Vizir: High-order mesh and solution visualization using OpenGL 4.0 graphic pipeline

Adrien Loseille, Rémi Feuillet

Vizir: High-order mesh and solution visualization using OpenGL 4.0 graphic pipeline

Adrien Loseille[1)], Rémi Feuillet[2)]
GAMMA3 Team, INRIA Saclay Ile-de-France, Palaiseau, France

OpenGL 4 with GLSL shading language have become a standard on many common architectures (Mac, Linux, Windows, , ...) from a couple of years. In the mean time, high-order methods (for flow solution and for meshing algorithm) are emerging. Many of them have proven their abilities to provide accurate results on complex (3D) geometries. However, the assessment of a particular meshing algorithm or of a high-order numerical scheme strongly relies on the capacity to validate and inspect visually the current mesh/solution at hand. However, having at the same time, an accurate and interactive visualization process for high-order mesh/solution is still a challenge as complex process are usually involved in the graphic pipeline: non linear root finding, ray tracing, GPU programming, ... In this paper, we discuss the current status and issues of using the (raw) OpenGL 4 pipeline to render curved high-order entities, and almost pixel-exact solutions. We illustrate this process on meshes and solutions issued from high-order curved from CAD and with high-order interpolated solutions.

I. Introduction

When developing meshing algorithms for surface, volume or adapted meshes, the capacity of quickly inspect, query and interactively observe a mesh in real-time is usually of great benefit to improve, correct and validate visually a new algorithm. This is used on a daily basis by programers at each step of the development process.

The same observation holds with the development of numerical schemes in the flow solver. This can be used to inspect wrong shaped finite-volume cell, track negative pressure, invalid Jacobians . . .
Note that this kind of visualization is different from complex rendering techniques that tend to provide new insight in the solution or improve the understanding of a computation. We do no intend to compete with visualization techniques and enhacement provided by Ensight, Paraview, tecPlot, . . .

High-order visualization techniques are more and more used. In the beginning it was used for surface smoothing in computer graphics (see [1]) but it arrived quickly in scientific computing (see [2, 3]) as high-order numerical schemes and meshes are becoming successful. In this paper, we focus on the development of an interactive mesh and solution visualization component in the case of high-order meshes thanks to OpenGL 4.0 framework.

The paper is organized as follows. In Section 2, we review the main issues arising when dealing with direct high-order mesh and solution visualizations. We then described in Section 3, the graphic pipeline use to display surface entities, volume entities are quickly reviews in Section 5. In Section 6, we detail the issue of pixel exact rendering when curved elements implies a strong linearity. Finally, in Section 7, we describe the Vizir-API.

[1)]Researcher, adrien.loseille@inria.fr
[2)]Ph.D. Student, remi.feuillet@inria.fr

II. Issues with high-order solutions rendering and proposed solution

A standard procedure to visualize high-order (curved) meshes is to subdivide the initial mesh into multiple simple primitives as linear triangles (see [4]). More precisley, it adapts the mesh to eye-sensitive features on every element. Nonetheless, this solution has several drawbacks:

- The memory footprint may be huge as the subdivided mesh is done in the RAM, especially in the case the old display-list setting
- The rendering of the solution may be poor as it is linearly interpolated by element.
- The interactivity may be limited

Another procedure is to use ray tracing techniques for elements rendering (see [5, 6]). But in this case, the solution rendering has the following drawback : given a pixel, there is no easy way to find the intersection of the ray with the surfaces. Actually, this is a non linear problem. Moreover, ray-tracing techniques are really greedy in computations.

To overcome these issues, OpenGL 4.0 graphic pipeline (see [7]) offers a possibility to render curved high-order entities, and almost pixel-exact solution rendering.

