Bachelor Project

June 9, 2018

Barycentric Data Visualization for Triangle Meshes

Costanza Volpini

Abstract

The usual way to visualize data for a triangle mesh is to associate the data with the vertices and then use linear interpolation over the mesh triangles. While this is the obvious way to go for data given at the mesh vertices, it is less natural for data given at the edges or triangles, since it requires to first aggregate the data neighbouring each vertex, thus introducing an additional averaging step. In this project we want to explore alternative data visualization techniques, using the power of barycentric coordinates and GPU programming.

Advisor Prof.Dr. Kai Hormann
Assistant Jan Svoboda

Advisor's approval (Prof.Dr. Kai Hormann): Date:

Contents

1 Introduction

1.1 Barycentric coordinates

Barycentric coordinates, discovered by Möbius in 1827, represent one of the most progessive area of research in computer graphics and mathematics thanks to the numerous applications in image and geometry processing. [10] The position of any point in a triangle can be expressed using a linear combination of barycentric coordinates:

$$p = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3$$

where p_1, p_2 and p_3 are the vertices of a triangle and λ_1, λ_2 and λ_3 (the barycentric coordinates) are three scalars that respect the following barycentric coordinates properties.[7]

- partition of unity: $\sum_{i=1}^{3} \lambda_i(p) = 1$
- reproduction: $\sum_{i=1}^{3} \lambda_i(p) p_i = p$
- Lagrange-property: $\lambda_i(p_j) = \delta_{i,j}$
- linearity: $\lambda_i \in \prod_1$
- non-negativity: $\lambda_i(p) \geq 0$ for $p \in [p_1, p_2, p_3]$

A point is inside the triangle if and only if $0 \leq \lambda_1, \lambda_2, \lambda_3 \leq 1$. If a barycentric coordinate is less than zero or greater than one, the point is outside the triangle. Barycentric coordinates allow the interpolation of values, from a set of control points over the interior of a domain, using weighted combinations of values associated with the control points (Fig. 1). [10]

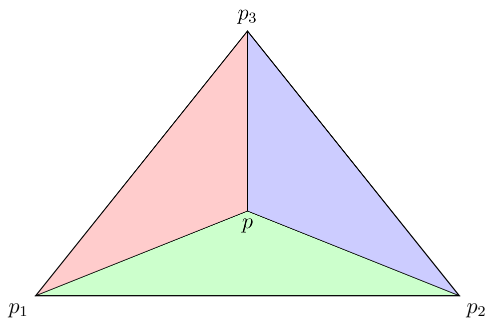

Figure 1: Let w_1 be the blue area, w_2 the red one and w_3 the green one. Normalizing each of them by the area of the triangle will return three values $(\lambda_1, \lambda_2, \lambda_3)$ that are the barycentric coordinates of p with respect to the triangle $[p_1, p_2, p_3]$.

1.2 Triangle meshes

A collection of triangles without any particular mathematical structure is called *triangle meshes*. To derive a global parameterization for an entire triangle mesh we can define a 2D position for each vertex. Let $\mathcal{M}$ be a triangle mesh that consists of a geometric and topological component represented by a graph structure with a set of vertices $\mathcal{V} = \{v_1, ..., v_V\}$ and a set of triangular faces connecting them $\mathcal{F} = \{f_1, ..., f_F\}$ with $f_i \in \mathcal{V} \times \mathcal{V} \times \mathcal{V}$. The connectivity of a triangle mesh can be expressed in termes of the edges of the respective graph $\mathcal{E} = \{e_1, ..., e_E\}$ where $e_i \in \mathcal{V} \times \mathcal{V}$. [4]

