STAG: Smart Tools & Apps for Graphics (2015)
Silvia Biasotti and Marco Tarini and Andrea Giachetti (Editors)

Distributed Processing of Large Polygon Meshes

D. Cabiddu[1] and M. Attene[1]

[1]CNR-IMATI Genova, Italy

Abstract

A system is described to remotely perform complex geometry processing on arbitrarily large 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 hosting the input mesh. The user interface allows to build complex pipelines by stacking geometric algorithms and by controlling their execution through conditions and cycles. Besides the technological contribution, an innovative mesh transfer protocol is described to treat large datasets whose transmission across scattered servers may represent a bottleneck. Also, efficiency and effectiveness are guaranteed thanks to a 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. Based on this paradigm, a distributed simplification algorithm has been implemented which proves that the overhead due to data transmission is negligible, as it is much lower than the gain in speed provided by parallel processing.

1. Introduction

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, archeology, medicine and entertainment. Since digital 3D models generated in these application areas (eg. original raw scans and their elaborations) are easily made of millions of geometric elements, processing and analysing them are non trivial tasks.

Geometry processing is now a mature research area where new algorithms and methods are continuously being developed on the top of the state-of-the-art previous works and allow to analyse and process 3D models. 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.

Out-of-core approaches can be exploited 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 are exploited since they provide the possibility to process different subparts of the input simultaneously, but the available memory is shared among the current processes and imposes a sequentialization of I/O operations in any case.

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 approach already provides the possibility to perform remote computantion in many life science areas, but it is scarcely considered in geometry processing literature where the transmission of large 3D models can easily slow the process down too much.

In this paper we prove that distributed environments can be actually exploited for geometry processing too, while making efficient sharing of 3D geometric data possible and giving the possibility to remotely run algorithms from different machines with no need of any local software installa-

© The Eurographics Association 2015.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

DOI: 10.2312/stag.20151301

tion or any specific hardware or software requirement. Our system allows to define geometry processing pipelines as a composition of several existing algorithms and exploits Web services to remotely run them.

Since the possible slow data transfer among available servers and their possible limited main memory would represent a limitation, specific approaches are necessary to avoid bottlenecks and crashes during the remote processing of large-size inputs. Specifically, we designed an optimized mesh transfer protocol to reduce the amount of data to be transmitted among distributed servers, and an innovative partitioning method for large input meshes that enables distributed parallel processing. 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.

2. Related Works

2.1. Remote Geometry Processing

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. In computer graphics and geometry processing, polygon meshes are the dominant representations for 3D objects, and diverse mesh processing software tools exist. Among them, MeshLab [CCR08] and OpenFlipper [MK12] allow to interactively edit a mesh, save the sequential list of executed operations and locally re-execute the workflow from their user interfaces. Pipelines can be shared in order to be rerun on different machines where the stand-alone applications need to be installed.

To get rid of any specific software, hardware, and operating system, Campen and colleagues published an online service called WebBSP [Cam] which is able to remotely run a few specific geometric operations. The system is accessible from a standard web browser and the user is required to upload an input mesh; then, a single geometric algorithm must be selected from a set of available operations. The algorithm is actually run on the server and a link to download its output is sent to the user by email. 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.

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. Though in Pitikakis's work Web services are stacked into hardcoded sequences, users are not allowed to dynamically construct workflows, and geometric issues such as the evaluation of mesh qualities (necessary to support conditional tasks and loops) and the transmission of large models are not dealt with.

2.2. Out-of-core and Parallel Mesh Processing

Traditional incore algorithms are not suitable for managing large input datasets. In these cases, the exploitation of out-of-core techniques is mandatory and parallel approaches may be used to reduce the overall elaboration time. The state-of-the art includes several different solutions, and most of them focus on mesh simplification methods.

In parallel algorithms [BP02] [DLR00] [FS00] [TJL07] a "master" processor partitions the input mesh and distributes the portions across different "slave" processors that perform the partial simplifications simultaneously. When all the portions are ready, the master merges the results together. The many slave processors available in modern GPU-based architectures are exploited in [SN13], while multi-core CPUs are exploited in [TPB08]. In these methods the main goal is to speedup the process and, with the exception of [BP02], the typical approach to partition the input exploits in-core algorithms.

