Visual Analysis for Extremely Large-Scale Scientific Computing

D4.1 – Specification of the GPU-Driven Representations and Architecture of the GPU-Based Scientific Visualization Pipeline

Version 1

Deliverable Information

Grant Agreement no	619439
Web Site	http://www.velassco.eu/
Related WP & Task:	WP4, T4.1, T4.2
Due date	March 31, 2015
Dissemination Level	PU
Nature	R
Author/s	Andreas Dietrich, Frank Michel
Contributors	Miguel Pasenau, Abel Coll

Approvals

	Name	Institution	Date	OK
Author	Andreas Dietrich	**FRAUNHOFER**		
Task Leader	Andreas Dietrich	**FRAUNHOFER**		
WP Leader	Frank Michel	**FRAUNHOFER**		
Coordinator				
Contributor	Miguel Pasenau, Abel Coll	**CIMNE**	13.04.2015	

Change Log

Version	Description of Change
Version 0.1	Topics outline
Version 0.2	Added architecture scheme and GiD and Analytics sections
Version 1.0	First complete version
Version 1.1	Review from Abel with some corrections

Table of Contents

1 Introduction

The Vision of VELaSSCo is to provide new visual analysis methods for large-scale simulations serving the petabyte era and preparing the exabyte era. VELaSSCo does this by adopting Big Data tools and architectures for the engineering and scientific community and by leveraging new ways of in-situ processing for data analytics and hardware accelerated interactive visualization.

The design of the VELaSSCo Platform has to account for user requirements and assessment, namely the demand for real-time visualization, fast interactivity and interoperability, realistic visualizations, and so on.

To address this challenge, VELaSSCo uses the following set of strategies:

- Take advantage of Big Data technology (in a broad sense, architecture, methods, philosophy, etc.), to perform the **post-processing operations faster and in a distributed way**. As Big Data works with highly scalable parallel systems, the post-processing algorithms should be programmed to fit these characteristics.
- **Reduce and simplify the data to be visualized**, in order to reduce the time for sending the information to the client. For example, although the results of a simulation can come in form of a volume mesh, the user may want to visualize datasets in surface or line meshes (like iso-surfaces or stream-lines).
- Use new **GPU-based algorithms** and formats to achieve interactive visualization rates in the visualization client.

This document gives an overview and detail on all aspects concerning the GPU-based visualization pipeline and the GPU-optimized data format used for transferring the data from the VELaSSCo platform to the visualization client.

We first present an overview of the VELaSSCo platform architecture and the communication pipelines. Based on this, the access, graphics and analytics modules, which are the main parts of the visualization pipeline, are described in more detail. Afterwards, the two visualization clients used in the frame of VELaSSCo project to connect to the Platform are described from a general and architectural point of view. Finally, the GPU-optimized data representation envisioned in VELaSSCo is presented in detail.

2 Overview over the VELaSSCo Platform Architecture

The VELaSSCo platform is the core block in the VELaSSCo architecture based on the Lambda Architecture. From a layered perspective, the platform is composed of three layers (Figure 1):

- The first layer, I/O or data layer is composed by a gathering tool (Flume) next to a monitoring tool to know the status of different process. The data query layer is connected to the I/O interface with the Hadoop ecosystem. Example of this interfaces are Hive, Storm, Hbase, etc.

- The second layer (query engine layer) concerns the analytics, batch and real time modules of the platform. This layer is managed by YARN [9]. It is the specific resource manager for Hadoop 2.x. With this software Hadoop supports any kind of computational model. This strategy enables to support more complex computational models, and MapReduce is not anymore the only option. Thus other tools like multi-resolution, convertors, etc. are feasible.
- Finally, the storage component, concerns the storage system of the platform. With Hadoop extensibility, most existing storage systems are supported. VELaSSCo will not only work with Hadoop storage system but also provide new storage facilities with the EDM-DB (similar to the HadoopDB approach).

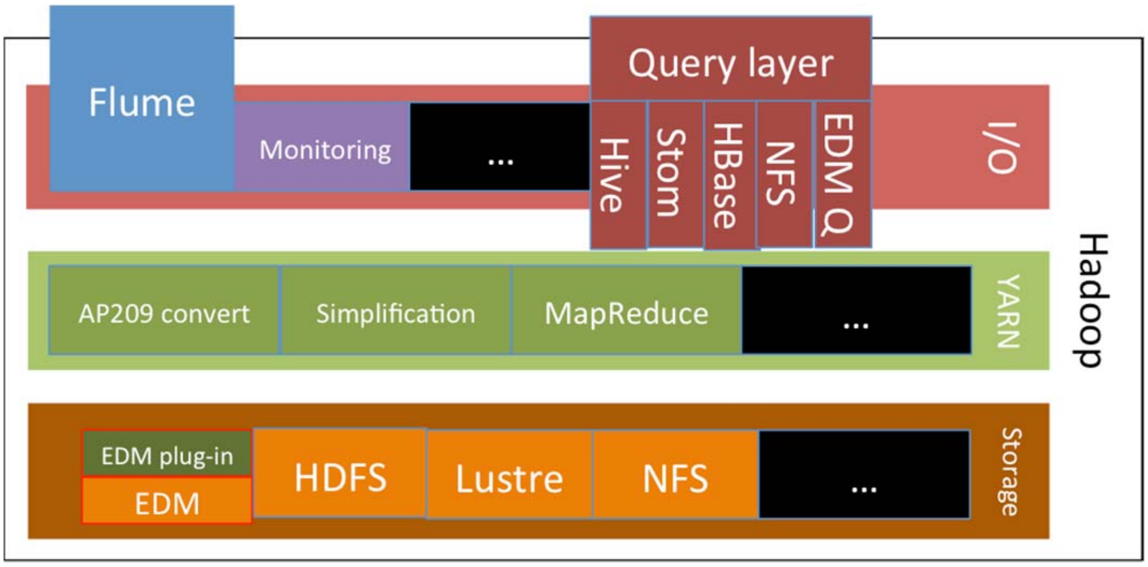


Figure 1. VELaSSCo core scheme.

A logical view of the platform disclosing the different layers in which the platform is composed is depicted in .

There will be two layers:

- **Data layer**: which is responsible to store, access and translate the data and will answer the data queries coming from the "Query Engine Layer" and, eventually, when data is ingested in the platform, it may generate simplification queries, to create coarser views of the simulation data. This layer corresponds to the I/O and Storage layer of Figure 1.
- **Query engine layer**: which is the responsible for receiving the user requests (VELaSSCo queries) from the visualization client, processing them, generating data queries to the Data Layer, processing the returned results, eventually merging them, and format these results in a gpu-friendly way for the visualization client. This layer corresponds to the YARN layer of Figure 1.

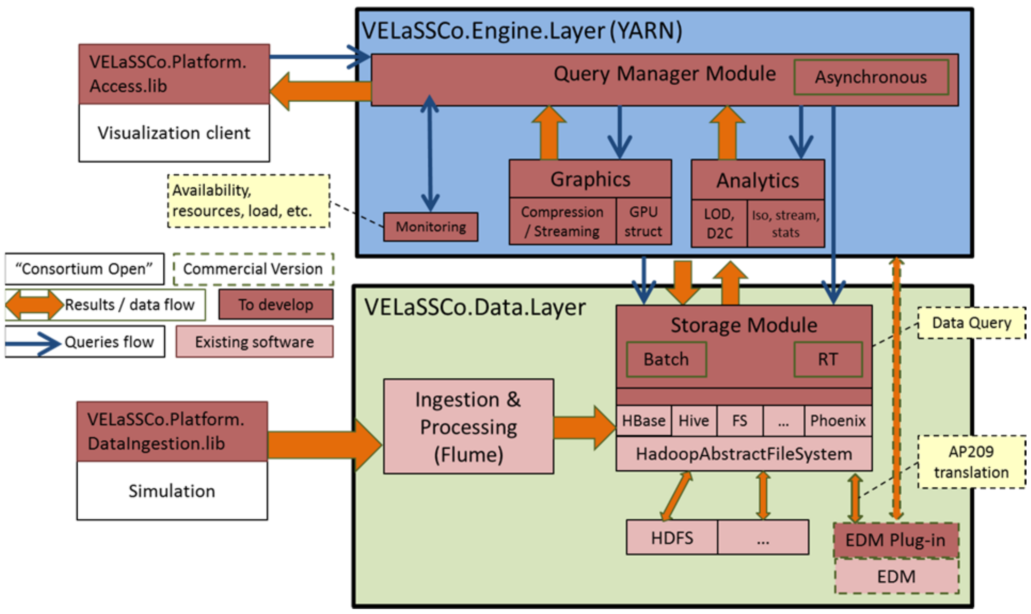


Figure 2. Schema of the VELaSSCo architecture.