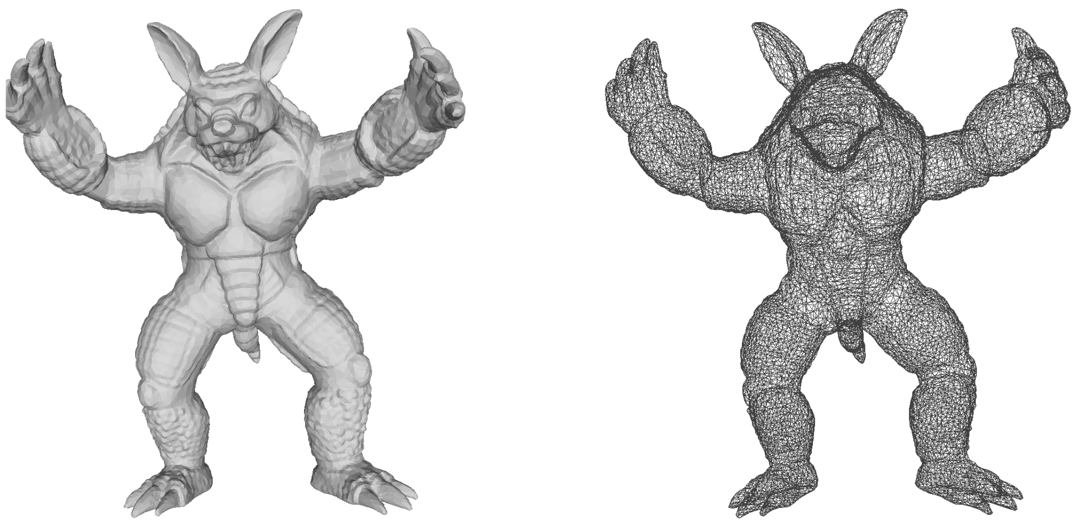

Figure 2: 3D triangle meshes.

1.3 Lighting - Phong lighting model

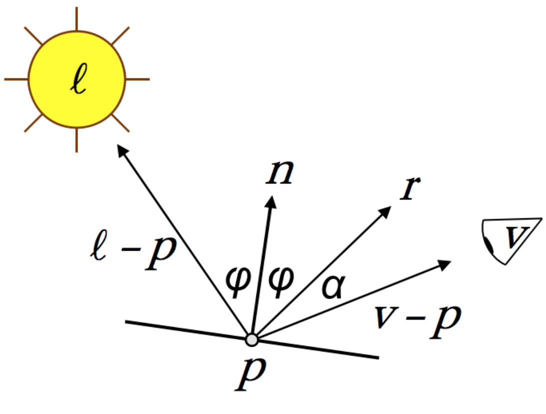

Figure 3: Lighting notation for Phong lighting model. [7]

Given a light source at position l with intensity I_l and a surface point at position p with normal n, we can define the angle between the incident light $(l-p)$ and the normal n as φ. Let r be our reflected light vector defined as $r = 2n \cdot < n, l-p > -(l-p)$ and α the angle between that vector and the view direction $(v-p)$.

The *Phong lighting model* is defined as the sum of the self-emitting intensity, ambient term, diffuse reflection and specular reflection.

$$I = I_e + \rho_a \cdot I_A + \sum_{j=1}^{n} (\rho_d \cdot cos\varphi_j + \rho_s \cdot cos_{\alpha_j}^{k}) \cdot I_j$$

where I_e is the self-emitting intensity, ρ_a, ρ_d, ρ_s are the reflection constants (surface properties), n is the number of lights sources with intensities I_j and k is the shininess. [7]

1.4 Linear interpolation

Linear interpolation is a method that will return equal spacing between the interpolated values. Given two numbers n_1 and n_2 (the start and final values of the interpolant), a linear interpolation can be carried out using a parameter t ($t \in [0,1]$). [3]

$$n = n_1 + t(n_2 - n_1)$$

The standard linear interpolated visualization is made passing three attributes (colors) to each vertex of a triangle. OpenGL will interpolate these colors linearly thanks to the barycentric coordinates that will tell how much of each color is being mixed at any position. Given a triangle $[p_1, p_2, p_3]$, where the color blue is passed to vertex p_1, red to p_2, and green to p_3, let be w_1 the blue area, w_2 the red area and w_3 the green area (See Fig. 1). Let us define the value at p as a *barycentric interpolation*

$$(w_1 p_1 + w_2 p_2 + w_3 p_3)/W$$

where W is the area of the triangle $[p_1, p_2, p_3]$.

