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

Adrien Loseille¹ and Rémi Feuillet²

GAMMA3 Team, INRIA Saclay-Ile-de-France, Palaiseau, France

AIAA SciTech Forum, January 2018
Motivations

- High-order CFD methods: SUPG, DG, ...
- High-order adaptivity: curved surface mesh
- More $p > 1$, infinite class of basis functions/solutions representations
- OpenGL 4 is standard on Linux, Mac, Windows

(final) Intent

- A visualization solution built to help for the development of meshing algorithms.
- A convenient way to check mesh and/or CFD solution validity.
- A mandatory tool for future research in high-order schemes and meshes.
- No intention to compete with advanced visualization software like Ensight, Paraview, tecPlot, FieldView, ...
Outline

1. Issues with standard high-order rendering
2. Curved surface elements visualization
3. Rendering of volume entities
4. Almost pixel exact solution rendering
5. Technical details about Vizir
Issues with standard high-order solution rendering

Fixed-pipeline/Legacy pipeline

- Build displayList and VBOs that are send to the GPU
- All entities/arrays are allocated on the CPU (subdivision, interpolation)
- Going to be deprecated . . .
- Mainly linear interpolation : solution rendering
- Memory footprint

Previous works

- Ray tracing techniques for pixel-exact solution rendering [Peiro et al., *High-Order Visualization with ElVis*, 2015]
  - The resolution of two nonlinear problems (root finding problem and reverse mapping problem).
  - Greedy computations.
- Adaptive subdivision [Remacle et al., IJNME, 2006]
  - Still end-up with linear interpolation . . .
Issues with standard high-order solution rendering

Classical rendering techniques based on subdivision are on the left and the proposed solution with Vizir is on the right.
Curved surface elements visualization

- The visualization process relies on the use of the OpenGL 4.0 graphic pipeline.
- The OpenGL 4.0 pipeline can be customized with up to five different shader stages.

![Diagram of the OpenGL 4.0 pipeline]

- Among built-in variables, we can pass through our own variables:
  - for a pixel we then know: \((x, y, z)\), or \((u, v)\), or primitive ids.
- Textures are used to store raw data (HO-Solutions)
Curved surface elements visualization

The important shaders are:
- The Tessellation Control Shader is used to create the subdivision of the parameter space (tessellation) of the HO element.
Curved surface elements visualization

The important shaders are :

- The Tessellation Evaluation Shader then computes the wanted position for each element of the subdivision.

![Diagram of shader pipeline]

- Vertex Shader .vs
- If no .tcs and no .gs
- Tessellation Control Shader .tcs
- Tessellation Evaluation Shader .tes
- Geometry Shader .gs
- Fragment Shader .fs
Curved surface elements visualization

The important shaders are:
- The Geometry Shader emits the vertex of geometry defined either by the Vertex Shader or the TES.
Curved surface elements visualization

The important shaders are:

- The Fragment Shader computes the color for each element of the geometry generated by the GS or the VS.

```
Vertex Shader .vs

| If no .tcs and |
| no .gs |

Geometric Shader .gs

| If no .tcs |

Tessellation Control Shader .tcs

Tessellation Evaluation Shader .tes

Fragment Shader .fs
```
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.

Anti-aliasing (no Z-buffer fighting for the edges of a same triangle)

It is not an edge on top of a triangle
The TES is using the Bézier representation of the elements to plot it (here an edge in $\mathbb{P}_2$).

**Input:** Three Lagrange points: $M_{20}, M_{11}, M_{02}$. 

Three Bézier control points are deduced:

\[
\begin{align*}
P_{20} &= M_{20} \\
P_{11} &= \frac{4M_{11} - M_{20} - M_{02}}{2} \\
P_{02} &= M_{02}
\end{align*}
\]

**Output:** The position of each point of the patch defined by a $u$ and a $v$.

\[
P(u, v) = u^2 P_{20} + 2uv P_{11} + v^2 P_{02} \quad \text{with} \quad v = 1 - u \quad \text{and} \quad u \in [0, 1]
\]
Curved surface elements visualization

Example of a TES used for a $\mathbb{P}_2$ triangle.

```c
//--- 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);
```
Curved surface elements visualization

- Elements of degree 2 (up) and degree 3 (bottom) displayed with Vizir.
- From left to right: edge, triangle and quadrilateral.
Illustration of an anisotropic $\mathbb{P}_2$ surface mesh approximating a shuttle NURBS with 2nd order (curved) elements.
Rendering of volume entities

