INTERACTIVE VISUALIZATION OF LARGE AND ARBITRARY POLYGONAL AND POLYHEDRAL MESHES WITH OPENGL 4

Matthieu Maunoury[1] Rémi Feuillet[2] Adrien Loseille[3]

Inria Saclay, Gamma Team, 1 Rue Honoré d'Estienne d'Orves, 91120 Palaiseau, France.
{matthieu.maunoury[1], remi.feuillet[2], adrien.loseille[3]}@inria.fr

ABSTRACT

This paper describes an efficient strategy to visualize polygons and polyhedra using OpenGL 4 flexibility. Such meshes offer flexibility as the number of vertices and faces are arbitrary. Dual meshes are examples of polygonal and polyhedral meshes. We give explanations on how polygons and polyhedra can efficiently be stored in mesh files. Algorithms to tessellate polygons into triangles are described. Many examples and comparisons with another visualization software show that our methodology is efficient (about 40 times faster than ParaView). Interactivity is also ensured with post-processing tools such as picking and cut planes.

Keywords: Visualization, Polygonal Meshes, Polyhedral Meshes, OpenGL 4, GLSL, Shaders

1. INTRODUCTION

The design of efficient meshing techniques or the developments of new numerical schemes requires the ability to quickly load, visualize and inspect meshes and solutions. The efficiency is bounded by what we can see and should be possible on classic laptops and workstations. This process becomes critical when non linear elements are used. This is the case for polygonal meshes where a few effective rendering techniques exist.

The goal of numerical simulations is to predict the behavior of physical phenomenons without using prototypes or experimentations. Many domains are involved such as Computational Fluid Dynamic, acoustics, electromagnetism, or biomedical. In general, the numerical simulations pipeline is composed of a mesh generation step [1], then a problem is numerically solved with the help of this mesh and finally a numerical solution is obtained. All along the process, visualization is needed to check and validate the mesh and the solution, and give tools to analyze the results. The choice of the elements types in the mesh depends on the type of equations studied or on the solver. The most common elements are triangles and quadrilaterals for surfaces and tetrahedra and hexahedra for volumes but prisms or pyramids are sometimes used, especially when hybrid meshes are involved. Unlike the latter elements, one interest to use polygonal (for surfaces) and polyhedral (for volumes) meshes is the flexibility as elements have an arbitrary number of vertices. Figure 1 shows examples of such meshes.

Only a few commercial meshers and simulation packages such as `Simcenter StarCCM+` [2] or `OpenFOAM` [3] handle generic polygons and polyhedra. There has been little works on generation of polygonal and polyhedral meshes [4, 5, 6], some of them are generated as the dual of tetrahedral meshes. Other works focus on the construction of finite element interpolants on polygonal and polyhedral meshes [7, 8, 9, 10, 11, 12].

However, many visualization software programs do not handle polygons and when it is the case, the interactivity is often limited. It is also the case when high-order elements and solutions are considered. There are two main strategies: ray-casting with possibly

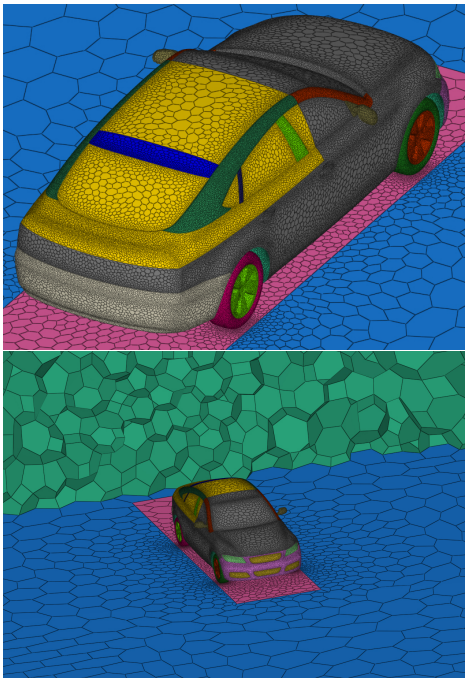

Figure 1: Examples of meshes composed of polygons only (top) and polygons and polyhedra (bottom). The number of vertices is not constant by element.

volume visualization and low-order remeshing. The first approach is ray-casting with volume visualization [13, 14]. A significant limitation of this technique is the cost and as a consequence it does not compete with the interactivity of the standard linear rendering methods. Furthermore, in the case of polygons and polyhedra, the cell-to-cell connectivity is required to traverse the mesh and needs lots of memory that is a limiting factor. The second approach is the low-order remeshing: the idea is to tessellate each element into triangles for surfaces or tetrahedra for volumes. Then, any visualization software is able to render these elements. For instance, `ParaView` [15] or `VisIt` [16] use this technique to represent polygons and polyhedra. New mesh visualization software solutions have also emerged [17, 18, 19, 20, 21, 22].

The goal of this paper is to explain how polygonal and polyhedral meshes are visualized using OpenGL 4. For this purpose, many points are detailed such as: storage and I/O, how tessellation of polygons into triangles is done, how post-processing tools like picking, clip planes is done. The tessellation algorithms presented in this paper do not add extra vertices in the tessellation as the aim is to minimize the number of triangles to maximize the rendering performances. All works presented in this paper have been developed in `ViZiR 4` that is freely available in its dedicated web site `http://vizir.inria.fr`.

The paper is outlined as follows. Section 2 is devoted to storage and I/O of polygons and polyhedra. Section 3 gives a presentation of the OpenGL 4 graphic pipeline. Section 4 tackles the problem of tessellation of polygons. Section 5 deals with post-processing tools and interactivity. Examples are given all along the paper and more complex examples are described in Section 6. Comparisons are done in this paper between `ParaView`, `VisIt` and the current approach.

2. STORAGE AND I/O OF POLYGONS AND POLYHEDRA

All the results collected in this paper have been generated with the same laptop: a MacBook Pro with details given in Table 1.

Hardware	Details
CPU	Intel Core i7 2.6 GHz 6-core
GPU	AMD Radeon Pro Vega 20 4 Gb
Mem	32 Gb of RAM 2400 MHz DDR4
OS	Mac

Table 1: Hardware used for testing.

A key to have an efficient visualization is to be able to quickly open mesh and solution files. Input and output are handled by the `libMeshb`[1] library. The files follow the `GMF` format provided by this library. For instance, the mesh of Lucy (see Fig. 2) with more than 14 millions vertices and 28 millions triangles (642 Mb) is opened in less than 1.5 seconds.

Figure 2: Rendering of a large mesh of 14M vertices and 28M triangles in 7.5 seconds (total time) on a laptop.