1.5 Flat Shading

Flat shading is a way to compute the color at each pixel (at a corner or at the barycentre) using the triangle normal. Given a triangle $[p_1, p_2, p_3]$, the lighting is computed using the normal n

$$\widehat{n} = (p_2 - p_1) \times (p_3 - p_1) \qquad n = \frac{\widehat{n}}{||\widehat{n}||}$$

at $p = (p_1 + p_2 + p_3)/3$. This color is then used for all pixels. Flat shading gives objects with flat facets. [7]

1.6 Gouraud Shading

Gouraud Shading is a way to compute the color at each pixel assigning a normal to each corner of a triangle and after having computed the color for each corner it linearly interpolates these color values (see Sections 1.1 and 1.4). Given a triangle $[p_1, p_2, p_3]$ and the normal at each corner n_1, n_2, n_3, the lighting is computed at p_i using the normal n_i. This process applied to each corner returns the color values c_1, c_2, c_3 respectively for p_1, p_2 and p_3. These colors are then linearly interpolate $c = \mu_1 c_1 + \mu_2 c_2 + \mu_3 c_3$. Gouraud shading gives objects that appear more smooth. [7]

2 Discrete differential geometry

2.1 Normals

For each triangle $T = [p_1, p_2, p_3]$ of a triangle mesh, the normal is defined as

$$n(T) = \frac{(p_2 - p_1) \times (p_3 - p_1)}{||(p_2 - p_1) \times (p_3 - p_1)||}$$

The normal along each edge E is the halfway between the normals of two adjacent triangles T_1 and T_2

$$n(E) = \frac{n(T_1) + n(T_2)}{||n(T_1) + n(T_2)||}$$

The normal at vertex V is obtained averaging the normals of the n adjacent triangles

$$n(V) = \frac{\sum_{i=1}^{n} \gamma_i n(T_i)}{||\sum_{i=1}^{n} \gamma_i n(T_i)||}$$

where γ_i can be a constant value, equals to the triangle area or equals to the angle θ_i of T_i at V. [6]

2.2 Local averaging regions

Let assume that a mesh is a piecewise linear approximation of smooth surface. A mesh can be constructed either as the limit of a family of smooth surfaces or as a linear approximation of an arbitrary surface. To derive a spatial average of geometric properties we mix finite elements (a linear interpolation between three vertices of a triangle) and finite volumes (finite-volume region on a triangulated surface using Voronoi cells or Barycentric cells, Fig. 4). Restricting the average to the neighbouring triangles (*1-ring*) for each vertex, we can choose an associated surface patch over which the average will be computed. Let $\mathcal{A}_{Barycenter}$ be the area formed using barycenters and $\mathcal{A}_{Voronoi}$ the one formed using *Voronoi* cell. The general case is represented by a point that can be anywhere, let's denote this surface area $\mathcal{A}_M$.

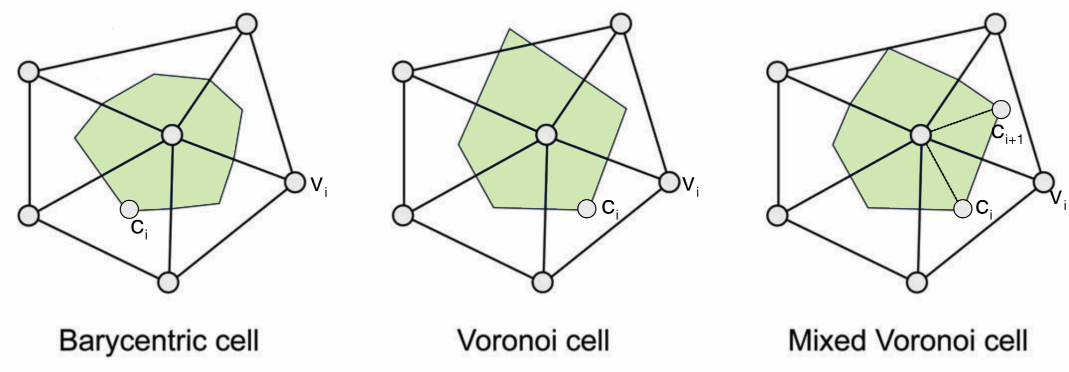

Figure 4: Local averaging regions used for computing discrete differential operators associated with the center vertex of the one-ring neighborhood. c_i corresponds to the barycentre of the triangle in the Bayrcentric cell, corresponds to the circumcenter of the triangle in the Voronoi cell. Let denominate θ the angle between c_i and c_{i+1}. If $\theta < \frac{\pi}{2}$ then c_i is on the circumcenter of triangle $[v_i, v, v_{i+1}]$, else c_i is the midpoint of the edge $[v, v_{i+1}]$. [4]

