Camarón: An Open-source Visualization Tool for the Quality Inspection of Polygonal and Polyhedral Meshes

Aldo Canepa[1], Gonzalo Infante[1], Nancy Hitschfeld[1] and Claudio Lobos[2]
[1]*Departamento de Ciencias de la Computación, FCFM, Universidad de Chile, Santiago, Chile*
[2]*Departamento de Informática, Universidad Técnica Federico Santa María, Santiago, Chile*

Keywords: Polygonal Meshes, Polyhedral Meshes, Quality Criteria, Data Visualization, GLSL.

Abstract: The numerical simulation of phenomena requires a good quality discretization (mesh) of the domain. Depending on the problem to be simulated, the mesh has to fulfil different quality criteria. Because of geometry restrictions or point density requirements, several mesh elements might not satisfy the required quality criteria and sometimes it is also not required that all elements fulfil them. Then, it would be helpful to know where unwanted elements are located in order to see if they need to be repaired or not. That is why a visualization tool that allows the user to inspect a mesh before a simulation is performed can be useful to prevent simulation problems. Moreover, if data from simulations is available, the visualization of geometrical properties together with simulation data could be also helpful to understand not expected results. These challenges have motivated us to develop *Camarón*, a visualization tool for large surface and volume meshes described in this paper. The surface meshes can be composed any polygonal cell and the 3D meshes can include any convex polyhedral cell. This tool was implemented in C++ and the OpenGL Shading Language (GLSL). We discuss the design and implementation issues that make our software portable, extensible and different from other visualization tools. We also compare the performance between *Camarón* and GeomView, TetView and MeshLab.

1 INTRODUCTION

The numerical simulation of complex objects requires a good domain discretization (mesh). In 2D, meshes are usually composed of triangles and/or quadrilaterals and, in less frequent cases, of convex polygons. In 3D, meshes are commonly composed of tetrahedra and/or hexahedra. In case of mixed element meshes, pyramids, prisms and other convex polyhedra might also be included. A good quality mesh depends on the problem to be solved and the chosen numerical method. Different quality criteria have been defined using geometric properties of the mesh elements such as minimum (dihedral) angle, maximum (dihedral) angle, and aspect ratio, among others. Quality criteria are used to control the refinement and improvement process of a mesh. Because of geometry restrictions or point density requirements, frequently not all mesh elements fulfil the quality criteria required by the user. It would be helpful to know where bad elements are located in order to try to improve them. That is why a visualization tool that allows the user to inspect a mesh before a simulation is performed can be useful to prevent simulation problems.

Currently, there are several open-source visualization tools, some of them only dedicated to visualize the spatial discretization and others to visualize scientific data together with the mesh. Both types of visualization tools render meshes composed of only one element type or a limited type of elements, and are usually associated to a mesh generator. For example, TetView (Si, 2013) is related to the tetrahedral mesh generator TetGen (Si, 2015) and it is able to visualize tetrahedral meshes. GeomView (Amenta et al., 1995) was specially designed for the visualization of surface meshes composed of any polygonal cell. A tool that integrates mesh generation and visualization is MeshLab (Cignoni et al., 2008). It is oriented to generate, repair and process 3D triangular meshes. MayaVi (Ramachandran and Varoquaux, 2011), VisIt (Childs et al., 2011; VisIt, 2015) and ParaView (Henderson and Ahrens, 2004) are popular visualization tools oriented to visualize large datasets by providing techniques to analyze them. These last tools are implemented on VTK (Vtk, 2015; Hanwell et al., 2015), a software system that supports a variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods. None of

Canepa, A., Infante, G., Hitschfeld, N. and Lobos, C.
Camarón: An Open-source Visualization Tool for the Quality Inspection of Polygonal and Polyhedral Meshes.
DOI: 10.5220/0005830501280135
In *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2016) - Volume 1: GRAPP*, pages 130-137
ISBN: 978-989-758-175-5

them allow the user interactive evaluation of the geometrical mesh quality and the visualization of meshes composed of any convex polyhedral cells. Note that the use of different element types reduces the amount of edges, faces and elements in the final mesh that is why the generation of polyhedral meshes is starting to be considered (Garimella et al., 2013; Contreras and Hitschfeld-Kahler, 2014).