In addition to these two layers, two libraries/modules will allow the connection between the platform and the final user:

- **Data Ingestion library**: which will provide a mechanism to ingest data from the computer engineering simulation solvers to the VELaSSCo platform. After establishing the communication with the Data Layer, including user validation, the ingestion module will send the data to the Hadoop Abstract File System. The data layer will store the data in HBase tables if the HDFS is used as storage or in EDM-DB system (previous translating, if needed, the data to STEP/AP209) when using the EDM-plugin and EDM-DB system.
- **Platform access library**: will communicate the visualization client and translate user requests into VELaSSCo queries, receive the results and show them to the user. The module will also generate VELaSSCo queries automatically, for instance when the user just navigates through the simulation model, or to ensure interactivity it will handle coarser models and issue full-detailed queries when the user zoom on parts of the simulation model.

Inside the query engine layer the following modules are to be developed:

- **Query manager**: responsible for managing the VELaSSCo queries, evaluate their costs and issue Fast queries and, eventually, Batch queries. If a query is too expensive, i.e. it will take too much time to evaluate, this module will launch both, a simplified version of the query (using a simplified version of the simulation data) to provide feed-back to the visualization client and a full detailed batch query, which will access the full-detailed simulation data. The query manager will also synchronize and control the visualization parameters and issue visualization queries to the Real Time module, like the 'traveling' through the model for an interactive navigation.

- **Analytics module**: responsible to perform data analytics over the stored simulation results. It will also provide a cost estimation of the data analytic query to help the query manager evaluate in which mode is the analytics to be evaluated: in speed mode or in batch mode. This module will also generate coarser versions of the simulation data in order to provide fast feedback to the user and the discrete to continuum transformation needed by the DEM simulation data.
- **Batch module**: responsible to perform heavy data analytics and to provide feedback on the progress of the query.
- **Real time module**: responsible of providing feed-back to the user whether in full detail (for instance of detailed views of navigation through the simulation model), simplified version (for heavy queries and when the coarse view is present) or progress indicator of the batch queries being executed; also for preparing the data for a fast visualization in the visualization client.

The communication between the Data Ingestion library (glued to the simulation program) and the Data Layer will utilize Apache Flume, and the communication between the visualization client + platform access library and the Query Engine Layer will be done using Apache's Thrift.

As for the communications between modules, as they are modules integrated inside a Hadoop framework, it will follow Hadoop recommendations just adapted for the FEM/DEM analytics needs.

The data to be passed between components will be reduced to basic types (string, integers, long integers (int64, doubles) or basic type lists/arrays.

A detailed overview of all the communication pipelines is presented in Figure 3. A deeper representation of the infrastructure with the communication data flow is presented in that Figure. Blue arrows represent the data transfers, while green arrows represent queries.

In the engine layer (Yarn), some examples of plug-ins are presented; each module (batch, RT and analytics) can be extended with other applications. This strategy allows bringing new capabilities to the platform in the future. In Figure 3, two execution queries are presented: on the server side (black octagons) and on the user side (purple circles).

The first pipeline is representative of a storage plan. For this purpose, only compute nodes are involved in the process. In (1), data is produced by different nodes using their simulation engine. Then, Flume agents aggregate this data and write it into Hadoop through the abstract file system. This abstract file system then writes data into the appropriate file storage system, for example on a standard storage system (HDFS, Lustre, etc. 3a) or using EDM, with a conversion from a file format to EDM/Step AP209 (3b).

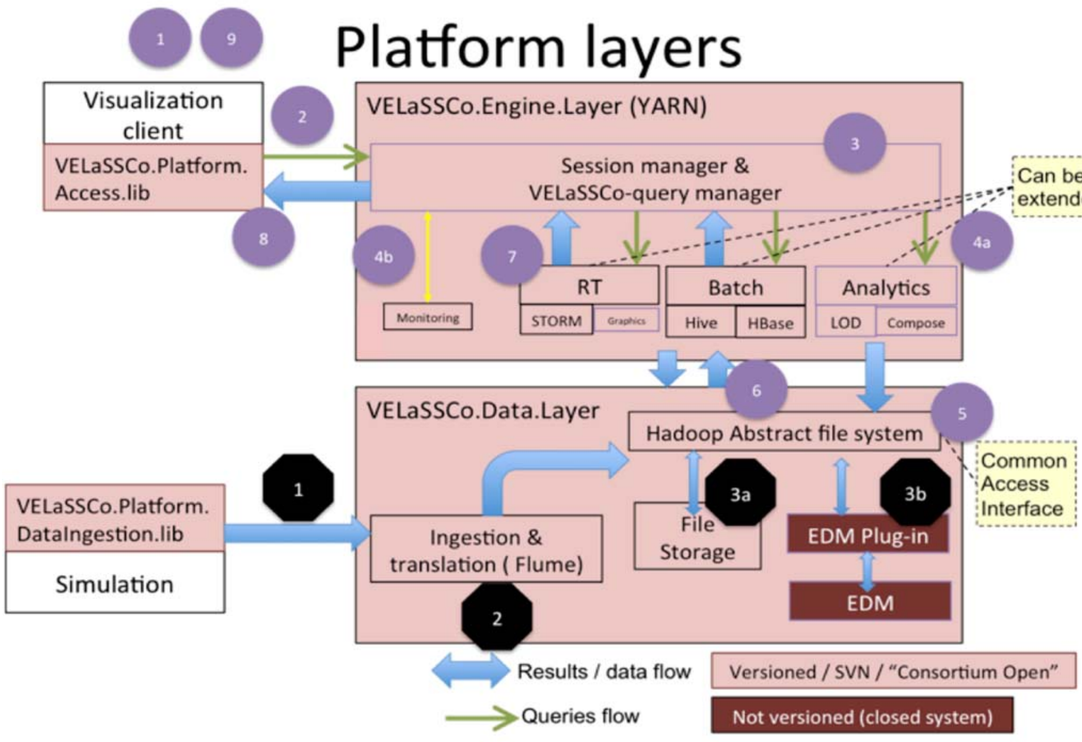

Platform layers
1
9
Visualization client
VELaSSCo.Platform. Access.lib
2
8
VELaSSCo.Engine.Layer (YARN)
3
Session manager & VELaSSCo-query manager
Can be extended
4b
7
4a
Monitoring
RT
STORM
Graphics
Batch
Hive
HBase
Analytics
LOD
Compose
6
5
VELaSSCo.Data.Layer
Hadoop Abstract file system
Common Access Interface
VELaSSCo.Platform. DataIngestion.lib
Simulation
1
Ingestion & translation (Flume)
2
3a
File Storage
3b
EDM Plug-in
EDM
Results / data flow
Queries flow
Versioned / SVN / "Consortium Open"
Not versioned (closed system)

Figure 3. Communication pipelines.

For the second workload, a query based on visualization is performed, and transmitted to the query layer (2) using Thrift. The user account is validated and the query is transformed into an analytic query (for example mesh simplification) (query 4a). The analytics module gathers information from the file system and processes it. Eventually a quick simplification is performed and the produced data is stored (and obviously dispatched to the correct storage system) (5 and 6). Then the produced data is sent to the real time and transmits the result data, with Thrift, in a suitable format for the visualization engine and finally, the rendering is performed by the viewer (8 and 9). Information concerning the query and progress is sent to the monitoring tool (4b) which also monitors availability and resources statistics of the platform

This example shows how storage and queries are processed in the platform. An important point of this architecture is the support for extensibility. All necessary plug-ins can be deployed, regarding which data will be stored and which computations are performed.