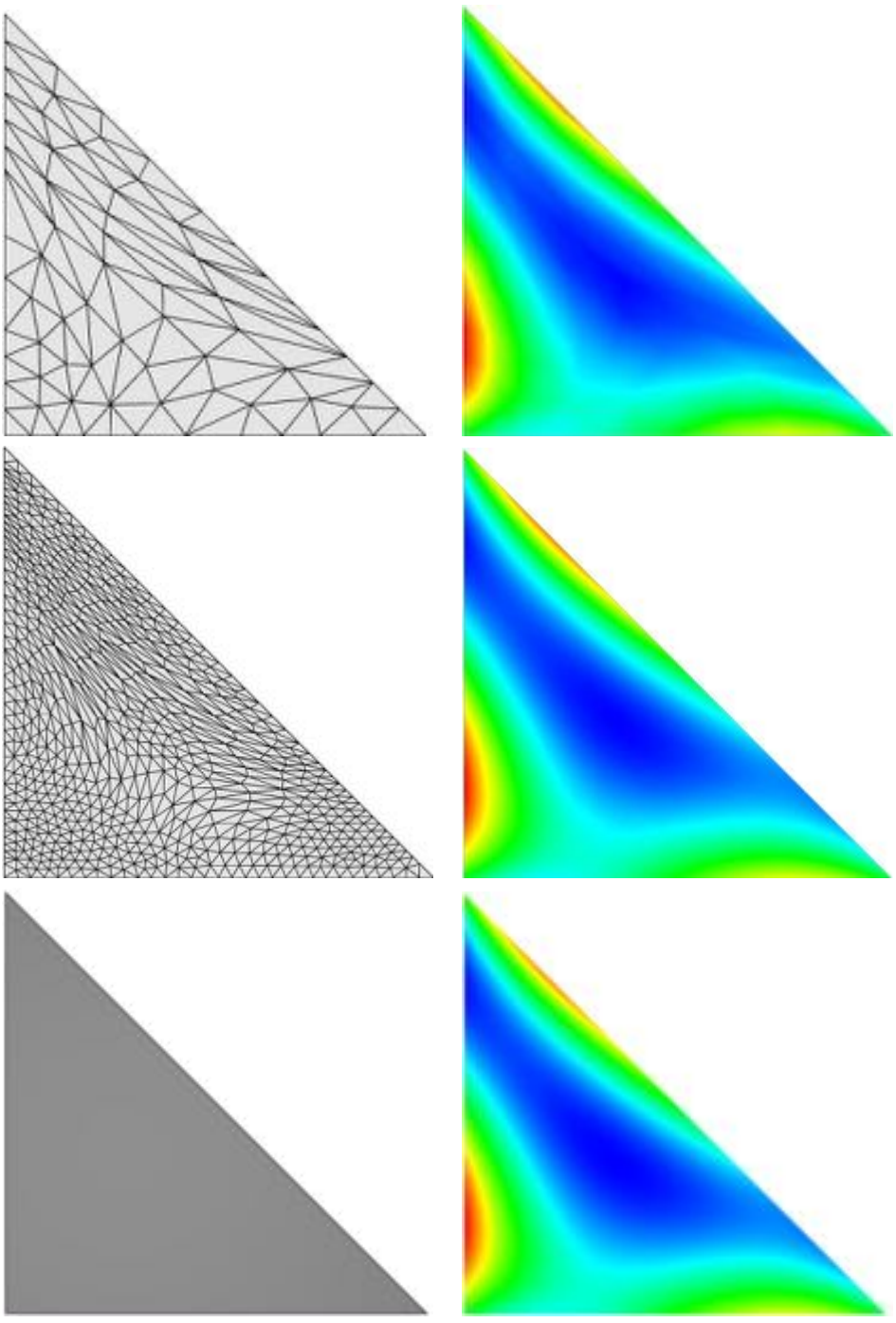

Fig. 1 2 subdivisions of a triangle and their associated $\mathbb{P}_3$ solutions. In the third line, computed solution on a single triangle with OpenGL 4.0 and the pixel exact approach.

III. On the visualization of curved surface elements

The visualization process relies on the use of the OpenGL 4.0 graphic pipeline.
Shaders replace parts of the OpenGL pipeline. More precisely, they are parts of the programmable pipeline. The main pros of redefining all the shader stages are:

- the memory footprint in RAM is limited to the size of the mesh
- Addition (subdivided) entities are created on the graphic cards directly on the fly
- Solutions are computed in the fragment shader so that new shapes functions and interpolation schemes can be interchanged.

The OpenGL 4.0 pipeline can be customised with up to five different shader stages (see Fig. 2). Also, among built-in variables, we can pass through our own variables which means that for a pixel, we then know (x, y, z), (u, v), or primitive ids. For th storage of raw data (like high-order solutions), texture are used.
More precisely, the description of the shaders is the following:

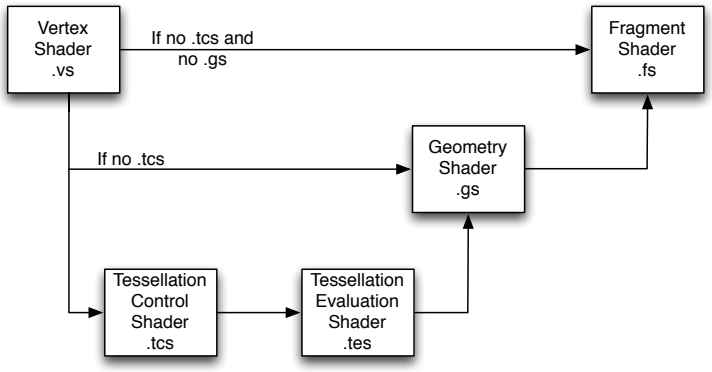

Fig. 2 Shaders used for the OpenGL graphic pipeline.

- The **Vertex Shader** (VS). Its input variables are vertex attributes. This shader can also send other information down the pipeline using shader output variables. For instance, the vertex shader can compute its color and give it to the pipeline. The data corresponding to the vertex position must be transformed into clip coordinates and assigned to the output variable `gl_Position`. The vertex shader is executed (possibly in parallel) once for each vertex.
- The **Fragment Shader** (FS). Its input variables come from the graphic pipeline and is a transformation of other shader's outputs. The fragment shader determines the appropriate color for the pixel (see Fig 3) and sends it to the frame buffer using output variables. The fragment shader is executed (possibly in parallel) once for each fragment (pixel) of the polygonal object being rendered.

These two shaders can be enough to define a customization of the graphic pipeline. In this case, between the two shaders, the vertices are assembled into primitives, clipping takes place and the viewport transformation is applied. Also, data provided from the fragment shader is by default linearly interpolated and provided to the fragment shader.
However, the OpenGL pipeline can be even more customized thanks to the following shaders:

- The **Geometry Shader** (GS). This shader cannot be used without the two previous shaders and is executed once for each primitive. It has access to all the input data of the vertices of the primitive that can be provided either by the vertex shader or by the tessellation evaluation shader. As a consequence all variables are arrays. The GS can receive some primitives and can emit other primitives as long as only one type of primitive is in input or output. For instance, it can receive triangles and output lines. The GS functionality is centered around two primitives: `EmitVertex` and `EndPrimitive`. To each vertex of the primitive, all useful data are linked with it and then the vertex is emitted thanks to the first primitive. Once all vertices of the primitive are processed, the second primitive is called. In our case, the GS is used to display the edge of a primitive, following the idea of NVIDIA Whitepaper WP-03014-001_v01.
- The **Tessellation Control Shader** (TES) and the **Tessellation Evaluation Shader** (TCS). These shaders cannot be used without the three previous shaders. As soon as tessellation shaders are used, the only used primitives are patches. A patch primitive is a part of the geometry defined by the programmer. The number of vertices

by patch is configurable as well as the interpretation of the geometry. For instance, the patch can be used as a set of control points that defined an interpolated curve or surface (Bézier curve for instance). More precisely, the tessellation control shader sets up the tessellation primitive generator (TPG) by defining how the primitives should be generated by it. For example, it describes how a geometry should be divided into sub-entities (e.g. tessellated, see Fig. 4). Then, the role of the tessellation evaluation shader is to determine the position of the vertices of the patch primitive. The TCS is executed once for each vertex in the output patch and it can compute additional information and give it to the TCS. Also, it tells the TPG how many primitives it should produce e.g. the granularity of the tessellation. The TPG computes in the parameter space, the uniform discretization of parameterized entities like quads, triangles and isolines. Once the discretization is done, the entities are sent to the TES. The TES is executed once for each parameter space vertex. For a given vertex, it determines the position of the vertices in the physical space thanks to the coordinates of the parameter space and also other vertex-related data. All the data is then sent to the GS that process it like any other entity. In our case, these shaders are used to render approximate high-order curved elements such as $\mathbb{P}_k$-triangles (inspired by [1]) and $\mathbb{Q}_k$-quads.

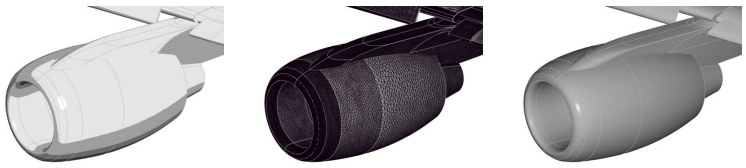

Fig. 3 Example of three different displays on an engine of a Dassault Falcon.

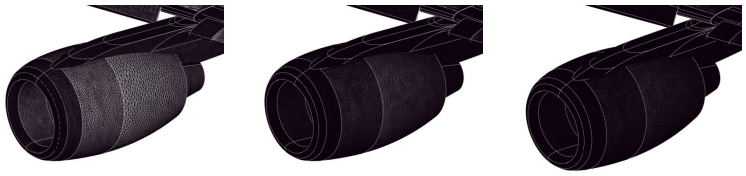

Fig. 4 Example of three different tessellation levels on an engine of a Dassault Falcon.

With Vizir, the rendering shape of the high-order curved element is performed with the tessellation shaders thanks to the Bézier representation of high-order elements. Note that this could be done on the GPU, but each unit has a limit amount of available memory As example, let us see the case of the edges:

- $\mathbb{P}_2$: In input, the user gives the coordinates of the three Lagrange points: M_{20}, M_{11}, M_{02}.
  The three Bézier control points are then deduced:

$$\begin{cases} P_{20} = M_{20} \\ P_{11} = \dfrac{4M_{11} - M_{20} - M_{02}}{2} \\ P_{02} = M_{02} \end{cases}$$

  The position of a point on the curve can be then deduced via the following formula:

$$P(u,v) = u^2 P_{20} + 2uvP_{11} + v^2 P_{02} \quad \text{with} \quad v = 1 - u \quad \text{and} \quad u \in [0,1]$$

- $\mathbb{P}_3$: In input, the user gives the coordinates of the four Lagrange points: M_{30}, M_{21}, M_{12}, M_{03}.

The four Bézier control points are then deduced:

$$\begin{cases} P_{30} = M_{30} \\ P_{21} = \dfrac{18M_{21} - 9M_{12} - 5M_{30} + 2M_{03}}{6} \\ P_{12} = \dfrac{18M_{12} - 9M_{21} + 2M_{03} - 5M_{30}}{6} \\ P_{03} = M_{03} \end{cases}$$

The position of a point on the curve can be then deduced via the following formula:

$$P(u,v) = u^3 P_{30} + 3u^2 v P_{21} + 3uv^2 P_{12} + v^3 P_{03} \quad \text{with} \quad v = 1 - u \quad \text{and} \quad u \in [0,1]$$

The same type of formula stands for the triangle and the quadrilateral and can be done at any degree.
With Vizir, the control points are precomputed on CPU and are then sent to the shader program. The TES computes then the position of any point of the curve for a given u (and v for triangles and quads) that are the coordinates of the parameters space.
An example of a Tesselation Evaluation Shader is given here:

```
//--- barycentric (with shrink)
float us3 =  1.0/3.0;
float u   = us3 + shrink*(gl_TessCoord.x - us3);
float v   = us3 + shrink*(gl_TessCoord.y - us3);
float w   = 1.0 - u - v;

//--- control points
vec3 P200 = gl_in[0].gl_Position.xyz;
vec3 P020 = gl_in[1].gl_Position.xyz;
vec3 P002 = gl_in[2].gl_Position.xyz;
vec3 P110 = gl_in[3].gl_Position.xyz;
vec3 P011 = gl_in[4].gl_Position.xyz;
vec3 P101 = gl_in[5].gl_Position.xyz;

//--- Bernstein and extension
float B200 = u*u;
float B020 = v*v;
float B002 = w*w;   //(1-u-v)*(1-u-v)
float B110 = 2*u*v;
float B011 = 2*v*w; // 2*v*(1-u-v)
float B101 = 2*u*w; // 2*u*(1-u-v)

vec3 pos = B200*P200 + B020*P020  + B002*P002   + B110*P110  + B011*P011 + B101*P101;

gl_Position =  MVP * vec4(pos,1.0);
```

Thanks to these visualization features, Vizir is able to determine if a CAD model and/or a mesh of degree 3 or less is well-made or not. In figure 6, we show $\mathbb{P}_2$ mesh of a shuttle CAD model where an anisotropic mesh controlling anisotropically the third order of the surface is rendered.