One difficulty to define polygons and polyhedra is that the number of vertices may be different for each element. It means that the number of vertices and number of faces for polyhedra must be defined for each element. In the following, some vocabulary is given and the storage of polygons and polyhedra is explained.

[1] `https://github.com/LoicMarechal/libMeshb`

- Boundary polygons: polygons that are displayed. Each element is characterized by an arbitrary number of vertices and the indices of these vertices. In practice, a list of all boundary polygons vertices is defined and for each element the beginning index and a reference are given. The number of vertices is deduced by looking at the index of beginning of the next element (i.e. the ending index is the one prior the starting index of the next element). It allows to access any element independently of all the previous polygons and very quickly.

- Inner polygons: polygons that are not displayed but are useful to define polyhedra: these polygons are faces of polyhedra. The definition of these inner polygons is the same as for boundary polygons: the beginning index of each inner polygon and the list of indices of these inner polygons vertices are given.

- Polyhedra: polyhedra that are displayed when intersecting the clip plane. In practice, a list of all polyhedra's faces (i.e. inner polygons) is defined and for each element the beginning index and a reference are given. Following the same idea than for polygons, the number of faces is deduced by looking at the index of beginning of the next element (i.e. the ending index is the one preceding the beginning index of the next element) and each element is accessed very quickly. Note that in practice, the faces of volume elements are rendering.

New keywords have been introduced to the `libMeshb` library to define these polygons and polyhedra. Furthermore, some functions have been introduced to ease the access of these data. All vertices are stored only once and are used to define boundary polygons, inner polygons and polyhedra. Then, all these elements are stored separately. This way, only boundary polygons are taken into account to display surfaces whereas inner polygons and polyhedra are only considered for cut plane. Information are generally stored in binary format to be more efficient but can also be written in ASCII format. The format follows the `libMeshb` library.

3. PRESENTATION OF THE OPENGL 4 GRAPHIC PIPELINE

The OpenGL 4 rendering pipeline can be customized with up to five different shader stages (see Fig. 3). These shaders are GLSL source code files that replace parts of the OpenGL pipeline. In general, a shader receives its input via developer-defined input variables, and the data for those variables come either from the main OpenGL application or previous pipeline stages (other stages). Data can also be provided to any shader using uniform variables or textures [23]. More details on OpenGL 4 and in particular OpenGL Shading Language (GLSL) can be found in [23, 24].

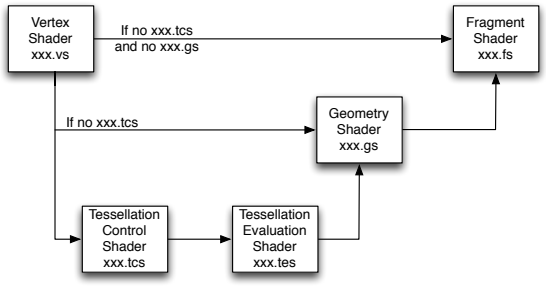

Figure 3: Shaders used for the OpenGL graphic pipeline.

Two shaders are enough to define a graphic pipeline, the vertex shader and the fragment shader. The vertex shader handles the vertices. The data corresponding to the vertices positions are tranformed into clip coordinates. The fragment shader determines the color for each pixel. Many parameters affect the color like a shading, a solution, an isoline, or a wireframe rendering. For the storage of raw data (like high-order solutions), textures are used.

Besides these two shaders, a geometry shader can be added to govern the processing of primitives. It allows to create new geometries on the fly. With this in mind, it can be preceded by the two tessellation shaders: the tessellation control shader and the tessellation evaluation shader. They are used to control the tessellation of the primitives, in other words, in how many sub-elements the elements should be divided.

OpenGL 4 graphic pipeline flexibility allows to compute on the fly the solution. It leads to a pixel exact rendering when flat elements (of degree one) are considered regardless of the degree of the solution. This recent language (GLSL) enables `ViZiR 4` to certify a faithful and interactive depiction. High order solutions are natively handled by `ViZiR 4` on surface and volume (tetrahedra, pyramids, prisms, hexahedra) meshes which can naturally be hybrid.

When more complex geometries are considered, curved elements perform a better approximation of the geometry. In this case, tessellation shaders occur in OpenGL pipeline (see [20, 21] for more details on the shaders pipeline) to tessellate all elements directly on the GPU. For solutions on such curved elements, almost pixel exact rendering is ensured [21].

4. TESSELLATION OF POLYGONS INTO TRIANGLES

4.1 Why tessellate polygons into triangles

As explained in Section 3, only the vertex and the fragment shaders are mandatory. For instance, in the case of triangles (of degree 1), the geometry shader could be useful for example to compute and display normals vectors but in many cases, this shader is avoided. The reason is that the use of this shader is expensive. To illustrate this cost, a comparison is done on the mesh of Lucy (28M triangles, see Fig. 2) and is outlined in Table 2. In the first case, only the vertex and fragment shaders are used, and the number of Frames Per Second is 28. In the second case, a geometry shader is added, and does nothing more than pass data through itself (i.e. the rendering is exactly the same than in the first case) and the number of Frames Per Second falls to 6. The rendering of triangles has good performance because only vertex and fragment shaders are used while additional shaders are necessary for more complex elements. For this reason, polygons are tessellated into triangles. However, when High Order elements are displayed, Geometry and Tessellation shaders should be used as it is the only way to have a good rendering (done on the GPU) of them.

GLSL pipeline	FPS
Vertex + Fragment Shaders	28
Vertex + Geometry + Fragment Shaders	6

Table 2: Comparisons on Lucy mesh (28M triangles) of FPS when Geometry shader is used or not.

4.2 How to tessellate polygons

We consider only simple polygons, which means polygons with no two non-consecutive edges intersecting, these polygons are convex or concave. An example of tessellation used for the rendering is shown in Fig. 4.

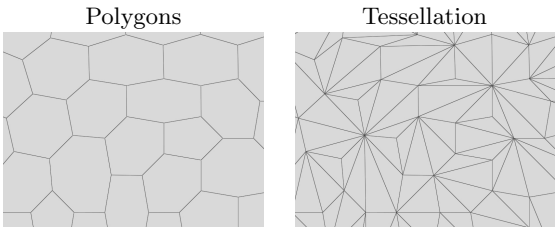

Figure 4: Rendering of polygons (left) and their tessellation into triangles (right) used for the rendering.

