University of Cagliari

PhD School of Mathematics
and Computer Science

xxviii cycle

Course in Computer Science

Distributed Processing of Large Triangle Meshes

INF/01

Supervisors:
Prof. Riccardo Scateni
Dr. Marco Attene

Author:
Daniela Cabiddu

PhD Coordinator:
Prof. G. Michele Pinna

2014 - 2015

Quello che non ho è un orologio avanti
per correre più in fretta e avervi più distanti
quello che non ho è un treno arrugginito
che mi riporti indietro da dove sono partito.
– *Fabrizio De Andrè*

Acknowledgements

There are a number of people without whom this work might not have been done and to whom I am greatly grateful.

To Marco ATTENE, whose constant guidance, encouragement and support have been key factors for the development of this work.

To Riccardo SCATENI, who introduced me to geometry processing research, started me down this road and encouraged me to purse my goals.

To Bianca FALCIDIENO, Michele PINNA, and Michela SPAGNUOLO, who opened this door of opportunities and gave me the strength to persist and succeed.

To the project "Tecniche di visualizzazione avanzata di immagini e dati 3D in ambito biomedicale", the European projects VISIONAIR and IQMULUS, and the international joint project "Mesh Repairing for 3D Printing Applications", who partially supported this work. To all the project partners, who got me involved in their activities and helpful discussions.

To all the colleagues at CNR-IMATI in Genova, who welcomed with open arms and helped me to grow professionally and as a person.

To all the colleagues at the Computer Graphics Lab (aka BATCAVE) in Cagliari, who kept in touch with me and succeeded in supporting me in spite of the distance.

To my FAMILY and my FRIENDS, who supported me although they are still wondering what I am actually doing and why I love my job.

THANK YOU!

Abstract

Thanks to modern high-resolution acquisition techniques, 3D digital representations of real objects are easily made of millions, or even billions, of elements. Processing and analysing such large datasets is often a non trivial task, due to specific software and hardware requirements.
Our system allows to process large triangle meshes by exploiting nothing more than a standard Web browser. A graphical interface allows to select among available algorithms and to stack them into complex pipelines, while a central engine manages the overall execution by exploiting both hardware and software provided by a distributed network of servers.
As an additional feature, our system allows to store workflows and to make them publicly available. A semantic-driven search mechanism is provided to allow the retrieval of specific workflows.
Besides the technological contribution, an innovative mesh transfer protocol avoids possible bottlenecks during the transmission of data across scattered servers. Also, distributed parallel processing is enabled thanks to an innovative divide and conquer approach. A simplification algorithm based on this paradigm proves that the overhead due to data transmission is negligible.

Sommario

Le nuove tecnologie di acquisizione permettono la ricostruzione digitale ad alta risoluzione di oggetti reali. I modelli 3D generati sono spesso composti da milioni, a volte miliardi, di elementi geometrici. La loro elaborazione richiede l'utilizzo di hardware e software specifici, spesso non disponibili.
Il nostro sistema permette di elaborare mesh triangolari di grandi dimensioni attraverso l'uso di un tradizionale browser Web. L'interfaccia utente consente di selezionare uno o più algoritmi e costruire pipeline complesse. Un motore centrale gestisce l'esecuzione sfruttando l'hardware e il sofwate messo a disposizione da una rete distribuita di server.
In aggiunta, il nostro sistema permette di archiviare le pipeline generate e renderele publicamente disponibili. Un meccanismo di ricerca semantica supporta la ricerca di pipeline che soddisfano particolari requisiti.
Oltre al contributo tecnologico, un protocollo di trasferimento per mesh triangolari limita il verificarsi di possibili colli di bottiglia sulla rete durante l'esecuzione delle pipeline. Elaborazioni parallele sulla rete distribuita sono inoltre rese possibili grazie ad un metodo innovativo di tipo divide et impera. Un algoritmo di semplificazione per mesh di grandi dimensioni sfrutta tale approccio e dimostra che il costo aggiuntivo di trasmissione dei dati sulla rete è trascurabile.

Contents

List of Figures

List of Tables

Part I

Introduction

1 Introduction

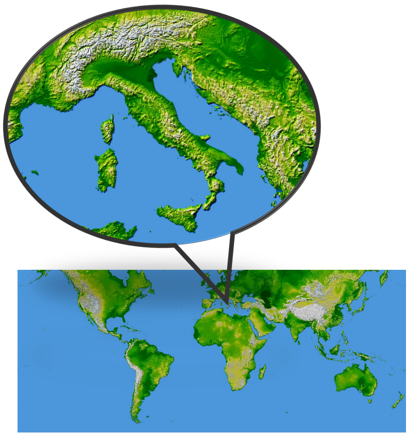
"Big Data is a vague term, used loosely, if often, these days. But put simply, the catchall phrase means three things. First, it is a bundle of technologies. Second, it is a potential revolution in measurement. And third, it is a point of view, or philosophy, about how decisions will be – and perhaps should be – made in the future."
– Steve Lohr, [Loh13]

Nowadays, the evolution of 3D data acquisition techniques provides fast and efficient means for generating extremely detailed digital representations of real objects in diverse industrial and research areas such as design, geology, archaeology, medicine and entertainment. Thanks to modern 3D scanners and their capability of acquiring data at very high resolution, both very small and very large objects can be digitally reconstructed by capturing their geometrical information from the reality.

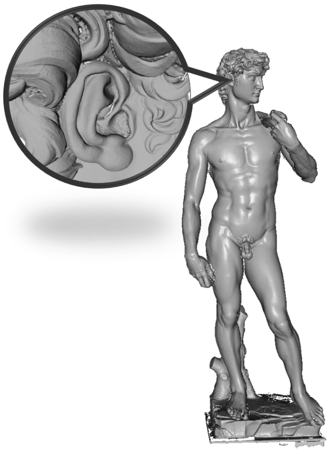
As an example, modern radar and lidar technologies allow to acquire both terrains and urban areas. Radar technologies transmit radio waves or microwaves and process their reflection to detect any object in their path, while airborne and stationary lidar allow to scan real scenes at very high resolution by illuminating a target with a laser and analyzing the reflected light. Both technologies allow to detect subtle topographic features (eg. river terraces and river channel banks, land-surface elevation beneath the vegetation canopy, ...) and have enabled the possibility to generate high-resolution digital elevation maps (DEMs) representing the acquired areas (Figure 1.1a) and to analyse them in order to retrieve useful information.

Another example is the generation of long-term digital archives of cultural artifacts. To achieve the goal, modern laser rangefinder technologies are exploited to reliably and accurately digitize the external shape and surface characteristics of many sculptures and architectures (Figure 1.1b).

Digital 3D models generated in both application areas (eg. original raw scans and their elaborations) are easily made of millions, or even billions, of geometric elements and huge disk memory is required to store them. Moreover, the limited RAM available on commodity PCs makes the processing and analysis of these models a non trivial task.



(a) *An image of the world generated using acquired elevation data (resolution is ≈ 1000m) [Fea07].*



(b) *A full-resolution 3D model of Michelangelo's David (≈ 900M points) [mic09]*

Figure 1.1: *Examples of high resolution digital models representing scanned objects from different application domains.*

To support and allow the analysis and processing of 3D models coming from different application areas, new geometry processing algorithms and methods are continuously being developed on the top of state-of-the-art previous works. Traditional algorithms require the input to be small enough to be completely loaded into main memory and sequential approaches are followed to perform the task. When large datasets appeared that could not fit into main memory, existing approaches needed to be redesigned in order to support these kind of inputs.

Out-of-core approaches are usually exploited to be able to deal with large-size inputs. Many of these methods subdivide the input into subparts, each of them sufficiently small to be processed with traditional incore approaches. In some cases the input can be partitioned using an incore algorithm: this is appropriate when memory is enough to store the model, but no further space is available to host all the support data structures necessary for elaboration which are often more memory-demanding than the input itself. Conversely, when even the plain mesh is too large, out-of-core partitioning is required to produce the sub-meshes.

An important aspect that should be taken into account when designing methods for processing large input datasets is efficiency. Typically, multi-core technologies and parallel approaches are exploited to accelerate the process and reduce the overall elaboration time. Multi-core architectures provide the possibility to process different subparts of the input simultaneously, but the available memory is shared among the processes and I/O operations are sequential in any case.

Nowadays, the well-known client/server model allows to distribute the computation on different machines that may be geographically scattered and communicate through a traditional Internet connection. This kind of architecture assumes that several servers are available, each of them exposing one or more remote services. The client is responsible of receiving the input from the user, managing the remote execution by properly invoking the available services one after the other, and finally returning the output. Web service technologies already provide the possibility to efficiently perform remote computations in many life science areas, but they are scarcely exploited in geometry processing since the transmission of very large inputs would represent a bottleneck and slow the elaboration down.

Summarizing, geometry processing is a mature research area where the state of the art brings together diverse solutions that allow to process meshes through incore approaches. Conversely, only a few algorithms are suitable for managing huge 3D geometric data that do not fit into main memory and specific hardware requirements need to be satisfied in order to run them efficiently. Finally, the possibility to exploit geographically distributed environments is scarcely considered in geometry processing literature.

1.1 Motivation

Researchers in geometry processing need to invent new algorithms and to fairly compare them against previous works, and such a need calls for approaches to share shape models and algorithms. In order to fulfill their needs, available source codes are recompiled on client machines where geometry processing libraries and tools are installed. Furthermore, algorithms that are not available need to be reimplemented from paper descriptions. Due to the required software and hardware compatibilities and to the complexity of algorithms described in scientific articles, these easily become costly and error prone operations.

Other than designing and implementing innovative algorithms, running experiments is a fundamental activity in geometry processing research. A typical experiment in this area considers an input data set, performs a sequence of operations on it, and analyzes the results. Sometimes a fixed sequence of operations is used to process a variety of data sets, whereas some other times the operation list is slightly changed while keeping the input constant. Proper pipelines of geometric algorithms to analyse and process digitized models have been defined in the literature and can be implemented as automatic processing workflows. Mesh processing and editing software tools allow to interactively edit a mesh, save the sequential list of executed operations and locally re-execute the workflow from their user interfaces. In most cases, the entire pipeline can take place in a completely automatic manner, that is, without user intervention. However, some workflows may need to iterate the execution of one or more basic algorithms to converge to an eventual clean result and user interaction may be required between sequential steps. Automatic re-execution of these pipelines on different input datasets are not always possible.

In the last decades we have assisted to an impressive growth of online repositories of 3D shapes and geometric software tools that allow researchers to share their input datasets and results of their experiments. Current repositories store the same shape model in several different versions (e.g. original raw scans, cleaned mesh, simplified mesh, remeshing, ...). This approach appears to be unsustainable since nowadays a single mesh can easily be made of millions of triangles that require significant storage resources, and for each model a repository may be called to expose the results of numerous algorithms in the long term. In view of a more intensive and collaborative use of the repositories to compare research works, innovative technologies must be provided, aimed to save as much storage as possible.

1.2 Objective

The main objective of this thesis is to provide an innovative tool to support geometry processing and sharing for large 3D models to make them independent of local hardware and software requirements. Our aim is to devise a Web-based platform that allows to run geometry processing pipelines on arbitrarily large inputs by using only a standard Web browser.

In our reference scenario, the user wants to analyse and elaborate a very large input mesh by performing a set of geometry processing operations and to make the result available for any other user. The input mesh is stored on the local disk of a commodity PC with no assumption on memory capacity or computational performance. The user should be allowed to access the system through a traditional Web browser, upload the input mesh, select the desired list of processing operations, ask the system to run the pipeline and finally download the result. Specifically, the user should be allowed to both create new workflows from scratch and to reuse existing workflows by simply selecting them from the graphical user interface. Non-expert users (eg. researchers in any field other than geometry processing) should be able to exploit the system to understand which tools and operations better fit with the specific purpose, and to define a workflow that allows to achieve the goal. Since understanding which software better fits with the specific purposes is frequently difficult for non-experts, the system should provide a high level description of available pipelines to guide the user to retrieve existing workflows or to define new processes. When the input mesh has been uploaded and the desired workflow has been defined, the user should simply ask the system to run the pipeline. At the end of the elaboration, the output should be available to the user for the download.

1.3 Impact and Applications

The advantages of such a system are diverse. First, researchers in the area of geometry processing can reuse simple geometric algorithms provided by the system, can stack them to construct workflows and can exploit them for extension or comparison purposes. Second, an online repository endowed with our system allows to rerun experiments on the fly from any location without the need to locally satisfy any software or hardware requirement. Therefore, results of short-lasting experiments can be recomputed on the fly when needed and there is no more need to keep them explicitly stored. Since experiments can be efficiently encoded as a list of operations, sharing them instead of output models sensibly reduces required storage resources. Finally, thanks to the high level documentation, researchers and practitioners in other fields (eg. mathematics, physics, biomedical imaging, ...) can exploit the most recent geometric algorithms without the need to be skilled programmers and to know the technical details of geometry processing.

1.4 Challenges and Scientific Contributions

The integration of Web service technologies and workflow-based frameworks already provides the possibility to remotely rerun experiments in many life science areas. The same approach can be used in computer graphics and geometry processing, but not surprisingly the possible slow data transfer among available servers and their possible limited main memory may cause bottlenecks and crashes during the remote processing of large-size inputs. Specifically, the distribution of the computation on geographically scattered machines requires to transfer the input from one to another by exploiting the Internet connection among them. The transfer of large-size meshes may constitute a bottleneck in the workflow execution, in particular when slow connections are involved. Moreover, remote servers may not satisfy specific hardware requirements (eg. huge main memory, high computational performance) necessary to efficiently process large datasets. As a consequence, the remote server that is asked to process a very large input may require a long time to finish its task or, even worse, a memory leak may occur and interrupt the elaboration.

In order to avoid that either a bottleneck or a memory leak occurs, and thus to efficiently support the processing of large meshes, solutions for the aforementioned problems need to be accurately designed. In order to improve the transfer speed and avoid possible bottlenecks, we propose an optimized mesh transfer protocol that manages the communication among servers and allows to reduce the amount of shared data that is flowed through the net during a workflow execution. Furthermore, possible memory leaks and very long computations are avoided by providing an innovative distributed *divide and conquer* approach that allows partitioning the large input mesh into subparts and remotely processing them simultaneously.

Summarizing, this thesis studies how large 3D models may be managed to efficiently implement distributed geometry processing workflows (Figure 1.2). Specifically, the following original contributions are provided:

1. definition of a formal representation for geometry processing pipelines that may be built as a composition of several existing algorithms;

2. design of an optimized mesh transfer protocol to reduce the amount of data shared among distributed servers;

3. design and implementation of an innovative partitioning method for large input meshes that enables distributed parallel processing;

4. design and implementation of a mesh simplification algorithm for large meshes that exploits our partitioning to distribute the computational load across multiple servers.

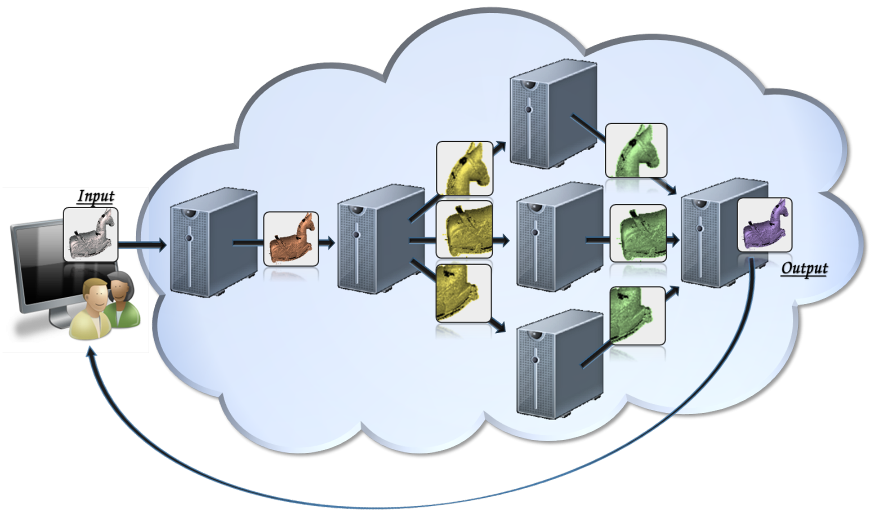


Figure 1.2: *Example of a distributed geometry processing workflow.*

1.5 Thesis Structure

The thesis is organized as follows.

Part I (Chapter 1) provides a general introduction to this dissertation. Both the motivation and main objectives have been described.

Part II describes the idea of exploiting a distributed environment to run geometry processing workflows and shows how we made it possible in practise. In particular, Chapter 2 discusses the existing tools that currently support both geometry processing research and workflow management. Chapter 3 describes our system and its three-layer architecture, while Chapter 4 goes on to explain the innovative mesh transfer protocol. Finally, Chapter 5 focus on workflow representation and provides a description of both the specialized XML grammar and the ontology exploited to store geometric workflows.

Part III shows how parallel approaches can be exploited to guarantee efficiency and effectiveness in a distributed environment. Specifically, Chapter 6 describes the limitations of our system when dealing with large datasets. Chapter 7 provides a summary description and analysis of existing approaches that allow to process large input meshes. In Chapter 8, an innovative partitioning approach is proposed that enables distributed parallel processing. Chapter 9 demonstrates the efficiency and effectiveness of our approach through an innovative distributed simplification algorithm that exploits our partitioning method.

Finally, Part IV (Chapter 10) draws the conclusions, describes the ongoing research and the plans for future work.

1.6 Reading Guidelines

Along the whole dissertation, technical aspects are provided through a customized style. Specifically, grey frame boxes are exploited to hold technical specifications. Each frame box is provided with a logo that represents the type of technicality and a title that specifies the main subject. The following technical types are considered:

Technology Exploitation

The list of technologies exploited for the implementation. (eg. development environment, programming language, ...).

File Management

Technical aspects related to a specific file or group of files (eg. access scheduling, encoding format, ...).

Geometry Processing

Technical aspects related to a geometry processing operation (eg. geometry and topology modifications).

Algorithms & Data Structures

Technical description of an algorithm (eg. pseudocode) or of a specific data structure.

Part II

Distributed Environments

2 Related Works

*'A **workflow** is the computerized facilitation or automation of a process, in whole or part".*

– D. Hollingsworth, [Hol95]

*'A **Workflow Management System** is a system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic".*

– D. Hollingsworth, [Hol95]

The aforementioned definitions were provided in The Workflow Reference Model [Hol95] [HSK04] that still forms the basis of most frameworks in use today. To put it simply, a workflow can be considered as a list of operations that can be automatically executed. In [Hol95], the structure of a generic WorkFlow Management system (WFM) is described by identifying the interfaces which enable products to interoperate at a variety of levels. A minimal WFM usually includes a *syntactical* tool to turn user-defined workflow specifications to a machine-readable form, and an interpreter (known as *workflow engine*) that turns the specifications to an actual sequence of processes. Where user interaction is necessary to build and/or control the process, a WFM may also include a user interface and a worklist handler which manages the interaction between parts.

In geometry processing and engineering applications, the effective use of 3D models frequently requires some model processing to satisfy the application requirements. Typically, a sequence of geometric operations are performed on the input dataset in order to obtain the desired result. This processing can involve conversion of the representation and format as well as modifications in object geometry and/or topology. In the literature, proper pipelines of geometric algorithms have been defined [Att10] and, most of them, can be implemented as automatic processing workflows.

2.1 Offline Workflows

The idea of allowing automatic replication of workflows is not new in life science research areas and many business activities already exploit specific frameworks that allow to reapply the same list of operations on different inputs. Many business and research activities take advantage of recently-designed WFMs specialized in different kinds of computation. Noticeable examples can be found in life science areas [TS07] like drug discovery, genomics, gene analysis and biology [WHF+13] [GNTT10] [LAB+06].

In geometry processing research areas, several standalone applications are now available that allow to exploit already implemented methods to edit a mesh on the user local machine and to visualize the results through a graphical interface [CCR08] [MK12] [AF06]. In order to create a new shape or elaborate an existing model, the user is required to:

1. search the net for the available tools and select the appropriate one by making sure of hardware and software compatibility between the application and the local operating system;

2. download and install the software, possible dependencies, and plugins on the local machine;

3. manually select the desired sequence of operations and set possible input parameters.

Understanding which tools and operations better fit with specific purposes is frequently difficult for non-experts. The Web provides plenty of documentation and tutorials on the use of many geometric tools, but they are weakly organized and generally related to specific software. Moreover, new geometry processing algorithms are continuously being developed on the top of already complex previous works. In order to take advantage of them, researchers often need to install required geometry processing libraries and recompile available source codes. This kind of activity may require lots of efforts and, even worse, algorithms may be described in scientific papers, but no corresponding executable is available. In these cases, it is necessary to reimplement the entire pipeline from paper description, but, due to the level of complexity, this easily becomes a costly and error prone operation.

To support the idea of replicating the same list of geometry operations on different inputs, MeshLab [CCR08] and OpenFlipper [MK12] allow to interactively edit a mesh and save the sequential list of executed operations, so that it can be re-executed locally from the tool user interface. Unfortunately, to reproduce such an experiment, a researcher needs a similarly performing machine, and needs to install the same software tool (eg. MeshLab) and possible plugins.

2.2 Online Workflows

In the last decades, service-oriented architectures have been exploited to make system independent from local hardware and software requirements. Users are allowed to use tools and applications that, in their turn, can be provided by others (i.e. service providers) who publish them as online services. Mainly, service providers offer their services according to two models, namely *platform-as-a-service* and *software-as-a-service*. In the former, providers deliver a computing platform (typically including operating system, programming-language execution environment, database, and Web server) that can be exploited by developers to implement and run their software solutions, while, in the latter, applications are delivered over networks as Web services and end users are allowed to access them remotely using Internet.

The recent concept of executable paper is just an example that exploits the platform-as-a-service model. In this new concept, experiments reported in an article should be reproducible directly from the paper itself which, in its electronic version, encapsulates the data and the code, and not just the results as traditionally done. This concept of executable publication was addressed by Nowakowski et al. through the Collage Authoring Environment [NCH+11]. The system is fully Web-oriented, and all the resources necessary to reproduce and reuse executable code (eg. hardware, servers, storage) are publicly provided. The platform allows scientists to enrich the digital publication with executable code and gives the possibility to readers to rerun it remotely.

Conversely, the software-as-a service paradigm is exploited in life science research and business areas. As an example, Taverna [WHF+13] is an open source domain independent WFM that allows to design and execute scientific workflows by exploiting a number of different Web services from a diverse set of domains (eg. biology, chemistry, medicine).

Web service technologies are scarcely considered in geometry processing literature, since the transmission of large 3D models can easily slow the process down too much. Campen and colleagues published an online service called WebBSP [Cam] which is able to remotely run a few specific geometric operations. The user is required to upload an input mesh from a standard Web browser and select a single geometric algorithm from a set of available operations. The algorithm is actually run on the server and a link to download its output is returned. The available operations are not customizable by users, only one of them can be run at each call, and the service is accessible only from the WebBSP graphical interface. Similarly, the possibility to remotely run a hex mesh optimization algorithm is provided at [LSVT15]. Again, a Web interface allows users to upload a poor quality hex mesh and to provide an email address to receive an optimized version of the mesh.

Conversely, MeshLabJS [Mes15] provides a Web-based graphical interface that allows to interactively edit triangle meshes by exploiting a modern Web browser that is able to run C++ code compiled into a javascript at a pretty decent speed. Unfortunately, it is still rudimental, minimal and does not consider the idea of supporting automatic pipeline replication.

Also, it is worth mentioning that geometric Web services were previously considered by Pitikakis [Pit10] with the objective of defining semantic requirements to guarantee their interoperability. Unfortunately, in [Pit10] Web services are stacked into hardcoded sequences, users are not allowed to dynamically construct workflows, and the transmission of large models is not dealt with.

