. Therefore, write as the grid type:

   Tree_float_5_4_3_HalfFloat

   If the _HalfFloat suffix is omitted, use 32-bit floating point data. Other formats are possible by replacing float with the appropriate string.

3. Instance parent as a u32. Use zero here since as no instancing is used.
4. Byte offset of the 'Grid Descriptor' as a u64. The grid descriptor starts 3 u64s after the current byte offset, so write just that, the current stream position plus 24 bytes

5. The two u64s before the grid descriptor, namely, the grid byte offsets that are disabled in the header of the file. Therefore, write zeros for both u64s. If grid offsets are enabled, the first u64 is the start of the grid data and the second the end of it
6. A u32 indicating if any compression is used for the grid data. If not, write a zero. This is also where the grid data starts and is at the byte offset that the first byte in the previous point should contain if grid offsets are enabled.
7. The grid metadata as a u32 indicating the number of metadata entries followed by the metadata itself. Writing one metadata entry involves 3 things:

    a. Writing the name of the entry as a length-based string
    b. Writing the type of the entry as a length-based string
    c. Writing the entry data itself based on the type specified

8. The transform. This one starts with the name of the mathematical map used for the transform as a length-based string. Use an 'AffineMap', so write that length-based string followed by 16 f64s that are the entries of the 4x4 matrix representing the affine transformation. OpenVDB uses the convention of right-multiplying the matrix by a vector which transposes the meaning of the entries.
9. The tree data itself.