4.2.1 Edge visibility

During the creation of the tessellation, the visibility (i.e. a Boolean) of the 3 edges of each triangle is defined. Indeed, some edges need to be visible as their correspond to the boundary of a polygon while others should not be visible as lying inside the polygon. Textures are used to send the information on visibility (booleans) to the fragment shader so that the appropriate color can be set according to the position (inside or on the boundary of the polygon) of the edge.

4.2.2 Definitions of normal by polygon

Normals of elements are important because they are used in the shading, for instance in Phong model [25]. Usually, the normal is computed for each element and given to the shaders by textures. To have a smoother shading, one normal n_{poly} for each polygon is defined. Let's first define:

$$\mathcal{A} = \sum_{i=1}^{d-2} (P_{i+2} - P_1) \wedge (P_{i+1} - P_1) \qquad (1)$$

where d is the number of vertices of the polygon, P_i the points of the polygon and $\wedge$ the usual vector cross product. Note that this definition of normal depends on the choice of the first vertex. Actually, the choice of the first vertex is not important as it can be modified, the crucial thing is to keep this definition during the whole process. Finally, the normal of polygon n_{poly} is obtained after normalization:

$$n_{poly} = \frac{\mathcal{A}}{\|\mathcal{A}\|} \qquad (2)$$

Fig. 5 shows a comparison of these two types of normals (by triangle and by polygon) and the definition of normal by polygon given by (2) gives a better smoothness of the shading.

4.2.3 A first naive tessellation algorithm

A first naive tessellation algorithm is to create all triangles from one vertex, for instance the first one. The number of created triangles is $d - 2$ where d is the number of vertices of the polygon. This algorithm is described in Algorithm 1. Fig. 6 shows an example of use of Algorithm 1 for a very simple 5-sides convex polygon.

Note that if the polygon is a triangle (3-sides), all the edges are set to visible. Now, let us consider another 5-sides polygon but which is concave. To do so, the second point of Fig. 6 is simply moved to become a concave point as shown in Fig. 7. Algorithm 1 is used,

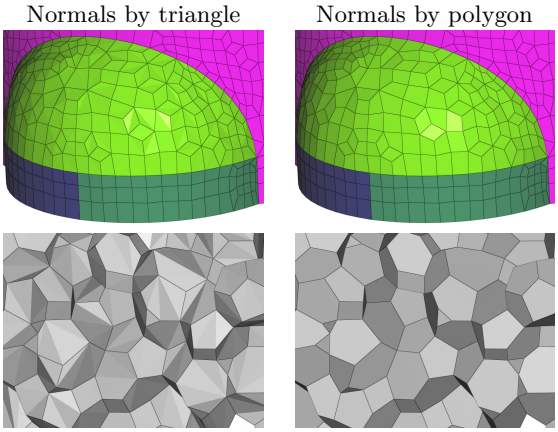

Figure 5: Comparison of normals by triangle (left) and by polygon (right) with a Phong model as shading.

Algorithm 1: A first naive tessellation algorithm

Input: d, `pol` (list of vertices of size d).
Output: `NmbTri` (number of triangles), `Tri` (list of indices of triangles), `VisEdg` (edges visibility)

```
(1)  NmbTri = d - 2 ;
     if NmbTri < 1 then
     |  return 0 ;
     for i = 1 to NmbTri do
(2)  |  Tri[i][1] = pol [1] ;
(3)  |  Tri[i][2] = pol [i + 1] ;
(4)  |  Tri[i][3] = pol [i + 2] ;
(5)  |  VisEdg [i][1] = 1 ;
(6)  |  VisEdg [i][2] = 0 ;
(7)  |  VisEdg [i][3] = 0 ;
     end
(8)  VisEdg [1][3] = 1 ;                    //- First triangle
(9)  VisEdg [NmbTri][2] = 1 ;               //- Last triangle
     //- If only 1 triangle, all edges are visible
     if NmbTri == 1 then
(10) |  VisEdg [0][3] = 1 ;
```

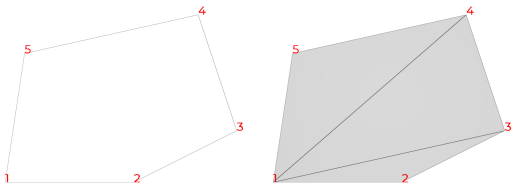

Figure 6: A 5-sides convex polygon. The naive tessellation algorithm 1 works.

and the same tessellation is constructed but this time it fails as the polygon is concaved. The tessellation does not span the polygon as the triangle {1, 2, 3} is outside the polygon.

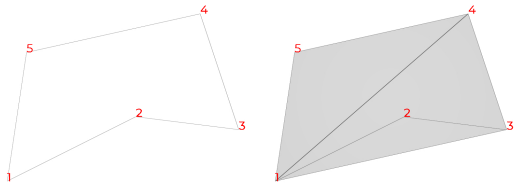

Figure 7: A 5-sides concave polygon. The naive tessellation algorithm 1 fails.

Fortunately, there is a very simple criterion to know if a triangle is outside the polygon. Once the tessellation has been created, it is sufficient to check if all dot products between the normal of the polygon, defined by (2), and the normals of triangles are positive. If all these dot products are positive, no triangle will lie outside the polygon and algorithm 1 is applied. For the first case (Fig. 6), this criterion is true while it is false for the second case (Fig. 7) as the dot product between the normal of the polygon and the normal of triangle {1, 2, 3} is negative. Thus, algorithm 1 can not be applied and a more general tessellation algorithm is needed.

4.2.4 Choosing a better a starting point in the naive tessellation algorithm

Algorithm 2 sums up the process to check if algorithm 1 should be used. If only one concave vertex has been found, the idea is to generate the tessellation from this point: Algorithm 1 is then used with this point as a starting point. An example is shown in Fig. 8 with the same concave polygon than for Fig. 7 but this time the tessellation is correct. Note that the criterion of dot products positiveness is also checked for every created triangles. Indeed, this new tessellation could be incorrect and in this case, a more general algorithm described in Section 4.2.5 should be used.

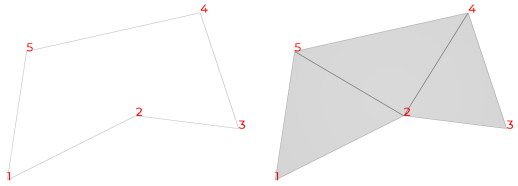

Figure 8: A 5-sides concave polygon. Following algorithm 2, a concave vertex (here vertex 2) has been used as a starting point to create the tessellation.