3 Access Library Module

A user accesses the VELaSSCo platform by operating a local visualization client (see Figure 3). The visualization client is separated from the database infrastructure, and communicates with the platform to send queries and to receive results. The visualization client basically consists of three main components: a visualization engine, the VELaSSCo access library, and one or more rendering nodes (Figure 4).

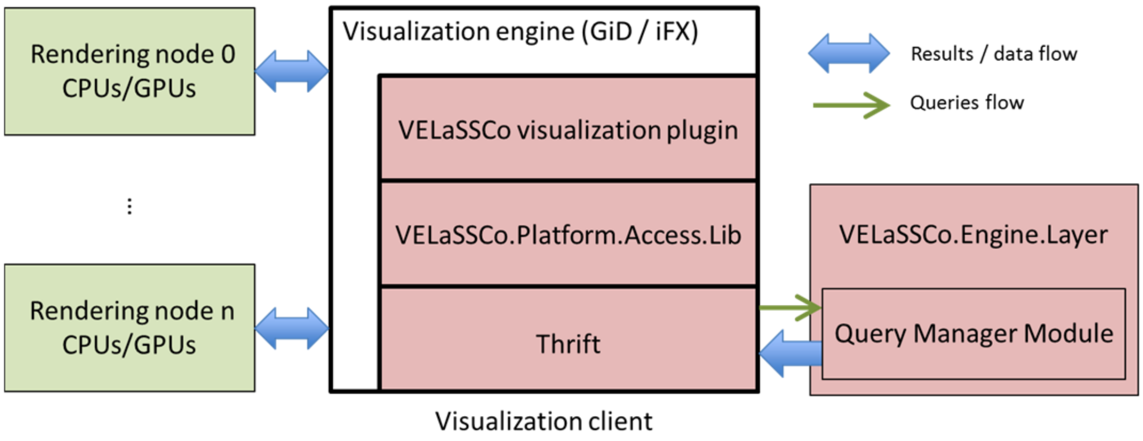


Figure 4. Integration of the VELaSSCo platform access library.

The visualization engine serves as rendering and post-processing framework, and is responsible for generating the final images that are presented to the user. Depending on the specific use case, this can either involve rendering of data that has been pre-computed by the VELaSSCo platform (e.g., a triangle mesh representing an iso-surface), or post-processing simulation data on the client side (e.g., generating an iso-surface from a volume dataset). As visualization engines GiD (CIMNE) [6] and iFX (Fraunhofer) [7] are employed (Chapter 6) in the frame of VELaSSCo project, although the architecture of the platform is designed to allow any other one.

The rendering nodes are dedicated machines (ideally fitted with multiple CPUs and GPUs) that can be utilized by the visualization engine. This enables distributed parallel visualization and post-processing on the client side, and allows for leveraging an available local HPC infrastructure.

The VELaSSCo access library acts as a communication layer, which connects the visualization engine to the VELaSSCo engine layer within the platform. To this end, the access library provides a specific application programming interface (API) for sending queries and receiving results. Both GiD and iFX feature a plugin-mechanism to enable extensions. To attach the access library to the visualization engines, a plugin for each framework will be developed, where each plugin will be linked to the library. Keeping platform access in a separated library allows for targeting other frameworks besides GiD and iFX.

On the client side, a user interacts with a graphical user interface of one of the visualization engines. User actions are translated by the plugin component into a query message that is sent to the access library. In order to interact with the engine layer, the library will make use of Apache Thrift [5]. Thrift is a framework for remote procedure calls (RPC) and data serialization, and is used to interchange information between the access library and the query manager module (see Chapter 2) in the

VELaSSCo engine layer. Sending a query will trigger either retrieving simulation results (to be post-processed on the client), or the computation of geometric data (to be rendered on the client). The resulting data is sent back the same way to the visualization engine, which then makes use of its rendering nodes to display or process the data.

Particularly, the access library is responsible for:

- opening, maintaining and closing the connection to the VELaSSCo platform,
- user validation,
- model selection,
- forwarding queries issued from the visualization client,
- decompress the data received and pass it to the client to visualize,
- providing information about the state of execution of the queries,
- providing interaction metaphors for specialized queries,
- monitoring the health state of the VELaSSCo platform.

4 Graphics Module

The VELaSSCo graphics module resides within the platform's engine layer (). Its primary purpose is to make use of server side HPC compute and memory resources to prepare query results in a suitable way, so that the information can be displayed by the visualization engines at high speed with minimal latencies. To this end, the graphics module converts the data into an internal representation. Data structures resulting from this conversion are handed over to the query manager module, which sends them back to the visualization client.

Moreover, the graphics module will handle streaming and progressive data transfer. Rather than sending the complete data set of a query in one step, information is sent on demand in small parts based on user input (e.g., depending on the current position of a moving camera).

It is important to note that the graphics module does not create new geometry as part of an analytics post-process. It does not compute attributes of glyphs, e.g., direction of arrows to visualize flow direction. It can, however, compute optimized geometric representations for glyphs, e.g., it could generate NURBS representations for glyph arrows.

4.1 Compression and Formatting

This conversion is done in two mayor steps. First, the data is optimized and compressed. In a second step the data is wrapped in an internal formal. The design of the data format is guided by the requirement to send data with minimal size and in a GPU-friendly layout that can be easily accessed by the rendering hardware of the visualization engines. Chapter 7 will describe this GPU-optimized data representation in more detail.

4.1.1 Content-Aware Compression

The (geometric) data, which is either coming from the analytics module or directly from the database layer, is usually not optimized in terms of storage space. For example, surface meshes may be represented as soup of triangles. It is often possible to transform the geometric representation into a more efficient form. Triangle soups, e.g., may be reorganized into triangles strips [3], where vertices are reused and edges are defined implicitly. Another method would be to compute a higher-order surface representation, such as NURBS [2]. In this case, curved surfaces would be defined as spline patches instead of large number of triangles.

4.1.2 General Compression

Additionally, the graphics module can handle on-the-fly compression of the data, which reduces communication bandwidth. This can either be a lossy compression (e.g., by using a decreasing floating point accuracy [4]), or it can be a lossless entropy-encoding (e.g., fast ZIP variants such as LZO or LZ4).

5 Analytics Module

To post-process, i.e., analyze and inspect the results of simulations, scientists not only visualize the calculated results on the provided model mesh using color maps or vectors among others. They also extract information of the provided data using a variety of techniques, e.g., calculating the 0-level iso-surface of the air pressure from a sea waves simulation, or cutting through the air volume mesh with several planes, or calculating stream-lines (which show the velocity pattern of the air surrounding a racing car). Also surface meshes of the embedded objects and the surrounding box of volume meshes are calculated in the post-process step.

All these analysis and information extraction algorithms are part of the analytics module, which together with the graphics, query manager and monitoring modules form the Engine Layer of the VELaSSCo platform (see).

VELaSSCo queries (short VQueries) are queries based on the demands of the visualization client. They can be classified according to the type of expected graphical output:

- **0D queries**: Result values over single vertices of the mesh or points in the domain.
- **1D queries**: Results that can be expressed along a curve, like result evolution on a single point, result variation along a straight line or curve in the domain or stream-lines or particle traces.
- **2D queries**: Results that can be expressed by a 2-variate function or vector valued function, e.g., results over a section plane of the 3D domain, one or several iso-surfaces of a scalar or vector field represented in the 3D domain.
- **3D queries**: Results that can be expressed by a 3-variate function or vector valued function. In the project the only 3D query is the discrete to continuum

transformation that from 4D simulation data (3D + time-steps) generates a 3D FEM mesh with results.

- **Animated queries**: The above three query types, but along some or all time-steps of the analysis.

5.1.1 Model View Queries

Some of these types of queries retrieve mesh information directly from the simulation data and the information is passed to the graphics module for later forwarding to the visualization client.

Other ones may require some analytics. The calculation of the boundary mesh is an example of these queries: Given a volume mesh, the analytics module outputs a surface mesh of the embedded objects and the outer surface of the volume. This triangle or quadrilateral mesh is then passed to the graphics module for later forwarding to the visualization client. Eventually, this boundary mesh may be simplified depending on the client's graphical capabilities. If so, then a data structure with interpolation information is stored, so that user-demanded simulation results can be interpolated on the simplified model.