Also, Vizir is able to perform the folowing things:

- The level of tessellation is determined in the TCS and can be controlled via error estimates. For instance, no tessellation is required when the element is straight.
- The distance of a point to an edge of an element is computed in the Geometry Shader for wireframe rendering, it is not an edge on a top of a triangle (see Fig. 7)
- There is anti-aliasing as there is no Z-buffer fighting for the edges of a same triangle.

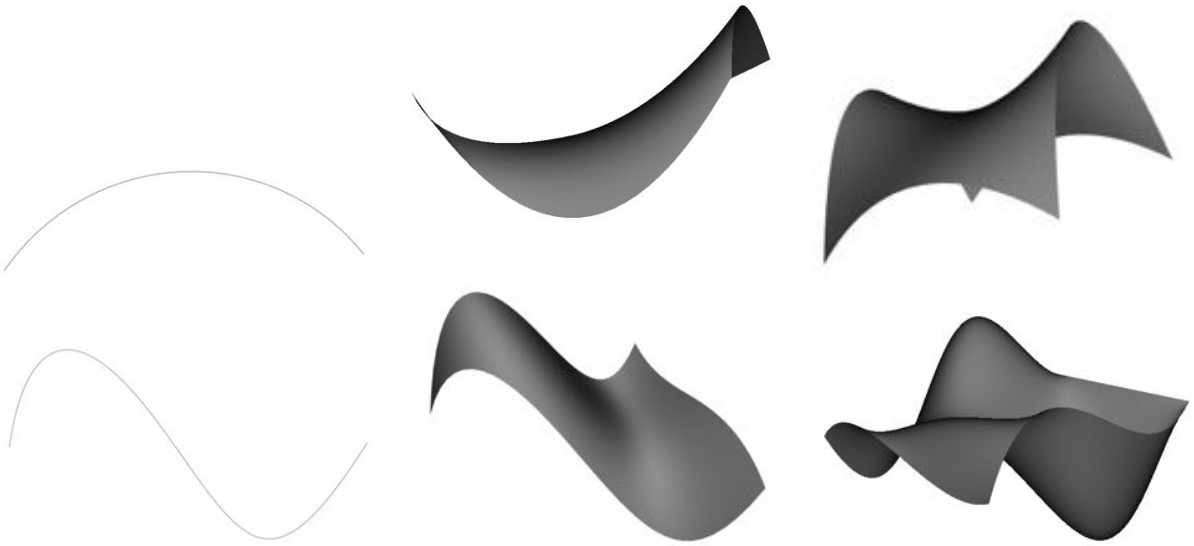

Fig. 5 From top to bottom: Elements of degree 2 and degree 3 displayed with Vizir. From left to right: Type of the elements : edge, triangle and quadrilateral.

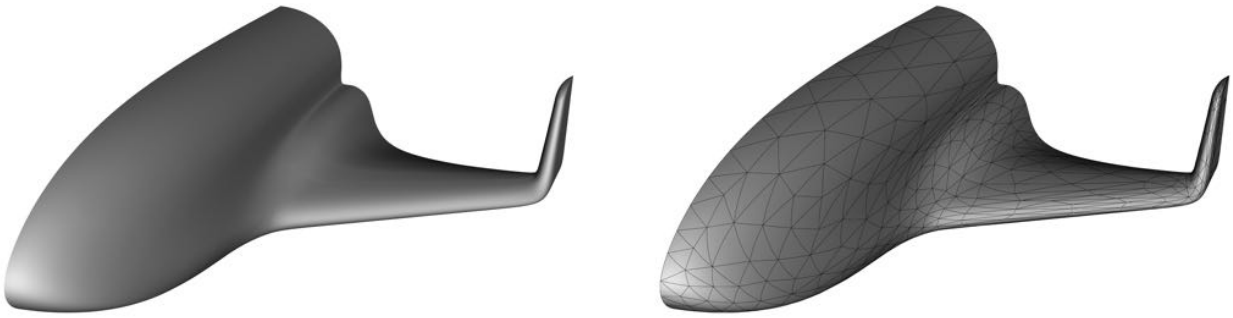

Fig. 6 Illsutration of an anisotropic $\mathbb{P}_2$ surface mesh approximating a shuttle NURBS with 2nd order elements.

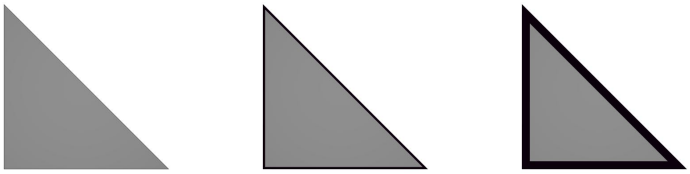

Fig. 7 Examples of wire-rendering on a triangle.