Algorithm 2: Check if the first naive tessellation algorithm 1 can be used

Input: d, `pol` (list of vertices of size d).
Compute the normal polygon n_{poly} with eq. (2) ;
Create the tessellation following algorithm 1 ;
`NmbVerConcave` $= 0$;
for $i = 1$ **to** `NmbTri` **do**
  Compute normal n_{tri} of triangle `Tri`$[i]$;
  if $n_{tri} \cdot n_{poly} < 0$ **then**
    Algorithm 1 can not be used. ;
    Update the list of concave vertices ;
    `NmbVerConcave` ++;
end
if `NmbVerConcave` $= 0$ **then**
  Tessellation created with algo. 1 is correct.
else if `NmbVerConcave` $= 1$ **then**
  Launch algorithm 1 with the concave point as a starting point to create new tessellation and check it with algorithm 2.;
else
  Too much concave points, another algorithm is needed (see algorithm 3). ;
end

4.2.5 A general tessellation algorithm

Triangulating a polygon, that is decomposing a polygon into a set of triangles is a problem that have been investigated for a long time. One of the most famous method is the ear clipping theorem [26, 27, 28, 29]. The principle is that in each polygon, an ear can be found and removed from the polygon. The result is a polygon whose area is smaller. By doing this recursively, a set of triangles is obtained and span all the polygon. Note that most of ear-clipping algorithms are for 2D (polygons in a plane) only whereas we are considering 3D meshes.

Algorithm 3: General tessellation algorithm

Input: d, `pol` (list of vertices of size d).
(1) Compute the normal polygon n_{poly} with eq. (2) ;
(2) Project all polygon vertices into an orthogonal plane of the normal polygon ;
(3) **while** $d > 3$ **do**
(4) Find a triangle which is admissible. ;
(5) Update all the lists: add this triangle to the tessellation, remove this triangle from the polygon list ;
end
(6) Generate the last triangle with the 3 last points. ;

Algorithm 3 gives the general steps to create a tessellation following the idea of the ear clipping theorem. All details of this algorithm are described now:

(1) Normals computations. The normal of the polygon is defined following eq. (2).

(2) Points projections. All vertices of the polygon are projected on a same plane that is orthogonal to the normal n_{poly} of the polygon. These projected points $\mathcal{P}_i$ are obtained following:

$$\mathcal{P}_i = P_i - (P_i, n_{poly})\, n_{poly} \tag{3}$$

where P_i denotes the vertex i of the polygon and $(.,.)$ is the usual dot product.

(3) Find an ear. Once a triangle has been found, the polygon changes, its size becomes smaller as one vertex is removed from the list of the polygon.

(4) Find an admissible triangle. To find a triangle that is correct, the idea is to take three consecutive vertices of the polygon and check if the dot product between the normal polygon and the normal of the triangle is positive and that no other projected point of the polygon lies inside this projected triangle. To do so, let's note $\mathcal{P}_{i_1}$, $\mathcal{P}_{i_2}$ and $\mathcal{P}_{i_3}$ the three consecutive projected points and $\mathcal{P}_j$ another projected point of the polygon, we define

$$\begin{aligned} u &= (\mathcal{P}_j\mathcal{P}_{i_1} \wedge \mathcal{P}_j\mathcal{P}_{i_2}) \cdot n_{poly} \\ v &= (\mathcal{P}_j\mathcal{P}_{i_2} \wedge \mathcal{P}_j\mathcal{P}_{i_3}) \cdot n_{poly} \\ w &= (\mathcal{P}_j\mathcal{P}_{i_3} \wedge \mathcal{P}_j\mathcal{P}_{i_1}) \cdot n_{poly} \end{aligned} \tag{4}$$

where $(. \wedge .)$ denotes the usual cross product. Then, if u, v and w are all positive, the point is inside the triangle. If none of the other projected points $\mathcal{P}_j$ lies inside the triangle $\{\mathcal{P}_{i_1}\ \mathcal{P}_{i_2}\ \mathcal{P}_{i_3}\}$, this triangle is admissible.

(5) Updates. All the lists must be updated. One more triangle is added in the tessellation, `Tri` is updated with these three vertices. The visibility of edges `VisEdg` is set by looking at the local index of the vertices. Indeed, if the two points of the edge are consecutive (or are the first and last vertices of the original polygon), `VisEdg` is set to 1, otherwise it is set to 0. Then, the vertex $\mathcal{P}_{i_2}$ is removed from the list `pol` of the polygon and d is decreased by one unit.

(6) Last triangle. Finally, a last triangle is created with the last three vertices. `Tri` and `VisEdg` are updated with the same way than (5).

Figure 9 shows an example of a 7-sides concave polygon handled with algorithm 3.

If the element is ill-defined, for instance not a simple polygon by with intersected edges, it is possible that neither algorithm 2 nor algorithm 3 work. In this case, algorithm 1 can still be applied to generate a tessellation in order to at least be able to see the polygon.

To sum up the tessellation process, algorithm 1 is first used. If the tessellation is correct according positiveness criterion, there is no need to use the other algorithms. If there is only one concave vertex, this point is

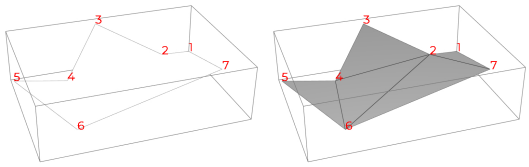

Figure 9: A 3-dimensional 7-sides concave polygon. With algorithm 3, even a polygon with two concave vertices is handled.

used as a starting point in algorithm 1 and we check if the tessellation is correct. Otherwise, the most general tessellation algorithm, algorithm 3 is launched to create the tessellation. All these tessellation algorithms are done in the CPU before the rendering of these triangles by the GPU.

4.3 Solution rendering with a tessellation of triangles

If a solution has been computed on polygons, a representation from a tessellation of triangles might be inaccurate. For instance, let's consider a mesh of quadrilaterals and a solution defined at vertices. Thus, the solution is Q^1-solution on a Q^1-quadrilaterals and is therefore bi-linear as shown in Fig. 10. If a tessellation of two triangles is generated with affine functions on them is done, an approximation is created and the representation in inaccurate (the tessellation in 2 triangles can be guessed in Fig. 10). Note that the rendering in both cases is pixel-exact and isolines are displayed to highlight the linearity or non-linearity of the solution plotted.

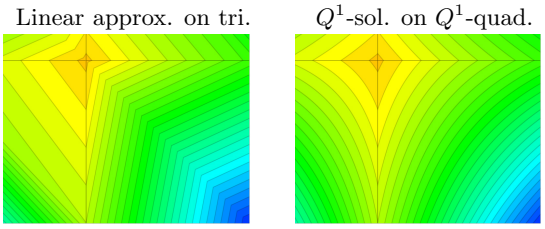

Figure 10: Rendering of Q^1-solution on Q^1-quadrilaterals (right) and tessellation into triangles with affine representation (left).