2.3 Workflows and Model Repositories

3D model repositories are already key instruments for researchers in the area of geometry processing and product design. Not by chance, in the last decades we have assisted to an impressive growth of online repositories of 3D shapes coming from different application areas. Current repositories store a large number of shape models to reduce the efforts in creating 3D objects from scratch and to provide a variety of input datasets.

The Stanford 3D Scanning Repository [sta96] is one of the earliest widely-used online collections of 3D models, and several authors used its meshes to demonstrate the benefits of their algorithms and the improvements achieved over the state of the art. Stanford's repository was focused on models coming from 3D digitization and both range data and reconstructed surfaces are available for many of the scanned objects. Conversely, the 3D CAD Browser [cad01] deals with synthetic CAD models. It provides high-quality 3D polygonal mesh as well as 3D CAD solid objects that can be downloaded by any registered user. Also, users have the possibility to upload new models and provide new input data to the community.

Some other available collections focus on specific algorithms and aim to evaluate the performance of such methods. As an example, several 3D shape retrieval benchmarks are available which aim to evaluate shape query and retrieval algorithms. Among them, the Princeton Shape Benchmark [SMKF04] is one of the mainly used collections and provides generic 3D polygonal models with a set of software tools that are widely used by researchers to compare their shape matching results with other algorithms, while Purdue Engineering Shape Benchmark [JKIR06] is a public 3D shape database for evaluating shape retrieval algorithms mainly in the mechanical engineering domain.

The more recent Digital Shape Workbench (DSW) [dsw12] tries to encapsulate most of the functionalities provided by previous repositories. The DSW is an e-Science support in the field of computer graphics and scientific visualization, initially developed by AIM@SHAPE [Aim04] and currently maintained by the VISIONAIR project [Vis12]. It provides a data repository and a knowledge management system able to perform data browsing and discovery. The data repository is aimed to collect and share a large number of 3D shapes and state-of-the-art tools that can be used to process digital models. Registered users are allowed to upload resources (eg. test models, prototype tools, algorithm results, ...) and make them available on the Internet. The DSW gives the possibility to share whole benchmarks by "grouping" shape models into collections. The shape repository explicitly stores the uploaded models, often requiring a huge amount of memory to save geometric and connectivity information. Redundancy is one of its main problems, due to the fact that outputs of similar processes, which are often very similar to each other, are individually uploaded with no attention to avoid possible duplicated models and data.

Since in several cases different shapes are modifications of the same original data, it should be possible in principle to reproduce these models on demand instead of storing them explicitly. In general, besides storing 3D data, a modern repository should provide the possibility to store and execute algorithms on such data, so that users can create a virtually infinite set of input models on the fly without struggling with software installations, compatibility issues, or hardware requirements. Moreover, executable workflows might be reused from any location to reproduce geometric experiments much more easily and fairly than current solutions. Hence, endowing modern 3D model repositories with geometric workflow capabilities is expected to further boost their efficacy.

At a more abstract level, the need of sharing scientific workflows is considered by the open-source Web infrastructure *myExperiment* [GBA$^+$10], where *in silico* experiments in any field of life science can be published. Most of the existing WFMs include a client application that needs to be installed and allows the user to graphically build workflows and run them either locally, involving local tools, or on remote computational infrastructures, by taking advantage of available Web services. To the best of our knowledge, no existing 3D model repository considers the idea of re-execute geometric workflows to allow the generation of 3D data by exploiting available Web services.

2.4 Applications

Nowadays the use of various systems for the creation and analysis of 3D digital models has moved most of the engineering applications into the digital world. Based on the end-use, input geometric models are required to satisfy specific application requirements. Algorithms to adapt input models to specific applications have been defined in literature [BPK$^+$07] and mostly involve operations to remove both geometric and topological defects. Some examples will be presented in which shape processing is generally performed according to steady sequences of operations resulting from technological constraints and experience.

Finite Element Analysis As an example, performing the finite element analysis during a product design not only requires the creation of a mesh model from the CAD B-Rep but also model adjustments, and a shape simplification involving both topological and geometric changes [FCF$^+$08]. The few long and thin triangles produced by a tessellation algorithm are perfect for a visualization setting because they allow higher frame rates. Unfortunately, these triangles are definitely not appropriate for a FEA application, where the regularity of the mesh and its density in regions affected by particular stress are determinant to guarantee faithful simulation results. In this case, a remeshing process is necessary to modify the shape of the triangles without modifying the overall shape of the model. Before remeshing, however, possible pre-processing might be necessary to guarantee that the mesh actually encloses a solid [Att14]. Also, after remeshing and depending on the analysis type, the surface mesh can be used to produce a conforming tetrahedral mesh.

3D Printing Another example is the process required to get correctly printable shapes. Today fabricating an appropriate 3D model using a low-cost 3D printer is as easy as printing a textual document, but creating a

3D model, which is actually "appropriate" for printing, is definitely complicated. A 3D model can be produced either from scratch by using traditional CAD software, or from real-world objects using 3D digitizers. In both cases, the raw model is likely to have a number of defects and flaws that make it unsuitable for printing [ACK13].

3D Visualization When a mesh model must be visualized, it is often important to convey a clear unbiased picture of the object. This requirement is in contrast with the characteristics of typical raw models coming from 3D digitization sessions, where a number of surface holes are commonplace just as surface noise, tiny disconnected components, gaps, and so on. Mesh visualization is not as demanding as 3D printing, but all the aforementioned defects should be removed or reduced to produce a nice and informative rendering. Geometric pipelines including surface smoothing, hole filling, gap closing and possibly simplification are therefore necessary. Automatic process workflows can be implemented in this case too, and a number of variations can be provided depending on the target visualization device (e.g. a powerful graphics workstation, a desktop PC, a smartphone). For example, the level of the simplification can depend on the rendering capabilities, whereas the amount of smoothing can depend on the specific rendering engine used.

Virtual Reality Virtual reality is the combination of real-time presentation and immersive interaction for the modification and evaluation of models and processes within a computer-generated environment. In order to use digital models in virtual reality environments, shape model adaptation is often required. Design reviews and simulations in virtual reality environments demands for high visualization capabilities obtained by processing polygon data, whereas CAD models are based on continuous surfaces. Therefore, CAD models need to be converted in a virtual reality compatible format, i.e. a polygonal representation. Various problems can be detected in this conversion [RCW$^+$06]. First, there is an inadequate treatment of the geometry with loss of precision leaving to inconsistent models with wrong surface orientation or cracks. In addition, the obtained models are too complex with unnecessary details, e.g. hidden areas, but at the same time, they miss realism. Finally, semantic information associated to each object, including its structure, is lost and frequently needs to be recreated. To overcome these problems, several adjustments have to be performed by virtual reality specialists using ad hoc tools.

3

The Framework

"A collection of independent computers that appears to its users as a single coherent system."
– A. S. Tanenbaum and M. van Steen , [TS06]

Our system allows to remotely perform complex geometry processing on triangle meshes. A distributed network of servers provides both the software and hardware necessary to undertake the computations, while the overall execution is managed by a central engine that both invokes appropriate Web services and handles the data transmission. Nothing more than a standard Web browser needs to be installed on the client machine.

3.1 The Architecture

The framework architecture (Figure 3.1) is organized in three layers, according to Hollingsworth WFM specifications [Hol95].

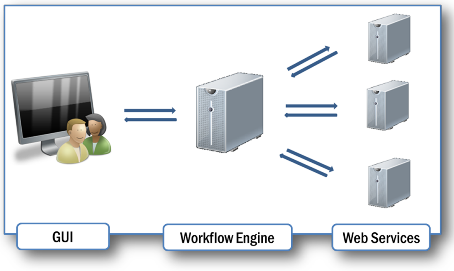
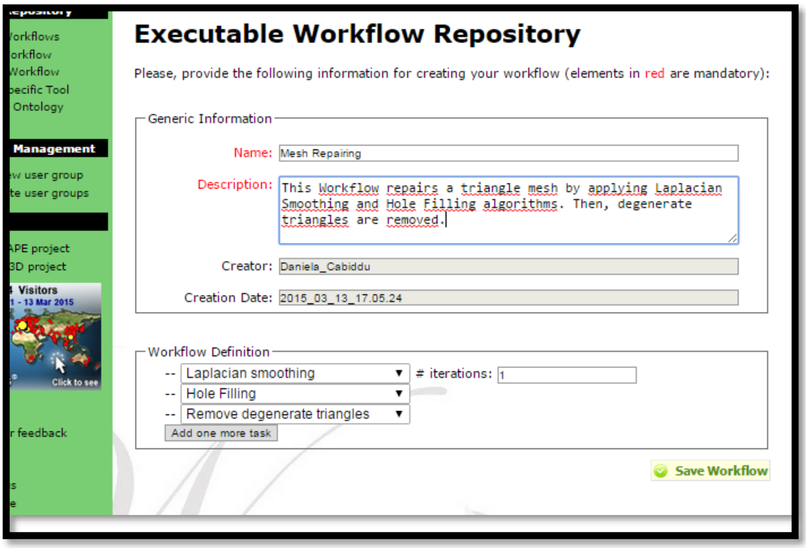


Figure 3.1: *The three-layered system architecture composed by a graphical user interface that allows to upload and run geometry processing workflows, the workflow engine responsible of workflow execution, and available Web services, each of them exposing a Web service to support a geometry processing algorithm.*

On one side, a Web-based user interface allows to choose the desired algorithms among the available ones and to combine them in order to define complex geometry processing pipelines. On the other side, a set of Web services is available, each of them able to run a specific geometry processing algorithm using possible input parameters and returning the generated output address. The workflow engine is the interface between the two sides and is responsible of the pipeline runtime execution. It receives the specification of a geometry processing workflow and the address of an input mesh. When all the data is available, the engine sequentially invokes the various Web services, manages the flow of data among them and returns the address of the eventual result to the user interface.

3.1.1 The Graphical User Interface

A dedicated user-friendly interface supports the creation of new geometry processing workflows. The user is asked at first to provide the information directly related to the workflow, such as its name, a description and the list of geometry processing algorithms that constitute the pipeline (Figure 3.2).



Executable Workflow Repository

Please, provide the following information for creating your workflow (elements in red are mandatory):

Generic Information

Name: Mesh Repairing

Description: This Workflow repairs a triangle mesh by applying Laplacian Smoothing and Hole Filling algorithms. Then, degenerate triangles are removed.

Creator: Daniela_Cabiddu

Creation Date: 2015_03_13_17.05.24

Workflow Definition

-- Laplacian smoothing ▼ # iterations: 1

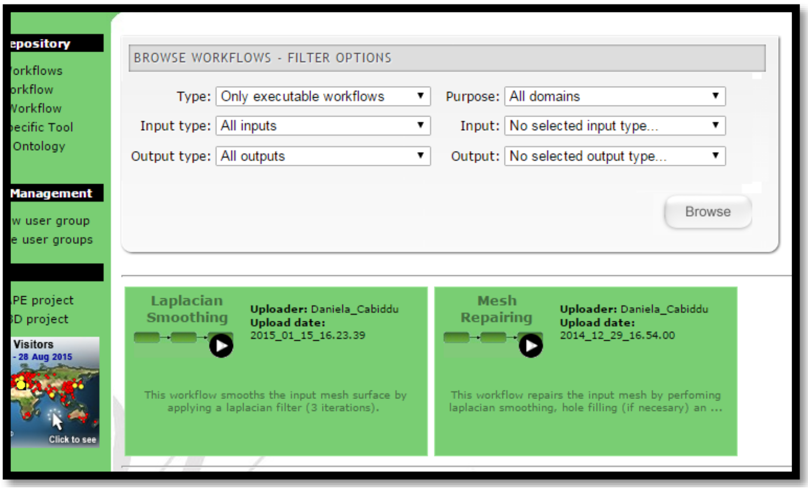
-- Hole Filling ▼

-- Remove degenerate triangles ▼

Figure 3.2: *Detail of the graphical user interface. The user can dynamically create a new workflow by selecting the operations and setting possible required parameters.*

The user may define a new workflow by selecting each task from a list of available ones and, for each task selected, the interface dynamically calls for possible parameters. Besides atomic tasks, the user defines possible conditional tasks or loops by specifying their conditions and by delimiting their execution bodies. Once the whole procedure is defined, the interface allows the user to associate an input mesh to the workflow in order to turn it into an actual experiment. If no input is associated, the workflow can be stored on the system as an abstract procedure to be necessarily instantiated on a specific input for each execution.

Other than creating new workflows from scratch, the user is allowed to browse the previously-defined workflows and to select one of them to be executed (Figure 3.3). To reuse one of the available workflows, the user is simply asked to select the desired one and provide an input mesh. The system is responsible of associating the identifier of the selected abstract procedure with the provided input model, running the actual pipeline and returning the result to the user.



BROWSE WORKFLOWS - FILTER OPTIONS

Type: Only executable workflows
Purpose: All domains
Input type: All inputs
Input: No selected input type...
Output type: All outputs
Output: No selected output type...

Laplacian Smoothing
Uploader: Daniela_Cabiddu
Upload date: 2015_01_15_16.23.39
This workflow smooths the input mesh surface by applying a laplacian filter (3 iterations).

Mesh Repairing
Uploader: Daniela_Cabiddu
Upload date: 2014_12_29_16.54.00
This workflow repairs the input mesh by perfoming laplacian smoothing, hole filling (if necesary) an ...

Figure 3.3: *Detail of the graphical user interface. The user can browse existing workflows and select one of them for actual execution.*

3.1.2 Web Services

Our system has been designed with the objective of being continuously extensible by indexing more and more Web services. Thus, in our context a Web service can be seen as a sort of "remote plugin" that any skilled student should be able to implement. In other words, a Web service can be considered as a black box able to perform a specific operation. A single server (i.e. a provider) can expose a plurality of Web services, each implementing a specific algorithm and identified by its own address. Also, each Web service provides the specification of its interface, that is the number and type of required input data and the type of returned output. Thanks to this information, any client is able to properly invoke the service.

Our system supports the invocation of two types of Web services, namely *atomic* and *boolean*. Atomic Web services are required to:

1. run a simple processing operation on a 3D triangular mesh using possible input parameters;
2. store the output on the server where it is located;
3. make the output available by returning its address.

Conversely, boolean Web services do not generate any output mesh and return a boolean value. Specifically they are required to analyze the input triangular mesh and check if a specific condition is satisfied. For example, if a watertight input mesh is required, a "isWatertight" Web service is invoked that reads the mesh and returns a boolean value depending on whether the mesh is watertight or not.

Theoretically, any atomic task may be made available as a Web service. Mainly, geometric tasks may be subdivided into two main categories, namely *generic* and *model-specific*. While the former includes all the methods able to perform a specific operation on any valid input mesh, the latter group together algorithms that are tailored to process one ore more elements of a specific mesh. It is worth noticing that only generic tasks are currently supported by our framework and can be stacked to constitute abstract pipelines.

3.1.3 The Workflow Engine

The workflow engine is the core of the system and is responsible of the workflow runtime execution. From the user interface it receives the specification of a geometry processing workflow, which can be either a new one or the identifier of one of the available pre-defined workflows, and possibly the address of an input mesh to be downloaded from the Internet. When all the data is available, the engine reads the encoded workflow and sequentially invokes the various Web services.

An example of workflow execution is shown in Figure 3.4 where three Web services are involved. The engine receives both the input mesh and the workflow from the user interface. Then, it reads the workflow, selects the involved Web services, and invokes the first one by sharing the input mesh. The first Web service runs its task and returns the generated output to the engine that shares it with the successive Web service to be invoked. All the successive Web services are sequentially invoked and each of them receives the output of the previous one as an input. Finally, the output generated by the last Web service is returned to the user.

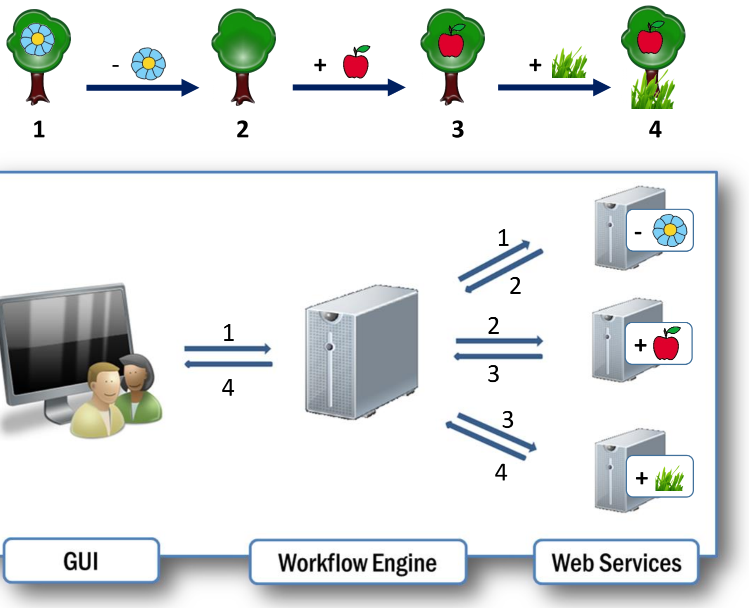


Figure 3.4: *An example of workflow execution. The workflow includes three operations. Each operation is performed by a specific Web service. The workflow engine is responsible of sequentially invoking the algorithms and managing the flow of data among them.*

Specifically, the engine stores the list of Web services that are available and able to perform specific tasks. For each workflow task, the engine selects and invokes the appropriate Web service. When triggered for execution, each Web service receives the address of the input mesh and possible input parameters. When the task terminates, the address of the generated output is returned to the Engine so that it can be forwarded to the next involved Web service. When the last Web service terminates its task, its output is returned to the engine that forwards it to the user by publishing the corresponding link.

In order to enable the definition of non-trivial workflows, the engine is also able to manage the execution of conditional tasks and loops, and the evaluation of the condition itself is delegated to boolean Web services. When a conditional task or loop is read in the workflow, the Web service able to evaluate the condition is invoked. It receives from the engine the address of the input mesh and possible information necessary for the evaluation. A boolean value is returned to the engine to indicate if the condition is satisfied. If so, the list of operations in the execution body is read and Web services are invoked to execute each of them. If not, the execution body is skipped and the Web service corresponding to the first operation outside it is invoked.

3.2 Technical Aspects

In practice, the three-layered architecture described in Section 3.1 and shown in Figure 3.1 has been carefully designed and implemented, in order to provide the desired features.

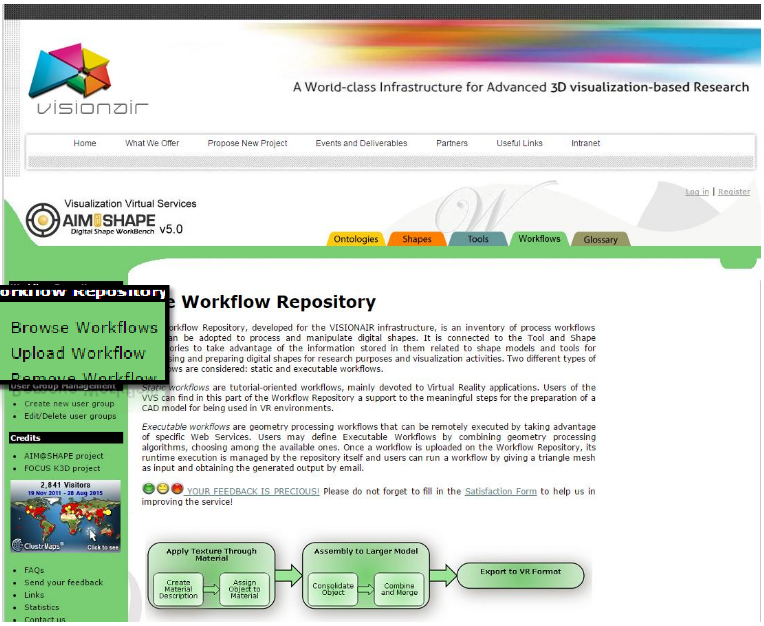
Development Environment

Our framework has been implemented by integrating technologies described in [WCL$^+$05] and exploiting Web Service development environments (e.g. NetBeans, GlassFish, OpenESB), specific programming languages (e.g. WSDL, BPEL) and communication protocols (e.g. SOAP).

Tool	**Version**	**Role**
Netbeans	6.5	Integrated Development Environment (IDE)
Java EE 6	1.6.0 u11	Platform for the Java programming language
GlassFish Server	2.1.1	Application Server
OpenESB	2.0	Enterprise Service Bus for communication among software applications

The Graphical User Interface

The Web application provides a home page where a summary description of the system is provided. Moreover, two main pages are available, each of them allowing to easily create a new workflow and browse existing ones respectively. Both pages are accessible from the left sidebar (Figure 3.5).



e Workflow Repository

orkflow Repository, developed for the VISIONAIR infrastructure, is an inventory of process workflows an be adopted to process and manipulate digital shapes. It is connected to the Tool and Shape ories to take advantage of the information stored in them related to shape models and tools for sing and preparing digital shapes for research purposes and visualization activities. Two different types of ows are considered: static and executable workflows.

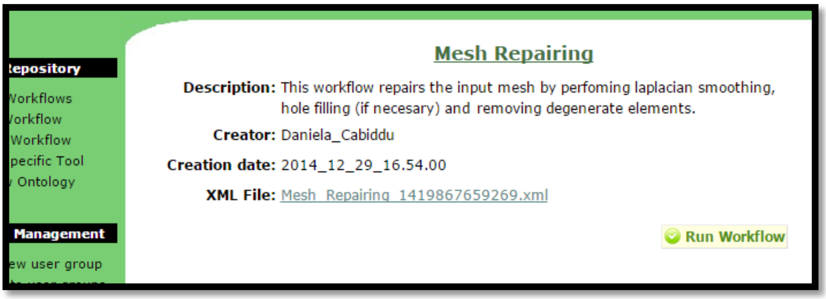
Static workflows are tutorial-oriented workflows, mainly devoted to Virtual Reality applications. Users of the VVS can find in this part of the Workflow Repository a support to the meaningful steps for the preparation of a CAD model for being used in VR environments.

Executable workflows are geometry processing workflows that can be remotely executed by taking advantage of specific Web Services. Users may define Executable Workflows by combining geometry processing algorithms, choosing among the available ones. Once a workflow is uploaded on the Workflow Repository, its runtime execution is managed by the repository itself and users can run a workflow by giving a triangle mesh as input and obtaining the generated output by email.

YOUR FEEDBACK IS PRECIOUS! Please do not forget to fill in the Satisfaction Form to help us in improving the service!

Figure 3.5: *The system home page. A summary description of the system is provided. Both pages that allow to create a new workflow and browse existing ones are accessible from the left sidebar.*

The "Browse Workflows" page allows the user to browse through the workflows stored in the repository and to select one of them to be executed. By clicking on a workflow, the user can access a new tab with a more detailed view on it (i.e. workflow name, creator name, creation data, link to the corresponding file) and a "Run Workflow" button is provided to allow the workflow execution (Figure 3.6). When the "Run Workflow" is clicked, the user is required to upload the input mesh and wait for the result. Users do not need to be registered to browse and run existing workflows. They are asked to provide only their email address where they wish to receive the link to the generated output.



Mesh Repairing

Description: This workflow repairs the input mesh by perfoming laplacian smoothing, hole filling (if necesary) and removing degenerate elements.

Creator: Daniela_Cabiddu

Creation date: 2014_12_29_16.54.00

XML File: Mesh_Repairing_1419867659269.xml

Figure 3.6: *Detailed view of an existing workflow. The page provides useful information about the author and the workflow itself. The "Run Workflow" button is provided to allow the workflow execution. When the "Run Workflow" is clicked, the user is required to upload an input mesh and wait for the result.*

Contrary to the browsing page, only registered users are allowed to access the "Upload Workflow" (Figure 3.2) page. From this page, the user can easily define new workflows by selecting the list of operations from the pull-down menus where the available ones are listed. When the complete workflow has been designed and the "Save Workflow" button is clicked, the system stores the pipeline as a file according to a specific XML format (see Chapter 5). From now on, the workflow is available as an abstract procedure that may eventually be turned into an actual experiment by associating an input triangle mesh to it.