Besides [BP02], other effective out-of-core partitioning techniques are described in [Lin00] [LS01]. These methods typically require their input to come as a triangle soup. When the input is represented using an indexed format, it must be dereferenced using out-of-core techniques [CSS98], but this additional step is time-consuming and requires significant storage resources. As an exception, 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. Instead of partitioning the input into fixed portions, processing sequences are used in [ILGS03]. This approach is very elegant and does not need to deal with boundary coherence. Even in this case, however, conversion to appropriate processing sequences is a non-trivial process that requires a significant time [IG03].

In [Lin00] the vertex clustering approach by Rossignac and Borrel [RB93] 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, and the use of a clustering-based simplification guarantees that adjacent mesh portions have coherent common boundaries. The adaptive clustering employed in [SG01] leads to an even higher quality result, whereas in [CMRS03] boundaries are kept consistent at each iteration thanks to a smart octree-based external memory data structure. Both [Lin00] and [SG01] solve the problem of boundary coherence, but the quality of their simplifications is still not comparable with traditional methods based on global priority queues [GH97]. On the other hand, [CMRS03] provides

© The Eurographics Association 2015.

high quality simplifications, but the approach is not suitable for a distributed setting.

The possibility to exploit distributed environments is scarcely treated in the literature. In [BP02] this possibility is considered but, due to the use of a distributed 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 is not supported.

3. The Geometric Workflow System

Our system have been designed with the objective of supporting computer graphics and geometry processing research activities. A standard Web browser is sufficient to remotely run complex geometric pipelines by distributing the various sequential steps on different servers that expose algorithms in form of Web services.

The framework architecture is organized in three layers, according to [Hol95]. The first layer includes a graphical user interface that allows building new workflows from scratch, and uploading and invoking existing workflows. The second layer contains the workflow engine responsible of runtime execution, while the third layer includes the web services that wrap geometry processing tools.

3.1. The Graphical User Interface

A dedicated user-friendly interface supports the creation of new geometry processing workflows. First, the user is asked to provide the information directly related to the workflow, such as its name, description and the list of geometry processing algorithms that constitute the pipeline. The user may define a new workflow by selecting atomic tasks from a list of available ones. Besides atomic tasks, the user is allowed to define possible conditional tasks or loops by specifying their conditions and by delimiting their execution bodies. Once the whole procedure is defined, the user can turn it into an actual experiment by uploading an input mesh. If no input is associated, the workflow can be stored on the system as an abstract procedure that can be selected later for execution.

3.2. The Workflow Engine

The workflow engine is the core of the system and orchestrates the invocation of the various algorithms involved. 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 workflow engine reads the encoded pipeline and sequentially invokes the various algorithms. For each operation, the engine searches the net to find an available Web service able to perform the task. When the selected Web service is triggered for execution, it receives from the engine the address of the input mesh and possible parameters, runs its task and returns the address of the generated output to the engine. This latter information is sent to the next involved Web service as input mesh or returned to the user interface when the workflow execution terminates.

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 specific Web services.

3.3. The Web Services

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. The system supports the invocation of two types of Web services, namely atomic and boolean. The former are required to:

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

while the latter just analyze the mesh and return a boolean value. Boolean Web services are used to support the execution of conditional tasks and loops.

4. Mesh Transfer Protocol

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 URL of the input mesh and to download it locally; also, after the execution of the algorithm, the output must be made accessible through a standard URL to be returned to the calling service. Not surprisingly, we have observed that the transfer of large-size meshes from a server to another according to the aforementioned protocol 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 a mesh transfer protocol inspired on the prediction/correction metaphor used in data compression [TG98].

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. Furthermore, in many cases modifications can be only

© The Eurographics Association 2015.