5. POST-PROCESSING TOOLS AND INTERACTIVITY

Many post-processing tools are available to make the analysis of results possible. Some of them are presented in this section. Such an interactivity is fundamental to develop and validate new algorithms.

5.1 Picking and hiding surfaces by reference

Any element can be picked to get information. When an element is picked, it is colored in light blue and the number of its vertices appear in red as shown in Fig. 11. More information is printed on the terminal, for example the element picked in Fig. 11:

```
Polygon (7-sides)      19793 : [   2863749
2863755    2863758    2863757    2863759
2863760    2863750]    Ref 3
```

The printed information: the number of vertices (sides), the index of element, the indices of all vertices and the reference number of the polygon.

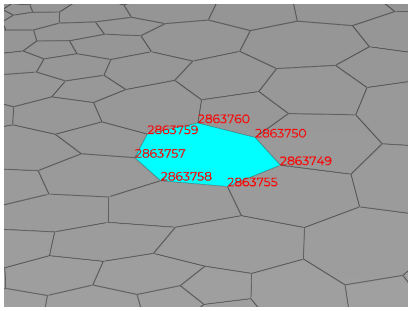

Figure 11: Picking a polygon (in lighe blue).

When a face of a volume element (here a polyhedron) is picked, all the faces of this volume elements are also set in light blue as shown in Fig. 12. In the same way that for polygons, information are printed on the terminal: the number of faces (inner polygons), the index of the polyhedron, the list of indices of these faces and the polyhedron reference. Then, for each face, the number of vertices and the list of vertices are printed. Here is an example for an hexahedron:

```
Polyhedron (6-faces)
952607 : [385682647 385683053 385685045
 385685051 385685052 385685053]    Ref 0
   Face (4-sides)          1 : [ 69594081
69594078   69581014   69581016]
   Face (4-sides)          2 : [ 69594081
69559331   69559333   69594078]
   Face (4-sides)          3 : [ 69559331
69594081   69581016   69559334]
   Face (4-sides)          4 : [ 69559333
69594078   69581014   69559341]
   Face (4-sides)          5 : [ 69559331
69559333   69559341   69559334]
   Face (4-sides)          6 : [ 69581016
69559334   69559341   69581014]
```

To inspect meshes, it is interesting to hide some elements. After an element is picked, it is possible to

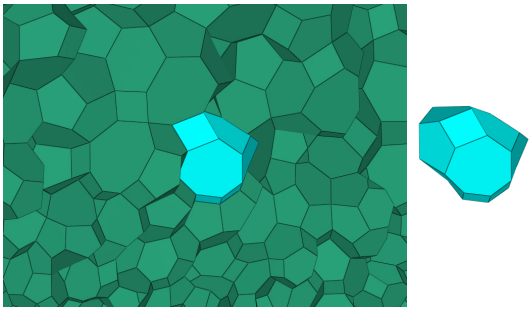

Figure 12: Picking a polyhedron (in light blue). Left: with all elements in the cut plane. Right: the polyhedron alone.

hide all elements having the same reference id (corresponding typically to a patch or a specific part of the object). An example is shown in Fig. 13 where the green surface (tire) is hidden to show the elements behind.

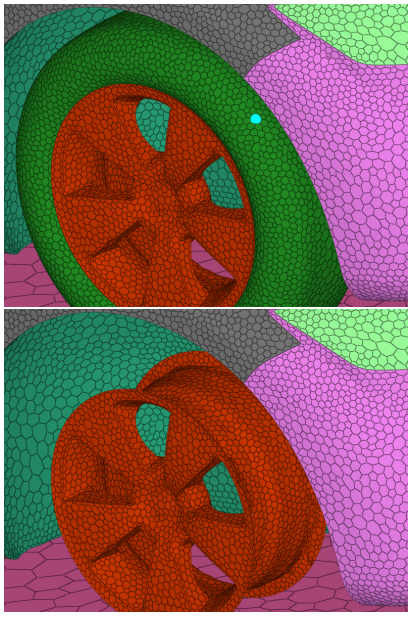

Figure 13: Example of picking (first picture) and hidding by reference (second picture).

In practice, here are the steps to set the whole element picked in light blue. A triangle has been picked. The index of the polygon or polyhedron is known (previously stored). All triangles belonging to the same element (polygon or polyhedron) are added to the picked list. During the creation of the texture that gives the rgba (red, green, blue and alpha) of the triangle to the shaders, if the triangle is in the picked list, its rgb is set to (0, 1, 1), that is light blue, instead of the color that should be displayed (for example its reference color or the usual signature grey). In the fragment shader, if the color texture is (0, 1, 1), that is light blue, we know that the triangle has been picked. If it is a polygon (i.e. not a volume element), `Fragcolor` is set to (0, 255, 255, 1), so that it will appear in light blue independently of the shading. Otherwise, the color is light blue but the shading can be seen as in Fig. 12.

5.2 Clip planes

To visualize polyhedra, clip planes are used. The clip plane can be defined by its equation. Otherwise, the clip plane can be translated or rotated with the mouse from an initial state. Then, all polyhedra belonging to this cut plane are displayed. In practice, the faces of these volume elements are rendered. To find if a polyhedron is intersected by the cut plane, one just have to look at the sign of all the element vertices in the cut plane equation $ax+by+cz+d$, where x, y and z are the coordinates of the vertex and a, b, c and d the parameters of the cut plane equation [20, 21]. If some of them are positive and some others are negative, the volume element lies in the cut plane. Fig. 14 shows an example of clip plane.

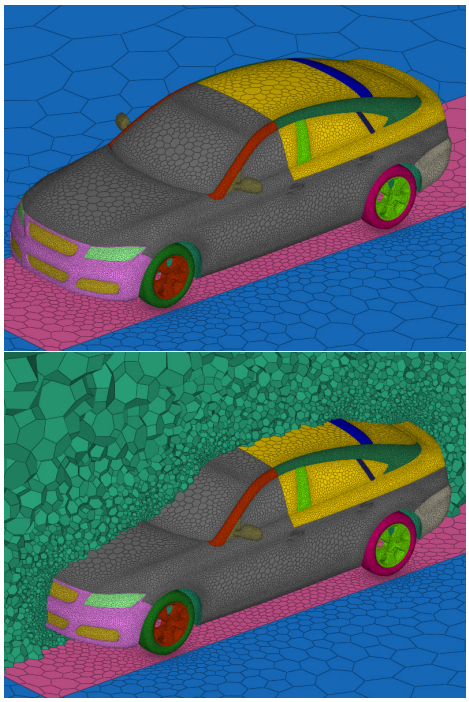

Figure 14: Examples of rendering without (first picture) and with (second picture) clip plane.