Graphical User Interface

The Web-based interface consists in a set of Web pages generated by exploiting JavaServer Pages (JSP) technologies. The layout, colors, and fonts of each page have been designed by using Cascading Style Sheets (CSS). Moreover, Javascript and JQuery are exploited to dynamically update the graphical interface based on the user interaction (eg. when a workflow operation is selected, the Web page dynamically updates to call for possible required parameters). Behind the graphical interface, dedicate Java servlets have been implemented that are responsible of storing new pipelines and uploaded input meshes into the system.

The Workflow Engine

Mainly, the engine is composed of two servlets, specialized in reading the workflow and managing its runtime execution respectively. The former is responsible of reading the encoded selected workflow and turning it into a list of *active tasks*, while the latter manages the workflow execution and the flow of data among involved Web services. Moreover, the engine stores the list of Web services that could perform a specific task. Each Web service is stored as a tuple of technical information, including an identification ID, the interface address, and possible required parameters. Also, for each Web service, the engine keeps track of the available bandwidth and the number of concurrent process executions at each moment. The selection of the most appropriate Web service favours the largest bandwidth and the smaller number of process executions at that moment.

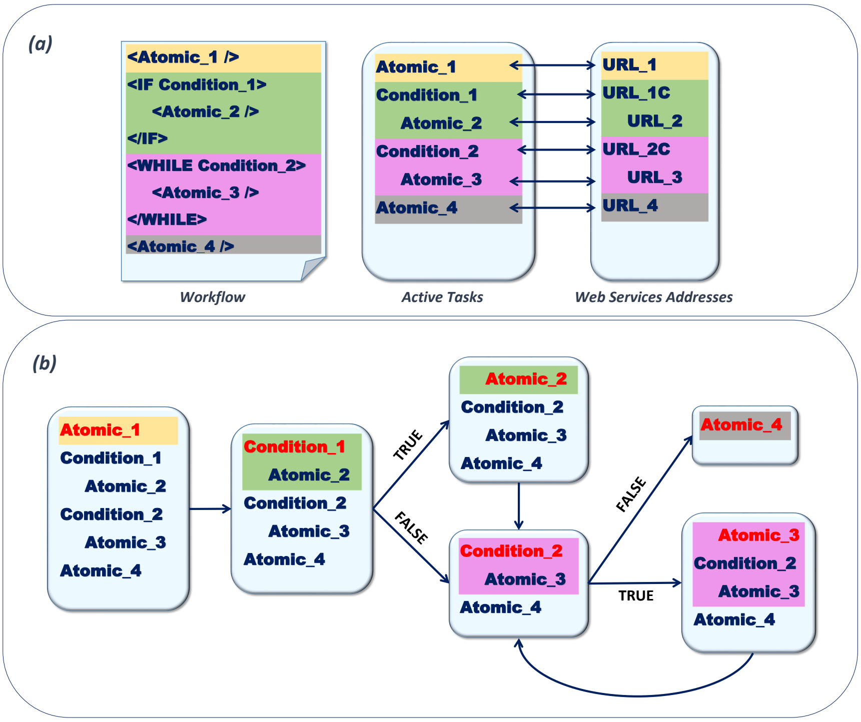


Figure 3.7: *An example of workflow execution with technical details. The workflow involves both atomic and conditional tasks. (a) The engine reads the input workflow, creates the corresponding list of active tasks with Web service addresses. (b) Then, the workflow is executed and the list of active tasks is updated at each Web service invocation.*

At the beginning of the workflow execution, all the tasks involved in the workflow are *active*. For each task, the address of the corresponding Web service is associated (Figure 3.7a). Then, the involved Web services are sequentially invoked. Upon termination, if the Web service runs an atomic task (i.e. neither an “IF” nor a “WHILE”), the corresponding active task is removed from the list meaning that it is no longer involved in the workflow. The same happens when an “IF” task is encountered with true condition. Conversely, after an “IF” or a “WHILE” whose condition is false, all the tasks constituting the body are removed from the list. After a “WHILE” with true condition, an additional copy of the condition’s task and of all the tasks in its body is added to the list (Figure 3.7b) right after the original copy.

Workflow Engine

The system core is implemented as a dynamic orchestration of services and deployed in a Java Business Integration (JBI) enabled platform. Both servlets are implemented in Java and their interfaces are described using the Web Service Description Language (WSDL), while their orchestration with the involved Web services is implemented using the Business Process Execution Language (BPEL). We took advantage of the BPEL dynamic binding technology and the WS-Addressing standard mechanism, namely Endpoint Reference, to sequentially invoke, for each process activity, the Web service able to execute it. The data exchange between Web services is implemented according to the Simple Object Access Protocol (SOAP).

Web Services

To make a new algorithm exploitable by our system, a properly designed executable must be wrapped within a Web service. Service providers should register their Web service by communicating the corresponding address and possible input parameters to the system. The system automatically updates the Web service database and the list of available Web services in the graphical user interface.

To support the idea of including Web services provided by third parties, and to allow input models to be stored on remote servers, we require that Web services are designed to receive the address of the input mesh and to download it locally. Each of these services can either execute an atomic task (i.e. a list of editing operations that modifies the mesh) or an analysis operation (i.e. a task that checks a condition of the mesh without modifying it). Specifically, Web services that run an atomic task should save the output model locally and make it available through the Internet through a public URL. Similarly, Web services that perform an analysis operation should download locally the received input mesh and return the quality value as an output, but no output mesh should be generated. Appendix A provides a detailed technical description of currently available Web services.

Web Services

We have implemented a first set of Web services according to the aforementioned specifications. Java has been exploited for the wrapper and C++ for the actual algorithms. For all these algorithms, we simply started from the corresponding C++ functions provided by ReMesh [AF06] and the Mesh Quality Tool [Att13]. The interface of each Web service is described using WSDL. SOAP is exploited to implement the communication between each Web service and the workflow engine. Each Web service is designed to receive a SOAP message where the address of the input mesh and possible input parameters are listed and return a SOAP message containing the result (eg. the URL of the output mesh or the quality value) to the workflow engine.

4

Mesh Transfer Protocol

"A cascade, a torrent, a deluge of data is going to want to move around the network"

– B. Golden, [Gol09]

Not surprisingly, we have observed that the transfer of large-size meshes from a server to another constitutes a bottleneck in the workflow execution, in particular when slow connections are involved. Mesh compression techniques can be used to reduce the input size, but they do not solve the intrinsic problem. In order to improve the transfer speed and thus efficiently support the processing of large meshes, we designed an optimized mesh transfer protocol that sensibly reduces the amount of data shared through the network. Our solution is inspired on the prediction/correction paradigm.

4.1 Background

The general idea of prediction/correction works as follows. A sender S needs to transmit a data set to a receiver R, but instead of transmitting the whole data set at once, S sends a piece of data only, let it be d_0. Then, R tries to *predict* what the next piece of data d_1 will be based on the previously received information d_0. At the same time S does exactly the same prediction and, instead of sending the next piece of data, sends the difference between the prediction and the actual data to be sent, that is $c_1 = d_1 - d_0$. Thus, R can calculate d_1 by *correcting* the prediction using c_1. The benefits of all this machinery become evident when the predictions are accurate enough: in this case the corrections to be sent are small if compared with the original data and thus can be encoded with fewer bits.

A typical example in geometry processing is the so-called "parallelogram rule" used for mesh compression [TG98]. According to such an approach, the position of a vertex D can be predicted by assuming that it completes the parallelogram formed by the vertices (A, B, C) of a neighboring triangle. When the predicted position D_P is different from the actual position of D, a corrector vector $\vec{D_P}$ is provided (Figure 4.1).

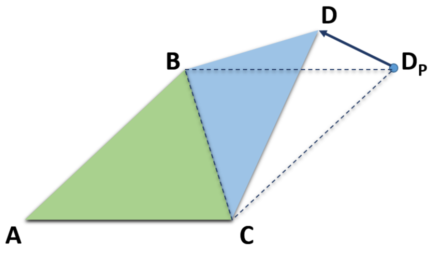


Figure 4.1: *An example of the parallelogram rule used in mesh compression.*

4.2 Concurrent Mesh Transfer

A triangle mesh can be defined by an abstract simplicial complex that specifies its connectivity endowed with a set of vertex positions that uniquely identify its geometric realization [Att13]. We have observed that there are numerous mesh processing algorithms that simply transform an input mesh into an output by computing and applying local or global modifications.

An algorithm that modifies an input mesh can act on its geometry only (e.g. by changing the position of the vertices), on its connectivity only (e.g. by triangulating boundary loops), or on both. Furthermore, in many cases modifications can be only local (e.g. sharp feature restoration). In all these cases, it is possible to predict the result by assuming that it will be identical to the input, and it is reasonable to expect that the corrections (i.e. list of applied editing operations) to be transmitted can be more compactly encoded than the explicit result of the process.

The aforementioned observation can be exploited in our setting as shown in Figure 4.2, where an example of execution of a simple workflow composed by three tasks is provided. The engine reads the whole workflow and, for each of the three tasks requested, looks for a server exposing an appropriate Web service (i.e. a Web service which implements the task). Then, the engine sends the address of the input mesh to all the servers that have

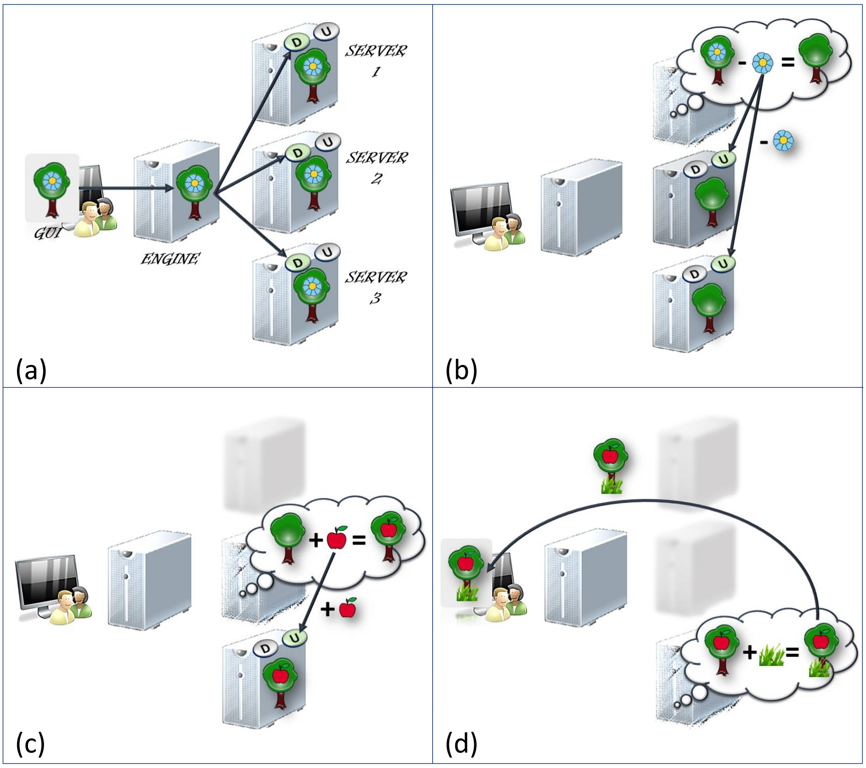


Figure 4.2: *Mesh Transfer Protocol Example. Three servers are involved into the workflow execution. Each of them exposes a Web service to support a geometry processing algorithm and two modules able to download (D) meshes and update (U) the previously downloaded mesh by applying the corrections. (a) The engine shares in parallel the address of the input mesh with all the involved servers that proceed with the download. (b) The first service runs the task, produces the corrections and returns the corresponding address to the engine that shares it in parallel to the following involved servers. Both download the file and correct the prediction. (c) The second service is invoked, runs the task and makes the correction available, so that the third involved server can download it and update its local copy of the mesh. (d) The engine triggers the third service that runs the algorithm and makes available the modified output mesh so that it can be directly downloaded by the user.*

been identified so that they can download it (Figure 4.2a). Right after having sent the address, the engine triggers the first Web service (Figure 4.2b) to locally run the algorithm. Such an algorithm produces both the output mesh and the list of changes applied on the input to obtain the result (e.g. vertices/edges/triangles that have been added, removed or modified).

Both the output mesh and the list of changes (i.e. the correction) are compressed and made available through two URLs which are communicated to the workflow engine. In its turn, the engine forwards this information to the next two Web services to be triggered, so that both of them can download the compressed correction from the first server, and can reproduce the output of the first step by decoding and applying the correction to the mesh that was previously downloaded. At this point the engine triggers the second Web service (Figure 4.2c) that follows the same protocol by running the algorithm and publishing the URLs of the output and the correction. Finally, the third Web service corrects its prediction, runs its task and returns the URL of the final result (Figure 4.2d).

In a more general setting, the protocol works as follows. Through the user interface, the user selects/sends a workflow and possibly the URL of an input mesh to the workflow engine. The engine analyses the workflow, locates the most appropriate servers hosting the involved Web services, and sends in parallel to each of them the address of the input mesh. Each server is triggered to download the input model and save it locally. At the first step of the experiment, the workflow engine triggers the suitable Web service that runs the algorithm, produces the result, and locally stores the output mesh and the correction file (both compressed). Their addresses are returned to the workflow engine that forwards them to all the subsequent servers involved in the workflow. Each server downloads the correction and applies it to the mesh it already has in memory in order to update the local copy of the model. Then, the workflow engine triggers the next service for which an up-to-date copy of the mesh is readily available on its local server. At the end of the workflow execution, the engine receives the address of the output produced by the last invoked Web service and returns it to the user interface, so that the user can proceed with the download.

In this scenario, the entire input mesh is broadcasted only once at the beginning of the process, whereas the final result is transmitted only once at the end. Inbetween, only the corrections are broadcasted to the subsequent servers. Thus, when the corrections are actually smaller than the partial results, this procedure produces significant benefits. In any case, each Web service produces both the correction and the actual result so, should the former be larger than the latter, the subsequent Web services can directly download the output instead of the corrections. Thus, our mesh transfer protocol improves the overall performances when the aforementioned conditions hold, while no degradation is introduced otherwise.

4.3 Technical Aspects

In order to make the transfer protocol work, service providers are required to set up their servers so that they are able to download the input mesh at the beginning of the workflow execution, save it locally, and update it at each workflow step by applying the corrections. Moreover, each algorithm provided as a Web service should be able to perform the streaming of applied editing operations according to our specific format. To simplify the work of potential contributors, we provide dedicate tools to set up their own server and a library for streaming editing operations according to our specification.

4.3.1 The Download and Update Module

Two modules must be installed on each service provider that take care of downloading the original mesh and downloading/applying the correction respectively (Figure 4.2a). Specifically, the "Download" module is triggered at the beginning of the workflow execution and receives the address of an input mesh to be downloaded, while the "Update" module is invoked when each atomic Web service finishes its task and receives the address of a compressed binary file representing the previously generated corrections.

Both modules pay attention not to download the same file twice. This latter check is necessary because the same service provider can host more than one Web service and thus the Workflow Engine may trigger the "Download" and "Update" modules more than once on the same machine. To perform the check, both modules exploit a database where the already downloaded files are stored. Specifically, each record of the database identifies a downloaded file by providing the address of the corresponding file and the local path where the file has been stored on the server.

When triggered, both the "Download" and the "Update" modules receive the address of the input file as an input and query the database to check if a record containing the received URL exists. If not, the input file, which can be either a mesh or a correction file, is downloaded, and a new record is added to the database. The "Update" module is responsible of updating the previously downloaded mesh by applying the received corrections. In addition to the address of the correction file, it also receives the address of the input mesh that should be updated. To achieve its tasks, the module downloads the correction file and queries the database to get the local path of the input mesh. Then, the model is loaded into main memory and each operation in the correction file is reapplied. Finally, the output result is stored by overwriting the original mesh file.

Download and Update Modules

We implemented both modules as one-way Web services by using Java. They receive a SOAP message with the URL of the file they will use as an input (the input mesh and the correction file respectively), open an HTTP connection with the sender and perform the download.

4.3.2 Correction Encoding

In our setting, each Web service runs a geometry processing algorithm, keeps track of the editing operations, and saves them along with the final result. To do this, each algorithm has been enriched with proper code to stream such operations into the correction file. Each operation is identified by a unique opcode, while each simplex is uniquely identified by an integer ID. Thus, to represent an “edge swap” we need an opcode representing the swap operation and an integer identifying the edge to be swapped.

Besides such atomic operations, we include some derived functionalities that group atomic changes for the most diffused editing operations. In many cases this allows to further save storage space (and thus transmission time). For example, let us suppose that we need to subdivide a triangle into three subtriangles by inserting a new vertex: in this case we would need to encode a “remove triangle” (1 opcode + 1 ID), a “create vertex” (1 opcode + 3 coordinates), and three “create triangle” (3 opcode + 9 IDs for the vertices) operations. Conversely, if we include a single “split triangle” operation in our set, we can simply use its opcode endowed with the identifier of the triangle to be split and the coordinates of the splitting vertex. Appendix B reports a comprehensive overview of currently supported editing operations.

Sometimes a careful analysis of the algorithm at hand allows to avoid streaming all the operations. For example, let us consider an algorithm that performs N iterations of Laplacian smoothing on a mesh with V vertices. At each iteration all the vertices are moved to the center of mass of their neighbors, thus by a naive approach we would stream $N * V$ vertex shifts. A more clever implementation, however, can simply stream the eventual global shift once for each vertex, thus reducing the size by a factor of N.

When an algorithm terminates, the produced sequence of operations is further compressed through arithmetic coding to minimize redundancy [Sai02]. The application of the correction by the subsequent Web services requires less computational efforts and time than the rerun of the algorithm because of the fact that its analysis part and the operation precondition checks are not needed anymore.

Streaming Editing Operations

We implemented a C++ library providing such functions. Mainly, we support atomic operations to encode the insertion, removal and modification of single simplexes of any order (i.e. vertices, edges and triangles).

4.4 Results

For the sake of experimentation, the proposed Workflow Management System has been deployed on a standard server running Windows 7, whereas our Web services have been deployed on different machines to constitute a distributed environment. However, since all the servers involved in our experiments were in the same lab with a gigabit network connection, we needed to simulate a long-distance network by artificially limiting the transfer bandwidth to 5 Mbps. Then, to test such a system we defined multiple processing workflows involving the available Web services. The dataset has been constructed by selecting some of the most complex meshes currently stored within the Digital Shape Workbench (see Table 4.1).

Mesh	$\lvert V\rvert$	$\lvert T\rvert$	$\lvert C\rvert$	$\lvert B\rvert$
Sicily	1.391.754	2.775.090	16	23
Rome	957.456	1.911.110	1	1
Dolomiti	810.000	1.616.420	1	1
Isidore	1.071.671	2.128.494	161	404
Nicolo	945.924	1.886.968	103	157
Neptune	1.321.838	2.643.684	1	0
Ramesses	775.712	1.537.462	308	824
Raptor	1.000.080	2.000.000	51	0
Dancers	703.207	1.399.805	1	105

Table 4.1: *Dataset extracted from the Digital Shape Workbench. Acronyms indicate Number of Vertices ($\lvert V\rvert$), Number of Triangles ($\lvert T\rvert$), Number of Components ($\lvert C\rvert$) and Number of Boundaries ($\lvert B\rvert$).*

As an example, one of our test workflows is depicted in Figure 4.3. The input model (Figure 4.3a) has 160 spurious disconnected components that are removed by the first Web service (Figure 4.3b). Then one iteration of laplacian smoothing is applied (Figure 4.3c) by the second Web service to reduce the noise on the surface, while its 404 holes are patched by the third Web service implementing Liepa's [Lie03] hole filling algorithm (Figure 4.3d). Finally, degenerate triangles are removed by the fourth Web service (Figure 4.3e). This test gives a first idea of the benefits provided by our transfer protocol: for example, consider that all the simplexes removed in the first step ($\approx 3K$ vertices, $\approx 7.5K$ edges and $\approx 4.5K$ triangles) could be encoded within a 11 KB correction file, whereas the compressed output mesh file size was 20.5 MB.

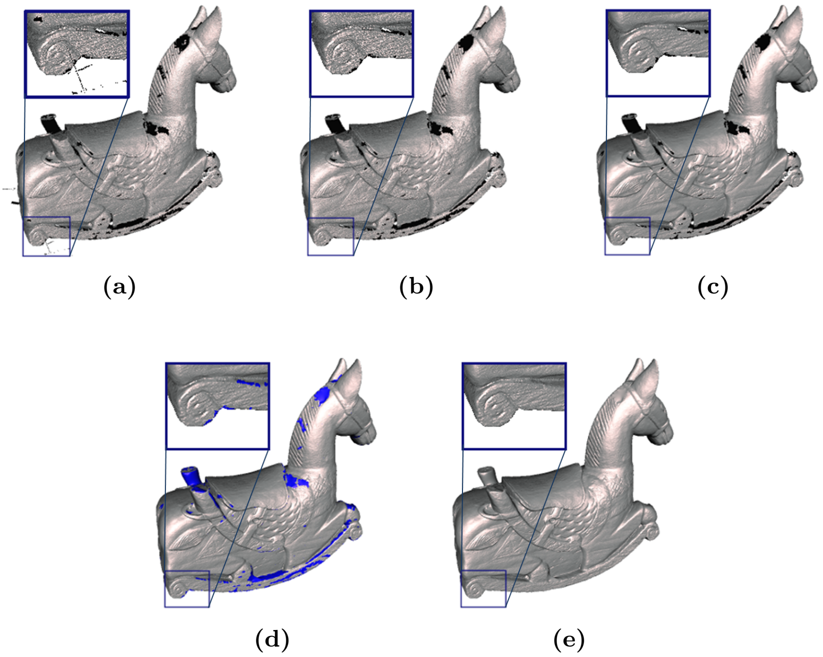


Figure 4.3: *A typical example of geometry processing workflow. (a) The raw model. (b) Smallest components removed. (c) Laplacian smooth applied. (d) Holes filled. (e) Degenerate triangles removed.*

The same workflow was run on all the other meshes in our dataset to better evaluate the performance gain achievable thanks to our concurrent mesh transfer protocol. Table 4.2 reports the size of the output mesh and the size of the correction file after each operation (both after compression), whereas Table 4.3 shows the total time spent by the workflow along with a more detailed timing for each single phase. In both tables, tasks are indicated by acronyms as follows: Removal of Smallest Components (RSC), Laplacian Smoothing (LS), Hole Filling (HF), and Removal of Degenerate Triangles (RDT).

Mesh	**RSC**	**LS**	**HF**	**RDT**
Sicily*	21.203 26	18.989 2.070	18.365 1	22.663 1
Rome*	14.915 1	15.551 1.425	14.915 1	13.166 1
Dolomiti*	11.146 1	11.637 1.402	11.146 1	10.588 1
Isidore	20.573 11	23.333 9.433	23.717 154	25.497 2
Nicolo	19.498 3	21.447 9.296	20.601 48	20.171 2
Neptune	39.881 1	40.131 15.237	39.891 1	39.937 1
Ramesses	17.484 3	19.544 8.754	19.934 149	19.802 3
Raptor	14.465 688	15.621 10.195	15.552 1	15.441 1
Dancers	16.457 1	18.037 7.220	18.325 80	18.116 1

Table 4.2: *Output sizes (in KB). For each mesh and for each task, the first line shows the size of the compressed output mesh, while the second line reports the size of the compressed correction. Average compression ratio is 5:1. Acronyms indicate Removal of Smallest Components (RSC), Laplacian Smoothing (LS), Hole Filling (HF), and Removal of Degenerate Triangles (RDT). Note that a modified version of the Hole Filling algorithm has been run to process "2.5D" geospatial data (*) in order to preserve their largest boundary.*

As expected, the corrections related to tasks that locally modify the model (eg. RSC, HF, RDT) are significantly smaller than the whole output mesh by several orders of magnitude; corrections regarding more "global" tasks (eg. LS) are also smaller than the output mesh, although in this latter case the correction file is just two/three times smaller than the whole output. Nevertheless, these results confirm that the proposed concurrent mesh transfer protocol provides significant benefits when the single steps produce mainly little or local mesh modifications.