In this paper, we present the design and implementation of an open-source portable and extensible visualization tool for large polygonal and polyhedral meshes. The current implementation allows the user to: (1) open and save a mesh from/to formats such as OFF, PLY, M3d, Ansys, TRI and Visf. Visf is an own binary format, designed to handle polyhedral meshes. It is an extension of the OFF format that can also store neighboring information, (2) apply to the mesh several rendering strategies such as flat and Phong shading, property and data visualization rendering, (3) evaluate the mesh using known quality criteria such as minimum and maximum (dihedral) angle, aspect ratio and volume, (4) select mesh elements that fulfil specific quality criteria or intersect a primitive such as a sphere, a set of polygons defining a convex polyhedron, or a set of planes, and (5) generate statistics to identify if the mesh has the expected quality or not. The tool was implemented in C++ and GLSL (Wolf, 2011). We did not implemented this tool on VTK because we wanted to use as efficient as possible the computational resources and permit the visualization of non-manifold meshes. We also wanted to design an extensible software in selection, rendering and evaluation strategies. The results of a performance and memory usage comparison among *Camarón*, GeomView, TetView and MeshLab shows that our tool is faster but uses more memory than the other tools.

This paper is organized as follows: in Section 2 we introduce the motivation and challenges of our work. In section 3 we describe the tool architecture, in particular, the methodology we have followed, and design and implementation issues. In section 4 we compare *Camarón* with Meshlab, GeomView and TetView both in performance and memory usage and in Section 5 we include the conclusions and future work.

2 MOTIVATION AND CHALLENGES

We are interested in developing new algorithms for the generation of polyhedral meshes for different kind of applications. Each application usually requires that the mesh fulfills particular quality criteria, but sometimes it is difficult or impossible to generate only good quality elements. Through the visual inspection of a mesh it can be detected where bad quality elements are located and decide if the mesh still needs improvements or it can be already used. In order to support this kind of inspection of a mesh, we have decided to develop a new visualization tool where a user can:

- Handle any element shape (convex polygons and polyhedra)
- Apply the most known quality criteria
- Generate statistics about the quality of the elements
- Detect where unwanted elements are
- Apply different rendering strategies
- Select mesh elements through several selection strategies
- Visualize large meshes
- Open and save meshes stored in the most known formats

In addition, it is important for us to develop software that is portable and easy to extend in quality criteria, input/output formats, rendering and selection strategies.

3 TOOL ARCHITECTURE

This section starts with the description of the methodology we have used to develop *Camarón*, then continues with main aspects of the design and finishes with implementation details and the user interface.

3.1 Methodology

In this work, we take advantage of our experience developing extensible object-oriented software for building families of meshing tools (Bastarrica and Hitschfeld-Kahler, 2006; Rossel et al., 2014). Our design considers (1) types and subtyping to model the objects whose instances can evolve in time (Halbert and O'Brien, 1987), (2) guidelines to write good classes (Eliëns, 1995) and (3) the use of design patterns (Gamma et al., 1995) and good software engineering practices (Lethbridge and Laganiére, 2005). These three points helped us to build software easy to understand, maintain and extend. Together with *Singleton* and *Factory Method*, the following design patterns are used:

- *Strategy*: to easy add and interchange different algorithms to solve a similar problem
- *Visitor*: to separate algorithms/strategies (class Visitor) from the objects (class Element) to be applied. The methods of the classes that extends Element use *Double Dispatch* in order to apply the right Visitor method during execution time.

Double Dispatch is a technique that helps one to choose what function to execute, when there exist several with the same name, depending on the objects involved in the call during the execution time.