Another example of model view queries that require analytics is the calculation of cut-planes of 3D volume meshes. These queries generate a surface mesh with triangles where the cut plane goes through the volume elements. In the case of cut-lines or cut-segments, the elements generated are lines. These cut-queries will also generate data structures with information to help interpolate the simulation results the user requests to be visualized on the created cuts. These data structure with interpolation information may also reside in the GPU of the visualization clients so that no interpolation is done in the platform but directly on the user's GPU. The above mentioned interpolation information will only live in the platform as long as the session remains open. Once the user closes the session this information will be deleted. Usually, when the user issues a cut VQuery (cut-plane, line-graph, etc.) he also selects the result to be visualized on it, so the interpolation information can be used right away.

A performance evaluation of the queries should be done. If the cut VQueries are fast enough, storing the interpolation may be unnecessary and each time the user changes the result visualized over the same cut plane, the cut can be performed again with the result interpolated.

5.1.2 Result View Queries

As mentioned above, some of the result view queries may not require any analytics at all, such as "get results for vertices", which will generate a data query and the information is passed to the visualization client.

This information flow can be optimized though, as they refer to mesh information that is already present in the visualization client. For instance, after getting the boundary mesh of a volume model, the user may visualize the pressure distribution on it, and the

VQuery "get results for vertices" will return the result values for the vertices of the previously returned boundary mesh.

The discrete to continuum transformation query, which transforms a dynamic particle mesh with results into a static FEM mesh with results, will only output, and return to the user, the result of such a conversion. The next query performed by the client, after this one, will retrieve the "discrete to continuum" transformation mesh, eventually also its boundary mesh.

The "calculate iso-surface" query can be seen as a generalized version of the "calculate cut-plane" query. For a user-specified result value, a surface mesh is generated with the calculated triangles for the volume elements whose vertex result values are above and below this specified value. The "cut-plane" query is a particular case where the result values are the oriented distances of the vertices to the cut-plane and the iso-value is 0. As with the "cut-plane" query some information must be stored, in the platform or in the client-side, to interpolate future results on the same iso-surface, like visualizing the velocity field on the 0-pressure iso-surface. Iso-surfaces can be very complex and NURBS compression can be used to reduce the required bandwidth, although, in general, it is hard to guarantee an accurate representation of the shape.

One of the most complex queries to implement in this platform is the calculation of stream lines, which, given a seeding point, outputs a curve as collection of lines, tangent to the vector field, that passes through this point. Usually, the user will select several seeding point, less than 1,000. The vector field may have vortices, and thus the generated stream-lines can be very long. B-spline compression/simplification will reduce the amount information that travels between platform and visualization client, and using the GPU capabilities it will speed-up the rendering process.

In the particle simulations, particle traces are also useful and will also profit from the B-spline compression.

5.1.3 Animated Queries

Three types of animated sequences can be requested by the user:

1. Traveling through the model from one point to another, including zooming in and zooming out.
2. In particles simulations: visualize the mesh on some or all time-steps.
3. Animating a result visualization, for instance the sea-level iso-surface as shown in Figure 5.

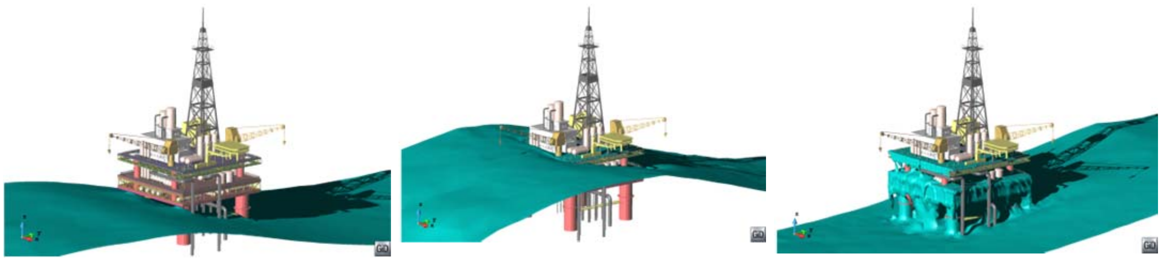


Figure 5. Example of different stages of an animation of the iso-surfaces, which represent the sea surface.

All these type of queries represent a stream of VQueries to the platform and results back to the visualization client.

In the first type of animated queries the client will send the world window (visualization bounding box) each time the user or the visualization client updates the position in the navigation query, and the platform will send back the new mesh information that enters in the provided world window. Eventually, this will represent also an update on the result query too, for instance, retrieve the values of the new vertices that enter in the world window or interpolate new result values.

In the second type of animated queries, when the user wants to visualize the mesh between two time-steps, the platform will access the simulation data to retrieve the mesh for each time-step between the two the user selected, and send this mesh information back to the visualization client. Eventually, this will represent also an update of the result query too, for instance, retrieve the result values for the new mesh, or perform again the cut-query as the domain mesh has changed.

An example of the third type of animated sequences is shown in Figure 5, where the image sequence corresponds to the simulation of the waves of the sea surface against an oil platform. The mesh representing the sea free-surface is not present in the simulation results. It is calculated from the distance field of the tetrahedral mesh. The 0-level iso-surface of the distance result is calculated and sent back to the visualization client at each time-step.

When a user defines a cut-plane or an iso-surface, he/she may want to visualize the effects of changing dynamically cut-plane parameters along an axis for instance, or to modify dynamically the result value to generate the iso-surface and visualize how it behaves. These are examples of the fourth type of animated sequences.

5.1.4 Model simplification

The query manager is responsible of receiving and managing the VELaSSCo queries (VQueries) and of ensuring feed-back to the user. For time consuming queries, the query manager will ask the analytics module for a simplified evaluation of the query, i.e., evaluate the query over a simplified version of the simulated data, while the query over the fully detailed model is passed to the batch module. If the simplified version does not exist, then the analytics module will create a simplified version of the model and store it for later use. This behavior is reflected in the workflow of.

To ensure a low latency and when the output of the queries of fully detailed models surpasses the graphical resources of the visualizations client (known at the beginning of the session) the outcome of the query will also simplified.

The volume simplification algorithm should preserve both the topology and the properties of the original mesh. For instance, it should avoid the generation of tetrahedrons with zero or negative volumes, new holes, or overlapped elements. Moreover, the boundary of the volume mesh and the interfaces between volumes of different materials should be preserved. Potentially interesting geometric details and attribute features such as discontinuities should be preserved, and the volume simplification process should be fast to avoid long waiting times.

If the process is too costly, then the usage of the algorithm may be limited to simplify the volume mesh only once, considering just the spatial information, but maintaining some data structure that allows for fast attribute interpolation at user request.