For each mesh in our dataset, Table 4.3 reports the time spent by each algorithm to process the mesh (columns RSC, LS, HF, RDT), the time needed to transfer the correction file to the subsequent Web service (columns $T_1 \dots T_3$), and the time spent to update the mesh by applying the correction (columns $U_1 \dots U_3$). For the sake of comparison, below each pair (T_i, U_i) we also included the time spent by transferring the whole compressed result instead of the correction file, and the overall relative gain achieved by our protocol is reported in the last column. It is worth noticing that, in all our test cases, the sum of the transfer and update times is smaller than the time needed to transfer the whole mesh, with a significant difference when the latter was produced by applying little local modifications on the input. Clearly, the additional instructions introduced in the geometry processing algorithms to stream out the corrections should be considered for a fair comparison, but we have verified that such an overhead is definitely negligible with respect to the overall processing time of each algorithm, and therefore has not been reported in Table 4.3.

Typically, geometry processing algorithms include (1) an “analysis” part that performs the calculations to derive what to add, modify or remove, and (2) an “editing” part that applies such changes to the mesh. Depending on the algorithm these two parts may be not necessarily sequential, but the editing operations can always be tracked and are sufficient to reconstruct the result. The editing part is usually faster and it is the only one that must be replicated by the subsequent servers. In the worst case, when the mesh is completely rebuilt from scratch, this list is a sequence of “add vertex” and “add triangle” operations preceded by a “clear all” (see Appendix B).

As additional examples, Tables 4.4 and 4.5 show results concerning the execution of workflows involving conditional tasks and loops respectively. Table 4.4 shows execution times of a workflow that removes disconnected components and fills holes. The corresponding Web services are invoked only if appropriate precondition checks return true, that is that the input mesh has components to be removed and/or holes to be filled. Table 4.5 is related to a workflow involving a “while” loop where Laplacian smoothing is applied as long as the average normal instability [Att13] exceeds a threshold value. Times are related to a single workflow step (eg. a Web service execution or a data transfer), while numbers between parenthesis indicate how many times the corresponding step is run during loop execution.

Mesh	**IB**	**RSC**	**T$_1$**	**U$_1$**	**LS**	**T$_2$**	**U$_2$**	**HF**	**T$_3$**	**U$_3$**	**RDT**	**Total**	**Benefits**
Sicily*	30,4	9,1	0,0	6,5	12,6	3,3	14,1	8,6	0,0	6,4	10,7	101,7	62%
			33,9			30,4			29,4			165,1	
Rome*	20,4	5,8	0,0	0,0	8,4	2,3	9,4	5,5	0,0	0,0	6,9	58,7	104%
			23,9			24,9			23,9			119,7	
Dolomiti*	15,8	4,9	0,0	0,0	7,2	2,2	7,8	4,6	0,0	0,0	5,7	48,2	92%
			17,8			18,6			17,8			92,4	
Isidore	33,0	7,7	0,0	5,8	12,4	15,1	7,1	8,4	0,2	6,0	13,8	109,5	67%
			32,9			37,3			37,9			183,4	
Nicolo	31,2	6,5	0,0	4,8	10,5	14,9	6,1	7,5	0,1	4,9	11,5	98,0	69%
			31,2			34,3			33,0			165,7	
Neptune	63,8	13,0	0,0	0,0	18,6	24,4	11,0	12,6	0,0	0,0	14,4	157,8	99%
			63,8			64,2			63,8			314,2	
Ramesses	28,0	6,7	0,0	4,3	9,6	14,0	5,4	7,0	0,2	4,5	10,3	90,0	70%
			28,0			31,3			31,9			152,8	
Raptor	26,7	7,7	1,1	5,6	9,6	16,3	5,8	6,0	0,0	0,0	9,3	88,1	50%
			23,1			25,0			24,9			132,3	
Dancers	26,3	4,9	0,0	0,0	7,3	11,6	4,3	5,2	0,1	3,6	7,0	70,3	92%
			26,3			28,9			29,3			135,2	

Table 4.3: *Elaboration times (in seconds). Acronyms indicate Input Broadcast (IB), Removal of Smallest Components (RSC), Laplacian Smoothing (LS), Hole Filling (HF), and Removal of Degenerate Triangles (RDT). Cells labelled by* $\boldsymbol{T_i}$ *indicate the time needed to transfer the correction file. Cells labelled by* $\boldsymbol{U_i}$ *indicate the time needed to update the mesh by applying the correction. Note that a modified version of the Hole Filling algorithm has been run to process "2.5D" geospatial data (***) in order to preserve their largest boundary.*

In our test cases, the mesh transfer protocol reduces the execution time if the mesh satisfies the required mesh quality and therefore the subsequent Web service is invoked. No advantage and no degradation are introduced when the operation precondition does not hold due to the fact that no output file is transferred from one server to the others.

Mesh	**IB**	$\lvert C\rvert$	**RSC**	**T**$_1$	**U**$_1$	$\lvert B\rvert$	**HF**	**Total**	**Benefits**
Sicily*	30,4	5,0	8,6	0,0	6,6	5,1	8,5	64,2	43%
				33,9				91,5	
Rome*	20,4	3,3	–	–	–	3,3	—	27,0	0%
				–				27,0	
Dolomiti*	15,8	2,9	–	–	–	2,9	—	21,6	0%
				–				21,6	
Isidore	33,0	4,3	7,7	0,0	5,7	4,4	8,2	63,3	43%
				32,9				90,5	
Nicolo	31,2	3,5	6,5	0,0	9,5	4,4	13,1	68,2	32%
				31,2				89,9	
Neptune	63,8	7,8	–	–	–	7,7	–	79,3	0%
				–				79,3	
Ramesses	28,0	3,4	6,1	0,0	4,5	3,60	6,8	52,4	45%
				28,0				75,9	
Raptor	26,7	4,2	7,7	1,1	5,6	3,8	–	49,1	33%
				23,1				65,5	
Dancers	26,3	3,0	–	–	–	2,7	5,5	37,5	0%
				–				37,5	

Table 4.4: *Elaboration times (in seconds). Acronyms indicate Input Broadcast (IB), Check Number of Components ($\lvert C\rvert$), Removal of Smallest Components (RSC), Check Number of Boundaries ($\lvert B\rvert$) and Hole Filling (HF). Cells labelled by* ***T***$_1$ *indicate the time needed to transfer the correction file. Cells labelled by* ***U***$_1$ *indicate the time needed to update the mesh by applying the correction. Cells whose value is "–" indicate that the corresponding service has not been invoked because the operation precondition did not hold. A modified version of the Hole Filling algorithm has been run to process "2.5D" geospatial data (*) in order to preserve their largest boundary.*

To summarize, our tests show that the concurrent mesh transfer protocol considerably reduces the amount of data transferred among the servers, and thus the total elaboration time. As previously mentioned, if the output mesh produced by a Web service is smaller than the correction file, then such an output is forwarded to the subsequent servers that simply replace their local copy of the mesh. In an ideal system, the time needed to apply the correction should be taken into account as well before choosing whether to forward the whole mesh or the correction file. Unfortunately the update time depends on too many factors (i.e. architecture of the host server, current workload, ...) to be accurately guessed, but we argue that this is not a real issue because this case appears to be unlikely to happen in practice. Regarding the tasks that globally modify the mesh, such as the Laplacian Smoothing, we suspect that a clever analysis combined with coordinate quantization can provide a much more compact representation of the correction file.

Mesh	**IB**	*AVG_NI*	**LS**	$\mathbf{T}_N$	$\mathbf{U}_N$	**Total**	**Benefits**
Sicily	30,4	7,4 (4)	12,8 (3)	5,4 (3)	14,2 (3)	157,2	30%
				35,1 (3)		203,7	
Rome	20,4	4,7 (4)	8,4 (3)	3,6 (3)	9,4 (3)	103,4	31%
				23,6 (3)		135,2	
Dolomiti	15,8	4,2 (6)	7,3 (5)	4,1 (5)	8,3 (5)	139,5	29%
				20,5 (5)		180,0	
Isidore	33,0	6,3 (4)	12,7 (3)	20,1 (3)	7,1 (3)	177,9	26%
				42,8 (3)		224,7	
Nicolo	31,2	5,4 (4)	10,7 (3)	17,8 (3)	6,3 (3)	157,2	22%
				35,7 (3)		192,0	
Neptune	63,8	11,2 (5)	19,2 (4)	24,8 (4)	11,7 (4)	342,6	32%
				64,3 (4)		453,8	
Ramesses	28,0	5,4 (5)	9,8 (4)	14,7 (4)	5,7 (4)	175,8	28%
				32,7 (4)		225,0	
Raptor	26,7	6,3 (5)	11,5 (4)	19,3 (4)	7,1 (4)	209,5	5%
				29,1 (4)		220,7	
Dancers	26,3	4,5 (4)	8,5 (3)	13,3 (3)	5,3 (3)	125,6	28%
				30,48 (3)		161,3	

Table 4.5: *Elaboration times (in seconds). The test has been run on our dataset after the artificially addition of noise. Acronyms indicate Input Broadcast (IB), Laplacian Smoothing (LS) and Check Average Normal Instability (AVG_NI). Cells labelled by* $\boldsymbol{T}_N$ *indicate the time needed to transfer the correction file. Cells labelled by* $\boldsymbol{U}_N$ *indicate the time needed to update the mesh by applying the correction. Each elaboration time is related to a single workflow step. The number between parenthesis indicates how many times the workflow step is run during loop execution. Note that the precondition check is run once more, the last time it returns false and breaks the loop.*

Finally, we recognize that some algorithms that completely rebuild the mesh (e.g. from an intermediate representation such as in [Ju04]) can hardly be reproduced in a compact way through our current set of local editing operations (Table B.1). In these cases our mesh transfer protocol does not provide any advantage and the whole output mesh should be transmitted.

5 Workflow Formalization

*"A **process definition** normally comprises a number of discrete activity steps, with associated computer and/or human operations and rules governing the progression of the process through the various activity steps. The process definition may be expressed in textual or graphical form or in a formal language notation."*
– D. Hollingsworth, [Hol95]

Our system allows to create geometry processing workflows and to store them as abstract pipelines that can be eventually turned into actual mesh elaborations. To provide such functionalities, the system includes a specialized module responsible of storing workflows according to a formal representation and of supporting browsing and retrieval activities (Figure 5.1). Specifically, such a module is the interface between the graphical user interface and the workflow engine.

In more details, a *syntactical* tool turns the user-defined workflow specifications to a machine-readable form. The list of geometric operations and possible input parameters are stored in a file according to a specific XML format that takes into account the requirements of geometry processing. When an existing workflow is selected to be re-executed, the engine reads the corresponding XML file and re-builds the list of involved Web services.

Besides the list of geometric operations, additional information may be associated to a geometric workflow, such as a description and a purpose. A *semantically*-enriched workflow representation is exploited to store such an information and to support the retrieval of specific workflows. Thus, a specialized ontology is provided. The system is able to reason on the data stored in the ontology and to retrieve workflows that satisfy the user-provided requirements. Thanks to the workflow ontology, users are allowed to retrieve existing workflows suitable for their purpose by simply providing the desired requirements through the graphical interface.

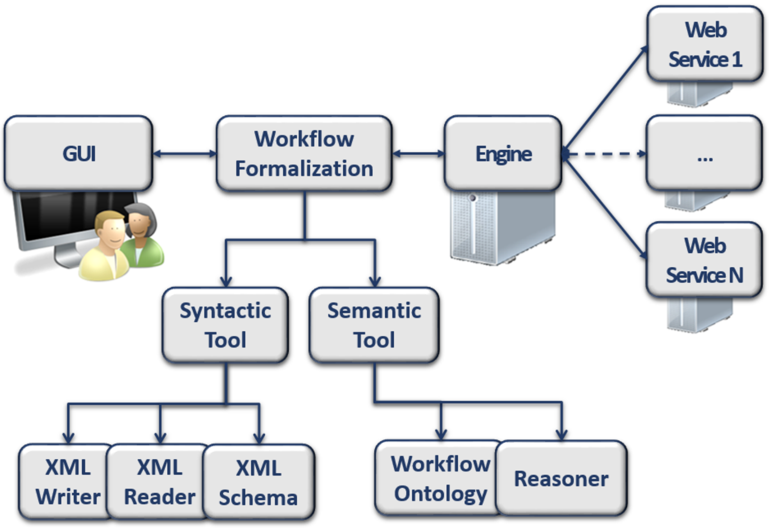


Figure 5.1: *The "Workflow Formalization" interface in the system architecture. Both a syntactical and a semantic tool are included. The former is responsible of storing workflows as XML files, while the latter supports the semantic search and retrieval of pre-defined workflows.*

5.1 The Workflow XML Language

Currently, each workflow system comes with its own workflow language designed to satisfy the needs of its specific target community. Based on existing workflow definition languages [FQH05] [PMP11] [vdAtH05] used in life science WFM systems, we defined a simple XML-based workflow representation format where the specific requirements of geometry processing are taken into account.

Workflow Language Definition

Constraints on the structure of a document representing any geometric workflow are formally defined through a XML-Schema.

XML Parsers

The Java Architecture for XML Binding (JAXB) [jax03] has been exploited to implement the modules responsible for reading and writing XML documents respectively. The former turns the user-defined workflow into a file, while the latter receives the identification of an existing workflow and turns it into a list of executable operations.

Such a formalization defines a workflow as an execution body that comprises an ordered list of tasks. Both atomic and conditional tasks (if-then-else) can be included in the workflow body, as well as sequential loops (while). The reference to a specific input model can be optionally included in the workflow to turn it to a specific elaboration.

Workflow XML File

The structure of a XML file representing a geometric workflow.

```
<?xml version="1.0" encoding="utf-8"?>
<workflow>
   <input> http://... </input>
   <body>
      <atomic> ... </atomic>
      <if> ... </if>
      <while> ... </while>
      ...
   </body>
</workflow>
```

5.1.1 Atomic Tasks

Each atomic task has a unique name and possibly a list of parameters. The name is meant to make the user immediately aware of which operation is performed by the task (e.g. "LaplacianSmoothing"), but it also represents a unique identifier used by the system to locate the proper Web service that actually performs the task. Appendix A.1 summarizes the list of currently supported atomic tasks.

Atomic Task Element

The structure of a XML element representing an atomic task.

```
<atomic>
   <name> ... </name>
   <parameters>
      <parameter> ... </parameter>
      <parameter> ... </parameter>
      ...
   </parameters>
</atomic>
```

5.1.2 Sequential Loops

Each sequential loop has a condition to be checked and an execution body. The condition is meant to describe the mesh quality type that must be satisfied to run all the tasks included in the following body (see Section 5.1.4).

Sequential Loop Element

The structure of a XML element representing a sequential loop.

```
<while>
   <condition> ... </condition>
   <body> ... </body>
</while>
```

5.1.3 Conditional Tasks

Similarly to sequential loops, each conditional task has a condition to be checked and an execution body. Optionally, an additional body (namely, *else*) can be included in such a task to describe a list of operations that must be executed if the required condition is not satisfied.

Conditional Task Element

The structure of a XML element representing a conditional task.

```
<if>
   <condition> ... </condition>
   <body> ... </body>
   <else> ... </else>
</if>
```

5.1.4 Analysis Tasks

Each analysis task has a unique name, a conditional operator and a target value. The name is meant to make the user immediately aware of which quality type is checked by the task (e.g. "NumberOfVertices"), but it also represents a unique identifier used by the system to locate the proper Web service that actually performs the task. Appendix A.2 summarizes the list of currently supported analysis tasks.

Condition Element

The structure of a XML element representing a condition.

```
<condition>
   <analysis>
      <name> ... </name>
      <operator> ... </operator>
      <value> ... </value>
   </analysis>
</condition>
```

5.2 Semantically Enriched Workflows

The aforementioned XML grammar defines a basic vocabulary and a fixed structure to describe and store a geometric workflow. Such a grammar allows to encode the list of geometric operations as a textual file that is both human- and machine-readable. The engine exploits such a XML file to re-execute an existing workflow, while users are allowed to read the file contents to improve their technical knowledge about a selected pipeline (eg. which operations are involved, possible input parameters, ...).

Nevertheless, such an approach does not provide the possibility to store any semantic information. Existing workflows can be retrieved, but users are required to manually "analyze" each of them to understand which ones are suitable for their purposes. Such an operation is often time-consuming and it is difficult for non-expert users to understand which worklow better fits with their requirements.

In our reference scenario, the user wants to retrieve existing workflows that satisfy specific requirements (eg. suitable for a specific purpose). To achieve the goal, the user should be allowed to access the browsing page (see Figure 3.3) and to query the system by simply providing semantic information (Figure 5.2). In the background, the workflow engine should be able to reason on existing workflows and to return the ones that satisfy the requirements.

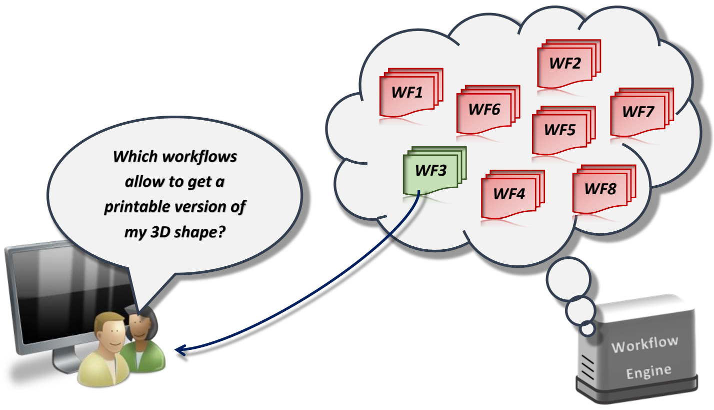

Which workflows allow to get a printable version of my 3D shape?
WF1
WF2
WF6
WF5
WF7
WF3
WF4
WF8
Workflow Engine

Figure 5.2: *Example of our reference scenario. Users should be allowed to query the system by providing a description of their purposes. The system should be able to reason on existing workflows and to retrieve the ones that satisfy the user requirements.*

5.2.1 Background

Ontologies represent a key element in knowledge management and content-based systems. Formally, two main definitions of ontology are provided in the literature.

> *'An* ***ontology*** *is an explicit specification of a conceptualization."*
>
> – T. Gruber, [Gru93]

> *'An* ***ontology*** *is a formal, explicit specification of a shared conceptualisation."*
>
> – R. Studer, [SBF98]

According to such definitions, an ontology is a set of concepts related to a specific domain that are defined in a structured manner. Designing an ontology actually means to define a set of semantic categories which reflect the conceptual organization of a specific domain. Thus, an ontology enables to express knowledge about the domain of interest by organizing such a knowledge into semantic entities.

Ontologies provide diverse advantages in terms of:

- *Sharing*: ontologies allow to share a common understanding by defining unambiguously the meaning of terms used in a specific domain;
- *Re-usability*: a knowledge domain defined through an ontology is reusable and interoperability among applications in such a domain is improved;
- *Reasoning*: an ontology allows to retrieve implicit knowledge about the domain by reasoning on explicitly encoded information

In general, ontologies have been exploited in different research and business areas (eg. medicine, biology, engineering, ...) to provide specific semantic-based information systems. Ontologies have also been exploited for the definition of specific process pipelines. We refer to [GBCL04] for a wide overview of the state-of-the-art and a deeper analysis of existing approaches.

5.2.2 The Workflow Ontology

The Workflow Ontology (WO) [ACG$^+$ed] is the knowledge base that allows to formally describe process pipelines in the Digital Shape Workbench (DSW) [dsw12]. The ontology is built on the top of the Common Info Ontology and of the Common Tool Ontology which organize the information about users and shape processing tools within the infrastructure.

In origin, the main idea behind the development of the WO was just to formally describe the so-called *static workflows*, which are a sort of tutorial/documental pipelines used for sharing the knowledge of expert persons in the area of a specific process. In its initial version, the WO was explicitly focused on processes for the transition of CAD models to their use in Virtual Reality.

Thanks to its modular structure, the WO can also be exploited and extended to conceptualize other shape-oriented workflows with different aims. Technically, we could achieve such a goal by defining a main class, namely *Workflow*, where pipelines coming from any application domain can be instantiated. A specific subclass, namely *WorkflowStatic*, provides the support to the main purpose of the ontology and contains the instances of tutorial/documental workflows.

The Workflow class

The *Workflow* class has a property, namely *WorkflowDomain*, that specifies the purpose of the workflow and its context of use. Moreover, the following attributes are required to define a workflow:

- *Name*: a user-understandable name for the workflow;
- *Description*: a textual description for the workflow;
- *Creator*: the registered user who created the workflow;
- *Creation Date*: date and time of the workflow creation.

Thanks to the aforementioned adaptation, we could extend the Workflow Ontology to support the user in browsing existing workflows and finding the ones that satisfy specific requirements. To achieve the goal, we extended the aforementioned ontology by providing a new subclass of the *Workflow* class, namely *WorkflowExecutable*. Such a subclass contains the instances of workflows generated through our system that can be remotely executed by taking advantage of specific Web Services.

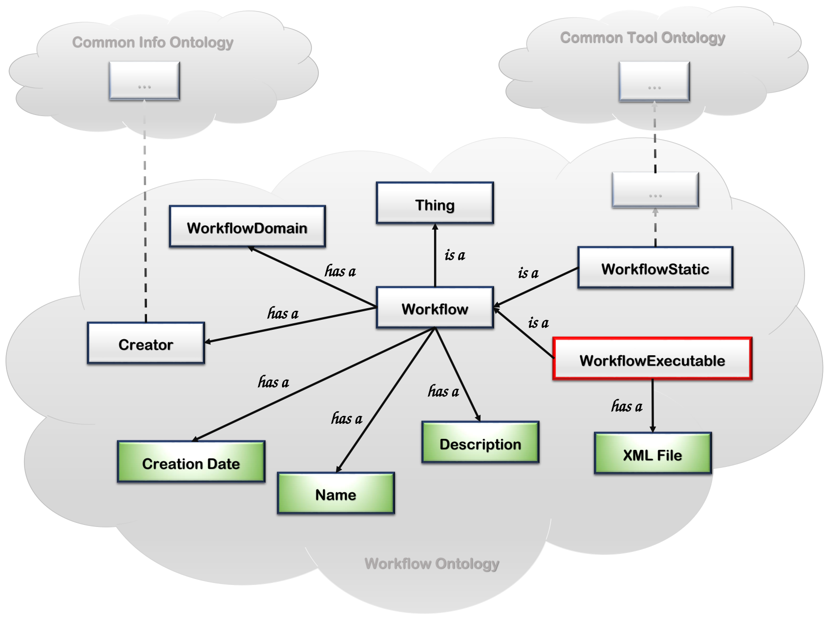


Figure 5.3: *The structure the WorkflowExecutable class in the Workflow Ontology. Grey boxes represent classes, while green boxes represent textual attributes. The label on each arrow shows the relationship between the two connected components.*

Technically, the *WorkflowExecutable* class inherits the attributes and properties of the *Workflow* class. As a consequence, a name and a description are required to define an executable workflow, as well as the name of the creator and the creation date. In addition to such an information, an extra attribute is required to store the name of the XML file generated by the engine. Figure 5.3 shows the structure of the *WorkflowExecutable* class.