3.2 Design

In order to build a visualization tool that can render point clouds, polygonal and polyhedral meshes, we have modeled the basic element by the class Element, from which inherits the classes Point, Polygon and Polyhedron. From Polygon inherits the class Triangle and Quadrilateral, because this allows us to handle efficiently triangular and quadrilateral meshes. From Polyhedron inherits Tetrahedron and Hexahedron in case of tetrahedral or hexahedral meshes are drawn.

To manage several input/output formats, rendering algorithms, selection modes and evaluation strategies we used the *Strategy* pattern. For example, we have designed the abstract class EvaluationStrategy to apply any quality criterion to the individual mesh element types. From EvaluationStrategy inherits MinimumAngle and MaximumDihedralAngle, among others.

The selection strategies can be applied to the whole mesh or to a subset of elements that was already selected. We created the class SelectionStrategy from which inherits SelectById and SelectByProperty, among others. SelectById allows a user to select elements by using their indices and SelectByProperty to select the elements that fulfil or not the specified quality criterion. The *Visitor* pattern is implemented between the selection strategies and the mesh elements so that by using the *Double Dispatch* technique the right strategy is applied to a particular mesh element.

The rendering strategies are also modeled in a similar way. There exists a base class called RendererStrategy and several subclasses that implements specific rendering algorithms. For example, NormalRenderer displays the model with normal vectors at each face vertex and PropertyRenderer draws each element with different colors according to the values of the applied quality criterion. For rendering the part of a model that intersects 3D convex shapes, we design the class ConvexGeometryIntersectionRenderer which inherits from RendererStrategy and SelectionStrategy. The idea was to use the specified 3D convex geometry for both to compute and render its intersection with the mesh and to select elements.

In order to extend the software without modifying the user interface or already implemented classes we have implemented dynamic registers. Each registry class is modeled with the *Singleton* pattern so that only one instance of each of them is created. Each extensible module must have an associated registry class. So, there exist the ModelLoadingFactory class, the RendererRegistry class and the SelectionStrategyRegistry class, among others. All these registries inherit from a RegistryTemplate class, which stores the different class instances associated to a key and a priority queue that allows the programmer to organize the order in which the different strategies will appear in the user interface.

Finally, a Controller class, contains references to all registry classes and is in charge of coordinating all the operations according to the user requests.

3.3 Implementation

3.3.1 General Issues

Meshes are expected to be composed of convex polygons and convex polyhedra. If non convex elements are included, some rendering errors can appear. Since in this application all geometrical properties remain constant while the same model is displayed, we send the computed values once to the GPU and keep them in the VRAM until a new model is loaded. We manage attributes per triangle vertex in order to store different attributes when the same vertex belongs to several triangles. So two triangles that share a vertex can be rendered, for example, with a different flat color. The polygons of surface meshes or the polygonal faces of polyhedra are triangulated first before sending their information to the VRAM, because GPUs only manage triangles. Each renderer class has a configuration widget, which allows the user to choose colors and other attributes for the selected and unselected elements. Each renderer has an associated Vertex shader, Fragment shader and Geometry shader if it is required. Several shaders to render the models were taken from the book (Wolf, 2011).

In order to develop software independent of the platform, we decided to use the Glew library (the OpenGL Extension Wrangler library) (Glew, 2012). This library provides efficient mechanisms to obtain which OpenGL functionalities are available on target GPU hardware. QtCreator was used to build the user interface.

3.3.2 Geometric Model Renderers

The property renderer was designed for the visualization of the elements that fulfill or not a particular quality criterion. The user chooses the property (quality criterion) he or she wants to inspect (minimum angle, aspect ratio, etc) and selects which colors will be used. A new attribute array is created to store the computed property values for the vertices and/or triangles. The user defines three different colors: null for the elements where the property does not apply, C_{max} for elements with the maximum value of the property and C_{min} for elements with the minimum value of the property. For elements where the property has a value between the minimum and maximum, the color is obtained by doing linear interpolation.