IV. Rendering of volume entities

The volume entities are displayed thanks to their faces or their intersections with planes. In the first case, it is the display of the volume elements thanks to a cut plane, and the features of the previous section can be reused. In the second case, it is mandatory to compute the intersection of (possibly) high-order elements with a plane. In this case, the intersection can lead to new shapes, like ellipsoids, pentagons, ... To overcome these issues, OpenGL 4.0 graphic pipeline (see [7]) offers a possibility to render curved high-order entities, and almost pixel-exact solution rendering. Currently, we only consider rendering in cut planes with a linear approximation. Some Illustrations of rendering are provided in Figure 8.

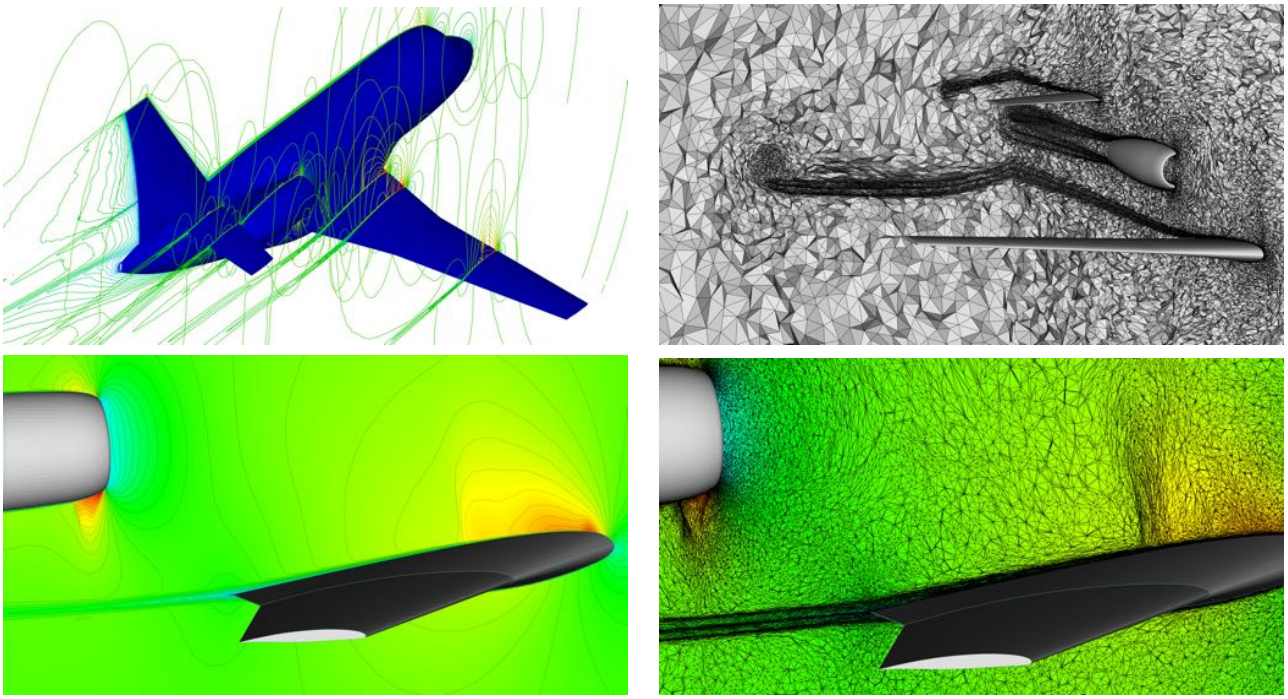

Fig. 8 Example of volume rendering of an anisotropic adaptive solution of a RANS computation around a Falcon geometry.

V. Almost pixel-exact solution rendering

One of the other features offered by Vizir is the almost pixel-exact solution display. This feature is also based on the use of the shaders with the textures. An example showing the benefit of exact solution rendering is given in Figure 9

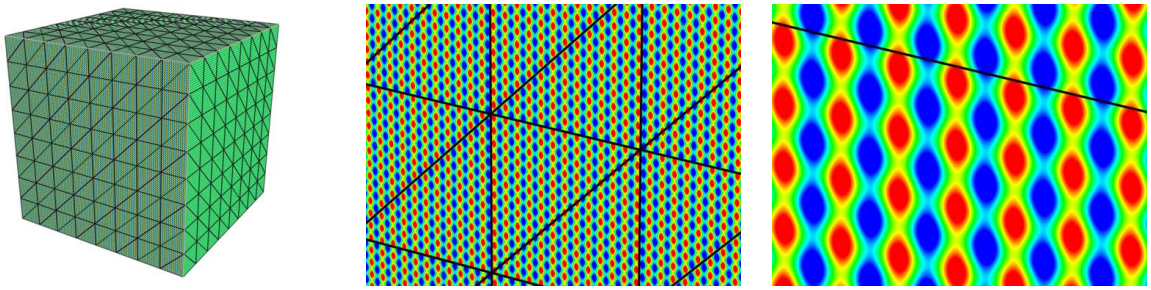

Fig. 9 Illustration of pixel-exact rendering of function $\sin(100\,\pi x) + \sin(100\,\pi y) + \sin(100\,\pi z)$ **on a simple mesh cube (at no cost for the CPU). From left to right, different level of zooms.**