5.2.3 Workflow Retrieval

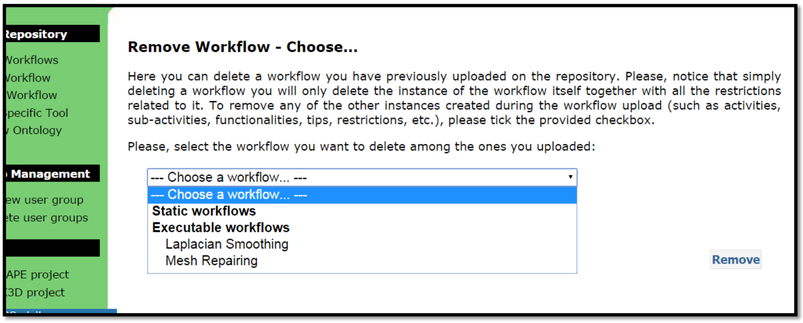
The graphical user interface exploits the extended version of the WO to support both the creation and the browsing activities. When a new workflow is created (see Figure 3.2), a new instance of the *WorkflowExecutable* class is generated. Both the name and the description provided by the user are exploited to set up the corresponding attributes, while the creator name and the creation date are automatically retrieved by the system.

On the other hand, the browsing page (see Figure 3.3) exploits the data stored in the ontology to retrieve specific workflows. The graphical user interface allows the user to set a few filter options (i.e. type of workflow, workflow purpose, input and output data types) for the retrieval of specific documental and executable workflows. When an executable workflow is retrieved, the user is allowed to visualize related information (see Figure 3.6) and to execute it on a selected input mesh.

Currently, the WO supports simple queries, such as:

- *Which workflows are currently available?*
- *Which workflows can be actually executed?*
- *Which workflows allow to achieve a specific purpose?*

As an additional example of exploitation of the WO, users are allowed to visualize all the workflows created by themselves and to potentially remove some of them (Figure 5.4).



Remove Workflow - Choose...

Here you can delete a workflow you have previously uploaded on the repository. Please, notice that simply deleting a workflow you will only delete the instance of the workflow itself together with all the restrictions related to it. To remove any of the other instances created during the workflow upload (such as activities, sub-activities, functionalities, tips, restrictions, etc.), please tick the provided checkbox.

Please, select the workflow you want to delete among the ones you uploaded:

Figure 5.4: *Detail of the graphical user interface. The user is allowed to manage his own workflows and possibly remove one of them.*

The WO can be further extended to provide support to more complex queries. An advanced browsing mechanism for workflows should be able to reason on the single steps that compose a pipeline and to "understand" if their combination allows the user to get a shape that satisfies specific requirements. To achieve the goal, both the main *Workflow* class and its subclasses must be provided with more attributes and properties for the description of each atomic operation. Both the Common Shape Ontology and the Common Tool Ontology within the DSW infrastructure already provide some formal conceptualizations of 3D shapes and software tools (eg. Web services) that can be exploited to extend the WO.

Part III

Parallelization

6 Introduction

"Parallelism refers to executing more than one task at the same time [...] It's important to distinguish ***parallelism*** *from* ***distribution****. Distributed computing is a specialization of parallel computing where the processors don't reside in the same computer and where tasks are distributed to computers over a network."*

– D. Higginbotham, [Hig15]

Our workflow-based framework has been designed to support collaborative research in geometry processing and any other research area where the generation and elaboration of 3D models is necessary. Expert programmers can avoid reimplementing or adapting known algorithms, for which the available Web services can be used, while they are free to focus on the development of the actually innovative parts. Conversely, scientists in any other field can exploit state-of-the-art algorithms and already defined compositions of them with no longer need to be skilled programmers or experts in geometric modelling. The intuitive graphical interface guides the user in both creating a new pipeline and browsing/reusing existing ones.

The platform is accessible from any operating system through a standard Web browser with no hardware requirements, and does not need any local software installation on the user machine. By distributing the workload, the system can count on considerable computational resources. Each invocable Web service is hosted by a remote server that makes its hardware and software configuration available for possible elaborations. Moreover, the system can be easily extended when a researcher wishes to make its own algorithm available as a Web service. There is no need to distribute the source code or executable files to improve the system capabilities, but the developer is simply asked to communicate to the system the address of its Web service and possible required input parameters.

6.1 Motivation

Although our system theoretically allows to process any input mesh, remote servers may not satisfy specific hardware and software requirements (eg. huge main memory, high computational performance) necessary to store and process extremely high resolution meshes. We expect that Web server computers used for research purposes are much more similar to commodity PCs than high-performance servers. Moreover, the algorithm provided by any hosted Web service may not be designed to efficiently perform the processing of large inputs.

The limited RAM provided by each available server may not be sufficient to load the entire input into main memory or to host all the support data structures required for specific elaborations, which are sometimes more memory-demanding than the model itself. As a consequence, the elaboration may be interrupted by the remote server whose available memory is not sufficient to perform the task.

It should also be considered that, even if the algorithm provided by the hosted Web service is suitable for managing large 3D geometric data (eg. out-of-core and parallel approaches are exploited), the memory available on the remote server imposes a sequentialization of I/O operations in any case. Since reading and writing files are time-consuming operations, the remote server may require a very long time to finish the elaboration.

6.2 Objective

In order to avoid excessively long computations and possible interruptions due to limited available memory during the runtime workflow execution, the system should be able to recognise large inputs and properly manage them. The traditional *divide and conquer* approach and parallel computing strategies should be exploited to efficiently process any valid input mesh, independently from its size.

In our reference scenario, the user wants to process a large mesh by exploiting the system. To achieve this goal, the user accesses the system through the graphical interface, uploads the input mesh and selects the desired operations. In the background, the Engine should analyze the size of the input mesh and check if the required servers satisfy the requirements to efficiently perform their tasks. If not, the Engine should be able to partition a large input mesh into smaller subparts, to distribute them across multiple available Web services for simultaneous processing, and to merge the processed subparts at the end of the elaboration to generate the final output (Figure 6.1). Both partitioning and merging operations should assure the possibility to process arbitrarily large inputs.

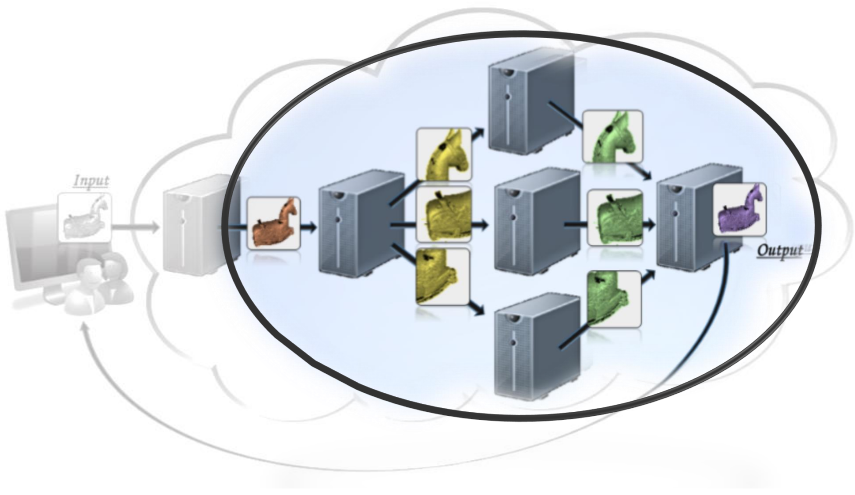


Figure 6.1: *An example of workflow execution where a parallel divide and conquer approach is exploited to allow the processing of a large dataset. The input mesh is partitioned into three portions that are simultaneously processed by three services. Finally, the processed portions are merged together to generate the final output.*

6.3 Challenges and Scientific Contributions

Several parallel approaches have been proposed to efficiently process large datasets. Typically, such methods subdivide a large input into subparts and elaborate them simultaneously by exploiting multi-core or many-core architectures (eg. GPUs). Although these approaches speed up the elaboration, they assume that a fast-access shared memory is available for communication among concurrent processes. Since such a memory is not available in distributed environments, these methods are not suitable and innovative solutions are required.

In most of existing parallel methods, the mesh is partitioned using an in-core algorithm. When the mesh is too large and available memory is not sufficient to load all the geometric and connectivity information, out-of-core partitioning is required to produce the subparts. Effective out-of-core techniques have been proposed, but they typically assume that the input mesh is represented as a list of triangles, each directly encoded by the coordinates of its three vertices. The high redundancy of such "triangle soups" represents a severe limitation when even more compact representations (i.e. indexed meshes) require giga or even terabytes of disk space [LPC$^+$00] [iqm13].

Moreover, after having processed each of the sub-meshes separately, these partial objects should be merged back into a single mesh. Some algorithms avoid to modify the sub-mesh boundaries to guarantee an exact match after the elaboration. Typically, post-processing is performed through in-core algorithms to enhance the output quality. When the resulting elaboration is still too large for in-core post-processing, contact borders can either be kept unprocessed or treated using a sub-optimal approach.

In order to support the distributed processing of large input meshes, an innovative divide and conquer approach needs to be accurately designed. Specifically, we propose an innovative out-of-core partitioning method for large indexed meshes that enables distributed parallel processing. Differently from the previous published methods, our divide and conquer approach allows to modify any part of generated submeshes, including their boundaries, and enables an exact match among them after the elaboration. Finally, an out-of-core merging algorithm is provided that assures the possibility to generate arbitrarily large outputs, with no need to perform any post-processing to enhance the final result.

As a proof-of-concept, we have implemented an innovative distributed mesh simplification algorithm that exploits our partitioning to distribute the computational load across multiple servers.

Summarizing, the following chapters aim to demonstrate how it is possible to efficiently process large meshes in a distributed environment through an innovative divide and conquer approach. Specifically, the following original contributions are provided:

- design and implementation of an out-of-core partitioning algorithm for indexed meshes
- design and implementation of an out-of-core merging algorithm that enables the generation of the final output, independently of the size of the processed submeshes

- design and implementation of a simplification algorithm that exploits our divide and conquer approach and proves the benefits provided by distributed processing

7 Related Works

"The divide-and-conquer technique is a natural way to express parallelism, since it repeatedly divides a problem into two or more smaller subproblems whose solutions can be computed simultaneously and independently."

– A. Zorat, [Zor79]

The well-known divide and conquer paradigm is often used in many application areas to efficiently solve conceptually difficult problems. The main idea is to break the problem into sub-problems, solve each of them and combine the results to generate the final solution. As underlined in [Zor79], the divide and conquer approach is "a natural one when a parallel processor is envisioned because each pair of recursive calls generate subproblems that can be solved in parallel".

In computer graphics and geometry processing, such an approach has often been exploited to allow the processing of large datasets. Specifically, the input mesh is partitioned into submeshes, each small enough to be processed with a traditional incore algorithm. Finally, when all the submeshes have been processed, they are merged together to generate the final output. To support the processing of input meshes that do not fit into main memory, the design of a divide and conquer algorithm should include the exploitation of out-of-core techniques. Moreover, parallel approaches may be used to assure efficiency and reduce the overall elaboration time.

The state-of-the-art includes several different solutions for processing large datasets. Mainly, the existing approaches include cutting the mesh into pieces, clustering mesh vertices, using external memory data structures, and processing stream data. The following sections describe the most relevant existing techniques. Most algorithms focus on mesh simplification, whereas other methods exist as a solution of specific problems, such as out-of-core mesh compression [HLK01] [IG03] and remeshing [AGL06]. For a comprehensive survey of existing out-of-core approaches used in visualization and computer graphics, we refer to [SCC$^+$02].

7.1 Sequential Processing

Out-of-core approaches assume that the input does not need to be entirely kept in memory, and the computation operates on the loaded parts at each time.

Processing sequences are used in [ILGS03]. A processing sequence represents a mesh as a particular interleaved ordering of indexed triangles and vertices. This representation allows streaming very large meshes through main memory while maintaining information about the visitation status of edges and vertices. At any time, only a small portion of the mesh is kept in-core, with the bulk of the mesh data residing on disk (Figure 7.1). Mesh access is restricted to a fixed traversal order, but full connectivity and geometry information is available for the active elements of the traversal.

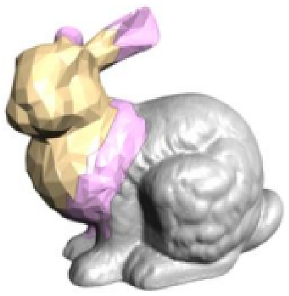


Figure 7.1: *Example of simplification using processing sequences. This snapshot shows the yet unprocessed part of the input data (grey), the current in-core portion (pink) and the already decimated output (gold). Courtesy of [ILGS03]*

Conversely, streaming meshes are exploited in [WK03] and [IL05]. These algorithms read the input from a data stream in a single pass and write the output to another stream while using only an in-core buffer (Figure 7.2). Since the stream algorithms use an in-core buffer of limited size, they assume that the geometry stream is approximately pre-sorted. When this mild sorting requirement is not satisfied, an out-of-core pre-sorting step is required before the processing [LS01].

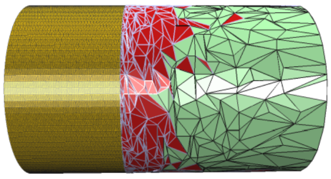


Figure 7.2: *Example of simplification using streaming meshes. This snapshot shows the yet unprocessed part of the input data (left), the current in-core portion (middle) and the already decimated output (right). The data in the original file happened to be pre-sorted from right to left. Courtesy of [WK03]*

A different approach is proposed in [CMRS03], where a smart hierarchical external memory data structure is proposed. It provides support for the management of generic processing on large meshes, under the constraint of limited core memory. Specifically, only the hierarchical structure is maintained in main memory, while only a small portion of the whole mesh is loaded in each instant of time.

The aforementioned approaches are very elegant and assure the possibility to manage arbitrary large datasets. However, conversion to appropriate processing sequences, mesh pre-sorting operations and generation of external memory data structures are non-trivial processes that require a significant time [IG03]. Moreover, since these methods are based on the idea of repeatedly loading parts of the input mesh, they are not suitable for a distributed setting.

7.2 Parallel Processing

To speed up the elaboration, parallel approaches are often exploited. Typically, existing parallel methods involve a "master" processor that partitions the input mesh and distributes the portions across different "slave" processors that perform the partial elaborations simultaneously (Figure 7.3). Multi-core technologies are exploited since they provide the possibility to process different subparts of the input simultaneously.

As an example, the many slave processors available in modern GPU-based architectures are exploited in [SN13], while multi-core CPUs are exploited in [TPB08]. Such methods assume that the available memory is shared among the current processes. For this reason, they are not suitable for distributed environments and a sequentialization of I/O operations is required in any case. Conversely, [TJL07] can operate without any shared memory and is designed for distributed environments, but input meshes are required to be small enough to be loaded into main memory.

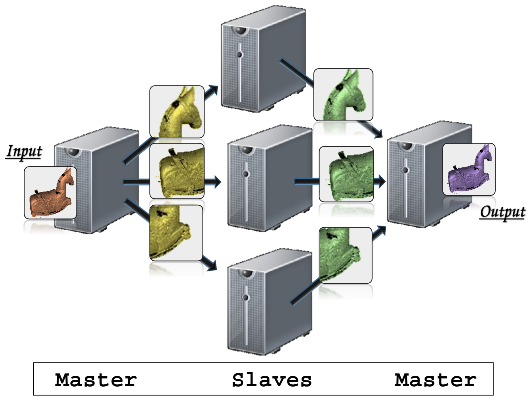


Figure 7.3: *The processing flow of a generic divide and conquer approach with parallel elaboration. In this example, the input mesh is subdivided into three portions and each of them is process by a different slave.*

7.3 Input Partitioning

Although the existing parallel methods are designed according to the general schema described above, the initial mesh partitioning can be performed through different approaches. Based on the mesh representation format and the input size, the appropriate partitioning algorithm needs to be exploited.

In [DLR00] [FS00] [TJL07], the mesh is partitioned using an incore algorithm: this is appropriate when memory is sufficient to store the entire mesh. In these cases, the mesh is partitioned by accumulating vertices and faces in subsets when travelling through the model. Typically, a starting vertex is chosen; then, the accumulation process is performed by selecting the neighbors of the starting vertex, and the neighbors of the neighbors, until the subset has reached the required size.

Conversely, when the plain mesh is too large, out-of-core partitioning is required to produce the submeshes. Effective out-of-core partitioning techniques are described in [Lin00] [LS01] [BP02]. These methods typically require their input to come as a triangle soup, that is a list of triangles, each directly encoded by the coordinates of its three vertices. In these cases, the bounding box for the input mesh is subdivided into cells and partitioning is performed according to vertex coordinates. Both in [Lin00] and [LS01], a vertex clustering approach is used, while in [BP02] each triangle is read from the input file and directly assigned to a specific cell according to its vertex coordinates.

The most diffused formats employ an indexed mesh representation: a first block in the file represents the vertex coordinates, whereas a second block represents each triangle as a triplet of indexes referred to the first block.

Before an indexed mesh can be used by the aforementioned partitioning methods, it needs to be converted into a triangle soup. This additional step is time-consuming and requires significant storage resources, since triangle indexes must be dereferenced using out-of-core techniques [CSS98]. Conversely, the method proposed in [SG01] is able to work with indexed representations by relying on memory-mapped I/O managed by the operating system; however, if the face set is described without locality in the file, the same information is repeatedly read from disk and thrashing is likely to occur.

7.4 Output Generation

Divide and conquer approaches typically produce the final output by merging the processed submeshes into a single mesh. Again, this operation can be performed according to different approaches, based on the kind of outputs returned by each partial elaboration.

When the resulting elaborations are comprehensively small enough to fit in memory, in-core methods can be exploited to merge and polish the final result. Processes responsible of running partial elaborations are often required to keep the sub-mesh boundaries unchanged [TJL07]. Such a restriction guarantees an exact match among contact regions of neighbor submeshes after the elaboration. The final output is then generated by unifying processed mesh portions in an in-core merging phase. If necessary, the quality of such a generated mesh is enhanced by exploiting traditional in-core algorithms in a final post-processing step. Conversely, when the resulting elaboration is still too large for in-core post-processing, contact borders can either be kept unchanged or enhanced using sub-optimal methods.

Differently, the smart octree-based external memory data structure proposed in [CMRS03] allows to keep the boundaries consistent at each iteration. Specifically, the data structure allows to dynamically load in main memory the selected portion to be processed and all its neighbors. During the elaboration, both inner and boundary elements in the selected portion may be modified, while the boundary of each neighbor is kept consistent. Unfortunately, as already mentioned above, the approach is not suitable for a distributed setting.

Depending on the specific type of elaboration, different approaches may be exploited to guarantee the boundary coherence. Vertex clustering is just an example used in mesh simplification, but such a method has often a cost in terms of output quality (see Section 9.2).

8

Out-of-core Processing

"But requiring all data to fit in memory means that if you have a dataset larger than your installed RAM, you're out of luck."
– A. Jacobs, [Jac09]

To support the process of large input meshes and avoid exceeding the memory capacity during execution, our workflow engine is able to partition a large input mesh into smaller subparts and to merge the processed subparts at the end of the elaboration to generate the final output. Both partitioning and merging operations are performed through out-of-core approaches.

Definitions From now on, we assume that the input mesh is encoded as an indexed mesh, since the most diffused file formats are based on this representation. Thus, the input mesh M is defined as a pair $\langle V, T\rangle$, where V is a list of vertices and T is a set of triangles. Each vertex v_i in V is encoded by its three coordinates, whereas each triangle t_i in T is encoded by three integer indexes: an index k identifies the k'th vertex in the list V. An analogous encoding is used to describe each submesh $M_i = \langle V_i, T_i\rangle$. Also, when dealing with submeshes we distinguish between local indexes and global indexes: a local index k in a submesh $M_i = \langle V_i, T_i\rangle$ identifies the k'th vertex in the list V_i, whereas a global index j identifies the j'th vertex in the overall V. In analogy with previous work on parallel processing, we considered our workflow engine as a "master", while the available servers as "slaves". For the sake of simplicity, our exposition assumes that all the servers have an equally-size memory and comparable speed.

8.1 Mesh Partitioning

Our solution requires two integer parameters: the number of vertices N_v that we wish to assign to each submesh (based on the memory available on each of the slaves) and the number of available slaves N_s that will run the partial mesh processing.

First, the mesh bounding box $B(M)$ is computed by reading once the coordinates of all the vertices V. At the same time, a representative vertex down-sampling V' is computed and saved to a file (one vertex out of 1000 is randomly picked from V in our implementation). Starting from $B(M)$, an in-core binary space partition (BSP) is built by iteratively subdividing the cell with the greatest number of V' points. Each cell is split along its largest side. The root of the BSP refers to the whole downsampling file V'. For each subdivision, each vertex in the parent cell is assigned to one of the two children according to its spatial location. If the vertex falls exactly on the splitting plane, it is assigned to the cell having the lowest barycenter in lexicographical order. The process is stopped when the number of vertices assigned to each BSP cell is at most equal to a given threshold, based on the available memory on each of the slaves and the ratio between M and the subsample size ($N_v/1000$ in our implementation).

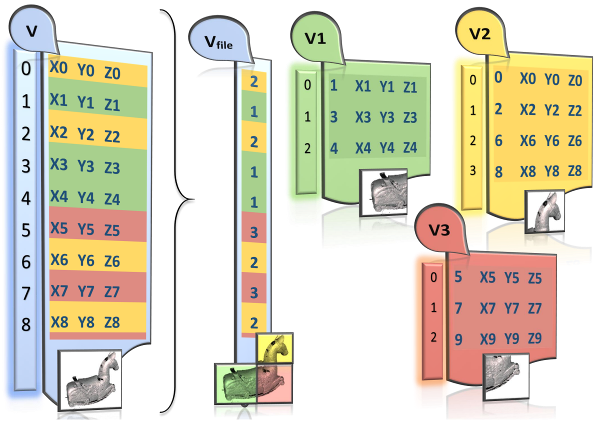


Figure 8.1: *Partitioning of vertices. For each BSP cell, a corresponding file is created. Vertices are read one by one and assigned based on their spatial location. Global indexes are shown on the left of the original V, while local indexes are on the left of each V_i. Global indexes and coordinates are written on each V_i. V_{file} stores, for each vertex, the ID of the corresponding BSP cell.*

Once the BSP is built based on V' as described above, all the vertices V and triangles T of the original M need to be assigned to the appropriate BSP cell. Vertices are read one by one and assigned based on their spatial location as above. Then, for each BSP cell C_i, a corresponding file V_i is created where both the global index and the coordinates of all the assigned vertices are stored (see Figure 8.1). Simultaneously, a global vector file V_{file} is created where, for each vertex, the ID of the corresponding BSP cell is stored. Then, the partitioner assigns to each BSP cell the corresponding inner triangles. For each BSP cell, a corresponding file T_i is created where triplets of global indexes are stored for all the triangles assigned to that cell. Triangles are read one by one from T and assigned depending on their vertex position as follows:

1. All the three triangle vertices belong to the same BSP cell C_A (Fig. 3a). The triangle is assigned to that same cell, that is, its three global indexes are written to T_A.
2. Two vertices belong to cell C_A while the third vertex belongs to cell C_B (Fig. 3b). The triangle is assigned to cell C_A along with a copy of the third vertex. Namely, the three global indexes are written to T_A, and a pair $\langle$*global index, coordinates*$\rangle$ is added to V_A to represent the third vertex.
3. The three vertices belong to three different cells C_A, C_B, and C_C (Fig. 3c). The triangle is assigned to the cell having the smallest barycenter in lexicographical order (let it be C_A), and a copy of each vertex belonging to the other two cells is created. Thus, the three global indexes are written to T_A, and two pairs $\langle$*global index, coordinates*$\rangle$ are added to V_A to represent the other two vertices.