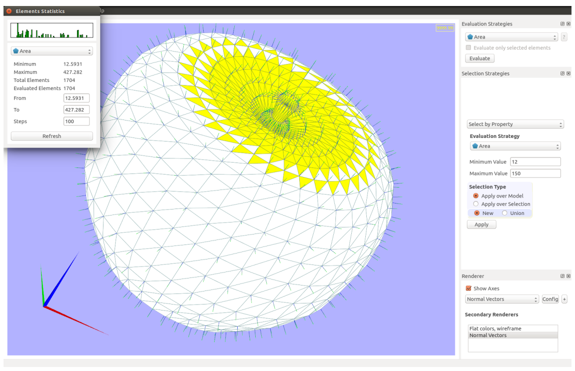
When rendering an intersected model, the 3D convex geometry intersecting a model can stay in a fixed position or can move with the model. In case the 3D convex geometry moves together with the model, the intersection is computed only once and the shader only takes care of rendering the elements by using other viewing parameters. In case the 3D convex geometry stays at a fixed position, the associated shader computes in real time its intersection with the model.

In order to draw the wireframe of the model, the algorithm described in (Bærentzen et al., 2006) is used. This algorithm achieves in a single pass to display the wireframe view of the mesh by handling the triangles through a geometry shader which calculates internal triangle distances from each vertex to its corresponding opposite edge. The smallest distances are then filtered in a fragment shader and colored different. As result it produces a wireframe view at a fast rate for large meshes.

3.3.3 Data Visualization Renderers

The isoline renderer uses a geometry shader for checking if an isoline value is within the bounds of each assembled triangle primitive. If this is the case, a new line segment is created by linearly interpolating the values on the edges of the triangle. The performance of the above implementation was poor in practice and a better solution for processing a large model is presented in the isosurface renderer description.

The isosurface renderer was implemented using a processing stage and a render stage. Both stages are executed on the GPU with their own shaders programs. The algorithm used to build the isosurfaces is based on the geometry shader marching cubes algorithm (Icare3D, 2015). In the first stage, *Camarón* subdivides every convex polyhedron into tetrahedra in a straightforward way. It then pass them to a geometry shader as 4-vertex primitives of type



Area
Minimum 12.5931
Maximum 427.282
Total Elements 1704
Evaluated Elements 1704
From 12.5931
To 427.282
Steps 100

Area
Select by Property
Area
Minimum Value 12
Maximum Value 150
[✓ Apply over Model] Apply over Selection
[✓ New] Union
[✓ Show Axes]
Normal Vectors
Flat colors, wireframe
Normal Vectors

Figure 1: Visualization of an apple model using simultaneously three rendering strategies.

GL_LINES_ADJACENCY. The four vertices are interpreted as a tetrahedron. Then, up to two triangles are interpolated per tetrahedron and their vertex positions are sent to the output of the geometry shader. Next we use the OpenGLs Transform Feedback functionality to save selected output variables into a buffer in VRAM. Once the isosurfaces are generated the second step takes place to render the new geometries. Note that the surface generation step is executed only when the isosurface values are changed by the user or a new model is loaded. In all other cases like rotation, zooming or panning only the render stage is applied.

3.4 User Interface

Figure 1 shows the interface of our tool by using a simple apple model of 1,704 triangles. At the right, the user can interact with the application. For surface meshes, minimum angle, maximum angle, aspect ratio, dihedral angle, and area, among others are available. Specific criteria for volume meshes are edge-radius ratio, aspect ratio, solid angle, and volume among others. The criteria for surface meshes can also be applied to the faces of the polyhedra. Histograms are generated automatically for any criterion. Elements can be visualized using different colors according to their property value. In Figure 1, the yellow triangles are the one whose area is between 12 and 150. The histogram shows that there are more smaller than larger triangles. The range of area values is shown below the histogram. To see interior mesh elements, the mesh can be cut by several convex shapes: sphere, several planes and several polygons defining polyhedron or selected with the mouse.