6. EXAMPLES AND COMPARISONS

6.1 Comparisons with other visualization software

Some comparisons are made with `ParaView` and `VisIt`. Several meshes of different sizes are studied where the geometry is the car plotted in Fig. 14. The `VTK Unstructured Grid` (`vtu`) format is used in `ParaView` and in `VisIt`. Meshes were converted from `CGNS` (CFD General Notation System) to `vtu` format. The versions 5.7 of `ParaView` and 3.2.1 of `VisIt` are used. For `ViZiR 4`, meshes were converted from `CGNS` to `libMeshb` format. Table 3 compares the total rendering time that is the time to open the mesh file, add objects to the scene and render the mesh. Three cases, similar than Fig. 14, are taken into account. The number of boundary polygons are 439 170, 1 101 804 and 2 649 542 and the number of triangles created by the tessellation are respectively 1 705 918, 4 333 706 and 10 505 154 in `ViZiR 4`. It gives an average of 3.88, 3.93 and 3.96 triangles per polygons and show that the number of vertices is truly arbitrary with an average of 6 for each boundary polygon. The ratio are huge and is explained by the fact that the time to open the mesh file in `ParaView` and in `VisIt` is long and because a surface reconstruction is done in `ParaView` and in `VisIt` and this step is very expensive. In `ViZiR 4`, the surface reconstruction is not done (even if it could be called) as this information (boundary polygons) is already in the mesh file. Otherwise, the surface reconstruction can be done in a pre-processing step, and then should be saved in the mesh. Note that this step can still be done in the visualization software but is useless and time-consuming if it has already be stored in the mesh file.

	Case 1	Case 2	Case 3
# vertices	9 600 780	24 551 880	61 321 116
# polygons	439 170	1 101 804	2 649 542
# polyhedra	2 652 618	6 603 843	15 055 285
`ViZiR 4` (s)	1.93	4.48	10.98
`ParaView` (s)	81.7	204.0	505.8
Ratio / `ParaView`	42.3	45.5	46.1
`VisIt` (s)	86.9	219.3	582.8
Ratio / `VisIt`	45.0	48.9	53.1

Table 3: Comparison of total rendering wall time (s) including mesh files opening.

Table 4 compares the time to generate cut planes. In `ParaView`, these cut planes are crinkle clips. `VisIt` was not able to generate crinkle clips with these meshes. Again, the ratio are huge and mainly due to the slow surface reconstruction done in `ParaView`.

Fig. 15 shows a comparison of rendering obtained from `ParaView` and `ViZiR 4`. It is clear that the shading in `ParaView` is done by triangles while it is done by polygon in `ViZiR 4` as explained in Section 4.2.2 and gives a better smoothness. Note that with `VisIt`, instead of the polygons, the triangles (i.e the tessellation) are displayed.

	Case 1	Case 2	Case 3
# vertices	9 600 780	24 551 880	61 321 116
# polygons	439 170	1 101 804	2 649 542
# polyhedra	2 652 618	6 603 843	15 055 285
`ViZiR 4` (s)	0.6	1.4	3.1
`ParaView` (s)	57.9	147.0	357.9
Ratio	98.1	106.5	114.0

Table 4: Comparison of wall time (s) to generate cut planes (clip).

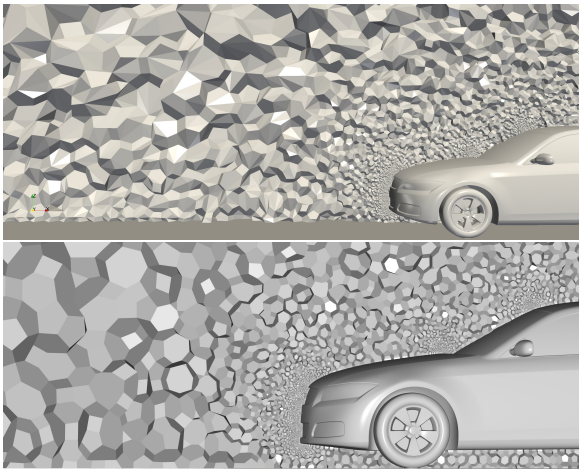

Figure 15: Comparison of rendering obtained from `ParaView` (first picture) and `ViZiR 4` (second picture). In the first case, shading is done by triangle while it is done by polygon in the second case.

Additional metrics, mesh file size, memory and video memory used, and the number of frames per second have been sum up in Table 5 for the 3 same cases than the previous tables. For `ViZiR 4`, the format `meshb` (binary) from `libMeshb` is used while the `vtu` format is used in `ParaView`. The mesh size in `cgns` format has also been added. Both software programs have good FPS for all cases and therefore interactive enough. `ParaView` needs much more memory especially during the preparation of the rendering however requires much less video memory.

Finally, last comparisons have been done with another CPU-GPU combo: Windows 11-Nvidia. The laptop is a 6-core i7 3 Ghz with 32 GB of RAM and the graphic card a Nvidia quadro T1000 (4Gb). Results are very

	Case 1	Case 2	Case 3
Size .meshb (us)	753 M	1.9 G	4.6 G
.vtu `ParaView`	1.8 G	4.5 G	11 G
.cgns `ParaView`	1.0 G	2.6 G	6.4 G
FPS (us)	60	60	52
FPS `ParaView`	58	57	46
RAM (us)	1.59 G	3.91 G	9.47 G
RAM `ParaView`	2.14 G	4.98 G	12.03 G
Peak RAM `ParaView`	2.95 G	7.46 G	17.59 G
VRAM (us)	214 M	453 M	1.01 G
VRAM `ParaView`	182 M	250 M	435 M

Table 5: Comparisons of additional metrics: mesh sizes, FPS (frames per second), memory RAM and VRAM (video RAM) for `ParaView` and `ViZiR 4`.

similar than Table 3 which is not surprising as the two laptops have similar features. For the first case, `ViZiR 4` has a total rendering wall time of 1.68 s and 1.37 Gb RAM while `ParaView` needs 50 s (ratio 30) and 2.4 Gb of RAM. For the second case, `ViZiR 4` has a total rendering wall time of 4.92 s and 4.67 Gb RAM while `ParaView` needs 2 min 17 (ratio 32) and 5.3 Gb of RAM. In all cases, both programs have very good frame rates.