Volume entities are rendered as surface

- pre-treatment
- Volume rendering is be done via cut planes and plane clipping.
- Example of volume rendering of an anisotropic adaptive solution of a RANS computation around a Falcon geometry.
Almost pixel exact solution rendering

Rendering of analytical function

- The fragment has the real $x, y, z$ pass through the vertex array and **linearly** interpolated
- The palette is a uniform array (and can be updated on the fly)
- Color lookup in the palette according to the solution

**Exact-rendering of** $\sin(100 \pi x) + \sin(100 \pi y) + \sin(100 \pi z)$
Almost pixel exact solution rendering

Rendering a numerical solution

- Depending on Basis functions: recompute on the fly the solution
- Real coordinates \((x, y, z)\), barycentric coordinates \((u, v, w)\)
- Solution is stored in texture

Example of a frag shader with 2nd order Bezier solution

```c
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 + (1.0-u-v)*(2.0*(p101*u + p011*v) + p002*(1.0-u-v));
}
```
Almost pixel exact solution rendering

- When the element is straight with a linear mapping, the solution is pixel exact.
- The value of the solution at the high-order nodes is sent to the GPU via a texture buffer.
- The solution is computed pixel by pixel thanks to user defined function.
- Adaptation on the fly of the palette to the variations of the solution.
- Example of a $\mathbb{P}_3$ solution on a straight triangle.
Almost pixel exact solution rendering

- Example of 3 different solutions rendering on a Dassault Falcon $\mathbb{P}_1$ mesh with $\mathbb{P}_1$ and $\mathbb{P}_3$ solution.
Almost pixel exact solution rendering

- Example of 3 different solutions rendering on a Dassault Falcon $\mathbb{P}_1$ mesh with $\mathbb{P}_1$ and $\mathbb{P}_3$ solution.
Example of 3 different solutions rendering on a Dassault Falcon $\mathbb{P}_1$ mesh with $\mathbb{P}_1$ and $\mathbb{P}_3$ solution.
Almost pixel exact solution rendering

- Issues with elements defined by a non-linear mapping.
- Pixel-exact \((u, v, w)\) but wrong \((x, y, z)\), solution is not at the right place
- Example with a non linear triangle with a mapping whose reverse mapping is analytically known and a \(P_2\) solution on it.
Approximated solutions with a non linear mapping for two different tessellation levels.
Technical details about Vizir

In a nutshell

- Vizir is a stand alone package or a library.
- C++ (C, Fortran bindings)
- It can be used as an external library to display high-order entities on the fly (reading directly data-structures) for co-visualization.
- Vizir is based on the Qt (≥ 5) component for the GUI and for creating the graphic window.

Memory footprint: 2 399 914 ver., 1683292 tets and 153 476 tris

- Display of a $\mathbb{P}_2$ solution
- The old version using openGL legacy and generating the tessellation on CPU does not launch as it is out of memory more than 1GB.
- The new version using openGL 4.0 launches in less than 1s and uses a memory of 45 MB.
Technical details about Vizir

Initialization of the library packages with Qt.

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

// Create the window : show and wait for openGL initialization
VizSceneWindow window(format);
window.resize(800, 600);
window.show();
window.waitOpenGLinit();
```

Display of a $P_2$ triangle with a $P_2$ solution on it.

```cpp
double crdP2[6][3] = {{0,0,0}, {1,0,0}, {0,1,0}, {0.3,0,0.2}, {0.3,0.7,0.1}, {0,0.5,0.3}};
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);
```
Technical details about Vizir

Native mesh format

- INRIA .meshb: binary, threads safe, parallel IO, hybrid, curved to P4/Q4, any order for solution
- CAD bindings via EGADS lite, B. Haimes
- We do not impose an ordering for HO elements, ordering pass through the list of parametric coordinates, . . .
- Long Long int, big/little endian
- Native format for the UGAWG working group
- see libmeshb, libhom on github
  https://github.com/LoicMarechal/libMeshb

Native solution format

- Basis functions have to be implemented in the user-fragment shaders
- No ordering required (up to user) and no limit on the order of the solution
A practical approach for high-order meshes and solutions interactively

A library for developers, see what you have

- Based on OpenGL 4 profile only.
- No subdivision On CPU
  Only the mesh/solution stored or shared
- Reduction of memory footprint.

Future work

- High-order volume rendering.
- Iso-surfaces/Iso-lines rendering.
- Improvement of tessellation algorithms (de Casteljau’s algorithm).
- Coupling with GPU features: clipping, ...