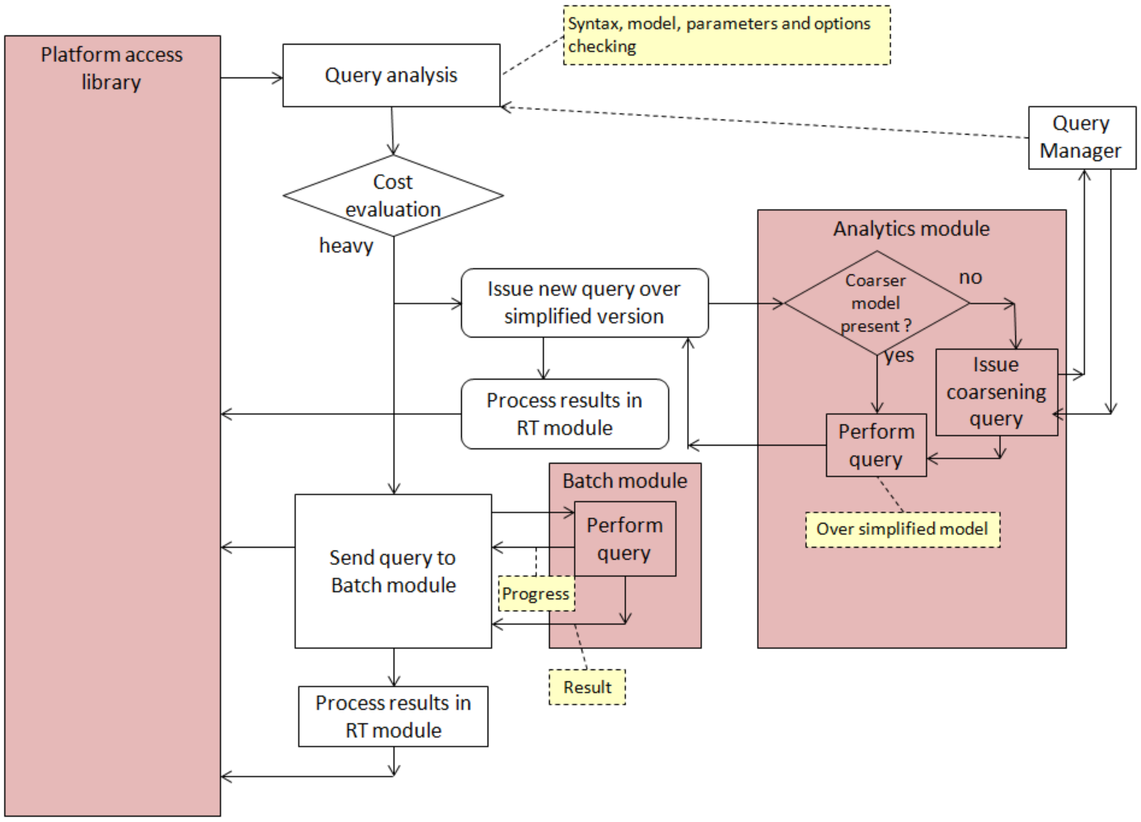


Figure 6. Workflow to process heavy VQueries, which will use simplified models.

The surface simplification algorithm with attributes will simplify large triangle meshes with attributes, which will be used on-demand, and it should be as fast as possible to ensure low response times for the user. The two possible use cases are:

- to simplify the mesh using only the spatial coordinates of the vertices and each time the user chooses a new attribute to visualize, it is interpolated on the simplified model,
- to simplify the mesh using both spatial coordinates and the user chosen attribute field.

6 Visualization Engines

As mentioned above, the visualization engines targeted in the VELaSSCo project are the GiD and iFX frameworks. In this chapter, we will provide a short overview over their architecture and capabilities.

6.1.1 GiD

GiD is a universal pre- and post-processor for numerical methods in engineering developed at CIMNE [6].

As a pre-processor, GiD has its own geometrical engine (actually, GiD is a CAD system itself), and several kinds of meshers able to generate meshes for different numerical methods, both for 2D and 3D cases. It also manages the simulation data, conditions, material properties, etc. by means of the 'Problem-Type' concept. The pre-processing operations involve the creation of the geometry, or the import of it from other software, the repairing and cleaning of it, and the generation of a suitable mesh for the calculation part. Furthermore, the data needed for the solver to compute the simulation must be assigned to the domain (either to the geometry or to the mesh).

As a post-processor, GiD can visualize the results coming from any kind of simulation using the most common post-processing and visualization techniques such as stream-lines, cuts, iso-surfaces, deformation, etc… Also 2D graphs can be processed from the 3D data, as well as animations of the results along different time steps or load cases.

The specification of the simulation conditions, material properties and the connection to the simulation solver is considered from the GiD point of view as a plug-in, called 'Problem-Type', to connect any solver to GiD, and take advantage of its full capabilities for pre- and post-processing. Thus, GiD is in charge of generating all the data for the simulation, send it to the solver, and receive the results from it to post-process and visualize them as shown in Figure 7.

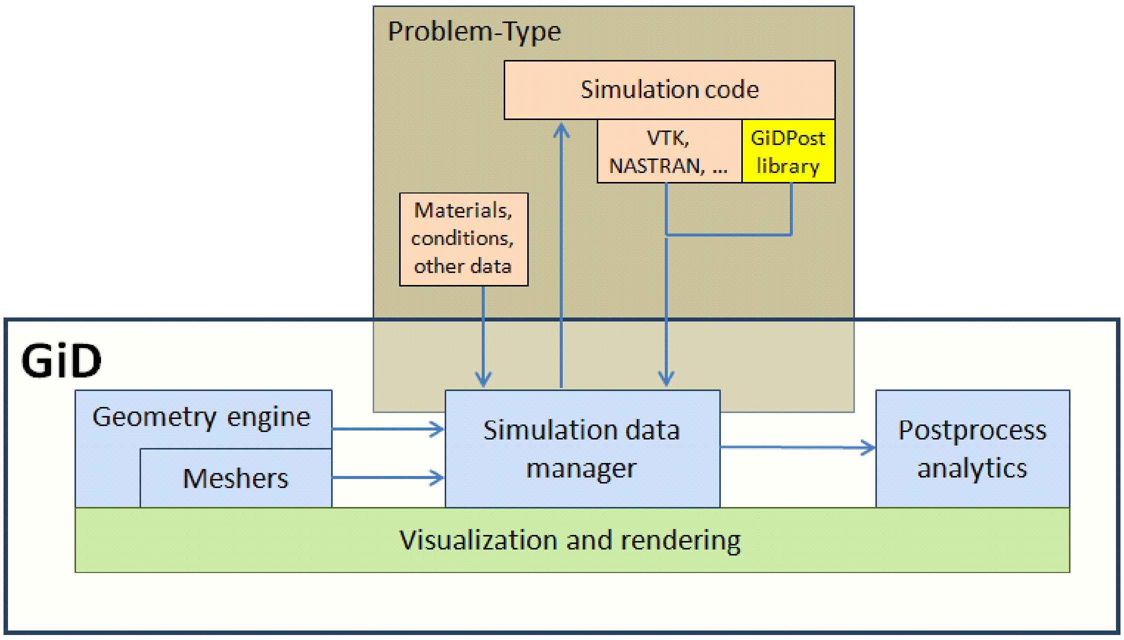


Figure 7. GiD architecture.

Both pre- and post-processing parts of GiD uses advanced visualization techniques in order to visualize the geometry or the mesh of the model, as well as all the results desired in the different available modes. Among the main visualization features present in GiD are: 3D stereoscopic view (both with anaglyphic mode and hardware quad buffers), shadows treatment or different perspective settings.

6.1.1.1 Architecture

The different parts of GiD concerning its architecture (depicted in Figure 7) are explained hereafter.

- **Geometry engine**. The geometry engine of GiD has been entirely developed at CIMNE, and is programmed in C++. All the geometrical entities, trimmed NURBS surfaces, Coon surfaces, etc., can be visualized in different modes: normal (with specific colors for the different geometrical entities: points, lines, surfaces and volumes), render flat or render smooth, depending on the smoothness of normal performed in the adjacent entities for rendering purposes.
- **Meshers**. The meshing layer of GiD consists in different meshing algorithms (based on Delaunay, advancing front or octree), some of them developed at CIMNE, and some others taken from third parts. Different kinds of meshes can be generated depending on the simulation requirements: Cartesian, structured, semi-structured or unstructured ones. Different element types are supported: line elements, triangles, quadrilaterals, circles, spheres, tetrahedra, prisms and hexahedra. These elements can be linear or different kinds of quadratic.
- **Simulation data management**. This layer is in charge of dealing with the simulation specific data to be assigned to the geometry (or the mesh) and which must be transferred to the solver. When GiD is to be used for a particular type of analysis, it is necessary to predefine all the information required from the user, like materials and conditions, and to define the way the final information is given to the solver module. Each solver must define, using a specific template, the way it wants to receive from GiD all the data needed for the simulation. Some of the parts of this layer are programmed in C++ and other ones in the Tcl scripting language. GiD provides with a wide range of Tcl commands for accessing the information contained in the model by any third software.
- **Post-processing operations**. This layer is in charge of performing the wide range of post-processing operations that can be made using GiD. Among others, it can generate stream-lines (when the result is a vector field), iso-surfaces, cuts, deformations, graphs or animations. The implementation of this layer allows GiD to display at the same time, and using different visualization options, different results onto the same model.
- **Visualization and rendering**. This layer is in charge of visualizing the model (in geometry or mesh mode inside the pre-processing part of GiD), and with the results integrated following different visualization modes (in the post-processing part of GiD). Commonly, the post-processing part of GiD is dealing with meshes to define the model, but nowadays GiD can also visualize results directly onto geometry, which is a requirement for Iso-Geometric Analysis (IGA) based codes.