where an highly oscillating function is displayed. (Re)Computing this solution in the fragment shader is at not cost while trying to subdivide a mesh to reach the necessary level of accuracy will lead to quickly intractable mesh size, although the initial geometry mesh are very simple.
The display of the solutions can be done via two methods. First case, the solution is given at the vertices, the solution must have the same degree as the one of the elements of the mesh, also known as iso-parameterization. Second case, the solution is given at the elements. In this case, the order of the solution can be different from the order of the element.
Before explaining how the solution rendering works with Vizir, let us have a clear definition of what a $\mathbb{P}_k$ solution is:
A $\mathbb{P}_k$ solution on an element is a solution that can be written as a polynomial function in the **reference element**, in other words, as a polynomial function of the parameters defining the element.
The major consequence is that a $\mathbb{P}_k$ solution is **not necessarily a polynomial function** of the coordinates of the physical space.
As a $\mathbb{P}_k$ solution is defined by its value at the Lagrange points, it is always possible to rewrite it into a *Bézier form* with control solutions. The process is exactly the same as in section III. In the shaders, the parameterization is either computed in the tessellation shaders for high-order elements or easily found in the geometry shader for straight elements. In the same way as the coordinates, computations of control solutions are made on CPU and then sent to the shader via textures. These textures are evaluated in the vertex shader which determines the color of the pixel thanks to the Bézier representation of the solution. An example of the solution evaluation in the fragment shader is given here:

```
float solp2(float u, float v)
{
 int idx    = gl_PrimitiveID*6;
 float p200 = texelFetch(tex, idx     ).x;
 float p020 = texelFetch(tex, idx + 1 ).x;
 float p002 = texelFetch(tex, idx + 2 ).x;
 float p110 = texelFetch(tex, idx + 3 ).x;
 float p011 = texelFetch(tex, idx + 4 ).x;
 float p101 = texelFetch(tex, idx + 5 ).x;

 return  u*(p200*u  + 2.0*p110*v) + p020*v*v +  (1.-u-v)*( 2.0*(p101*u + p011*v) + p002*(1.-u-v) );
}
```

The main advantage of this method is that it is pixel-exact for a solution of any order on a straight element, because the solution becomes a polynomial function of the coordinates of the physical space as the mapping with the reference element is linear (see Fig. 1 at the bottom and Fig 10).
However, the problem becomes non-linear for curved elements as the mapping is not linear anymore. The reverse mapping is only known at the points on which the parameters space is sampled e.g. the tessellation points. Consequently, the solution rendering at a point of the physical space is not pixel-exact anymore apart from tessellation points. The reverse mapping is approximated by a linear function between between two consecutive tessellation point. This approximation means that the solution rendering is a polynomial function of the physical space at the order of the solution between two consecutive tessellation points.
For instance, let us consider the following straight $\mathbb{P}_2$ element defined by 6 points so that there is a non linear mapping with the parameter space(see Fig 11).
The mapping can be written as:

$$\begin{cases} x = 0.8v^2 + 0.2v \\ y = 1 - u - 0.2v - 0.8uv - 0.8v^2 \end{cases}$$

Consequently, the reverse mapping can be computed and is:

$$\begin{cases} u = -\dfrac{2(x + y - 1)}{\sqrt{0.04 + 3.2x} + 1.8} \\ v = \dfrac{-0.2 + \sqrt{0.04 + 3.2x}}{1.6} \end{cases}$$

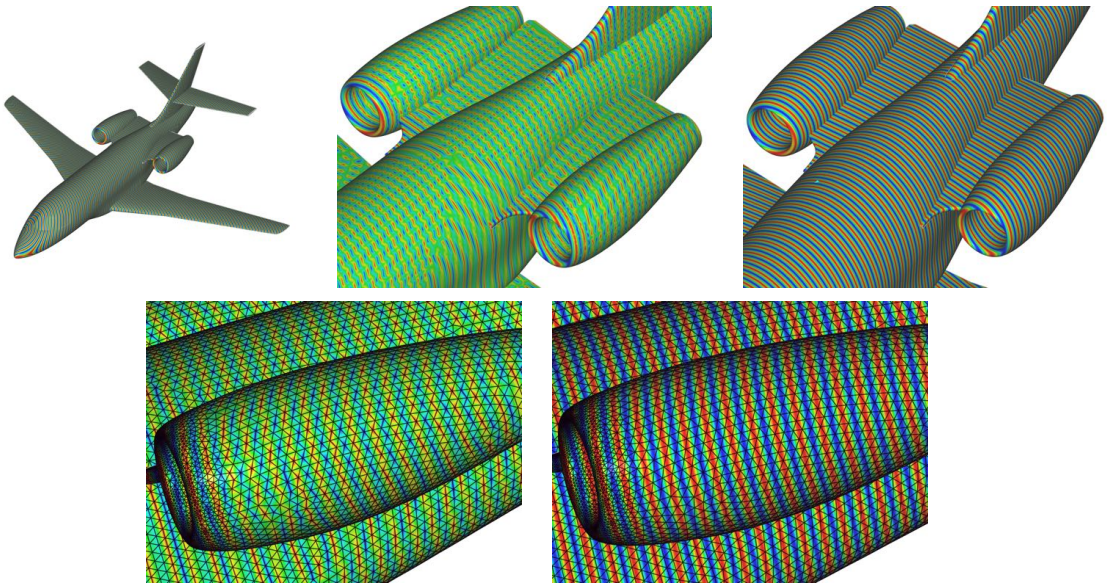

Fig. 10 Example of three solution rendering: $\mathbb{P}_1$ rendering (in the middle of the first line, and on the left of the second line) and $\mathbb{P}_3$ rendering (other figures) of the same analytical solution on a Dassault Falcon $\mathbb{P}_1$ mesh.