Voronoi cell of each vertex is an appropriate local region that provide a stable error bounds. The *Voronoi* region for a point P of a non-obtuse triangle $[P, Q, R]$ is expressed as $\frac{1}{8}(|PR|^2 cot\angle Q + |PQ|^2 cot\angle R)$. The sum of these areas for the whole *1-ring neighborhood* gives the non-obtuse *Voronoi* area for a vertex. The above expression for the *Voronoi* finite-volume area does not hold in case of obtuse angles. Let's define a new surface area for each vertex denoted $\mathcal{A}_{Mixed}$. Essentially the idea is to use the circumcenter point for each non-obtuse triangle and to use the midpoint of the edge opposite to the obtuse angle in case of an obtuse triangle. (See Pseudocode 4). [8]

2.3 Gaussian Curvature

The *Gaussian curvature* K is defined as the product of the principal curvatures:

$$K = k_1 k_2$$

A basic interpretation would be to imagine the *Gaussian curvature* as a logical `AND` since it checks whether there is a curvature along both directions. The curvature of a surface is characterized by the principal curvatures. [2] Surfaces that have a zero gaussian curvature are called *developable surfaces* because they can be flattened out into the plane without any stretching. *Gaussian curvature* should be zero inside each mesh triangle and the same along edges since it can be flattened symmetrically into the plane by simply rotating one triangle about the common edge into the plane defined by the other. Consequently the *Gaussian curvature* is concentrated at vertices of a triangle and it is defined as the *angle defect*

$$K(V) = 2\pi - \sum_{i=1}^{n} \theta_i$$

where θ_i is the angles of the triangle T_i adjacent to the vertex V at V. This should be seen as the integral of the Gaussian curvature over a certain region $S(V)$ around V, where

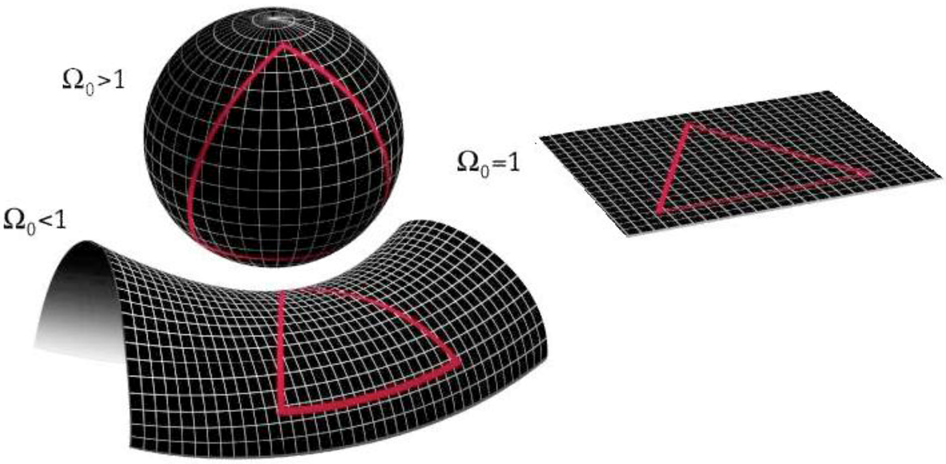

Figure 5: Positive curvature, negative curvature and zero curvature.

these $S(V)$ form a partition of the surface of the entire mesh.

$$K(V) = \int_{S(V)} K dA$$

Negative curvature can be recognized by the fact that external directions curve in opposite directions, *zero curvature* has one external direction that has zero curvature, *positive curvature* has external directions that curve in the same direction (Fig. 5). The *Theorema Egregium*, discovered by C.F. Gauss in 1827, states that the *Gaussian curvature* is an intrinsic property of the surface that does not depend on the space, despite by the fact that it is define as the product of the principal curvatures (whose value depends on how the surface is immersed in the space). We can then notice that triangle angles add up to less than 180° in negative curvature, exactly 180° in zero curvature, and more than 180° in positive curvature. [6]