6.2 Examples

Dual meshes are implicit supports for many CFD solvers, for instance the ones based on finite volume methods. Here we illustrate with Fig. 16, 17 and 18 some rendering to study the differences between adaptive meshing techniques. Examples of Fig. 18 come from the same airplane geometry than Fig. 17 with a zoom on the wings. Classic mesh adaptation techniques are based on a sequence of local mesh modifications. These techniques consist in an advancing-point methods using metric fields. We can see that the dual patterns are different.

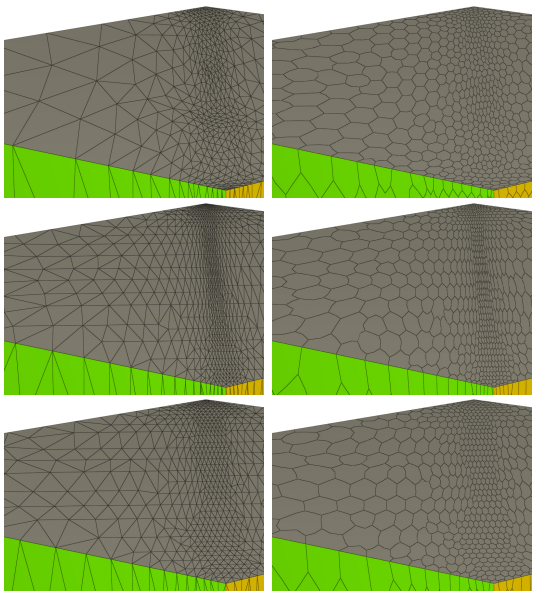

Figure 16: Primal (left) and dual (right) meshes for standard adaptation (first line), and two metric-aligned techniques (second and third lines).

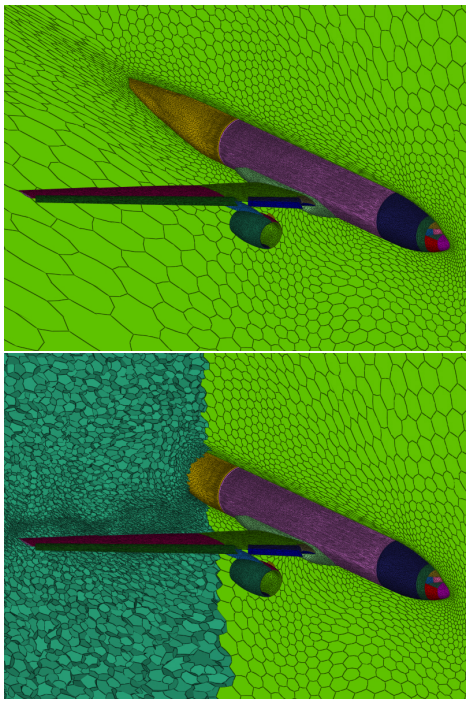

Figure 17: An example of polyhedral mesh without (first picture) and with (second picture) clip plane.

7. CONCLUSIONS

In this paper, we presented how OpenGL 4 can be used to visualize polygonal and polyhedral meshes. In particular, we discussed how the storage in the mesh file is done. We showed that the use of triangles in the OpenGL graphic pipeline is the most efficient as it is the simplest. For this reason, polygons are tessellated into triangles. Algorithms and criteria are given to create a good tessellation. Many examples show the efficiency of our method and comparisons with `ParaView` and `VisIt` show that `ViZiR 4` is much faster.

A first perspective of this work could be to use polygons when capping is done. Indeed, when cut planes are generated, two modes can be used: cut plane (or

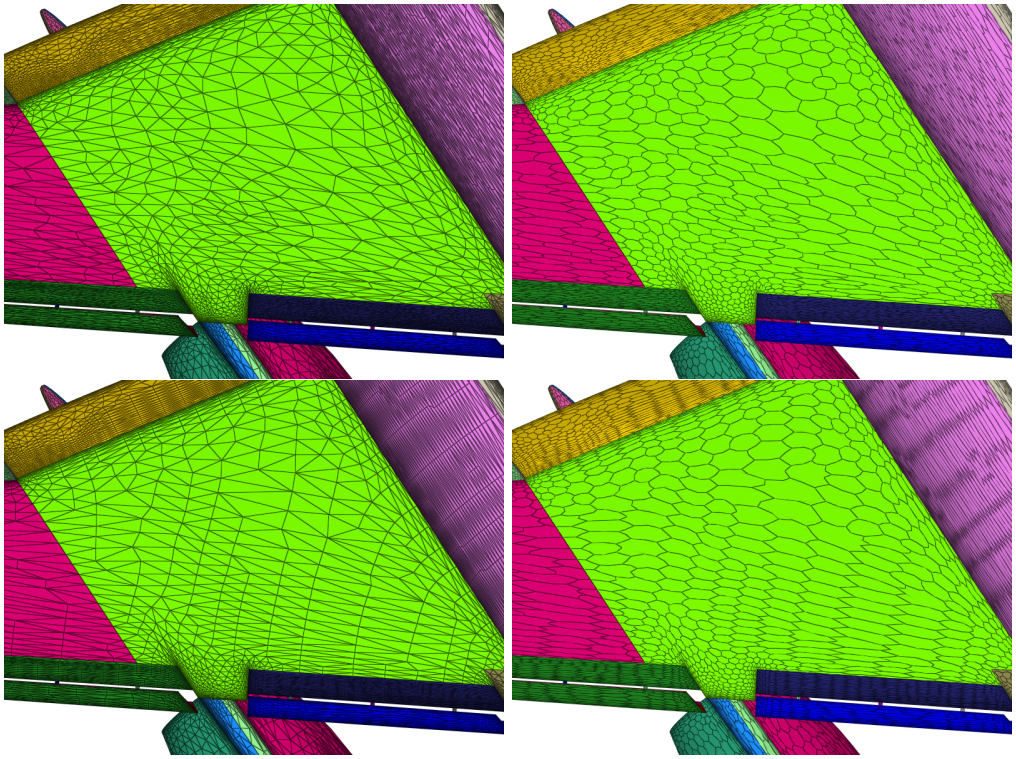

Figure 18: Primal (left) and dual (right) meshes for standard adaptation (top) and metric-aligned adaptation (bottom).

crinkle) when the faces of the volume elements are shown or capping (or slice), when the intersection of the volume element with the plane is computed. In the latter case, when non simplicial elements such as prisms, pyramids or hexahedra are considered, the intersection is in fact a polygon. At the moment, triangles are used to display the capped elements as shown in Fig. 19 where the mesh is composed of quadrilaterals and hexahedra. Another perspective would be to handle solutions on polygonal and polyhedral meshes. Finally, as OpenGL 4 is able to handle high-order elements, one can imagine that high-order polygons, when and if it will exist, could also be visualized with OpenGL 4.