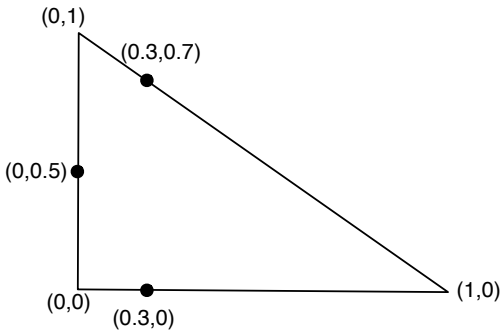

Fig. 11 Triangle defined by a non-linear mapping

Let us impose the following analytical $\mathbb{P}_2$ solution in order to compare it with the displayed one at two different tessellation levels:

$$P(u,v,w) = 0.1\times v(2v-1)+0.5\times w*(1-2w)+1\times 4uv-1\times 4vw+\frac{1}{3}\times 4uw \quad \text{with} \quad w = 1-u-v \quad \text{and} \quad (u,v)\in[0,1]^2$$

On the previous triangle, if the result is analytical imposed, e.g. P is analytically expressed with x and y in the FS, the rendered solution is in Fig. 12.

And if we use the expression of P with u and v that are approximated for two different tessellation levels, the rendered solution is in Fig 13.

The main remark that can be done, is that without any tessellation, the rendered solution is the one that would be if the mapping was linear with the parameter space. Moreover, it can be noticed that for a tessellation level that is not really deep, the solution rendering is already almost pixel exact.

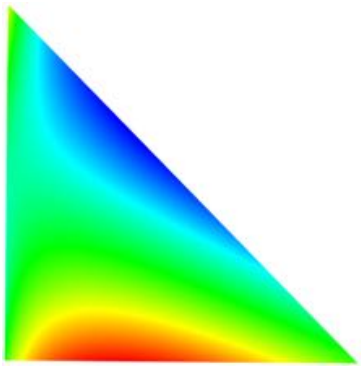

Fig. 12 Pixel exact analytical solution

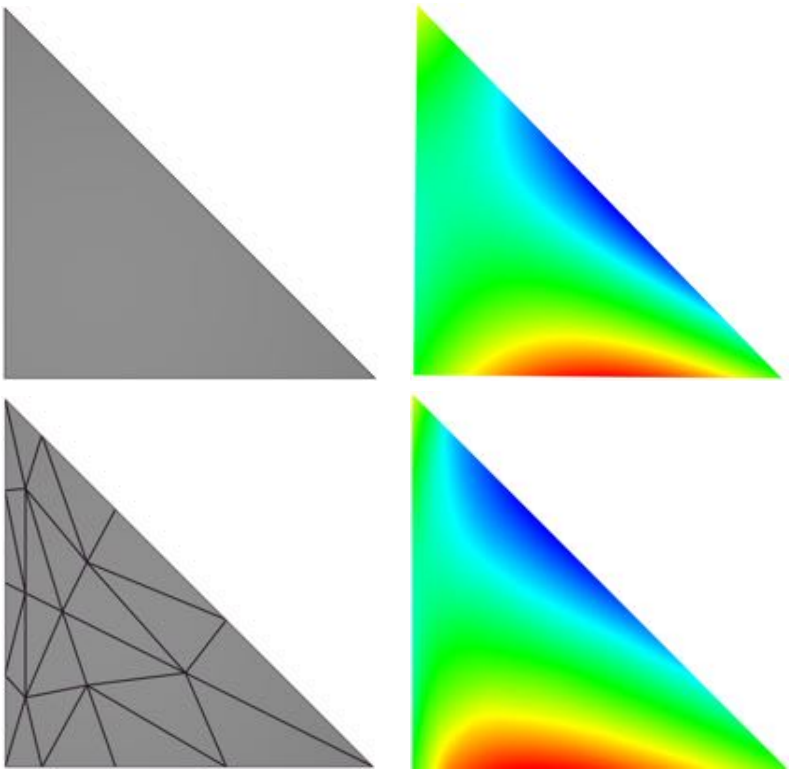

Fig. 13 Approximated solutions with a non linear mapping (see Fig. 12) for two different tessellation levels: If the barycentric coordinates are exactly interpolated along with the solution, the extact solution may be displayed at the wrong (x, y, z). Increasing the tessellation level will reduce this misfit.

VI. Technical details

If Vizir is bundled as a stand alone package, it can be used as an external library to display high-order entities on the fly. Vizir is based on the Qt (≥ 5) component for the GUI and for creating the graphic window. Note that is is easy also to use an other GUI system. The main advantage of Qt is that there is no include of specific OpenGL headers and the use of explicit OpenGL context help to stay away from using deprecated or old pipeline strategies.
Vizir reads mesh and solution files in the GMF format (.mesh or .meshb) provided by the libMeshb (see https://github.com/LoicMarechal/libMeshb for more details). We now describe the simple steps to display a full mesh and solution. First you have to create the required OpenGL format:

```
QSurfaceFormat format;
format.setVersion(4,0);
format.setProfile(QSurfaceFormat::CoreProfile);
format.setDepthBufferSize(24);
format.setSamples(16);
```

Then declare a `VizSceneWindow` for display and wait for OpenGL initialization. It is necessary to "show" the window to trigger the openGL initialization.

```
VizSceneWindow window(format);
window.resize(800, 600);
window.show();
window.waitOpenGLinit();
```

If the initialization is not completed correctly, you will get the following error message when trying to perform openGL calls: *warning : openGL context is not active (recall setGLfunc after initialization of the graphics window)*. Then to display a triangle, define the array of coordinates and array of indices :

```
double  crd[3][3]     = {{0,0,0},{1,0,0},{0,1,0}};
unsigned int index[3] = {1,2,3};
```