2.4 Mean Curvature

The *mean curvature* H is defined as the arithmetic mean of principal curvatures

$$H = \frac{k_1 + k_2}{2}$$

A basic interpretation would be to imagine the *mean curvature* as a logical `OR` since it checks if there is a curvature along at least one direction.[2] The *mean curvature* inside each mesh triangle is zero, but it does not vanish at edges. The *mean curvature* associated with an edge is defined as $H(E) = \parallel E \parallel \theta_E/2$, where $\theta_E/2$ is the signed angle between the normals of adjacent triangles (see Fig. 6).

Let us think of an edge as a cylindrical patch $C(E)$ with a radius r that touches the planes defined by adjacent triangles. The *mean curvature* at any point of the cylindrical patch is defined as $1/(2r)$ and the area of $C(E)$ is $r||E||\theta_E$

$$H(E) = \int_{C(E)} H dA$$

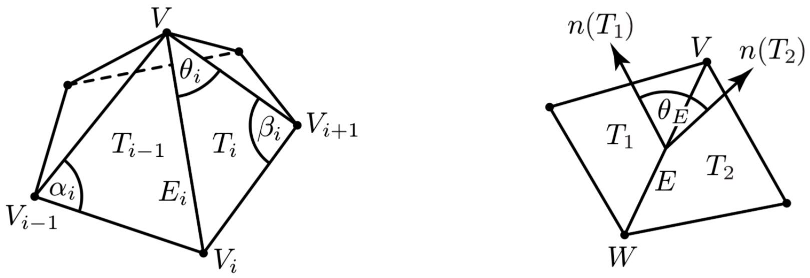

Figure 6: A vertex V with its neighbouring vertices V_i and adjacent triangles T_i. Angles opposite the edge E_i are denoted by α_i and β_i. The angle between the normals of adjacent triangles T_1 and T_2 with positive or negative sign is denoted as θ_E. [6]

. The *mean curvature* at the vertex V is defined as

$$H(V) = \frac{1}{2}\sum_{i=1}^{n} H(E_i)$$

Averaging the mean curvatures of its adjacent edges guarantee that *mean curvature* of an edge is divided uniformly to both end points. $H(E)$ and $H(V)$ would be seen as integral curvature values associated to regions $S(E)$ and $S(V)$. [6]

2.5 Mean Curvature Vector

Let be $H = Hn$ the surface normal vector scaled by the *mean curvature* to derive the discrete mean curvature vector associated to the mesh edge $E = [V, W]$

$$H(E) = \int_{C(E)} H dA = \frac{1}{2}(V - W) \times (n(T_1) - n(T_2))$$

The length of $H(E)$ gives the edge mean curvature $H(E) = ||H(E)|| = ||E|| sin(\theta_E/2)$. The discrete mean curvature vector associated to V can be obtained averaging $H(E)$ over the edges adjacent to a vertex V

$$H(V) = \frac{1}{2}\sum_{i=1}^{n} H(E_i) = \frac{1}{4}\sum_{i=1}^{n}(\cot\alpha_i + \cot\beta_i)(V - V_i)$$

where α_i and β_i are angles opposite to E_i (see Fig. 6). [6]

3 GPU program

A program that runs on GPU is called *shader*. Shaders are principally used to modify the representation and the behaviour of 3D objects. They are also used to create lighting effects. Shaders can perform tasks efficiently thanks to the GPU. That guarantees faster results than CPU since GPU is designed to work in parallel.

3.1 GPU pipeline

A program allows us to control the rendering pipeline since by default there is no pipeline set in OpenGL. It takes a set of vertices as input, then the *vertex shader* transforms them (translation, rotation, projection...) and passes the transformed vertices to the *geometry shader*. This shader takes vertices to create primitive shapes and then it rasterizes them. These rasterized flat images are then passed as input to the *fragment shader* that adds the lighting, apply textures and color these images (Fig. 7).