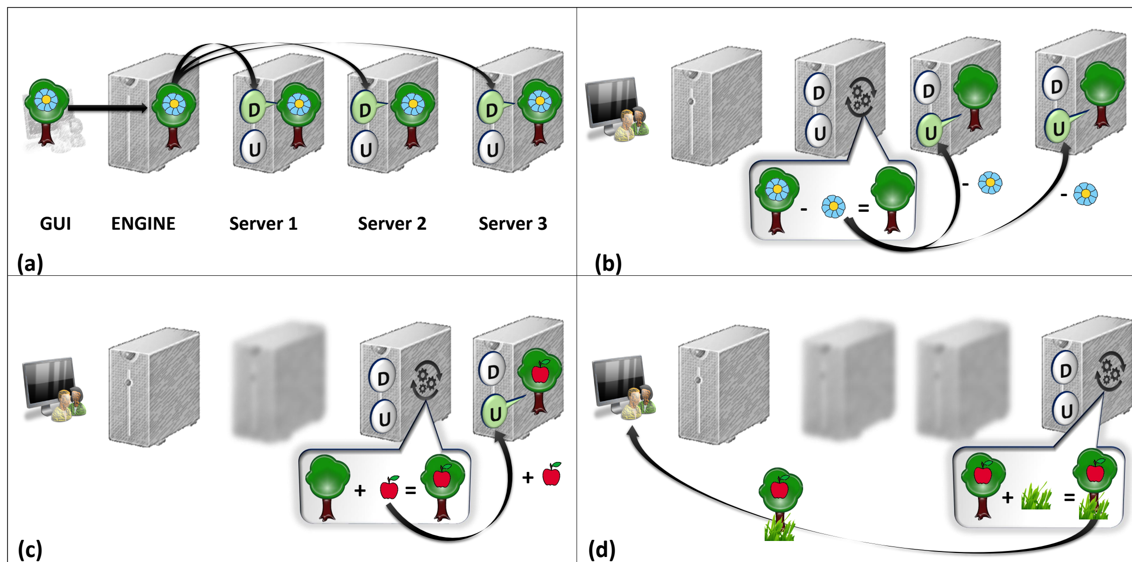

Figure 1: *Mesh Transfer Protocol Example. Three servers are involved into the workflow execution. Each of them exposes a web service to support a geometry processing algorithm and two modules able to download (D) meshes and update (U) the previously downloaded mesh by applying the corrections. (a) The 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 successive servers involved. 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 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 the modified output mesh available so that it can be directly downloaded by the user.*

local (e.g. sharp feature restoration), may involve the geometry only while keeping the connectivity unaltered (e.g. most mesh deformation algorithms), or may modify both geometry and connectivity while minimally changing the overall shape (e.g. remeshing). 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 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 1, where an example of execution of a simple workflow composed by three tasks is shown. 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 download it.

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.

© The Eurographics Association 2015.

5. Distributed Processing

Although our system theoretically allows to process any input mesh, remote servers may not satisfy specific hardware requirements (eg. huge main memory, high computational performance) necessary to efficiently process large data. As a consequence, the remote server that is invoked may require a very long time to finish its task or, even worse, a memory leak may occur and interrupt the elaboration. In order to avoid these situations, the workflow engine is responsible of managing the runtime execution by exploiting the traditional *divide and conquer* approach. Hence, it is able to partition a huge 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 in order to support input meshes that are too large to fit into main memory. Also, we assume that the input mesh is encoded as an indexed mesh, since the most diffused file formats are based on this representation.

In the reminder, 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, 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.

5.1. 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 server) and the number of available servers 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. 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 servers and the ratio between M and the subsample size.

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 2). Simultaneously, a global vector file V_{file} is created where, for each vertex, the ID of the corresponding BSP cell is stored.