- **GiDPost library**. The GiDPost library is an API for writing the results in binary GiD format. This library may be used for the simulation codes to write their results, but they also can write them by their own based on the documentation. This library is programmed in C++ and has a Fortran interface.
- **I/O.** This layer is in charge of writing and reading the models of GiD, as well as their properties assigned for the simulation, and the possible results coming from the solver.
- **GUI**. The Graphical User Interface (GUI) of GiD is written in the Tcl/Tk scripting language. It has different bars to for the user interaction such as the geometry or the postprocessing toolbars and the menu bars among others. The GUI can also be modified by the module connected to GiD (the solver) by adding new windows and menu entries or removing them. All the features of GiD can also be accessed via command line, or via batch file.

6.1.2 iFX

iFX [7] from Fraunhofer is a post-processing and visualization framework that has been designed for the engineer's graphic workstation. The objective is to provide best-in-class solutions for specific customer needs with the following aims in mind: leveraging GPU hardware (GPU shaders) and spatial acceleration structures, efficient visual analysis and goal orientation, and visual quality realized by topology-independent interpolation schemes.

iFX is built upon the Rapid Prototyping Environment (RPE) developed at Fraunhofer IGD. The design goals for RPE stem from the need to support a wide variety of research and industrial activities, particularly in areas such as geometry processing, simulation, rendering and scientific visualization. More precisely, the main design intention was to provide a common development environment for the mentioned scenarios, which facilitates rapid prototyping of applications, avoids code duplication, is easy to learn for new developers and fosters joint development of in-house staff and external project partners.

6.1.2.1 Architecture

The structure of an RPE-based iFX application is shown in Figure 8. The central approach of RPE is to encapsulate high-level features in dynamically loadable modules, i.e., plugins. Plugins enable the support of diverse functionalities, most notably these include:

- **Rendering methods.** This can be all kinds of different rendering algorithms, e.g., photorealistic rendering, volumetric rendering, or point rendering.
- **Interaction methods.** Not only various input and output devices (such as space mice or head-mounted displays) are supported, also multiple interaction metaphors and manipulators can be added.
- **Data import/export and processing methods.** A multitude of file and data formats can be read, written and processed. This can either include geometrically oriented formats such as VRML/X3D, or formats providing additional model information (e.g., vector fields) such as VTK.

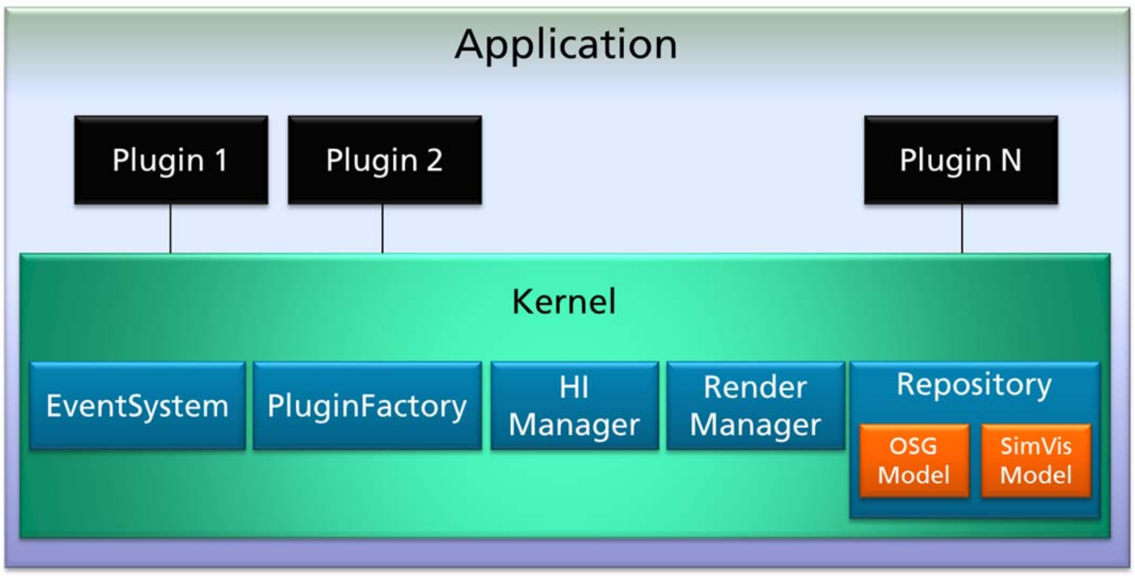


Figure 8. Architecture of an iFX/RPE application.

Plugins that extend the functionality of RPE make use of a core system (kernel), which integrates a number of fundamental sub-systems:

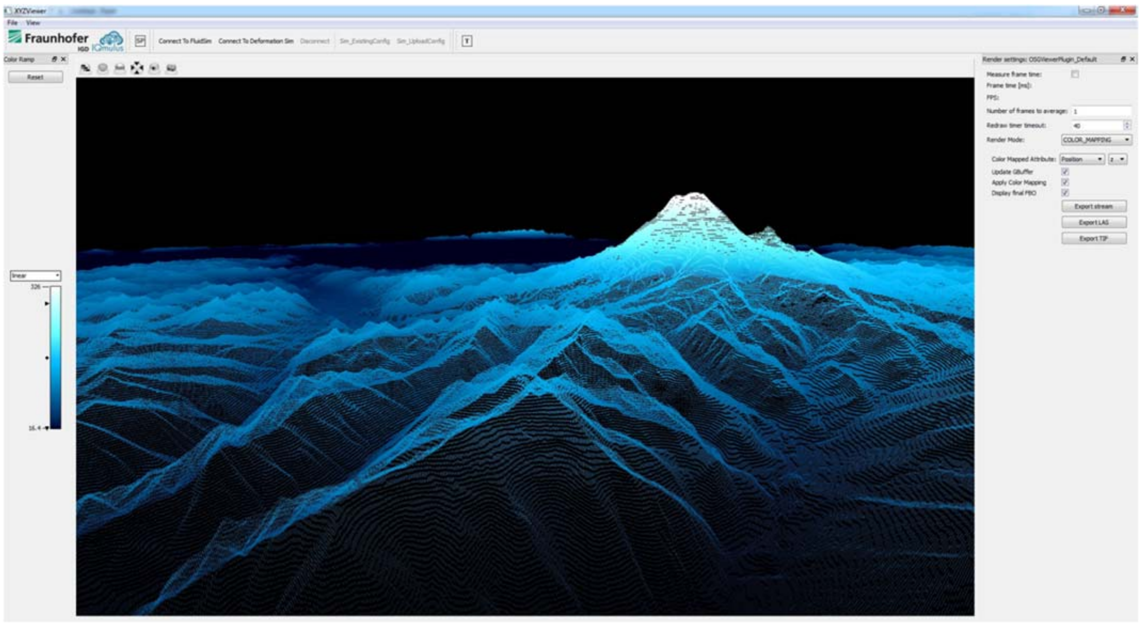
- **Event system.** This sub-system is responsible for system-wide (inter-module) communication.
- **Plugin factory.** Plugins are instantiated by the plugin factory, which acts as a portable module loader and registry.
- **Human interface manager.** Hardware and drivers of input/output devices are coordinated the human interface manager. It is also responsible for mapping input user action to suitable actions (e.g., mapping mouse movement to navigation schemes).
- **Render manager.** This component manages different types of render pipelines, which define specific rendering methods.
- **Repository.** The central data repository administrates centralized and synchronized data storage. (For example, geometric data is stored in the OpenSG (OSG) [8] model, and simulation data in the SimVis model.)

An iFX application that utilizes RPE is typically conceived as a stand-alone executable that links to RPE and all needed third-party-library. Depending on the required functionality, a number of plugins are loaded providing specific algorithms and data structures.