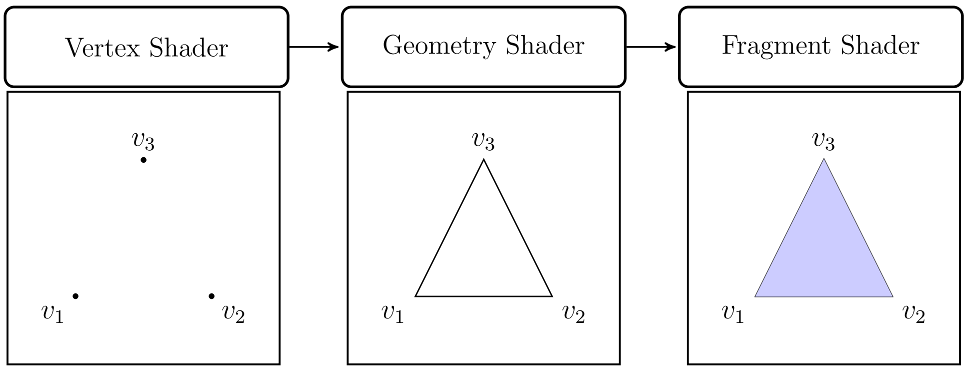

Figure 7: GPU pipeline [5]

3.2 Vertex Shader

The program that performs vertex operations is called *vertex shader*. It receives one vertex at a time and then it passes the output to a *fragment shader* or to a *geometry shader*, if any.

3.3 Fragment Shader

Fragment shader performs color computation for every visible pixel of the rasterized object. It works on a fragment at a time, but thanks to the power of GPU it can work in parallel for all vertices (*vertex shader*) and fragments (*fragment shader*).

3.4 Geometry Shader

Geometry shader is used for layered rendering. It takes as input a set of vertices (single primitive, example: triangle or a point) and it transforms them before sending to the next shader stage. In this way, we can obtain different primitives. Each time we call the function `EmitVertex()` the vector currently set to `gl_Position` is added to the primitive. All emitted vertices are combined for the primitive and output when we call the function `EndPrimitive()`. [1]

4 Vertex area based

This section shows alternative methods to extend the idea of flat shading from triangles to vertices. The idea of flat shading is to draw all the pixels of a triangle with the same color. The extension of this approach is to split the surface of the triangle mesh likewise into regions around vertices and draw all pixels in these regions with the same color (Fig. 9), thus visualizing data given at the vertices of the mesh in a piecewise constant, not necessarily continuous way, resembles the classical triangle flat shading. The aforementioned regions can easily be defined using barycentric coordinates and a simple GPU fragment program (Fig. 8) can be used for each pixel to find out to which region it belongs and which color it should be painted with.

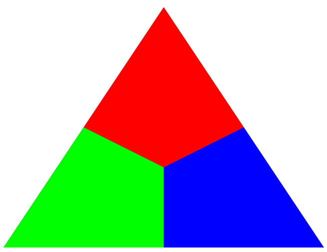

Figure 8: Max diagram

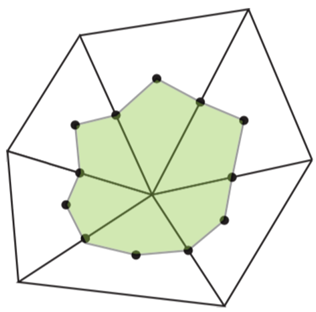

Figure 9: Region around a vertex

4.1 Max diagram - Vertex area

Passing barycentric coordinates to the *fragment shader* will clearly demonstrate that we can get different results from the classic color interpolation. [9] There are various approaches to color interpolation focusing on the distance from vertices. For each point in a triangle, we can easily determine its closest vertex, which we use as a cue for coloring. Another approach, different from the above, can be defined as coloring vertex areas based on the maximum barycentric coordinate. The color is given by the region closest to a vertex (Fig. 8, Pseudocode 1).