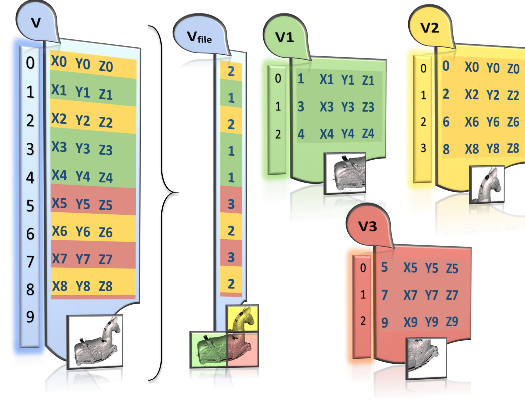

Figure 2: *Partitioning of vertices. For each BSP cell, a corresponding file is created. Vertices are read one by one and assigned based on their spatial location. Global indexes are shown on the left of the original V, while local indexes are on the left of each V_i. Global indexes and coordinates are written on each V_i. V_{file} stores, for each vertex, the ID of the corresponding BSP cell.*

Then, the partitioner classifies the 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. The triangle is assigned to that same cell.
2. Two vertices belong to cell C_A while the third vertex belongs to cell C_B. The triangle is assigned to cell C_A and a copy of the third vertex is added to V_A.
3. The three vertices belong to three different cells C_A, C_B, and C_C. 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 added to V_A.

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

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. 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.

© The Eurographics Association 2015.

5.1.1. 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.

The idea is to exploit the concept of independent sets to reduce communication among servers. During partitioning, generated submeshes are grouped into independent sets so that submeshes in the same group do not share any vertex. Thanks to this grouping, submeshes in the same independent set can be processed simultaneously by different processes without the need to communicate. Each process is asked to return both the output result and the list of applied modifications that involve neighbor submeshes (eg. boundary modifications). The synchronization among neighbor submesh can be handled in a dedicated computational step before starting the processing of the other independent sets.

To provide this feature, an adjacency graph for the submeshes is defined where each node represents a BSP cell, and an arc exists between two nodes if their corresponding BSP cells are "mesh-adjacent". 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, corresponding arcs are added to the graph. 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]. Clearly, the maximum number of nodes included in the same group is limited by N_s.

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

5.2. Output Merging

When all the sub-meshes have been processed, the engine is responsible of merging them and generating the output indexed mesh. To achieve the goal, each processing service is required to return both the output submeshes M_i' and an extra file B_i' where the list of boundary vertices of M_i' is saved, sorted by their local index. Each boundary vertex is described as a pair $\langle$*local index, global index*$\rangle$ (Figure 3).

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).

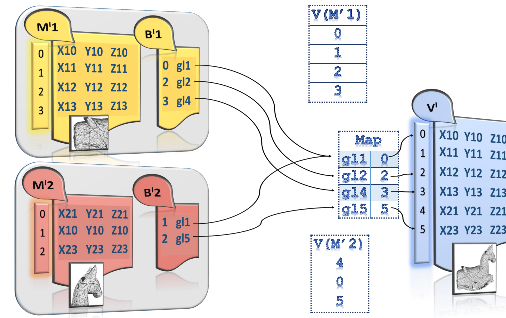

Figure 3: *Merging vertices. First, vertices in M_1' are read one by one and coordinates are added to V_f. For each boundary vertex in M_1', its global index is saved in Map. Map is sorted by global indexes ($gl_1 < gl_2 < gl_4 < gl_5$). Then, vertices in M_2' are read and coordinates are added to V_f. 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.*

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 master 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 $\log_2 n$ 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.

6. Distributed Mesh Simplification

In our reference scenario, the user wants to simplify a large mesh by exploiting the system. To achieve this goal, the engine stores the model on its local disk and invokes the re-

© The Eurographics Association 2015.

quired services. For the sake of simplicity, our exposition assumes that all the N_s available servers have an equally-sized memory and a comparable speed.

The distributed simplification algorithm works as follows. In the first step, the engine partitions the mesh into a set of submeshes using the previously described algorithm (Sec. 5.1). 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. 5.2) to join the simplified submeshes along their boundaries, which are guaranteed to match exactly.

6.1. Adaptivity

Each submesh is simplified by a single Web service through a standard iterative edge-collapse approach based on quadric error metric [GH97]. 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. In order to preserve the appearance of the original shape and support adaptivity, the simplification algorithm applied by each service stops when a maximum error max_E is reached.

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.