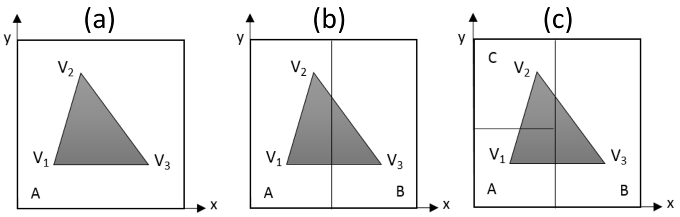


Figure 8.2: *Triangle Assignment. (a) The triangle is assigned to C_A, that is, its three global indexes are written to T_A (b) The triangle is assigned to cell C_A along with a copy of v_3. (c) The triangle is assigned to the cell having the smallest barycenter in lexicographical order (C_A), and a copy of v_2 and v_3 is created.*

To compute the cell containing each triangle vertex, the partitioner takes advantage of V_{file}.

Retrieving Vertex Coordinates

When a triangle is not completely included into a single BSP cell (e.g. one or two vertices are assigned to neighbor cells), coordinates of its vertices must be retrieved and assigned to the corresponding BSP cell to properly rebuild the face geometry (Figure 8.2b–8.2c).

In order to solve the issue, during $B(M)$ computation the engine creates a file V_{binary} that represents the original vertex list in binary format. Thanks to the constant size of vertices, V_{binary} can be randomly accessed to deference these few vertices (i.e. $O(\sqrt{N})$ out of a total of N vertices in M).

Note that V_{binary} is created only when the input mesh is encoded as a textual file. Conversely, when a binary file is provided as an input, it can be randomly accessed to deference the required vertices and no other support file needs to be created.

Loading V_{file}

Since V_{file} may be too large to be completely loaded, the engine is able to load it portion by portion.

Let max_v be the maximum size of V_{file} that the engine is able to load in memory. Thus, the partitioner keeps in main memory a sub-portion of it of size max_v. For each triangle vertex, if the corresponding index is in the loaded sub-portion, the ID of the corresponding BSP cell is retrieved; if not, the partitioner loads the sub-portion centered around the vertex index.

In practice, coherence among vertex IDs is usually satisfied, that is, most pairs of connected vertices have similar global indexes; thus, the engine needs to load a different sub-portion of V_{file} just a few times.

Also, the number of cells is usually small enough to assure that corresponding IDs can be encoded in 2 bytes. This means that two billion vertices may be indexed within a 4GB V_{file}. Based on this observation, we can affirm that the problem of loading V_{file} portion by portion is strictly limited to extreme cases (e.g. processing of huge meshes on very low memory engines).

At the end of the triangle classification, the BSP leaf cells represent a triangle-based partition of the input mesh geometry. Each sub-mesh is stored as a pair of files representing its vertices and triangles. Note that global indexes still need to be converted to local indexes, but this operation is delegated to the slaves that can undertake it in parallel. Also, an additional file B_i is created where the submesh boundary is described as a list of vertices, each encoded as its local index and the list of cells sharing it. This information will be reused during simplification for keeping boundary consistence among neighbors.

Opening and Closing Files

Since opening and closing files are time-consuming operations, the engine should avoid closing a file if not strictly necessary.

Let max_f be the maximum number of files that the operating system is able to open simultaneously. Thus, when the number of BSP cells is larger than max_f, an intelligent file access scheduling is necessary to efficiently create the submeshes.

In our implementation, a list of open files is maintained. Before any operation on a specific file, the engine checks if it is open; if not and if max_f files are already open, the list is browsed, the file whose last modification is oldest is closed and the required one is open.

8.2 Independent Sets

When a *divide and conquer* approach is exploited and subparts of the original input are processed in parallel, explicit communication and synchronization among processes is often required. Typical multi-core methods communicate based on a fast-access shared memory which is not available in a standard distributed environment. We propose a method to support distributed elaborations involving processes that require only local information (eg. elements that are inside or on the boundary of the considered submesh), but are allowed to modify any part of the submesh, including its boundary. Specifically, we exploited the concept of independent sets to reduce communication among servers.

During partitioning, the generated submeshes are also grouped into independent sets so that submeshes in the same group do not share any vertex.

Clearly, each independent set must be composed by at most N_s sub-meshes, thus our ISs are not necessarily maximal. An adjacency graph for the sub-meshes is defined where each node represents a BSP cell, and an arc exists between two nodes if their corresponding BSP cells are "mesh-adjacent" (Figure 8.3a). Two cells are considered to be mesh-adjacent if their corresponding submeshes share at least one vertex, that is, at least one triangle is intersected by the splitting plane between the two cells. Based on this observation, the adjacency graph is built during triangle partitioning and kept updated at each assignment. For each triangle whose vertices are assigned to different BSP cells (Figures (8.2b) and (8.2c)), corresponding arcs are added to the graph.

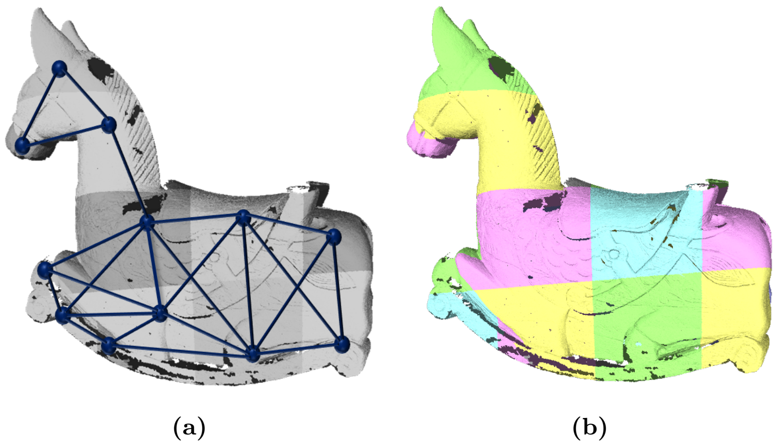


Figure 8.3: *The input mesh subdivided into independent sets of submeshes. (a) Adjacency graph. (b) Independent sets. Submeshes colored with the same color belong to the same set.*

The problem of grouping together submeshes that are independent (e.g. no arc exists between the corresponding nodes) is solved by applying a greedy graph coloring algorithm [HKK14]. The maximum number of nodes included in the same group (Figure 8.3b) is limited by N_s.

The final output of the partitioning step is a list of groups of sub-meshes where each group contains independent sub-models.

8.3 Output Merging

When all the sub-meshes have been processed, the engine is responsible of merging them and generating the output indexed mesh. Since the final output may be too large to fit into memory, the output merging is performed based on an out-of-core approach.

To allow the engine to achieve its goal, each processing service is required to return both the output submesh M'_i and an extra file B'_i where the list of boundary vertices of M'_i is saved, sorted by their local index. Specifically, M'_i must be represented as an indexed mesh, while each boundary vertex in B'_i must be described as a pair $\langle$*local index, global index*$\rangle$ (Figure 8.4).

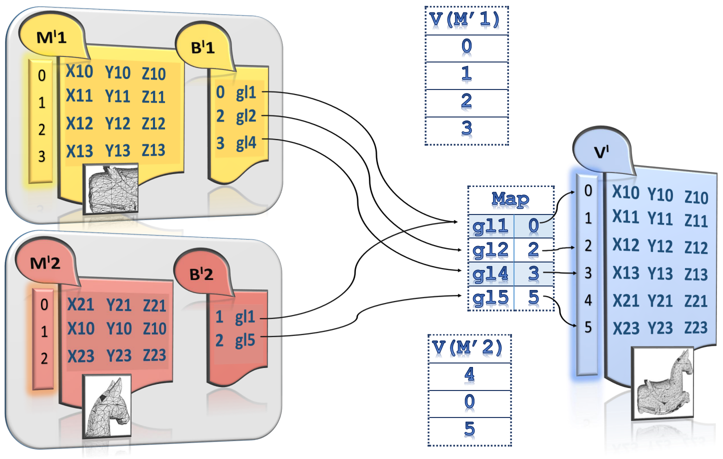


Figure 8.4: *Merging vertices. First, vertices in M'_1 are read one by one and coordinates are added to V_f. For each boundary vertex in M'_1, its global index is saved in Map. Map is sorted by global indexes ($gl_0 < gl_3 < gl_4 < gl_6$). Then, vertices in M'_2 are read and coordinates are added to V_f. Vertex 1 in M'_2 is not added to V_f since it is on the boundary and its reference is already in Map, that is, its coordinates are already written in V_f.*

Two temporary files (V_f and T_f for vertices and triangles respectively) are incrementally built to represent the overall mesh M'. A counter V_c is initialized to 0 and used to store the number of vertices written in V_f. Also, an in-core map Map is used to store, for each boundary vertex already written to V_f, a mapping between its global index and its position in V_f (i.e. final index).

Iteratively, each M'_i is handled. First, an additional in-core vector $V(M'_i)$ is allocated to host the final index of each vertex in M'_i. Then, the first pair $\langle l, g\rangle$ in B'_i is loaded into main memory and the list of vertices in M'_i and B'_i are read "in parallel" as follows. For each vertex v in M'_i, corresponding coordinates are read. If the local index of v is not equal to l, v is an inner vertex of M'_i. In this case, its coordinates are added to V_f, $V(M'_i)$ is updated by storing V_c as final index of v and V_c is incremented. This procedure is followed until a boundary vertex is found. When it happens (i.e. when v's local index is l), the engine checks if v is already in V_f. This check is performed by searching the global index g of v in Map. Since Map is sorted by global index, this search requires log_n operations, where n is the number of boundary vertices already added to V_f. If it is found, the final index of v is retrieved from Map and used to update $V(M'_i)$. If not, coordinates of v are added to V_f, $V(M'_i)$ is updated by storing V_c as final index of v, the mapping between the global index of v and its final index V_c is added to Map, and V_c is increased. When the boundary vertex v has been handled, the next pair in B'_i is read and loaded into memory.

$V(M'_i)$ is exploited to rebuild triplets representing triangle vertices of M'_i according to the final indexing. Each triangle t in M'_i is represented as a triplet (v_1, v_2, v_3), where v_1, v_2 and v_3 are local indexes of its vertices. To redefine t according to the final indexing, positions v_1, v_2 and v_3 in $V(M'_i)$ are directly accessed, and final indexes v'_1, v'_2 and v'_3 are retrieved and added to T_f. M'_i and B'_i files are closed as soon as they are completely scanned and $V(M'_i)$ is deleted to free occupied memory.

When all submeshes have been handled, V_f represents the list of vertices in M', while the list of triangles is saved in T_f. A final file M'_f representing the entire M' is built just appending an information header, V_f, and T_f in this order.

Out-of-Core Merge Algorithm

The following pseudocode shows the procedure to merge n submeshes M'_i into a single file M'_f.

```
procedure MERGE($M'_1, ..., M'_n, B'_1, ..., B'_n$)
    Create $V_f$ and $T_f$ files
    Create empty $Map$
    $V_c \leftarrow 0$
    for each pair $\langle M'_i, B'_i \rangle$ do
        $\langle l, g \rangle \leftarrow$ first pair in $B_i$
        Allocate $V(M'_i)$
        for each $v \in M'_i$ do
            $l_v \leftarrow$ local index of $v$
            if $l_v \neq l$ then                    ▷ $v$ is an inner vertex
                Write $v$ coordinates in $V_f$
                $V(M'_i)[l_v] \leftarrow V_c$
                $V_c \leftarrow V_c + 1$
            else                                    ▷ $v$ is a boundary vertex
                $f_v \leftarrow Map.find(g)$
                if $g$ is not found then            ▷ $v$ is not in $V_f$
                    Write $v$ coordinates in $V_f$
                    $V(M'_i)[l_v] \leftarrow V_c$
                    $Map.add(\langle g, V_c \rangle)$
                    $V_c \leftarrow V_c + 1$
                else                                ▷ $v$ is already in $V_f$
                    $V(M'_i)[l_v] \leftarrow f_v$
                $\langle l, g \rangle \leftarrow$ next pair in $B'_i$
        for each $t := (v_1, v_2, v_3) \in M'_i$ do
            $v'_1 \leftarrow V(M'_i)[v_1]$
            $v'_2 \leftarrow V(M'_i)[v_2]$
            $v'_3 \leftarrow V(M'_i)[v_3]$
            Write $v'_1$, $v'_2$ and $v'_3$ in $T_f$
        Delete $V(M'_i)$
    Create final file $M'_f$
    Write header information in $M'_f$
    Append $V_f$ and $T_f$ to $M'_f$
    return $M'_f$
```

9 Distributed Mesh Simplification

“Data matures like wine, applications like fish.”
– J. Governor, [Gov07]

As a proof-of-concept, a distributed simplification algorithm has been implemented that exploits our divide and conquer approach. Our algorithm allows to simplify arbitrarily large triangle meshes while leveraging the computing power of modern distributed environments. Our method combines the flexibility of out-of-core techniques with the quality of accurate in-core algorithms, while representing a particularly fast approach thanks to the concurrent use of several available servers. When compared with existing parallel algorithms, the simplifications produced by our method exhibit a significantly higher accuracy.

9.1 Objective

In our reference scenario, the user wants to simplify a large mesh by exploiting the system. To achieve this goal, the workflow engine should exploit the divide and conquer approach proposed in Chapter 8. Therefore, the engine should allow the processing of a large mesh by partitioning it through the previously described method (Section 8.1). Moreover, efficiency should be guaranteed by enabling parallel processing, that is, by invoking available Web services to simultaneously simplify the generated submeshes.

On its turn, each Web service should be able to load the received submesh and to simplify it by preserving the original appearance of the shape. To achieve this goal, an adaptive approach should be exploited, that is, flat areas should be strongly decimated, while shape features in morphological rich areas should be maintained. Moreover, boundaries of adjacent submeshes should be guaranteed to exactly match, so that the previously described merging method (Section 8.3) might be exploited to generate the final output.

To summarize, the following aspects should be taken into account when designing a distributed simplification algorithm for large meshes:

- Out-of-core initial partitioning;
- Out-of-core final merging/post-processing;
- High quality of the simplification/ adaptivity;
- Efficiency and distributable load.

9.2 Background

In the last decades, the evolution of 3D acquisition technologies called for methods to simplify meshes that have become large and larger. Earlier simplification algorithms [GH97] [HG97] required the whole mesh to be loaded into main memory and could focus on efficiency and accuracy only. Mainly, such methods exploit two different approaches, namely *edge collapse* and *vertex clustering.*

Typically, standard iterative edge-collapse approaches aim to preserve the appearance of the original shape. In this methods, every edge is assigned a “cost” that represents the geometric error introduced should it be collapsed. On each iteration, the lowest-cost edge is actually collapsed, and the costs of neighboring edges are updated. Today we know that methods based on iterative edge collapses driven by quadric error metrics are both efficient and, under certain conditions, provably optimal [HG99]. In such methods, a quadric matrix associated with each vertex represents a set of planes. This set is initially made by the planes of triangles that meet at the vertex. After an edge collapse, the resulting representative vertex v is associated with the sum of the quadric matrices of the original endpoints. In other words, the quadric Q at v represents the union of all the planes meeting at the original endpoints, and the error (i.e. the edge cost) at v is:

$$e(v) = v^T Q v = \sum_{i=0}^{N_p} d(v, \pi_i)^2 \tag{9.1}$$

where $d(v, \pi_i)$ is the distance from v to plane π_i and N_p is the cumulative number of planes.

Conversely, methods based on vertex clustering were originally proposed to handle meshes of arbitrary topological structure [RB93]. In these methods, a bounding box is placed around the mesh and subdivided into a 3D grid. Then, all the vertices in a given grid cell are clustered to the position of the most representative point. The algorithm is extremely efficient and simple to implement. However, vertex clustering can drastically alter the topology of the input mesh, and does not produce very faithful geometric approximations, especially when simplifying meshes with fair morphological variations.

Soon, however, too large meshes appeared that could not fit in main memory, and existing algorithms needed to be redesigned to account for an appropriate out-of-core elaboration. In most of these methods the mesh is partitioned in several sub-meshes, each small enough to be processed with traditional algorithms (see Chapter 7).

As an example, the vertex clustering approach by Rossignac and Borrel [RB93] is often exploited to perform out-of-core simplification [LS01] [SG01]. Such an approach guarantees that adjacent mesh portions have coherent common boundaries, but the output quality may be damaged. In [Lin00], the vertex clustering approach is modified to use a quadric error metric to compute the representative vertex. With respect to [RB93], this choice improves the quality of the resulting mesh. The adaptive clustering employed in [SG01] leads to an even higher quality result, but is still not comparable with traditional methods based on global priority queues [GH97].

On the other hand, [CMRS03] provides high quality simplifications by exploiting the traditional edge collapse approach, but is not suitable for a distributed setting. In [BP02] this possibility to exploit distributed environments is considered but, due to the use of a shared memory, the approach proposed is appropriate only on high-end clusters where local nodes are interconnected with particularly fast protocols. To the best of our knowledge, the only existing technique that can operate without any shared memory is described in [TJL07], but out-of-core partitioning/merging is not supported.

9.3 The Algorithm

The distributed simplification algorithm works as shown in Figure 9.1. In the first step, the engine partitions the mesh into a set of submeshes using the previously described algorithm (Sec. 8.1). Submeshes are then grouped into independent sets. Each independent set is guaranteed to contain at most N_s submeshes to be simultaneously sent to the services for simplification. In the first iteration, each submesh is simplified in all its parts according to the target accuracy. Besides the simplified mesh, each service is required to produce an additional file identifying which vertices on the submesh boundary were removed during simplification. This information is appended to adjacent submeshes and used as a constraint during their own simplification. When all the independent sets are been processed, the engine employs our out-of-core algorithm (Sec. 8.3) to join the simplified submeshes along their boundaries, which are guaranteed to match exactly.

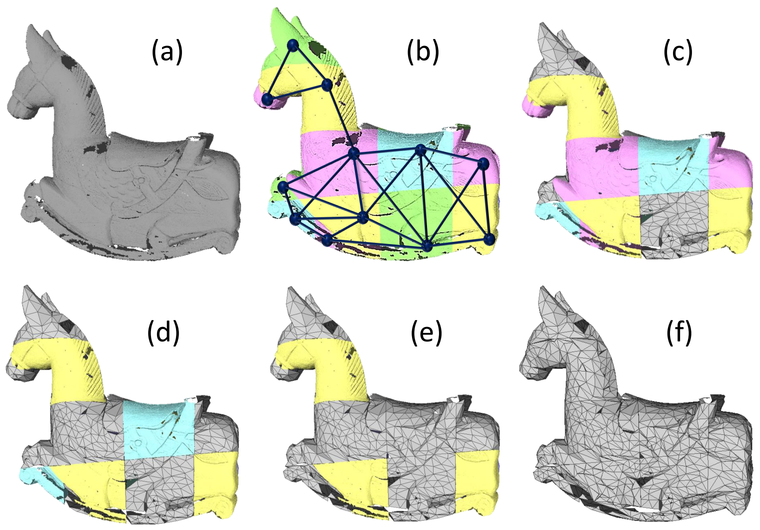


Figure 9.1: *Distributed simplification. (a) Input mesh. (b) Independent sets of submeshes. (c)(d)(e) Simplification steps. (f) Merged final output.*

9.3.1 Adaptivity

Each submesh is simplified through a standard iterative edge-collapse approach based on quadric error metric [GH97]. In order to preserve the appearance of the original shape and support adaptivity, the simplification algorithm applied by each server stops when a maximum error max_E is reached. Specifically, each server stops the simplification when the rooted mean square distance $\overline{d(v)}$ from a vertex v to its planes exceeds max_E. In our implementation,

$$\overline{d(v)} = \sqrt{\frac{e(v)}{N_p}} \tag{9.2}$$

where N_p is the cumulative number of planes.

Thanks to this approach, flat submeshes are strongly decimated, while shape features are fairly maintained in morphologically rich submeshes.

Loading A Submesh

Each server i receives its own submesh M_i represented as pair of binary files, storing vertices and triangles respectively. Each vertex is represented as a pair ⟨*global index, coordinates*⟩, while a triangle is a triplet of global indexes. Each server loads incore the list of n_v submesh vertices, sorted by global index. For each triangle in the submesh, its triplet of vertices is dereferenced by searching their global indexes in the sorted list. This operation requires $O(log_{n_v})$ operations for each triangle. Note that since M_i' is sufficiently small, standard data structures can be used to represent it incore.

Edge Collapses

Edge collapses are performed exploiting three different approaches, according to the position of the selected edge with respect to the submesh boundary. Half edge collapse is performed when one or both endpoints are on the border. In the former case, the edge collapses to its boundary vertex, while in the latter it collapses to the endpoint whose associated error is the lowest. Full edge collapse with optimal point placement [GH97] is performed otherwise.

9.3.2 Boundary Coherence

During the simplification of a submesh, elements that fall on its boundary are also reduced, both on the original boundary of the input mesh M (if any) and on the boundaries of submeshes. On the other hand, vertices shared among three or more submeshes are constrained and never removed to preserve the original topology.

In order to keep the boundary of the n neighbor submeshes consistent, each server is required to:

- check if some part of the boundary were previously simplified by some neighbor and, if so, reapply the same modifications before starting the simplification
- create a set of n files $\{Rem_k \mid k = 1, ..., n\}$, each containing the list of boundary vertices shared with the unprocessed neighbor k and removed during the simplification process

At the end of each iteration, the master receives some files from each server: the simplified submesh M'_i, the corresponding set of Rem files, and an additional file B'_i storing the global index of all the remaining (i.e. unsimplified) boundary vertices of M'_i. During the simplification of a single IS, no direct communication among the servers takes place. At subsequent iterations, the master is responsible of forwarding Rem_k to the server that will process the adjacent submesh k.

Boundary Information

Besides geometry and connectivity information, each server also receives one more file B_i that describes the submesh boundary and the (possibly empty) set of Rem files. The boundary B_i is described as a list of vertices sorted by their local index, while deleted vertices in each Rem_k are identified by their global indexes.

Boundary Update

Each vertex v in M_i shared with previously simplified submeshes is considered. For each v, if it was deleted by a neighbor, a half edge collapse is performed to delete the same vertex in M_i too, independently of the quadric error associated to v in M_i; otherwise, v is constrained and not deleted during the simplification of M_i.

9.4 Results

We performed our tests on a lab network of PCs, each equipped with Windows 7 64bit, an Intel i7 3.5 GHz processor, 4GB Ram and 1T hard disk. Connection among machines is provided through a 100 Mbps Ethernet. One machine plays the part of master, while the others are considered as servers.

Our dataset (Table 9.1) includes large meshes extracted from the Stanford online repository [sta96], from the Digital Michelangelo Project [mic09] and from the IQmulus Project [iqm13]. Some small meshes have been included in our dataset to evaluate and compare the error generated by the simplification by exploiting dedicated tools (Metro [CRS98]) used in previously published works.

For each input model, we run several tests by varying the number of servers and the maximum error threshold. We fixed the number N_v of vertices that should be assigned to each submesh to 1M for very large input meshes. Even if servers can manage more data, lower thresholding were used for the smaller meshes to provide a fair comparison with existing work. The initial vertex down-sampling is always performed with ratio 1:1000, since we empirically found that it provides a sufficiently representative subset.

Our experiments aimed to evaluate both elaboration times and quality of output meshes. Also, an analysis of the required memory space is provided, in terms of both main and secondary memory.

9.4.1 Elaboration Time

For each execution, our algorithm performs the following steps and the total elapsed time is the sum of times required to run each of them:

1. Computation of $B(M)$ and V'
2. Computation of the N cells of the BSP
3. Vertex classification
4. Triangle classification
5. Calculation of the independent sets
6. Delivery of first IS to the servers
7. Simplification of first IS and delivery of second IS
8. Simplification of second IS, delivery of third IS, and return of first simplified IS
   ...
   ...
9. Merge

Steps 1–5 are required to perform input partitioning. The time spent to undertake each step depends on the hardware performances and strictly depends on the number of I/O operations. The original list of vertices is read twice (step 1 and 3) and split into N files (step 3), while triangles are read and written once (step 4). Step 5 is performed in-core and does not require any I/O operation.