Several renderers are implemented: Glass renderer, Phong renderer, Normal Vector, Flat renderer, Property renderer and Intersection renderer, among others. More than one renderer can be applied at the same time. For the apple model three render-

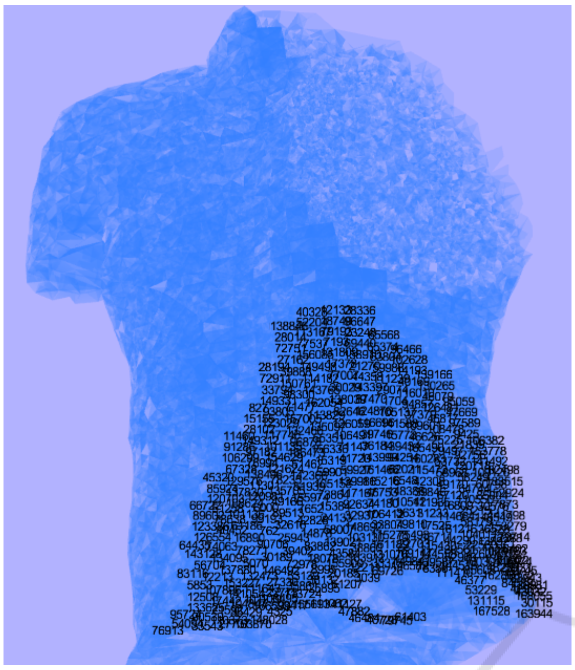


Figure 2: Visualization of a tetrahedral model using the Glass and the Intersection renderer.

ers have been applied: (1) the main renderer is always used when a selection strategy is chosen. In this case, it draws the yellow triangles, (2) the Flat renderer draws the white triangles and the black edges and (3) The Normal Vector renderer shows the normal vectors. Mesh elements can also be selected and visualized by specifying their vertex, edge, face and polyhedron indices. This can be appreciated in Figure 2 that shows a torso model composed of 168,930 points and 1,082,723 tetrahedra. The torso was drawn by using the Intersection renderer to cut part of its shoulder and show some interior elements, the Glass renderer and the renderer to show the indices of some selected vertices.

In order to illustrate the isoline renderer, we chose the Dragon mesh from the Standford Repository. The triangular mesh has 437,645 vertices and 871,414 triangles. The scalar fields for the models were generated with a simple pseudo-noise generation algorithm to ensure the visualization of smooth curves. Figure 3 shows the mesh rendered using gray colors at the left and the isolines corresponding to 20 scalar values at the right. In a similar way, Figure 4 shows the application of the isosurface renderer to the Permanent Magnetic DC motor (pmdc) mesh taken form the Tetview webpage. The pmdc mesh has 109,189 tetrahedron, 237,291 triangle faces and 27,196 vertices. The isosurfaces corresponding to 10 scalar values are shown.

We have also defined and added a new format called Visf for handling any polyhedral mesh. This

Table 1: Hardware used for testing.

Hardware	Detail
CPU	Intel Core i5-3210M 2.5 Ghz
GPU	Nvidia Geforce GT 650M
Mem	8GB, DDR3, 800 MHz
OS	Windows 7

format is similar to the OFF format: after the faces, the number of polyhedra must be specified and then each polyhedron with its number of faces followed by the face indices. *Camarón* can be downloaded from http://sourceforge.net/p/camaron/code/ci/master/tree

4 EXAMPLES AND EVALUATION

The performance comparison is done by measuring the number of frames per second each visualization tool can generate. This value is obtained through the application Fraps (Fraps, 2012) for MeshLab, GeomView and TetView. The characteristics of the hardware used for testing are shown in Table 1.