6.2. Boundary Problem

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, each containing the list of boundary vertices shared with the already processed neighbor k and removed during the simplification process.

Before starting the simplification, the service is asked 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. Then, the service is required to 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.

Thus, the service returns to the engine the simplified submesh M'_i, the corresponding set of *Rem* files, and and additional file B'_i storing the global index of all the remaining (i.e. unsimplified) boundary vertices of M'_i. The engine is responsible of sharing each Rem_k with the service that will be further invoked to manage the neighbor submesh k. Simplified sub-meshes along with B'_i files are used by the engine to generate the final output by exploiting the aforementioned out-of-core merge algorithm.

Our simplification algorithm proves the benefits provided by our partioning/merging approach, but it also has other noticeable characteristics. However, their description would bring us too far from the scope of this paper, hence we refer the reader to [CA15] for details.

7. Results

For the sake of experimentation, the proposed Workflow Engine has been deployed on a standard server running Windows 7, whereas other web services implementing atomic tasks 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. All the machines involved in the experimentation are equipped with Windows 7 64bit, an Intel i7 3.5 GHz processor, 4GB Ram and 1T hard disk.

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 [dsw12]. As an example, one of our test workflow is composed by the following operations: Removal of Smallest Components (RSC), Laplacian Smoothing (LS), Hole Filling (HF), and Removal of Degenerate Triangles (RDT). 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 1 reports the size of the output mesh and the size of the correction file after each operation (both after compression) whereas Table 2 shows the total time spent by the workflow along with a more detailed timing for each single phase.

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

© The Eurographics Association 2015.

Mesh	RSC	LS	HF	RDT
Rome*	14.915	15.551	14.915	13.166
	1	1.425	1	1
Dolomiti*	11.146	11.637	11.146	10.588
	1	1.402	1	1
Isidore	20.573	23.333	23.717	25.497
	11	9.433	154	2
Nicolo	19.498	21.447	20.601	20.171
	3	9.296	48	2
Neptune	39.881	40.131	39.891	39.937
	1	15.237	1	1
Ramesses	17.484	19.544	19.934	19.802
	3	8.754	149	3
Dancers	16.457	18.037	18.325	18.116
	1	7.220	80	1

Table 1: *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). A modified version of the Hole Filling algorithm has been run to process "2.5D" geospatial data (*) in order to preserve their largest boundary.*

significant benefits when the single steps produce mainly little or local mesh changes.

For each mesh in our dataset, Table 2 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.

To test our partitioning and simplification algorithm, large meshes extracted from the Stanford online repository [sta96], from the Digital Michelangelo Project [mic09] and from the IQmulus Project [iqm13] were used as inputs. Some small meshes have been included in our dataset to evaluate and compare the error generated by the part-by-part simplification.

For each input model, we ran several tests by varying the number of involved web services 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 available 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. Table 3 shows the time spent by the system to finish the elaboration. 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 services increases. More noticeably, speedup increases as the input size grows.

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. Results show that the number of services does not significantly affect the quality of the output.

For larger models, Metro cannot be used and quality can be assessed based on a visual inspection only. Figure 4 and 5 show that high quality is preserved in any case and is not sensibly affected by the number of involved services.

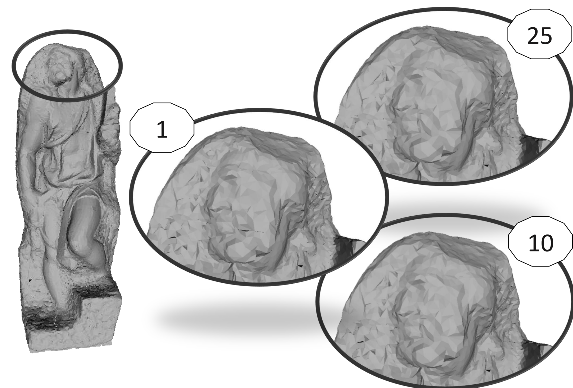

Figure 4: *Detail of St. Matthew model simplified by exploiting 1, 10 and 25 available services (original: ≈ 187M vertices, simplified: ≈ 119K vertices).*

8. Conclusions

We proposed a workflow-based framework to support collaborative research in geometry processing. It allows scientists to remotely run geometric algorithms provided by other researchers as Web services and to combine them to create executable geometric workflows.