Steps 6–8 are required to send submeshes to servers, perform simplification and return decimated models to the master. Time spent for slot 6 depends on the network speed. A mesh made of 1M triangles occupies $\approx$10 Mbytes which can be transferred in about 0.8 seconds on a typical lab network (100 Mbps). Such a mesh takes approximately 10 seconds to be simplified on a commodity PC. Processing incore data and transferring disk data exploit different hardware resources, which means that these two duties can be undertaken in parallel. Hence, in time slots 7 and 8 the time to tranfer data can be neglected, as it is shorter than the processing time. This means that the only overhead due to the use of a limited network speed is represented by time slot 6.

During the last step the master merges the simplified submeshes into a single final output. Time required to perform this operation is proportional to the final output size.

In order to evaluate execution time, we run distributed simplification on large models. Results are shown in Table 9.1. Total elaboration times refer to the entire processing, including partitioning, simplification and final merge. Each server exploits the incore simplification algorithm proposed in [fas14], slightly modified to stop when max_E is reached and to return both M'_i and B'_i. For each dataset, we evaluate the sequential runtime by using a single server and by summing the time required to process each submesh. Then, some experiments are performed by increasing the number of available servers and computing simplification time as sum of time required to process each IS. The achieved speedup S_i is also shown, computed as $S_i = \frac{Time_1}{Time_i}$, where $Time_1$ is the sequential time and $Time_i$ is the time required to run the simplification on i servers. As expected, speedups are higher when the number of available servers increases. More noticeably, speedup increases as the input size grows.

It should be considered that, in a theoretical scenario where infinite servers are available, the best performance is achieved by getting rid of the maximum number of submeshes in an independent set and exploiting a number of servers equal to the number of submodels in the largest group. In this case, the four colors theorem [Wil02] guarantees that four ISs can be computed in the worst case for any input mesh.

As a summarizing achievement, our method could simplify the 25GB OFF file representing the “Atlas” model (≈ 0.5 billion triangles) in ≈ 25 minutes. As a matter of comparison, the master’s operating system takes more than 8 minutes to perform a simple local copy of the same OFF file.

Input						Times				
Mesh (# vertices)	N_v	max_E	N_s	$\#ISs$	# Output Vertices	Step 1–5	Step 6–8	Step 9	Total	Speedup
Bunny (35947)	5000	0.00025	1	10	4563	0.25	0.33	0.25	0.83	–
			2	5	4456		0.20		0.70	1.19
			3	4	4533		0.18		0.68	1.22
Dragon (437695)	10000	0.00027	1	75	8839	3.5	2.73	0.5	6.73	–
			2	38	8844		1.73		5.73	1.17
			3	26	8837		1.28		5.28	1.27
			4	19	8834		0.95		4.95	1.36
Lucy (14027872)	1000000	1.92355	1	25	9453	50.5	56.40	0.5	107.40	–
			3	9	9474		28.40		79.40	1.35
			5	6	9469		20.25		71.25	1.51
Terrain (67873499)	1000000	0.00006	1	117	12166	497	302	1	800	–
			10	13	11697		64.45		562.45	1.42
			25	6	11660		13.37		511.37	1.56
St. Matthew (186836670)	1000000	3.01716	1	285	119121	1225.5	805.65	2.5	2033.65	–
			10	29	119035		104.05		1332.05	1.53
			25	13	119308		47.65		1275.65	1.59
Atlas (245837027)	1000000	3.35350	1	395	234084	1441	1481.25	4.5	2926.75	–
			10	42	234081		157.05		1602.55	1.83
			25	18	234091		72.95		1518.45	1.93

Table 9.1: *Execution times (in seconds). The speedup increases with the input size. Column labels: N_v is the number of vertices per-server, max_E is the threshold error (one thousandth of the bounding box diagonal of the input in all these experiments) expressed in absolute values, N_s is the number of available servers, $\#ISs$ is the number of generated independent sets, "Step 1-5" refers to input partitioning, "Step 6-8" refers to simplification, "Step 9" refers to output merging.*

9.4.2 Quality

To test the quality of output meshes produced by our algorithm, we used Metro [CRS98] to measure the mean error between some small meshes and their simplifications. Table 9.2 shows the mean error of the Dragon model simplified with different parameter settings (i.e. with different threshold errors and different number of servers). Results show that the number of servers does not significantly affect the quality of the output.

max_E	Servers		
	1	2	3
0.00053	**0.060** (4864)	**0.060** (4891)	**0.059** (4923)
0.00027	**0.027** (8839)	**0.027** (8844)	**0.027** (8837)
0.00013	**0.012** (18422)	**0.012** (18421)	**0.012** (18502)

Table 9.2: *Approximation mean error of Dragon model. For each experiment, the first line reports the mean error (in bold), while the second line shows the number of output vertices. Mean errors are expressed as a percentage of the bounding box diagonal. Cells labelled with max_E indicate the threshold errors (0.002, 0.001, and 0.0005 of the bounding box diagonal) expressed in absolute values.*

Table 9.3 shows a comparison of our evaluations with results reported in [TJL07] and [BP02]. Since our simplification stops on a threshold error, we tuned this parameter to generate output meshes whose vertex number is comparable with both works. Also, we compared with the traditional incore method based on global priority queue provided by the QSlim software [GH97]. Results show that the quality of our simplified meshes is higher than previous works providing parallel simplification. Also, Table 9.3 shows that the quality of our output is close to the quality obtained by QSlim and, in same cases, even better.

#v	[TJL07]	[BP02]	**Ours**	#v	[GH97]
5000	**0.348**	**0.067**	**0.060**	4864	**0.051**
10000	**0.187**	**0.042**	**0.027**	8834	**0.030**
20000	**0.130**	**0.026**	**0.012**	18422	**0.015**

Table 9.3: *Comparison of mean error on the Dragon model, simplified on 4 servers with different threshold errors (0.002, 0.001, and 0.0005 of the bounding box diagonal). The first three columns show mean errors on outputs of previous works, while the last three show mean errors on our results and the comparison with the incore method provided by the QSlim software [GH97]. Mean errors (in bold) are expressed as a percentage of the bounding box diagonal. Columns labelled with #v report the number of vertices in the evaluated simplified mesh.*

For larger models, Metro cannot be used and quality can be assessed based on a visual inspection only. Figures 9.2, 9.3, 9.4, and 9.5 show that high quality is preserved in any case and that increasing the number of involved servers does not sensibly affect the output size and quality.

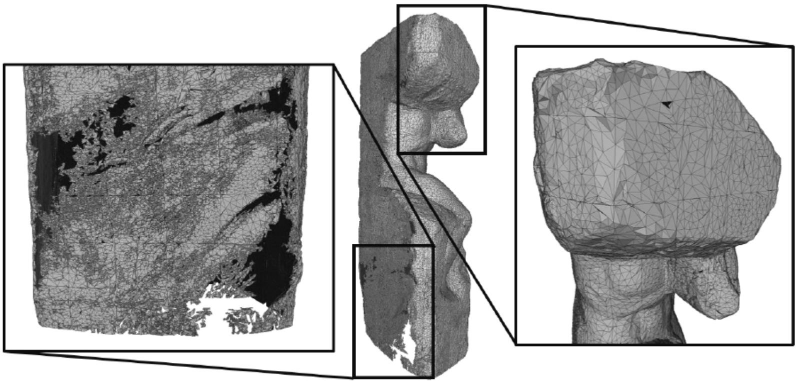


Figure 9.2: *Details of Atlas model simplified by exploiting 25 available servers (234091 vertices). The flat area is strongly decimated, while shape features are fairly maintained.*

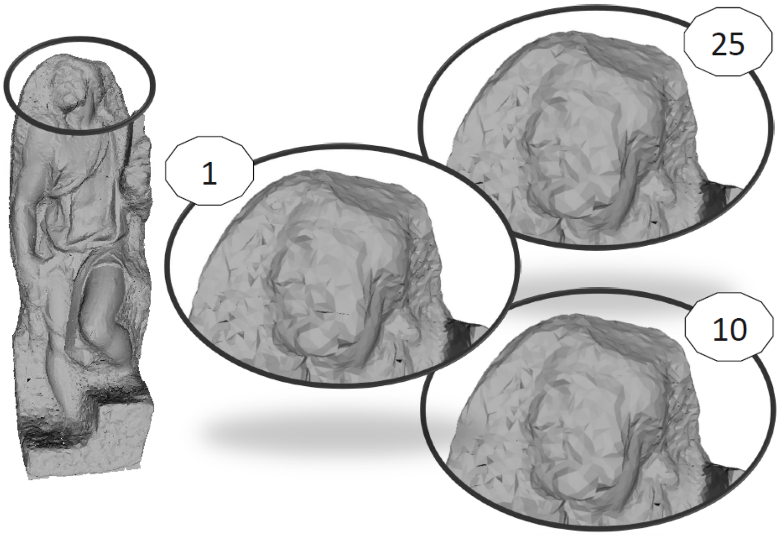


Figure 9.3: *Detail of St. Matthew model simplified by exploiting 1, 10 and 25 available services (original: ≈ 187M vertices, simplified: ≈ 119K vertices).*

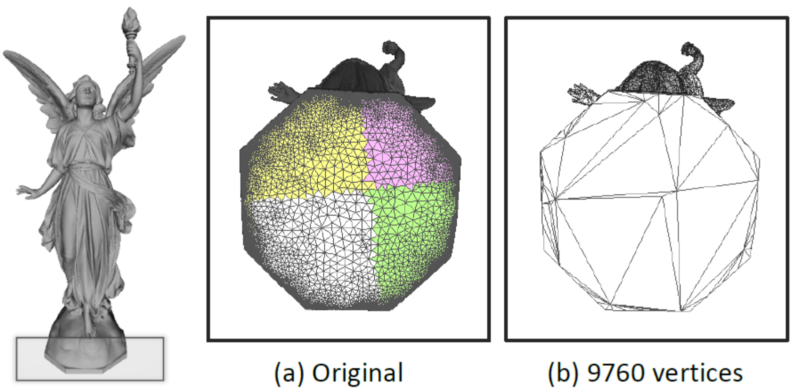


Figure 9.4: *Detail of Lucy model. The underside flat area is strongly decimated, while all the tiny features are preserved elsewhere. Methods based on vertex clustering cannot reach this level of adaptivity.*

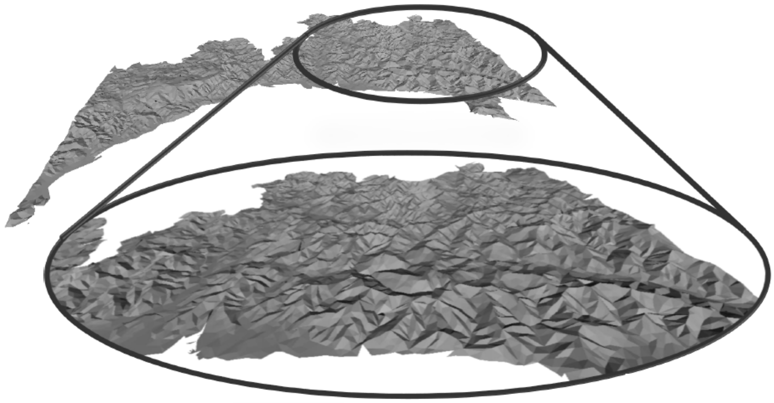


Figure 9.5: *Detail of Terrain model. Nearly height fields are naturally supported (original: ≈ 68M vertices, simplified: ≈ 11.5K vertices).*

9.4.3 Memory Space Evaluation

To evaluate the total amount of memory space required, both main (RAM) and secondary (disk space) memory must be considered. Our memory estimation is based on standard IEEE 32-bit integers and 32-bit floating point values.

During mesh partitioning, main memory is exploited to store the BSP structure, while geometric and connectivity information associated to each cell is stored on disk. Regarding the BSP, each cell includes an ID (16 bits), two vertices representing the bounding box ($6 \cdot 32$ bits), three references to file storing corresponding vertices, triangles and boundary description, and the list of IDs referring to neighbor cells (16 bits for each neighbor). We empirically found that each cell has 6 neighbors in average. Therefore, ≈ 50 bytes are sufficient to represent a single BSP cell and ≈ 25 MB are sufficient to store the entire BSP in our worst case.

During triangle classification a sub-portion of V_{file} is kept in main memory at any time. It is worth noticing that, in all our experiments, V_{file} is sufficiently small to be completely loaded into main memory by exploiting 16-bit representation for each cell ID. In our worst case where the partitioning of 250M vertices is made of 400 cells, V_{file} occupies ≈ 470 MB.

Secondary memory is exploited to store, besides the original input, three files for each BSP cell, representing corresponding vertices, triangles and boundary description B_i respectively. Since the sum of files associated with BSP structure represents the same geometric and connectivity information as the input data, roughly twice the input size is required. The boundary information associated with each cell (e.g. the list of cell IDs sharing each boundary vertex) can be efficiently encoded in a few bits for each boundary vertex and we empirically found that the total memory required to store this information can be neglected (less than 1% of input size in average).

Each server exploits its main and secondary memory to run its own simplification. Main memory is used to load geometric and connectivity information of corresponding submesh, to store and update quadric matrices and to store the priority queue of edges to be collapsed. This lead to an overall consumption of ≈ 630 Mb for 1M vertices. Secondary memory is exploited to save the input submesh, its simplified version and boundary information concerning input and output. As for the master, this latter information requires negligible space. Therefore, we can affirm that required memory space on servers is upper bounded by twice the submesh size, but in practice this limit is never touched.

At the end, the master receives simplified indexed submeshes and corresponding boundary information, that are saved on disk. Also, disk space must be sufficient to host the final output. During this step, required secondary space is almost twice the final output size. Main memory is used to store the Map structure and the $V(M_i')$ vector. Each entry in Map is a pair of indexes and occupies 64-bit (32 bits for each index). Since Map stores only boundary vertices, its final size is $O(\sqrt{N})$ for N vertices in the input mesh. Also, a single $V(M_i')$ is stored at each step and requires $n \cdot 32$ bits of space, where n is the number of vertices in the current M_i'.

To summarize, distributed simplification exploits secondary memory as mush as possible and sensibly reduces the required main memory. Since modern architectures provide huge hard disks and the cost of increasing their performance is much lower than providing larger main memories, our solution is suitable to be run on any commodity PC.

9.4.4 Summary of the features

Table 9.4 summarizes our achievements in terms of non-quantitative results and provides a comparison with previously publish work. This analysis shows that our algorithm is a winning proposal for efficiently processing very large indexed meshes by exploiting distributed environments, both in the case of full 3D models and in the case of height fields such as terrains (Figure 9.5).

	[LS01]	[TJL07]	[BP02]	Ours
Out–Of–Core Input	✓	✗	✓	✓
Out–of–Core Output	✓	✗	✗	✓
Adaptivity	✗	✓	✓	✓
Distributable	✓	✓	✗	✓
Indexed Mesh support	✗	✓	✓	✓

Table 9.4: *Feature-based comparison with the state of the art.*

Part IV

Conclusions and Discussion

10

Conclusions

'Big data is a nebulous term with many different definitions. The key thing to remember is that in this day and age, big data is distributed data. This means the data is so massive it cannot be stored or processed by a single node."

– R. Sobers, [Sob12]

We provided an innovative workflow management system to remotely perform complex geometry processing on large triangle meshes. Nothing more than a standard Web browser needs to be installed on the client machine hosting the input mesh, while a distributed network of servers provides both the software and hardware necessary to undertake the computations. The user interface allows to build complex pipelines by stacking geometric algorithms and by controlling their execution through conditions and cycles, while the overall execution is managed by a central engine that both invokes appropriate Web services and handles the data transmission.

10.1 Technological and Scientific Innovation

Our system represents a practical example of how it is possible to process arbitrarily large geometric data by exploiting existing distributed architectures. Differently from previously published works that allow remote geometry processing, our platform also enables the possibility to combine available algorithms to build complex pipelines and to customize each operation by setting possible parameters. Geometric issues such as the analysis and evaluation of mesh qualities are enabled thanks to specific Web services.

Currently, the system provides some “in-house” geometry processing algorithms and some workflows representing traditional mesh repairing pipelines, but the architecture is open and fully extensible. The list of available algorithms can be enriched by any developer without the need to distribute the source code or executable files. Furthermore, new complex pipelines can be easily defined through the user interface and stored on the system. Such a feature allows scientists in any field to perform complex elaborations with no longer need to be skilled programmers or experts in geometric modelling.

Besides the technological contribution, scientific innovative solutions are provided that enable the processing of large input meshes. First, the optimized mesh transfer protocol efficiently manages the data transmission across scattered servers and avoids possible bottlenecks. Second, efficiency and effectiveness are guaranteed thanks to the novel divide and conquer approach that the engine exploits to partition large meshes into smaller pieces, each delivered to a dedicated server for parallel processing. Thanks to such approaches, we demonstrated that the computing power of a network of PCs can be exploited to significantly speed up the elaboration of large triangle meshes. Sure enough, we proved that the overhead due to data transmission is negligible, as it is much lower than the gain provided by parallel processing.

Furthermore, the proposed distributed approach supports collaborative environments in efficiently sharing output results. Instead of storing 3D models explicitly, an online repository endowed with our system allows to generate meshes on demand without struggling with software installations, compatibility issues, or hardware requirements. Moreover, geometric workflows are available for any researcher who wants to reproduce exactly the same result. Such a repository requires significantly less storage resources, but at the cost of a more intensive computation that can be easily distributed among available servers.

Consistently with the Lohr's statement [Loh13] provided as an introduction to this dissertation, the threefold contribution of our work can be summarized as follows:

- **It is a bundle of technologies**: the integration of geometric algorithms, Web browsers, Web service technologies, and workflow-based frameworks allows to efficiently process large geometric data with a significant flexibility, scalability and speed;

- **It is a revolution of measurement**: as big geometric data can be continuously measured and updated, explicit results of elaborations on these datasets are subject to obsolescence. By replacing such explicit results with dynamic and efficient processing workflows, we can guarantee that the elaborations are always up-to-date and consider the new measurements;
- **It is a point of view of how decisions should be made in the future**: distributed environments are efficient, scalable, and fault-tolerant. Hence they can be exploited to perform elaboration of large geometric datasets for critical decision-making processes such as, e.g. disaster management.

10.2 Limitations

Although workflows provide a popular means for preserving scientific methods by explicitly encoding their process, they may be subject to a decay in their ability to be re-executed or reproduce the same results over time, largely due to the volatility of the resources required for executions [ZGPB+12]. Workflows generated through our system suffer of such a decay, that is experiments can be reproduced only as long as the involved Web services are available and are not modified by their providers. To reduce the possibility of workflow decay, a certain level of redundancy would be required, for example by uploading the same Web service on different machines, but in this case contributors should either distribute the code to someone else or directly take care of such a duplication.

Our system provides the possibility to easily process a variety of datasets using a fixed sequence of operations. Completely automatic pipelines can be executed by simply uploading the user-selected inputs. Currently, our system does not support semi-automatic pipelines, that is with user interaction. Such a functionality would require the engine to interrupt the execution waiting for the user intervention.

10.3 Work in progress

Currently, the system supports the elaboration of triangle meshes only. Although such a shape representation is a de facto standard in computer graphics and geometry processing, acquired 3D data coming from diverse industrial and research areas (eg. geology, archaeology, medicine, ...) are often represented as points sets. Motivated by this, we consider worthwhile to improve the capabilities of our system in order to enable the elaboration of large-size point clouds. We are now investigating the possibility to exploit a distributed environment to perform surface reconstruction from large point sets.

As a reference scenario, we consider a “server” (i.e. our engine) storing a very large point cloud representing a 3D object, and a user who wishes to analyze the 3D model on a standard “client” PC. The user must be able to select a Region Of Interest (ROI) on the model, and the client must be able to reconstruct a surface on the fly whose overall resolution adapts to both the client hardware capabilities and the specified ROI. In other words, the area selected by the user should be reconstructed at very high level of detail, while all the others should be reconstructed at lower resolution.

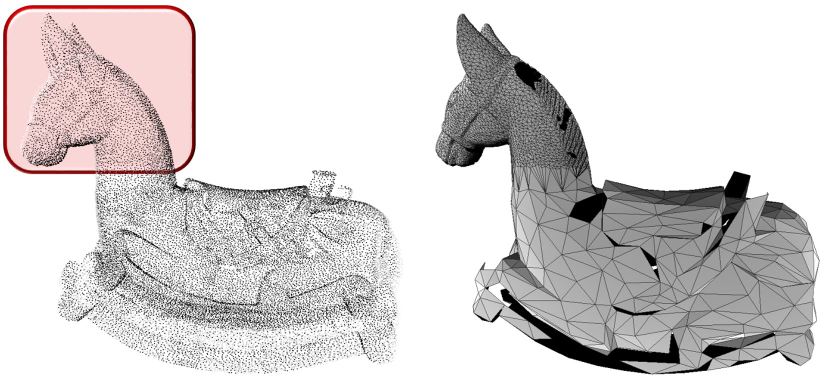


Figure 10.1: *Example of adaptive-resolution surface reconstruction (on the right) given an input point cloud (on the left). The selected area (in red) is reconstructed at very high level of detail, while all the others could be reconstructed at lower resolution.*

Our idea is to make the server able to subdivide a large point cloud. At an abstract level, our hierarchical data structure could be designed as a tree, where the root represents the whole input point cloud and each leaf represents a single point. Each middle node in the structure is a subset of parent points that satisfies a specific requirement (i.e. it can be easily approximated by a planar polygon in the 3D space). In practice, the original point cloud should be subdivided into files, each of them representing a subset of points that is “coherent with the shape surface”. Moreover, an extra file should be provided that describes the data structure in order to allow to access specific areas in an efficient way. Such a data structure could be used by the client to reconstruct the surface at different resolutions, according to the area of interest selected by the user.

10.4 Future Developments

Future works are also addressed to improve the system and provide new functionalities. As an example, the usability of the system should be enhanced by providing the possibility to graphically show and edit existing workflows. The user should be allowed to visualize a workflow through a flow representation and analyze each single operation. Editing operations should be allowed on such a representation in order to generate new similar workflows.

Furthermore, the user is required to install specific standalone applications on its local machine to visualize the received output. As already underlined, such an operation may be time-consuming, due to hardware and software compatibility issues. An innovative 3D visualization tool should allow to visualize the output meshes through the Web browser and directly interact with it [PCV$^+$13] [Mes15].

Finally, the platform currently provides only a small set of geometry processing operations. These operations have been sufficient to demonstrate the effectiveness of the system, but many more operations would be required to actually exploit our solution for real research purposes. Thus, future efforts should be addressed to provide a more complete set of Web services to improve the capabilities of the system and allow to run traditional mesh editing and repairing pipelines. Furthermore, some efforts should be done to support the processing of 3D models having additional properties (eg. texture coordinates, normals, ...), but such an upgrade would impact both on the transfer protocol and on the partitioning/merging approach.

10.5 Future Research

The distributed approach presented in this dissertation provides several directions for future research, both in terms of improvement of the platform capabilities and enrichment of available geometry processing operations.

Editing Operations Each algorithm provided as a Web service must be enriched with proper code to stream each editing operation and to return both the output mesh and the applied corrections. To further simplify the work of potential contributors, the engine should be able to automatically compute the list of editing operations by simply comparing the input and output of each Web service. A possible solution should be inspired on existing algorithms [DP13], but requiring less computational complexity so that no degradation is introduced in the system performance.

Input Formats Our system can load a single indexed mesh and produces a single file. Part of future plans include the study of methods to efficiently represent big meshes through several files that can be hosted on different machines. Moreover, some effort should be done to support other standard formats used in specific application areas.

Workflow Formalization The formalization of geometry processing workflows is at a very preliminary step and mainly consists of a XML file describing the workflow and an instance of the ontology that simply links to such a file. A deeper formalization process represents an idea for a further development of the Workflow Ontology. Such a formalization could be exploited to improve the browsing mechanism and allow the system to resolve queries that involve both geometric and semantic information.