4.1 Rendering of Surface Meshes

The surface meshes used for comparison were taken from free repositories available through the Internet. The models, number of vertices, number of polygons and formats are shown in Table 2. We chose these meshes because their vertex number is close to $2^i, i = 12, \ldots, 22$. Table 3 shows the frames per second measured for each visualization tool. The highest fps value Fraps gave us while measuring the frame rate of GeomView, MeshLab and TetView was 60fps. Then we also consider 60fps as the performance rate when *Camarón* generates frames at a higher rate. We can observe that *Camarón* gets the best performance rates because it does an intensively used of the GPU and the others not.

Table 4 shows the amount of RAM used for the three largest models. *Camarón* needs more RAM space than the other tools because it stores the geometric properties associated to each quality criteria and several neighbor relationships between mesh elements in order to compute the properties fast. We also observe as expected that the amount of memory increases linearly with the amount of vertices.

4.2 Visualization of Volume Meshes

We generated tetrahedral meshes of size $2^i, i = 14, \ldots, 19$ to compare the performance of *Camarón* and TetView. Figure 5 shows the number of vertices,

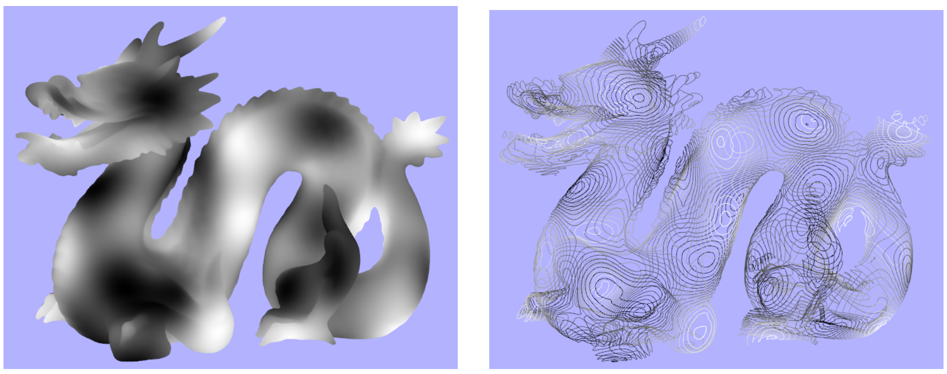


Figure 3: Isoline Renderer.

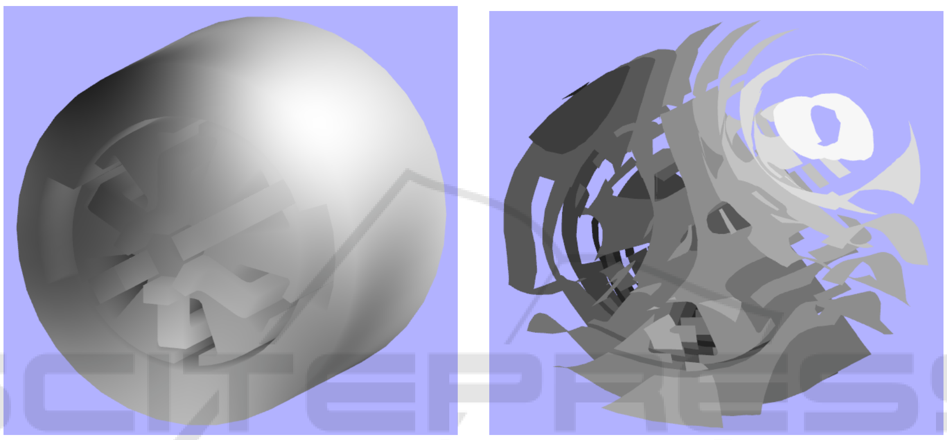


Figure 4: Isosurface Renderer.

Table 2: Geometrical information of the surface meshes.

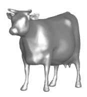
Object	Torus	Cow	Bunny	Lucy	Ramses	Neptuno	Thai
							
Points	4,800	16,612	35,947	262,999	826,266	2,003,932	4,999,996
Polygons	9,600	33,244	69,451	525,814	1,652,528	4,007,872	9,999,754
Format	TRI	OFF	PLY	TRI	OFF	OFF	PLY binary

Table 3: Number of frames per second (fps).