The platform is accessible from any operating system through a standard Web browser with no hardware or software requirements. A prototypal version is available at `http://visionair.ge.imati.cnr.it/workflows/`. Expert programmers can avoid reimplementing known algorithms, while scientists in other areas do

© The Eurographics Association 2015.

Mesh (# vertices)	IB	RSC	T$_1$	U$_1$	LS	T$_2$	U$_2$	HF	T$_3$	U$_3$	RDT	Total	Benefits
Rome* (957.456)	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* (810000)	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 (1071671)	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 (945924)	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 (1321838)	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 (775715)	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	
Dancers (703207)	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 2: *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. A modified version of the Hole Filling algorithm has been run to process "2.5D" geospatial data (*) in order to preserve their largest boundary.*

Input				#ISs	# Output Vertices	Times				Speedup
Mesh (# vertices)	N_v	max_E	N_s			**Partitioning**	**Simplification**	**Merging**	**Total**	
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 3: *Execution times (in seconds). The speedup increases with the input size. Column labels: N_v is the number of vertices per-service, 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 services, #ISs is the number of generated independent sets.*

no longer need to be skilled programmers or experts in geometric modelling to exploit state-of-the-art algorithms. Also, available pre-defined workflows can be reused by other researchers. Hence, results of short-lasting experiments can be recomputed on the fly when needed and there is no more need to keep output results explicitly stored on online repositories. Since experiments can be effciently encoded as a list of operations, sharing them instead of output models sensibly reduces required storage resources.

Currently, our framework provides some "in-house" geometry processing algorithms, but the architecture is open and fully extensible by simply publishing a new algorithm as a web service and by communicating its URL to the system.

Moreover, we have demonstrated that the computing power of a network of PCs can be exploited to significantly speedup the elaboration of large triangle meshes and we have shown that the overhead due to the data transmission is much lower than the gain in speed provided by parallel processing.

In its current form, our system has still a few weaknesses: experiments can be reproduced only as long as the

© The Eurographics Association 2015.

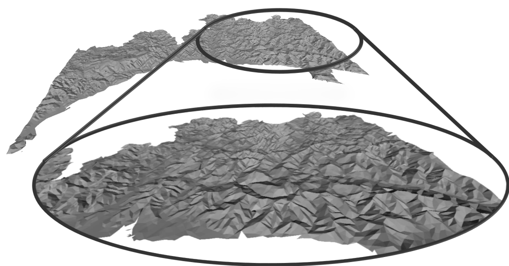

Figure 5: *Detail of Terrain model. Nearly height fields are naturally supported (original: ≈ 68M vertices, simplified: ≈ 11.5K vertices).*

involved Web services are available and are not modified by their providers. To reduce the possibility of workflow decay [ZGPB*12] a certain level of redundancy would be required, for example by uploading the same Web service on different machines.

Finally, our system can load a single indexed mesh and produces a single file. Part of our future plans include the study of methods to efficiently represent big meshes through several files that can be hosted on different machines.

References

[BP02] BRODSKY D., PEDERSEN J. B.: Parallel model simplification of very large polygonal meshes. In *Procs. of Parallel and Distributed Processing Techniques and Applications - Volume 3* (2002), PDPTA '02, pp. 1207–1215. 2, 3

[CA15] CABIDDU D., ATTENE M.: Large mesh simplification for distributed environments. *Computers & Graphics 51* (2015), 81 – 89. 7

[Cam] CAMPEN M.: WebBSP 0.3 beta. http://www.graphics.rwth-aachen.de/webbsp. URL: http://www.graphics.rwth-aachen.de/webbsp. 2

[CCR08] CIGNONI P., CORSINI M., RANZUGLIA G.: Meshlab: an open-source 3d mesh processing system. *ERCIM News*, 73 (April 2008), 45–46. 2

[CMRS03] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics 9*, 4 (oct 2003), 525–537. 2