6.1.2.2 Render Pipelines

An important part of RPE is the possibility to extend its visualization capabilities by exploiting programmable render pipelines. A render pipeline is an approach to expose a variety of rendering methods to the system. It realizes a particular rendering method by means of stage based rendering, where multiple render passes are executed and the individual results are combined into a final image.

An example of the output of a render pipeline can be seen in Figure 9. Here, a large point (particle) cloud, which has been collected by laser scanning a terrain, is color mapped based on the altitude of sample points.



Render settings: OSGViewerPlugin_Default
Measure frame time: ☐
Frame time [ms]:
FPS:
Number of frames to average: 1
Redraw timer timeout: 40
Render Mode: COLOR_MAPPING
Color Mapped Attribute: Position z
Update GBuffer ✓
Apply Color Mapping ✓
Display final FBO ✓

Figure 9. Deferred color mapping render pipeline.

In this scene, the dataset of the point cloud is very large. To make real-time adjustment of the mapping parameters possible, color mapping is realized with deferred shading: Points and their attributes are first drawn into a deep buffer. Then the actual coloring is performed in an additional stage as a post process.

7 GPU-Optimized Data Representation

As shown in the previous sections, a visualization client makes use of the VELaSSCo platform access library to send queries to the Query Manager, and to receive information based on the results of a query. Data that is sent back to the visualization client needs to be formatted in a specific way that enables the visualization engine to process the content efficiently.

This chapter provides an initial design of a possible format. Based on the experience gained during the course of the VELaSSCo project, this format will evolve over time and adapt to the supported usage scenarios. Ideally, such a format should be able to handle a wide variety of use cases, but also enable real-time performance. Unfortunately, typically there is a tension between generality and efficiency. To be able to get the best of both worlds, it is proposed to support actually two formats: an established standard format, and additionally a highly specialized internal format.

7.1 General Visualization Format

In cases where real-time performance is not required, i.e., result data is sent only occasionally and can be fetched as a single dataset, it may be beneficial to use a standard format. One candidate for such a format could be the VTK format [1]. It has the following properties:

- **XML encoding.** This enables the use of standard XML parsers. The file structure is human readable and extensible.
- **Binary data encapsulation.** Payload data can be represented in binary form, which reduces the amount of data to be transferred.
- **Support for parallel reading.** The dataset is broken into pieces, which can be read and imported in parallel. This also facilitates streaming.

Apart from the technical properties, there are some practical advantages:

- **Support for a variety of different data types.** For example, image data, polygonal data, or unstructured grids.
- **Open source.** Reader and writer code is readily available and does not need to be re-implemented and tested.
- **Widely used.** This would help the adoption of the VELaSSCo access library and enable the VELaSSCo platform to more easily connect to other visualization tools besides GiD and iFX.

So far, the VTK format does **not** support higher order curves and surfaces other than quadratic types. However, to support NURBS [2] or other higher order objects, we can still use VTK's capability of handling vertices to transfer control points. Since file metadata is in XML, it is possible to add tags that would identify the data as NURBS. (Of course, this would not be recognized by standard VTK readers.)

For high performance visualization, the VTK format, or any other open format, is most probably too inefficient as it still requires some minimal parsing, conversion, and interpretation of data. For such cases a highly specialized internal format is better suited. The remainder of this document describes the proposed structure for it.

7.2 Specialized Visualization Format

Unlike a general format, an internal VELaSSCo format aims for high performance, so that is applicable in real-time scenarios where information should be transferred at high speed with minimal latencies. The current proposal is guided by the following design requirements:

- **Minimal data size.** In order to avoid spending time for parsing and conversion, the amount of control data, such as headers, separators and meta-data, should be minimized.
- **Binary data.** Since data has to be transferred over a network, and also to avoid conversion from an ASCII representation, the payload data should be in binary form.
- **Processing friendly layout.** Ideally, all data should be in a form, where is can directly be accessed by processing units without any conversion. It should be possible to load the payload data into buffers that are directly uploaded to the GPU. While the GPU is the typical target, this also applies to CPUs.

Consequently, this means that the data has to be prepared in a form suitable for the target architectures as there will be no marshalling. Therefore, data type sizes and byte order need to match the target specification (in practice, this would be standard C data types with little endian byte order).

In this format, a batch of data is represented as a *file*. A file is a self-contained binary blob of data that includes all the information to display a scene. Such a file itself consists of two sections, a *header* section and a *data* section. The header contains some sort of description of how the payload is handled. The data section is simply a collection of buffers. Within a C/C++ program a file could just be accessed as the struct shown in Figure 10.

```
struct File
{
  // file information
  Header header;

  // actual data;
  Data data;
};
```

Figure 10. File structure.

Figure 11 shows a C/C++ code example of the header section. The header has a fixed size of `sizeof(Header)` bytes and can be loaded directly into a Header object. This structure consists of three parts. The first part identifies the file as a VELaSSCo data structure with an 8 byte ASCII UTF-8 string, followed by an integer version number.

```
struct Header
{
  // identification as VELaSSCo data
  uint8_t  magic[8];               // magic number ("VELaSSCo", UTF-8)
  uint32_t version;                // version number (100 = V1.00)

  // information about the content
  uint64_t descriptionBytes;       // size of description block in bytes
  uint64_t metaBytes;              // size of meta block in bytes

  // sizes of the contained buffers
  uint64_t vertexDefinitionsBytes; // size of vertex definition buffer in bytes
  uint64_t vertexAttributesBytes;  // size of vertex attribute buffer in bytes
  uint64_t edgeDefinitionsBytes;   // size of edge definition buffer in bytes
  uint64_t edgeAttributesBytes;    // size of edge attribute buffer in bytes
  uint64_t faceDefinitionsBytes;   // size of face definition buffer in bytes
  uint64_t faceAttributesBytes;    // size of face attribute buffer in bytes
  uint64_t cellDefinitionsBytes;   // size of cell definition buffer in bytes
  uint64_t cellAttributesBytes;    // size of cell attribute buffer in bytes
};
```

Figure 11. Header structure.

All following fields are simply buffer sizes. These allow the client to compute the total amount of memory that needs to be reserved to hold the actual data. Size values are provided individually for each buffer, therefore it is possible to allocate separated memory blocks if this is beneficial. The second part of the header holds the block sizes for semantic data that describes how the content buffers are to be interpreted. The third part includes the buffers sizes for the actual content buffers. These buffers will be described below. After reading the header, the client loads the rest of the file into the allocated memory. Of course, one can just sum up the buffer sizes, allocate a single contiguous buffer and just read the rest of the file into that buffer.

7.2.1 Data Section

After the header the actual data follows. Again, this is just a binary blob that contains all the buffers needed by the visualization client. Within a C/C++ program the data could be represented as shown in . Note that there are only pointers, which store the addresses of the respective buffers in memory. The data section file on disk would contain the concatenated buffer content.

```
struct Data
{
  // description of file content
  uint8_t* description;

  // additional data
  uint8_t* meta;

  // vertex block
  uint8_t* vertexDefinitions;
  uint8_t* vertexAttributes;

  // edge block
  uint8_t* edgeDefinitions;
  uint8_t* edgeAttributes;

  // face block
  uint8_t* faceDefinitions;
  uint8_t* faceAttributes;

  // cell block
  uint8_t* cellDefinitions;
  uint8_t* cellAttributes;
};
```

Figure 12. Data structure.

7.2.1.1 Description Buffer

The *description buffer* is meant to contain a semantic description of the data in the following buffers. Simply speaking, this is all the information the client needs to know to interpret the sent data. This would include high-level specifications, such as indicating that the data is an iso-surface, but would also include low-level specifications, such as data type sizes, e.g., whether to use float or double types.

Given that there can be countless cases one might want to handle, this document will no define the format of this field. It will be determined in the course of the VELaSSCo project how the description field should best look like. Ideally, the description should be in ASCII form, so that it would be human readable. A straightforward method would be to just have a list of key/value pairs. In the simplest case, the description field could contain a single value as code that identifies the following data. In a later section in this document some possible examples are presented.