	TView	GView	MLab	*Camarón*
Torus	60	40	60	60
Cow	60	13	60	60
Bunny	57	6	47	60
Lucy	10	1	7	60
Ramses	5	<1	2	60
Neptuno	-	<1	1	21
Thai	-	<1	<1	6.5

triangles and tetrahedra of each sphere. The comparison is only between *Camarón* and TetView because

Table 4: Use of RAM(MB).

	TView	GView	MLab	*Camarón*
Ramses	437	137	281	913
Neptuno	-	326	465	2,086
Thai	-	806	970	4,645

MeshLab and GeomView were designed to handle surface meshes.

Figure 5 shows two views of a tetrahedral mesh: at the left side, the sphere was cut by a plane and the intersection is shown, and at the right its surface is displayed. Table 6 shows at the left the frames per

Table 5: Volume meshes.

Object	Sphere 1	Sphere 2	Sphere 3	Sphere 4	Sphere 5	Sphere 6
Points	16,384	32,768	65,536	131,072	262,144	524,288
Polygons	219,642	439,898	882,974	1,768,357	3,540,332	7,085,413
Tetrahedra	109,776	219,890	441,409	884,080	1,770,053	3,542,559

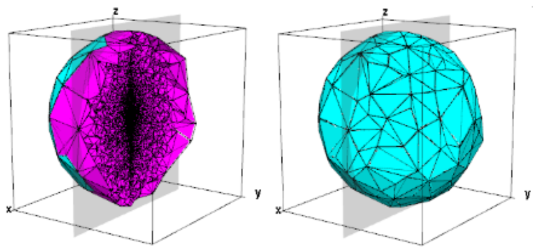


Figure 5: TetView visualization of the Sphere 3 cut by a plane (left) and its surface (right).

second each visualization tool took to visualize the cut sphere and at the right the whole sphere. We observe that TetView shows a good performance while drawing the whole sphere because in this case does not process all tetrahedra; it only draws the triangles at the surface and these are few. Our tool instead always processes all tetrahedra in parallel and in this case only draws the ones located at the surface. After cutting the model, the number of interior triangles to be rendered increases with the size of the tetrahedral mesh. That is why our tool shows a better performance than TetView. Note that TetView is quite slow if surface meshes have more than 200,000 points as shown in Table 3. Then it would be a larger performance difference between both visualization tools if larger surface meshes are drawn.

Table 6: Number of frames per second.

	TView	*Camarón*
Sphere 1	60:60	60:60
Sphere 2	58:60	60:60
Sphere 3	47:60	60:60
Sphere 4	33.60	60:60
Sphere 5	24:60	22:60
Sphere 6	14:60	9:21

Table 7: RAM usage (MB).

	TView	*Camarón*
Sphere 4	110	1,346
Sphere 5	170	2,478
Sphere 6	296	4,010

Table 7 shows the amount of RAM used for the three largest models. With volume meshes, our tool also requires more RAM than TetView because of the same reasons given for surface meshes.

5 CONCLUSIONS AND FUTURE WORK

We have presented the main characteristics of a new visualization tool designed for the quality inspection of surface and volume meshes composed of any convex polygons and polyhedra, respectively. This tool can also render non-manifold meshes and this is useful when partial meshes are built from images. The software design guarantees that the tool can easy evolve in new quality criteria, selection and render strategies, and new input/output formats.

The performance evaluation is still not complete. Since we have recently added the visualization of datasets, we also want to do a comparison with some of the mentioned data visualization tools (ParaView, VisIt). Until now, we have only compared its performance with MeshLab, TetView and GeomView and shown that our tool can render in real time large surface and volume meshes on standard desktop computers.