[CRS98] CIGNONI P., ROCCHINI C., SCOPIGNO R.: Metro: Measuring error on simplified surfaces. *Comput. Graph. Forum 17*, 2 (1998), 167–174. 8

[CSS98] CHIANG Y. J., SILVA C. T., SCHROEDER W. J.: Interactive out-of-core isosurface extraction. In *IEEE Visualization'98* (1998), pp. 167–174. 2

[DLR00] DEHNE F., LANGIS C., ROTH G.: Mesh simplification in parallel. In *Procs. of Algorithms and Architectures for Parallel Processing* (2000), ICA3PP 2000, pp. 281–290. 2

[dsw12] DSW v5.0 - visualization virtual services., 2012. URL: http://visionair.ge.imati.cnr.it. 7

[FS00] FRANC M., SKALA V.: Parallel triangular mesh reduction. In *Procs. of Scientific Computing* (2000), ALGORITMY 2000, pp. 357–367. 2

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Procs. of SIGGRAPH '97* (1997), pp. 209–216. 2, 7

[HKK14] HUTTER M., KNUTH M., KUIJPER A.: Mesh partitioning for parallel garment simulation. In *Procs. of WSCG 2014* (2014), pp. 125–133. 6

[Hol95] HOLLINGSWORTH D.: *Workflow Management Coalition - The Workflow Reference Model*. Tech. rep., Jan. 1995. 3

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. In *Procs. of SIGGRAPH '03* (2003), pp. 935–942. 2

[ILGS03] ISENBURG M., LINDSTROM P., GUMHOLD S., SNOEYINK J.: Large mesh simplification using processing sequences. In *Visualization, 2003. VIS 2003* (October 2003), pp. 465–472. 2

[iqm13] Iqmulus: A High-volume Fusion and Analysis Platform for Geospatial Point Clouds, Coverages and Volumetric Data Sets., 2013. URL: http://www.iqmulus.eu. 8

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *Procs. of SIGGRAPH '00* (2000), pp. 259–262. 2

[LS01] LINDSTROM P., SILVA C. T.: A memory insensitive technique for large model simplification. In *IEEE Visualization* (2001), pp. 121–126. 2

[mic09] The Digital Michelangelo Project., 2009. URL: http://graphics.stanford.edu/projects/mich/. 8

[MK12] MÖBIUS J., KOBBELT L.: Openflipper: An open source geometry processing and rendering framework. In *Procs. of Curves and Surfaces* (2012), pp. 488–500. 2

[Pit10] PITIKAKIS M.: *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. 2

[RB93] ROSSIGNAC J., BORREL P.: Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics*. 1993, pp. 455–465. 2

[SG01] SHAFFER E., GARLAND M.: Efficient adaptive simplification of massive meshes. In *Procs.of Visualization '01* (2001), pp. 127–134. 2

[SN13] SHONTZ S. M., NISTOR D. M.: CPU-GPU algorithms for triangular surface mesh simplification. In *Procs. of Meshing Roundtable* (2013), pp. 475–492. 2

[sta96] The Stanford 3D Scanning Repository., 1996. URL: http://graphics.stanford.edu/data/3Dscanrep. 8

[TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Graphics Interface* (1998), pp. 26–34. 3

[TJL07] TANG X., JIA S., LI B.: Simplification algorithm for large polygonal model in distributed environment. In *Advanced Intelligent Computing Theories and Applications*, vol. 4681 of *Lecture Notes in Computer Science*. 2007, pp. 960–969. 2, 3

[TPB08] THOMASZEWSKI B., PABST S., BLOCHINGER W.: Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics 32*, 1 (2008), 25–40. 2

[ZGPB*12] ZHAO J., GOMEZ-PEREZ J. M., BELHAJJAME K., KLYNE G., GARCIA-CUESTA E., GARRIDO A., HETTNE K., ROOS M., DE ROURE D., GOBLE C.: *Why workflows break: understanding and combating decay in Taverna workflows*. 2012, pp. 1–9. 10

© The Eurographics Association 2015.