We then add a drawable object (derived from `VizDrawObject` class) to the scene and transfer the previous arrays to the graphics card:

```
VizDrawTriangle vizTri;
window.addObject(&vizTri);
window.attachData(&vizTri, 3, crd, 1, index);
```

To force the rendering, call:

```
window.renderNow();
```

To display a $\mathbb{P}_2$ triangle with a solution

```
double  crdP2[6][3] = {{0,   0,   0},{1,     0,   0},{0,   1,    0},
                       {0.5, 0, 0.2},{0.5, 0.5, 0.2},{0, 0.5, 0.2}};
int     p2tri[6]    = {1,2,3,4,5,6};
VizDrawTriangleP2 vizP2Tri;
window.addObject(&vizP2Tri);
window.attachData(&vizP2Tri, 6, crdP2, 1, p2tri);
VizDrawSolution vizP2TriSol;
double   solP2[6]    = {0., 0.1, 0.5,1.,-1.,-1./3.};
window.addObject(&vizP2TriSol);
window.attachData(&vizP2TriSol, &vizP2Tri, pal, solP2, 1, 6);
```

The same approach holds for all kind of element and solution. Note that is possible to display as many entities as desired in one call. There exist also functions allowing to quickly display a mesh.
Also, the OpenGL 4 version of Vizir is way more faster than the old version based on OpenGL legacy. For instance, for the display of a $\mathbb{P}_2$ mesh of 2 399 941 vertices, 1 683 292 tetrahedra and 153 476 triangles with a solution on it. The old version that is generating the tessellation on CPU does not launch as it is out of memory with more than 1 GB required, whereas the new version using OpenGL 4 launches in less than 1s and uses 45 MB of memory.

VII. Conclusion

In this paper, we have described a practical approach to visualize high-order meshes and solutions interactively. This approach is fully developer-oriented. This means we intend to display what is really manipulated in flow solvers or mesh generation software. In particular, we do not offer smoothing techniques of geometries and solutions, and we do not implement very sophisticated filters usually used to magnify a numerical solution or a mesh.

The whole process is based on the OpenGL 4 Core profile, and have been proven to work on standard operating systems (Mac, Linux and Windows) with no particular effort during the compilation. Pixel-exact solution rendering is obtained for straight elements while an inner tessellation is used to approximate strongly non linear mapping arising with curved elements. For every case, the color computed in the fragment shader uses the real interpolation schemes. There is neither a linear interpolation involved nor the need to subdivide the mesh to approximate the solution. The high flexibility of the GLSL shading language allows to quickly implement new interpolation schemes. Currently, continuous and discontinuous interpolation schemes from $\mathbb{P}_0$ (respectively $\mathbb{Q}_0$) to $\mathbb{P}_5$ (respectively $\mathbb{Q}_5$) are implemented. For the rendering of high-order curved elements, the memory footprint is also minimal as the tessellation is generated directly on the graphic card. In comparison with CPU based techniques [4], there is no duplicated subdivided meshes/solutions in the main memory. In comparison with techniques based on ray-tracing [5, 6], we do not have to solve a non-linear problems. If the root finding problem for the ray tracing is naturally avoided, we still have to handle the correction of the estimated space coordinates. Instead of solving a non linear problem (that would be done in the fragment shader), we prefer the simple solution that consists in approximating the mapping with the tessellation shader.

Current efforts are directed at improving the rendering of volume entities and mesh-related features as Riemannian metric field, surface curvatures or frame fields. For the visualization of solutions, accurate iso-lines and iso-surfaces (real-time) rendering remains an important issue.

VIII. Acknowledgment

This work was supported by a french public grant as part of the *Investissement d'Avenir* project, reference ANR-11-LABX-0056-LMH, LabEx LMH.

References

[1] Vlachos, A., Jörg, P., Boyd, C., and Mitchell, J. L., "Curved PN Triangles," *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, 2001, pp. 159–166. doi:10.1145/364338.364387, URL `http://doi.acm.org/10.1145/364338.364387`.

[2] Haasdonk, B., Ohlberger, M., Rumpf, M., Schmidt, A., and Siebert, K. G., "Multiresolution Visualization of Higher Order Adaptive Finite Element Simulations," *Computing*, Vol. 70, No. 3, 2003, pp. 181–204. doi:10.1007/s00607-003-1476-2, URL `https://doi.org/10.1007/s00607-003-1476-2`.

[3] Ellis, D., Karman, S., Novobilski, A., and Haimes, R., "3D Visualization and Manipulation of Geometry and Surface Meshes," *44th AIAA Aerospace Sciences Meeting and Exhibit*, American Institute of Aeronautics and Astronautics, 2006. doi:10.2514/6.2006-944.

[4] Remacle, J. F., Chevaugeon, N., Marchandise, E., and Geuzaine, C., "Efficient visualization of high-order finite elements," *International Journal for Numerical Methods in Engineering*, Vol. 69, No. 4, 2007, pp. 750–771. doi:10.1002/nme.1787, URL `http://dx.doi.org/10.1002/nme.1787`.

[5] Peiro, J., Moxey, D., Jordi, B., Sherwin, S. J., Nelson, B. W., Kirby, R. M., and Haimes, R., "High-Order Visualization with ElVis," *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, Springer International Publishing, 2015, pp. 521–534. doi:10.1007/978-3-319-12886-3_24.

[6] Nelson, B., Haimes, R., and Kirby, R. M., "GPU-Based Interactive Cut-Surface Extraction From High-Order Finite Element Fields," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 17, No. 12, 2011, pp. 1803–1811. doi:10.1109/tvcg.2011.206, URL `https://doi.org/10.1109%2Ftvcg.2011.206`.

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