In the near future we plan to visualize data coming from simulations not necessarily associated to the mesh vertices. The idea is that a user can interactively detect and analyze how much not so good quality elements affect the simulation results. We also want to optimize the memory usage as far as possible.

ACKNOWLEDGEMENTS

This work was partially supported by Fondecyt Project N^o 1120495 and Enlace-Fondecyt N^o ENL009/15 (VID, University of Chile).

REFERENCES

Amenta, N., Levy, S., Munzner, T., and Phillips, M. (1995). Geomview: A system for geometric visualization. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, SCG '95, pages 412–413, http://www.geomview.org. ACM.

Bærentzen, A., Nielsen, S. L., Gjøl, M., Larsen, B. D., and Christensen, N. J. (2006). Single-pass wireframe rendering. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA. ACM.

Bastarrica, M. C. and Hitschfeld-Kahler, N. (2006). Designing a product family of meshing tools. *Advances in Engineering Software*, 37(1):1–10.

Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Bonnell, K., Miller, M., Weber, G. H., Harrison, C., Pugmire, D., Fogal, T., Garth, C., Sanderson, A., Bethel, E. W., Durant, M., Camp, D., Favre, J. M., Rübel, O., Navrátil, P., Wheeler, M., Selby, P., and Vivodtzev, F. (2011). VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *Proceedings of SciDAC 2011*. http://press.mcs.anl.gov/scidac2011.

Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., and Ranzuglia, G. (2008). MeshLab: an Open-Source Mesh Processing Tool. In Scarano, V., Chiara, R. D., and Erra, U., editors, *Eurographics Italian Chapter Conference*. The Eurographics Association.

Contreras, D. and Hitschfeld-Kahler, N. (2014). Generation of polyhedral delaunay meshes. *Procedia Engineering*, 82:291 – 300. 23rd International Meshing Roundtable (IMR23).

Eliëns, A. (1995). *Principles of object-oriented software development*. Addison-Wesley.

Fraps (2012). *Realtime Video Capture Software*. Beepa, Pty Lt. http://www.fraps.com.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Element of Reusable Object Oriented Software*. Addison-Wesley.

Garimella, R. V., Kim, J., and Berndt, M. (2013). Polyhedral mesh generation and optimization for non-manifold domains. In *IMR*, pages 313–330.

Glew (2012). *The OpenGL Extension Wrangler Library*. BSD License, MIT License. http://www.glew.sourceforge.net.

Halbert, D. and O'Brien, P. (1987). Using types and inheritance in object-oriented programming. *IEEE Software*, 4(5):71–79.

Hanwell, M. D., Martin, K. M., Chaudhary, A., and Avila, L. S. (2015). The visualization toolkit (vtk): Rewriting the rendering code for modern graphics cards. *SoftwareX*, 12:9 – 12.

Henderson, A. and Ahrens, J. (2004). *The Paraview guide : a parallel visualization application*. Kitware, Inc., New York.

Lethbridge, T. C. and Laganiére, R. (2005). *Object-oriented software engineering : practical software development using UML and Java*. Mc Graw Hill Education.

Ramachandran, P. and Varoquaux, G. (2011). Mayavi: 3D Visualization of Scientific Data. *Computing in Science & Engineering*, 13(2):40–51.

Rossel, P. O., Bastarrica, M. C., Hitschfeld-Kahler, N., Díaz, V., and Medina, M. (2014). Domain modeling as a basis for building a meshing tool software product line. *Advances in Engineering Software*, 70:77–89.

Si, H. (2013). *TetGen and TetView software*. http://tetgen.berlios.de.

Si, H. (2015). Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36.

VisIt (2015). *Open Source, interactive, scalable, visualization, animation and analysis tool*. Lawrence Livermore National Laboratory. https://wci.llnl.gov/simulation/computer-codes/visit.

Vtk (2015). *The visualization toolkit*. http://www.vtk.org/.

Wolf, D. (2011). *Open Gl 4.0. Shading Language Cookbook. Birminghang*. UK. Pack Publishing ltd.