The *meta buffer* is merely intended as some sort of reserve storage, where all additional information is stored that is not represented by the content buffers that follow. For example, this could be thumbnail images, statistics data, or other annotations.

7.2.1.2 Content buffers

Finally, the payload data is stored in *content buffers*. These buffers serve as representation of geometry plus associated attribute data. The buffers are organized as pairs, where each pair defines a set of specific primitives. The first buffer (*definition buffer*) of a pair holds a geometric definition, while the second buffer (*attribute buffer*) holds attribute data associated with the primitives. The intention is that the geometry buffers ideally map directly to OpenGL buffers, whereas the attribute buffer keep data that is related to simulation results, e.g., pressure, velocity vectors, or particle radii. In a file there is technically only one pair of buffers per primitive, however, internally they could be composed of several concatenated buffers. For example, a vertex position and a vertex normal buffer could be combined into one vertex definition buffer.

Currently, there are four content buffer pairs (*blocks*). The idea is that each block relates to a geometric combination of the previous pair. The four primitive blocks are: *vertex block*, *edge block*, *face block*, and *cell block*. Edges connect vertices, faces are composed of edges, and a number of faces make up a cell. Depending on the use case, this can be simplified. For example, in order to define a triangle strip one would only need vertices and edges. For defining an unstructured grid, only vertices and cells would be required.

7.2.2 Examples

In this section a few examples are presented to show how the proposed buffer structure might be used to cover a number of uses cases that will be common in the VELaSSCo project. As mentioned before, there are countless possible combinations, and how to specifically encode a certain case will be determined throughout the project.

7.2.2.1 Iso-Surface

The first example shows how to encode an iso-surface as a set of triangle strips. In this example, vertex data is interleaved, i.e., position, color, and normal data are not placed into several buffers, but are combined into groups. All vertex data is stored in the vertexDefinitions buffer. The edgesDefinitions buffer holds the index lists that combine the vertices to strips. Each strip is terminated by an end marker (here -1). These two buffers can directly be employed as OpenGL buffers when rendering GL_TRIANGLE_STRIPS. OpenGL also allows the user to define an arbitrary restart index. In addition, the file also uses the attribute buffer. Here, it is used to store one pressure value per vertex.

There is currently no definition for the description field. This example shows how one could look. Typically, this field needs to provide all the information to set up the required OpenGL context. For standard VELaSSCo use cases that happen often, a single keyword may also be sufficient, like the ISOSURFACE_OGL_TRIANGLE_STRIP identifier used here. It is important to note that modern OpenGL provides much flexibility in the data layout. Whether or not it is better to use interleaving or just concatenate position, color, and normal buffer is yet to be researched.

```
//
// VELaSSCo real-time file format example
//
// Iso-surface with per vertex pressure values encoded as OpenGL
// triangle strips with vertex interleaving.
//

#include "RealTimeFormat.hpp"

using namespace VELaSSCo;
using namespace RTFormat;

void WriteFile(const char filename[], File file)
{
  // ...
}

main()
{
  File file;

  //
  // Content definition -----------------------------------------------------
  //

  // vertex definitions
  static float vertices[] =
    {
      // vertex 0
      0.0f, 0.0f, 0.0f, // position
      1.0f, 1.0f, 1.0f, // color
      0.0f, 0.0f, 0.1f  // normal

      // vertex 1
      // ...

      // vertex n
      // ...
    };

  // edge definitions
  static int edges[] =
    {
      // strip 0
      0, 4, 1, 5, 2, -1, // indices + restart

      // strip 1
      4, 8, 5, 9, 6, -1, // indices + restart

      // strip 2
      // ...

      // strip n
      8, 9, 7, 2, 3      // indices, no restart
    };
  // vertex attributes
  static float attributes[] =
    {
      // pressure 0
      3.14f,

      // pressure 1
      6.28f

      // pressure 2
      // ...

      // pressure n
      // ...
    };

  //
  // Description definition ---------------------------------------------------
  //

  static char description[] =
    "# Iso-surface with per vertex pressure values encoded as OpenGL \n"
    "# triangle strips with vertex interleaving.                     \n"
    "                                                                \n"
    "ISOSURFACE_OGL_TRIANGLE_STRIP                                   \n"
    "                                                                \n"
    "VertexDefinitions     = position, color, normal                 \n"
    "VertexDefinitionsType = float                                   \n"
    "VertexAttributes      = pressure                                \n"
    "VertexAttributesType  = float                                   \n"
    "                                                                \n"
    "OGLPrimitiveRestartIndex = -1                                   \n";

  //
  // Header definition --------------------------------------------------------
  //

  // magic
  file.header.magic[0] = 'V';
  file.header.magic[1] = 'E';
  file.header.magic[2] = 'L';
  file.header.magic[3] = 'a';
  file.header.magic[4] = 'S';
  file.header.magic[5] = 'S';
  file.header.magic[6] = 'C';
  file.header.magic[7] = 'o';

  // version
  file.header.version = 100;

  // description
  file.header.descriptionBytes = sizeof(description);

  // meta
  file.header.metaBytes = 0;

  // buffers
  file.header.vertexDefinitionsBytes = sizeof(vertices);
  file.header.vertexAttributesBytes  = sizeof(attributes);
  file.header.edgeDefinitionsBytes   = sizeof(edges);
  file.header.edgeAttributesBytes    = 0;
  file.header.faceDefinitionsBytes   = 0;
  file.header.faceAttributesBytes    = 0;
  file.header.cellDefinitionsBytes   = 0;
  file.header.cellAttributesBytes    = 0;

  //
  // Data definition ----------------------------------------------------------
  //

  file.data.description = (uint8_t*)description;
  file.data.meta        = 0;

  file.data.vertexDefinitions = (uint8_t*)vertices;
  file.data.vertexAttributes  = (uint8_t*)attributes;
  file.data.edgeDefinitions   = (uint8_t*)edges;
  file.data.edgeAttributes    = 0;
  file.data.faceDefinitions   = 0;
  file.data.faceAttributes    = 0;
  file.data.cellDefinitions   = 0;
  file.data.cellAttributes    = 0;

  //
  // Write file ---------------------------------------------------------------
  //

  WriteFile("TestFile", file);
}
```

7.2.2.2 Volume Data

Similar to surfaces, it is possible to define volumetric data. A simple definition would be very similar to the triangle strip layout. The vertexDefintions buffer would hold the vertex positions. Cells would be defined in the same way as triangle trips, i.e., edgeDefinitions stores index lists with termination markers that show where a cell definition ends. If simulation data is attributed to cells (e.g., velocity vectors), it could be stored in the cellAttributes buffer.

7.2.2.3 NURBS

Obviously, NURBS [2] can also be defined in a similar way, although they require more parameters to be defined. NURBS points and weights could be kept in the vertex buffer, knots and connectivity information in the face or edge buffers. Like with other surfaces, the description buffer would contain the necessary information to let the client know how to interpret the data. Eventually, these buffers are uploaded to the GPU as buffer objects, from where they can be accessed by vertex and tessellation shaders, which generate the output primitives.

7.2.3 Progressive Visualization

In the current form, this format does not explicitly support streaming or level-of-detail display. The intention is that in such cases data is not sent as a single file, but rather as a stream of files (which are transferred one after another). Later, files could either contain new data that replaces old data (e.g., a new set of triangle strips), but they could also provide incremental data. For example, in case of progressive meshes, new files could add new vertices and edges that are used to expand available geometry to a new detail level. How new data is to be interpreted could be defined in the description Buffer.

8 References

[1] VTK file formats. http://www.vtk.org/VTK/img/file-formats.pdf

[2] Les Piegl, *The NURBS book*, Springer 2013.

[3] Michael Deering, *Geometry Compression,* Computer Graphics (Proceedings of ACM SIGGRAPH), 1995.

[4] Peter Lindstrom, *Fixed-Rate Compressed Floating-Point Arrays*, IEEE Transactions on Visualization and Computer Graphics (Proceedings of Visualization 2014), 2014.

[5] Apache Thrift. https://thrift.apache.org

[6] GiD. http://www.gidhome.com

[7] iFX. http://www.i-fx.net

[8] OpenSG. http://www.opensg.org

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator, Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC 2013.