8. ACKNOWLEDGMENTS

This work was supported by the public grant ANR Impacts, reference ANR-18-CE46-0003. The authors are also grateful to Loïc Maréchal (Inria) for providing the `libMeshb` library and his help in testing, Lucien Rochery (Inria) for fruitful discussions on tessellation algorithms and Siemens for providing meshes from `Simcenter StarCCM+`.

References

[1] Frey P.J., George P.L. *Mesh generation: application to finite elements*. Iste, 2007

[2] "Simcenter STAR-CCM+." `https://www.plm.automation.siemens.com/global/en/products/simcenter/STAR-CCM.html`

[3] "OpenFOAM." `https://www.openfoam.com`

[4] Oaks W., Paoletti S. "Polyhedral mesh generation." *Proceedings of the 9th International Meshing Roundtable*, pp. 57–67. 2000

[5] Paoletti S. "Polyhedral mesh optimization using the interpolation tensor." *Proceedings of the 11th International Meshing Roundtable*, pp. 19–28. 2002

[6] Garimella R.V., Kim J., Berndt M. "Polyhedral mesh generation and optimization for non-manifold domains." *Proceedings of the 22nd International Meshing Roundtable*, pp. 313–330. Springer, 2014

[7] Wachspress E.L., EL W. "A rational finite element basis." 1975

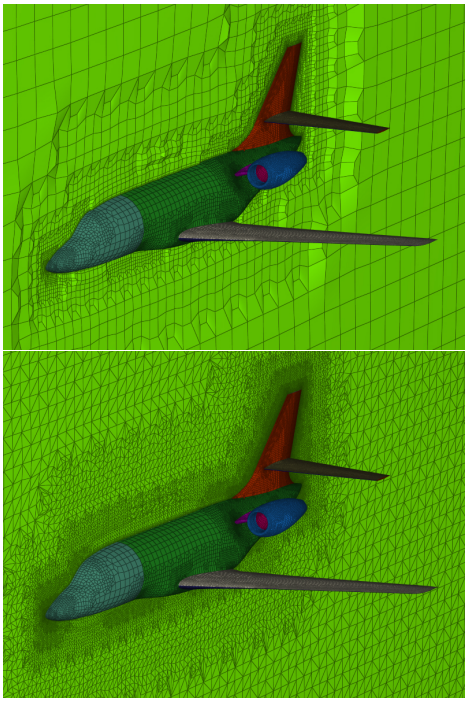

Figure 19: Example of clip plane (first picture) and capping (second picture) of a hexahedral mesh.

[8] Sukumar N. "Construction of polygonal interpolants: a maximum entropy approach." *International journal for numerical methods in engineering*, vol. 61, no. 12, 2159–2181, 2004

[9] Sukumar N., Malsch E. "Recent advances in the construction of polygonal finite element interpolants." *Archives of Computational Methods in Engineering*, vol. 13, no. 1, 129, 2006

[10] Cangiani A., Georgoulis E.H., Houston P. "hp-version discontinuous Galerkin methods on polygonal and polyhedral meshes." *Mathematical Models and Methods in Applied Sciences*, vol. 24, no. 10, 2009–2041, 2014

[11] Manzini G., Russo A., Sukumar N. "New perspectives on polygonal and polyhedral finite element methods." *Mathematical Models and Methods in Applied Sciences*, vol. 24, no. 08, 1665–1699, 2014

[12] Perumal L. "A brief review on polygonal/polyhedral finite element methods." *Mathematical Problems in Engineering*, vol. 2018, 2018

[13] Muigg P., Hadwiger M., Doleisch H., Hauser H. "Scalable hybrid unstructured and structured grid raycasting." *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, 1592–1599, 2007

[14] Muigg P., Hadwiger M., Doleisch H., Groller E. "Interactive volume visualization of general polyhedral grids." *IEEE transactions on visualization and computer graphics*, vol. 17, no. 12, 2115–2124, 2011

[15] KitWare Inc. "ParaView." `https://www.paraview.org/`

[16] Childs H., Brugger E., Whitlock B., Meredith J., Ahern S., Pugmire D., Biagas K., Miller M., Harrison C., Weber G.H., Krishnan H., Fogal T., Sanderson A., Garth C., Bethel E.W., Camp D., Rübel O., Durant M., Favre J.M., Navrátil P. "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data." *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372. Oct 2012

[17] Musy M., Dalmasso G., Sullivan B. "marcomusy/vtkplotter: vtkplotter.", 2019

[18] Sullivan B., Kaszynski A. "PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)." *Journal of Open Source Software*, vol. 4, no. 37, 1450, 2019

[19] Canepa A., Infante G., Hitschfeld N., Lobos C. "Camarón: An Open-source Visualization Tool for the Quality Inspection of Polygonal and Polyhedral Meshes." *International Conference on Computer Graphics Theory and Applications*, vol. 2, pp. 130–137. SCITEPRESS, 2016

[20] Loseille A., Feuillet R. "Vizir: High-order mesh and solution visualization using OpenGL 4.0 graphic pipeline." *56th AIAA Aerospace Sciences Meeting, AIAA Scitech*, 2018

[21] Feuillet R., Maunoury M., Loseille A. "On pixel-exact rendering for high-order mesh and solution." *Journal of Computational Physics*, vol. 424, 109860, 2021

[22] Bracci M., Tarini M., Pietroni N., Livesu M., Cignoni P. "HexaLab. net: An online viewer for hexahedral meshes." *Computer-Aided Design*, vol. 110, 24–36, 2019

[23] Wolff D. *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, 2011

[24] Sellers G., Wright R., Haemel N. *OpenGL SuperBible, Sixth Edition*. Addison-Wiley, 2013

[25] Phong B.T. "Illumination for computer generated pictures." *Communications of the ACM*, vol. 18, no. 6, 311–317, 1975

[26] Meisters G.H. “Polygons have ears.” *The American Mathematical Monthly*, vol. 82, no. 6, 648–651, 1975

[27] Tarjan R.E., Van Wyk C.J. “An O(n log log n)-time algorithm for triangulating a simple polygon.” *SIAM Journal on Computing*, vol. 17, no. 1, 143–178, 1988

[28] Chazelle B. “Triangulating a simple polygon in linear time.” *Discrete & Computational Geometry*, vol. 6, no. 3, 485–524, 1991

[29] ElGindy H., Everett H., Toussaint G. “Slicing an ear using prune-and-search.” *Pattern Recognition Letters*, vol. 14, no. 9, 719–722, 1993