Technology Exploitation Our workflow engine currently stores input and output data on its own disk and exploits HTTP protocol to distribute 3D models through the net. The possibility to extend the architecture by exploiting a distributed file system (eg. HDFS [SKRC10]) or an online database should be taken into account to optimize the data storage and allow Web services to directly access the data. Although such technologies already provide support for partitioning large data sets of small unstructured records (eg. Map-Reduce [DG08]), they are not suitable for managing 3D models that have a specific geometric structure.

10.6 Published As

Parts of this dissertation have been published before. The list of accepted papers is provided below.

Journal Papers

- D. Cabiddu and M. Attene. Large mesh simplification for distributed environments. *Computers & Graphics - Special Issue: Shape Modeling International*, 51:81 – 89, 2015
- M. Attene, D. Cabiddu, S. Gagliardo, F. Giannini, and M. Monti. A web repository to describe and execute shape oriented workflows. *Computer Aided Design and Applications (CAD)*, To be appeared

Conference Proceedings

- D. Cabiddu and M. Attene. Distributed triangle mesh processing. In *Proceedings of the 22nd International Conference in Central Europe* *on Computer Graphics, Visualization and Computer Vision (WSCG)*, Plzen, Czech Republic, 2014

- M. Attene, D. Cabiddu, S. Gagliardo, F. Giannini, and M. Monti. A web-based system to describe and execute shape processing workflows. In *Proceedings of Computer Aided Design and Applications (CAD)*, London, UK, July 2015
- D. Cabiddu and M. Attene. Distributed processing of large polygon meshes. In *Proceedings of Smart Tools and Apps in Computer Graphics (STAG)*, Verona, Italy, October 2015

Extended Abstracts

- D. Cabiddu and M. Attene. A web-based distributed system to process large geometric models. In *IQmulus Workshop for Big Data Processing*, Cardiff, Wales, July 2014

Appendix A

Available Web Services

Currently, our system provides a set of "in-house" Web services, able to perform atomic tasks and analysis operations on triangle meshes. All the provided Web services are designed to receive the address of the input mesh and to download it locally. Also, possible input parameters may be required in order to allow the service to perform its task. The list of available Web services is listed in the following sections.

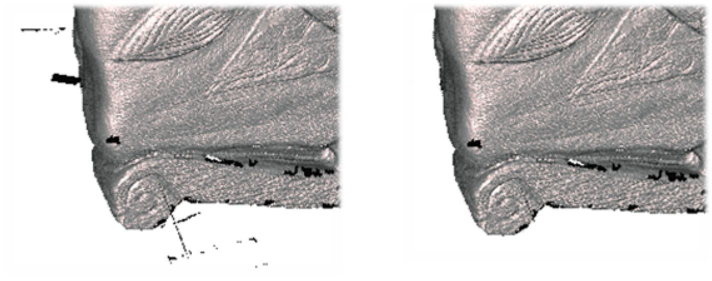
A.1 Atomic Tasks

Each atomic task aims to perform a specific geometry processing operation. The result of such an operation is stored locally and made available through the Internet through a public address. Available Web services are listed in Table A.1.

Web service	**Parameters**
Smallest Component Removal	–
Degenerate Triangles Removal	–
Noise Addition	Error Threshold
Laplacian Smoothing	# iterations
Hole Filling	–
Mesh Simplification	Error Threshold

Table A.1: *Available Web service that perform a geometry processing operation. Cells whose value is "–" indicate that no input parameter is required.*

Smallest Components Removal It keeps the largest connected component of the mesh while removing all the others [AF06]. No input parameters required.

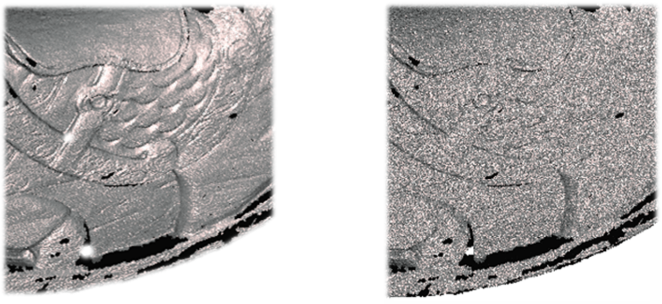


(a) *Input* **(b)** *Output*

Figure A.1: *An example of smallest component removal.*

Degenerate Triangles Removal It applies local mesh modifications to remove zero-area triangles [BK01]. No input parameters required.

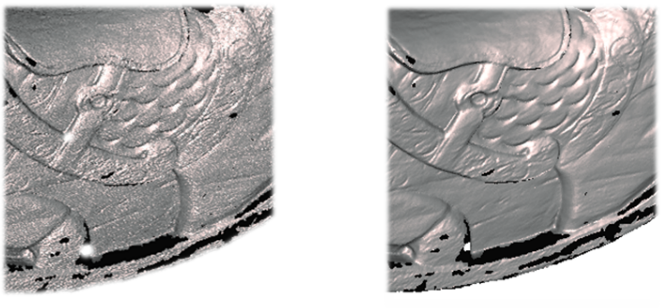
Noise Addition It displaces each vertex along its normal direction by a random distance. It requires the maximum distance to be provided as an input parameter. The user is required to specify such a distance as a percentage of the mesh bounding box.



(a) *Input* **(b)** *Output*

Figure A.2: *An example of noise addition.*

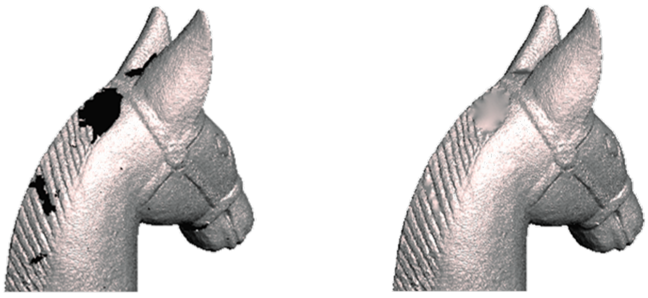
Laplacian Smoothing It moves each vertex to the center of mass of its neighbors. By default, the Web service applies the algorithm once. A different configuration can be set by providing the desired number of iterations as an input parameter.



(a) *Input* **(b)** *Output*

Figure A.3: *An example of laplacian smoothing (2 iterations applied).*

Hole Filling It patches all the boundary loops [Lie03]. Such a Web service assumes that the input mesh represents a full 3D object. No input parameters required.
A modified version of such a Web service is also provided to support height fields (eg. terrain reconstructions) as inputs. It patches all the boundary loop, but the largest one is preserved.



(a) *Input* **(b)** *Output*

Figure A.4: *An example of hole filling.*

Mesh Simplification It simplifies the input by reducing the number of its vertices [GH97]. It requires the maximum desired error to be provided as an input parameter. The user is required to specify such an error as a percentage of the mesh bounding box (see Chapter 9).

A.2 Analysis Tasks

Conditional tasks and loops are supported by specific Web services that check mesh qualities. Differently from atomic tasks, no output mesh is generated, but a boolean value is returned.

Each Web service receives the condition to be checked as an input. Such a condition includes a comparison operator and a reference value (see Section 5.1.4). Based on the set of the values that the mesh quality Q can assume, both the comparison operator and the reference value in the input condition are required to be set accurately. Specifically, the reference value must be of the same type of Q, while the list of supported comparison operators is provided in Table A.2.

Operator	**Symbol**	**XML Syntax**
Equal To	$=$	*EQ*
Not Equal To	$\neq$	*NEQ*
Greater Than	$>$	*GT*
Less Than	$<$	*LT*
Greater Than or Equal To	$\geq$	*GEQ*
Less Than or Equal To	$\leq$	*LEQ*

Table A.2: *The list of supported comparison operators.*

The list of currently supported mesh qualities is provided below, subdivided according to the type of Q.

- **Natural Numbers** ($Q \in \mathbb{N}$)

  All the comparison operators are supported. By default, the operator is set to *EQ* and the reference value is set to 0.

  - *Number of vertices*
  - *Number of triangles*

  - *Number of edges*
  - *Number of boundary loops*
  - *Number of disconnected components*

- **Real Numbers** ($Q \in \mathbb{R}$)

  All the comparison operators are supported. By default, the operator is set to EQ and the reference value is set to 0.

  - *Minimum triangle angle*
  - *Maximum triangle angle*
  - *Average edge length*
  - *Total surface area*
  - *Enclosed volume*
  - *Average normal instability*: the amount of noise in a mesh [Att13]

- **Boolean Values** ($Q \in \{true, false\}$)

  Only the EQ and NEQ operators are supported. By default, the operator is set to EQ and the reference value is set to $true$.

  - *Manifoldness*
  - *Orientation*
  - *Orientability*
  - *Watertightness*

Appendix B

Editing Operations

Our system requires each Web service to keep track of all the mesh modifications in terms of added/removed/modified simplexes. We implemented a C++ library providing functions to stream all the operations listed in Table B.1. For each of them, the third table column shows how the operation is encoded. For the purpose of reducing the correction file size as much as possible, we define some complex operations that group atomic changes (eg. split triangle) and we omit operation parameters when they can be easily recomputed during correction phase (eg. the split point if it is the center of the triangle).

Simplex	**Operation**	**Encoding**
Triangles	Add	T A v_1 v_2 v_3
	Remove	T *id* R
	Split (center)	T *id* SPC
	Split (generic point)	T *id* SP x y z
Vertices	Add	V A x y z
	Move	V *id* M x y z
	Move All	V MA x_1 y_1 z_1 x_2 y_2 z_2 ...
Edges	Add	E A v_1 v_2
	Swap	E *id* SW
	Collapse	E *id* C
	Split (midpoint)	E *id* SP
	Split (generic point)	E *id* SP x y z
All	Clear All	CLEAR

Table B.1: *Supported editing operations.* Italic *labels in the encodings indicate either simplex identifiers (i.e. indexes) or vertex coordinates.*

Bibliography

[ACG$^+$15] M. Attene, D. Cabiddu, S. Gagliardo, F. Giannini, and M. Monti. A web-based system to describe and execute shape processing workflows. In *Proceedings of Computer Aided Design and Applications (CAD)*, London, UK, July 2015.

[ACG$^+$ed] M. Attene, D. Cabiddu, S. Gagliardo, F. Giannini, and M. Monti. A web repository to describe and execute shape oriented workflows. *Computer Aided Design and Applications (CAD)*, To be appeared.

[ACK13] M. Attene, M. Campen, and L. Kobbelt. Polygon mesh repairing: An application perspective. *ACM Comput. Surv.*, 45(2):15:1–15:33, March 2013.

[AF06] M. Attene and B. Falcidieno. Remesh: An interactive environment to edit and repair triangle meshes. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)*, page 41. IEEE Computer Society, 2006.

[AGL06] M. Ahn, I. Guskov, and S. Lee. Out-of-core remeshing of large polygonal meshes. *IEEE Trans. Vis. Comput. Graph.*, 12(5):1221–1228, 2006.

[Aim04] AIM@SHAPE. EU FP6 IST project n. 506766. http://www.aimatshape.net, 2004.

[Att10] M. Attene. A lightweight approach to repairing digitized polygon meshes. *The Visual Computer*, 26(11):1393–1406, 2010.

[Att13] M. Attene. Surface mesh qualities. In *GRAPP/IVAPP*, pages 79–85, 2013.

[Att14] M. Attene. Direct repair of self-intersecting meshes. *Graphical Models*, 76(6):658–668, 2014.

[BK01] M. Botsch and L. Kobbelt. A robust procedure to eliminate degenerate faces from triangle meshes. In *Vision, Modeling and Visualization*, pages 283–290, 2001.

[BP02] D. Brodsky and J. B. Pedersen. Parallel model simplification of very large polygonal meshes. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 3*, PDPTA '02, pages 1207–1215. CSREA Press, 2002.

[BPK$^+$07] M. Botsch, M. Pauly, L. Kobbelt, P. Alliez, B. Lévy, S. Bischoff, and C. Rössl. Geometric modeling based on polygonal meshes video files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[CA14a] D. Cabiddu and M. Attene. Distributed triangle mesh processing. In *Proceedings of the 22nd International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, Plzen, Czech Republic, 2014.

[CA14b] D. Cabiddu and M. Attene. A web-based distributed system to process large geometric models. In *IQmulus Workshop for Big Data Processing*, Cardiff, Wales, July 2014.

[CA15a] D. Cabiddu and M. Attene. Large mesh simplification for distributed environments. *Computers & Graphics - Special Issue: Shape Modeling International*, 51:81 – 89, 2015.

[CA15b] D. Cabiddu and M. Attene. Distributed processing of large polygon meshes. In *Proceedings of Smart Tools and Apps in Computer Graphics (STAG)*, Verona, Italy, October 2015.

[cad01] 3D CAD browser. http://www.3dcadbrowser.com, 2001.

[Cam] M. Campen. WebBSP 0.3 beta. http://www.graphics.rwth-aachen.de/webbsp.

[CCR08] P. Cignoni, M. Corsini, and G. Ranzuglia. Meshlab: an open-source 3d mesh processing system. *ERCIM News*, (73):45–46, April 2008.

[CMRS03] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, oct 2003.

[CRS98] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Comput. Graph. Forum*, 17(2):167–174, 1998.

[CSS98] Y. J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Visualization'98*, pages 167–174, 1998.

[DG08] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DLR00] F. Dehne, C. Langis, and G. Roth. Mesh simplification in parallel. In *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing, Hong Kong*, ICA3PP 2000, pages 281–290, 2000.

[DP13] J. D. Denning and F. Pellacini. Meshgit: Diffing and merging meshes for polygonal modeling. *ACM Trans. Graph.*, 32(4):35:1–35:10, July 2013.

[dsw12] DSW v5.0 - visualization virtual services. http://visionair.ge.imati.cnr.it, 2012.

[fas14] Voxel Game Engine Development – quadric mesh simplification with source code. http://voxels.blogspot.jp/2014/05/quadric-mesh-simplification-with-source.html, 2014.

[FCF$^+$08] G. Foucault, J. C. Cuillière, V. François, J. C. Léon, and R. Maranzana. Adaptation of CAD model topology for finite element analysis. *Computer-Aided Design*, 40(2):176–196, 2008.

[Fea07] T. G. Farr and et al. The shuttle radar topography mission. *Rev. Geophys.*, 45, 2007.

[FQH05] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl: an abstract grid workflow language. In *CCGRID*, pages 676–685. IEEE Computer Society, 2005.

[FS00] M. Franc and V. Skala. Parallel triangular mesh reduction. In *Proceedings of the Conference on Scientific Computing (ALGORITMY 2000)*, ALGORITMY 2000, pages 357–367, 2000.

[GBA$^+$10] C. A. Goble, J. Bhagat, S. Aleksejevs, D. Cruickshank, D. T. Michaelides, D. R. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li, and D. D. Roure. myexperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic Acids Research*, 38(Web-Server-Issue):677–682, 2010.

[GBCL04] A. Gangemi, S. Borgo, C. Catenacci, and J. Lehmann. Task taxonomies for knowledge content d07. Technical report, Metokis Project, 2004.

[GH97] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[GNTT10] J. Goecks, A. Nekrutenko, J. Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, August 2010.

[Gol09] B. Golden. The skinny straw: Cloud computing's bottleneck and how to address it. http://www.cio.com/article/2425754/virtualization/the-skinny-straw–cloud-computing-s-bottleneck-and-how-to-address-it.html, 2009.

[Gov07] J. Governor. Why applications are like fish and data is like wine. http://redmonk.com/jgovernor/2007/04/05/why-applications-are-like-fish-and-data-is-like-wine/, 2007.

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[HG97] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. Technical report, CS Department, Carnegie Mellon, 1997.

[HG99] Paul S. Heckbert and Michael Garland. Optimal triangulation and quadric-based surface simplification. *Journal of Computational Geometry: Theory and Applications*, 14(1–3):49–65, 1999.

[Hig15] D. Higginbotham. Clojure for the brave and true. learn the ultimate language and become a better programmer. http://www.braveclojure.com/concurrency/, 2015.

[HKK14] M. Hutter, M. Knuth, and A. Kuijper. Mesh partitioning for parallel garment simulation. In *Proceedings of the 22nd International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2014 (WSCG 2014)*, 2014.

[HLK01] J. Ho, Kuang-Chih Lee, and D. Kriegman. Compressing large polygonal models. In *Visualization, 2001. VIS '01. Proceedings*, pages 357–573, Oct 2001.

[Hol95] D. Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, January 1995.

[HSK04] D. Hollingsworth, Fujitsu Services, and United Kingdom. The workflow reference model: 10 years on. In *Fujitsu Services, UK; Technical Committee Chair of WfMC*, pages 295–312, 2004.

[IG03] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 935–942, New York, NY, USA, 2003. ACM.

[IL05] M. Isenburg and P. Lindstrom. Streaming meshes. In *Procs. of Visualization'05*, pages 231–238, 2005.

[ILGS03] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *Visualization, 2003. VIS 2003. IEEE*, pages 465–472, October 2003.

[iqm13] Iqmulus: A High-volume Fusion and Analysis Platform for Geospatial Point Clouds, Coverages and Volumetric Data Sets. http://www.iqmulus.eu, 2013.

[Jac09] A. Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.

[jax03] JAXB Reference Implementation. https://jaxb.java.net/, 2003.

[JKIR06] S. Jayanti, Y. Kalyanaraman, N. Iyer, and K. Ramani. Developing an engineering shape benchmark for {CAD} models. *Computer-Aided Design*, 38(9):939–953, 2006.

[Ju04] T. Ju. Robust repair of polygonal models. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 23(3):888–895, 2004.

[LAB$^+$06] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, August 2006.

[Lie03] P. Liepa. Filling holes in meshes. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, pages 200–205, Aachen, Germany, 2003. Eurographics Association.

[Lin00] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 259–262, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Loh13] S. Lohr. Sizing up big data, broadening beyond the internet. http://bits.blogs.nytimes.com/2013/06/19/sizing- up-big-data-broadening-beyond-the-internet/, 2013.

[LPC+00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[LS01] P. Lindstrom and C. T. Silva. A memory insensitive technique for large model simplification. In Thomas Ertl, Kenneth I. Joy, and Amitabh Varshney, editors, *IEEE Visualization*. IEEE Computer Society, 2001.

[LSVT15] M. Livesu, A. Sheffer, N. Vining, and M. Tarini. Practical hexmesh optimization via edge-cone rectification - web application. http://polyhexweb.cs.ubc.ca:8090/untangler, 2015.

[Mes15] Meshlabjs. http://www.meshlabjs.net/, 2015.

[mic09] The Digital Michelangelo Project. http://graphics.stanford.edu/projects/mich/, 2009.

[MK12] J. Möbius and L. Kobbelt. Openflipper: An open source geometry processing and rendering framework. In J.-D. Boissonnat, P. Chenin, A. Cohen, C. Gout, T. Lyche, M.-L. Mazure, and L. Schumaker, editors, *Curves and Surfaces*, volume 6920 of *Lecture Notes in Computer Science*, pages 488–500. Springer Berlin Heidelberg, 2012.

[NCH+11] P. Nowakowski, E. Ciepiela, D. Harezlak, J. Kocot, M. Kasztelnik, T. Bartynski, J. Meizner, G. Dyk, and M. Malawski. The collage authoring environment. In Mitsuhisa Sato, Satoshi Matsuoka, Peter M. A. Sloot, G. Dick van Albada, and Jack Dongarra, editors, *ICCS*, volume 4 of *Procedia Computer Science*, pages 608–617. Elsevier, 2011.

[PCV$^+$13] P. Parascandolo, L. Cesario, L. Vosilla, M. Pitikakis, and G. Viano. Smart brush: a real time segmentation tool for 3d medical images. In *8th International Symposium on Image and Signal Processing and Analysis*, 2013.

[Pit10] M. Pitikakis. *A Semantic Based Approach For Knowledge Management, Discovery and Service Composition Applied To 3D Scientif Objects.* PhD thesis, University of Thessaly, School of Engineering, Department of Computer and Communication Engineering, 2010.

[PMP11] K. Plankensteiner, J. Montagnat, and R. Prodan. Iwir: a language enabling portability across grid workflow systems. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pages 97–106, New York, NY, USA, 2011. ACM.

[RB93] J. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In Bianca Falcidieno and TosiyasuL. Kunii, editors, *Modeling in Computer Graphics*, IFIP Series on Computer Graphics, pages 455–465. Springer Berlin Heidelberg, 1993.

[RCW$^+$06] A. Raposo, E. T. L. Corseuil, G. N. Wagner, I. H. F. dos Santos, and M. Gattass. Towards the use of cad models in vr applications. In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications*, VRCIA '06, pages 67–74, New York, NY, USA, 2006. ACM.

[Sai02] A. Said. Introduction to arithmetic coding - theory and practice. In *Lossless Compression Handbook*, pages 101–152. Academic Press, 2002.

[SBF98] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25(1):161–197, 1998.

[SCC$^+$02] C. Silva, Y. Chiang, W. Correa, J. El-sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *In Visualization'02 Course Notes*, 2002.

[SG01] E. Shaffer and M. Garland. Efficient adaptive simplification of massive meshes. In *Proceedings of the Conference on Visualization '01*, VIS '01, pages 127–134, Washington, DC, USA, 2001. IEEE Computer Society.

[SKRC10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[SMKF04] P. Shilane, P. Min, M. Kazhdan, and T. Funkhouser. The princeton shape benchmark. In *SMI '04: Proceedings of the Shape Modeling International 2004 (SMI'04)*, pages 167–178, Washington, DC, USA, 2004. IEEE Computer Society.

[SN13] S. M. Shontz and D. M. Nistor. CPU-GPU algorithms for triangular surface mesh simplification. In Springer, editor, *Proc. 21st International Meshing Roundtable*, pages 475–492, Berlin, Germany, 2013.

[Sob12] R. Sobers. 5 things you should know about big data. http://blog.varonis.com/5-things-you-should-know-about-big-data/, 2012.

[sta96] The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3dscanrep, 1996.

[TG98] C. Touma and C. Gotsman. Triangle mesh compression. In Wayne A. Davis, Kellogg S. Booth, and Alain Fournier, editors, *Graphics Interface*, pages 26–34. Canadian Human-Computer Communications Society, 1998.

[TJL07] X. Tang, S. Jia, and B. Li. Simplification algorithm for large polygonal model in distributed environment. In De-Shuang Huang, Laurent Heutte, and Marco Loog, editors, *Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues*, volume 4681 of *Lecture Notes in Computer Science*, pages 960–969. Springer Berlin Heidelberg, 2007.

[TPB08] B. Thomaszewski, S. Pabst, and W. Blochinger. Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics*, 32(1):25–40, 2008.

[TS06] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[TS07] A. Tiwari and A. K. T. Sekhar. Workflow based framework for life science informatics. *Computational Biology and Chemistry*, (5-6):305–319, 2007.

[vdAtH05] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, June 2005.

[Vis12] VISIONAIR. vision advanced infrastructure for research. EU FP7 project n. 262044. http://www.infra-visionair.eu, 2012.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[WHF+13] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic acids research*, 41(Web Server issue):W557–W561, July 2013.

[Wil02] R. Wilson. *Four colors Suffice.* Penguin Books, London, 2002.

[WK03] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. In *Procs. of Graphics Interface 2003*, pages 185–192, 2003.

[ZGPB+12] J. Zhao, J. M. GómezPérez, K. Belhajjame, G. Klyne, E. GarcíaCuesta, A. Garrido, K. M. Hettne, M. Roos, D. D. Roure, and C. A. Goble. Why workflows break understanding and combating decay in taverna workflows. In *eScience*, pages 1–9. IEEE Computer Society, 2012.

[Zor79] A. Zorat. *A Divide–and–Conquer Computer.* PhD thesis, University of Southern California, The Graduate School of Computer Science, 1979.