4.2 Vertex Flat Shading

An extension of *flat shading* would be to have each vertex area to be in one constant color. This color can be taken using the normal at the vertex and the vertex position. The color will then be computed as in *Gouraud shading*. The idea is to compute the color per vertex but instead of linearly interpolating it in each triangle (as *Gouraud shading* does) we color regions around a vertex with that constant color (using the GPU fragment program: max diagram 4.1). To implement this approach, the barycentric coordinates, the vertex color, the normal at the vertex and the lighting calculations must be passed to the *fragment shader*. We want to avoid the automatic interpolation of colors provided by OpenGL. In order to return the resulting color using the *max diagram* algorithm, we have used a *Geometry shader* that has access to all three vertex colors in *fragment shader*. (Pseudocodes: 5, **??**, **??**)

Figure 10: Vertex flat shading.

4.3 Comparison between triangle flat shading, triangle Gouraud shading and vertex flat shading

The standard approaches: *triangle flat shading* and *triangle Gouraud shading* are then compared with the new technique *vertex flat shading*. In Fig. 11 we can see that the icosahedron where we have applied the vertex flat shading shader seems to be a good compromise since it preserves the original geometry and avoids creating triangle-like artifacts in the final result. Vertex regions look more realistic and less noisy than triangle regions. Moreover, Gouraud shading is prone to smoothing and loosing many details in some meshes.

4.4 Gaussian curvature

Another interesting alternative data visualization technique is given computing the *Gaussian curvature* per vertex. That can be done summing up, for each vertex, angles at this vertex with adjacent triangles and then subtracting this value from 2π. Having obtained this value, called *angle defect* (Fig. 13), we map it linearly to a color range. The resulting

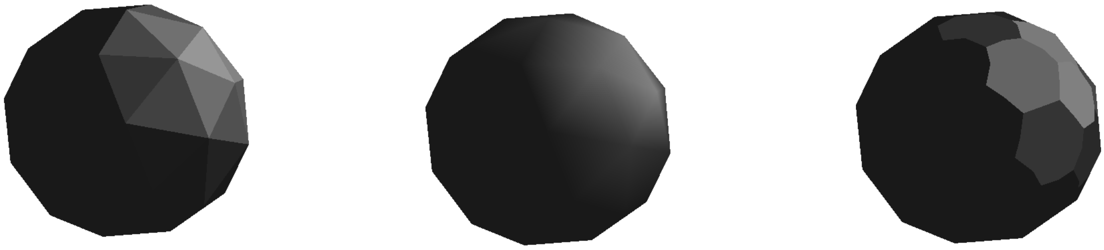

Figure 11: Comparison between: triangle flat shading, triangle Gouraud shading and vertex flat shading.

color will be the vertex flat shading visualisation of *Gaussian curvature* (See Section 2.2 and 2.3).

$$K(V) = (2\pi - \sum_j \theta_j)/\mathcal{A}_{Mixed}$$

4.5 Constant Gaussian curvature

Constant Gaussian curvature returns a constant color around each vertex using the max diagram algorithm(Fig. 13, Pseudocode **??**). The value $K(V)$ is mapped into a color range to get the corresponding curvature color (see Fig. 14). This process is made separately for each vertex of the triangle and consequently, using the technique of max-diagram explained above (See section 4.1), the final resulting constant color is returned.

4.6 Gouraud Gaussian curvature

Gouraud Gaussian curvature computes the curvature per vertex, convert it to color, and linearly interpolate it. The idea is to calculate the *Gaussian curvature* as explained above (mapping the color into a color range to get the corresponding color per vertex) but instead of returning the constant color using a max-diagram approach, we just return the interpolation of values obtained for each triangle.

4.7 Evaluation and Comparison between constant Gaussian curvature per vertex and Gouraud Gaussian curvature

We compare the *constant Gaussian curvature* (Fig. 15) with the *Gouraud Gaussian curvature* (Fig. 16). In Fig. 15 each vertex area is colored applying the *max diagram* algorithm. Instead, in Fig. 16 the color is obtained with a linear interpolation. Visualization of the principal curvatures of the model as colors from blue (highest values of curvature) to red (lower values of curvature) highlighs the geometry of meshes in Fig. 17. These changes of curvature, positive (blue), flat (green) and negative regions (red), better emphasizes the 3-dimensionality of the model. *Gouraud Gaussian curvature* is smoother,

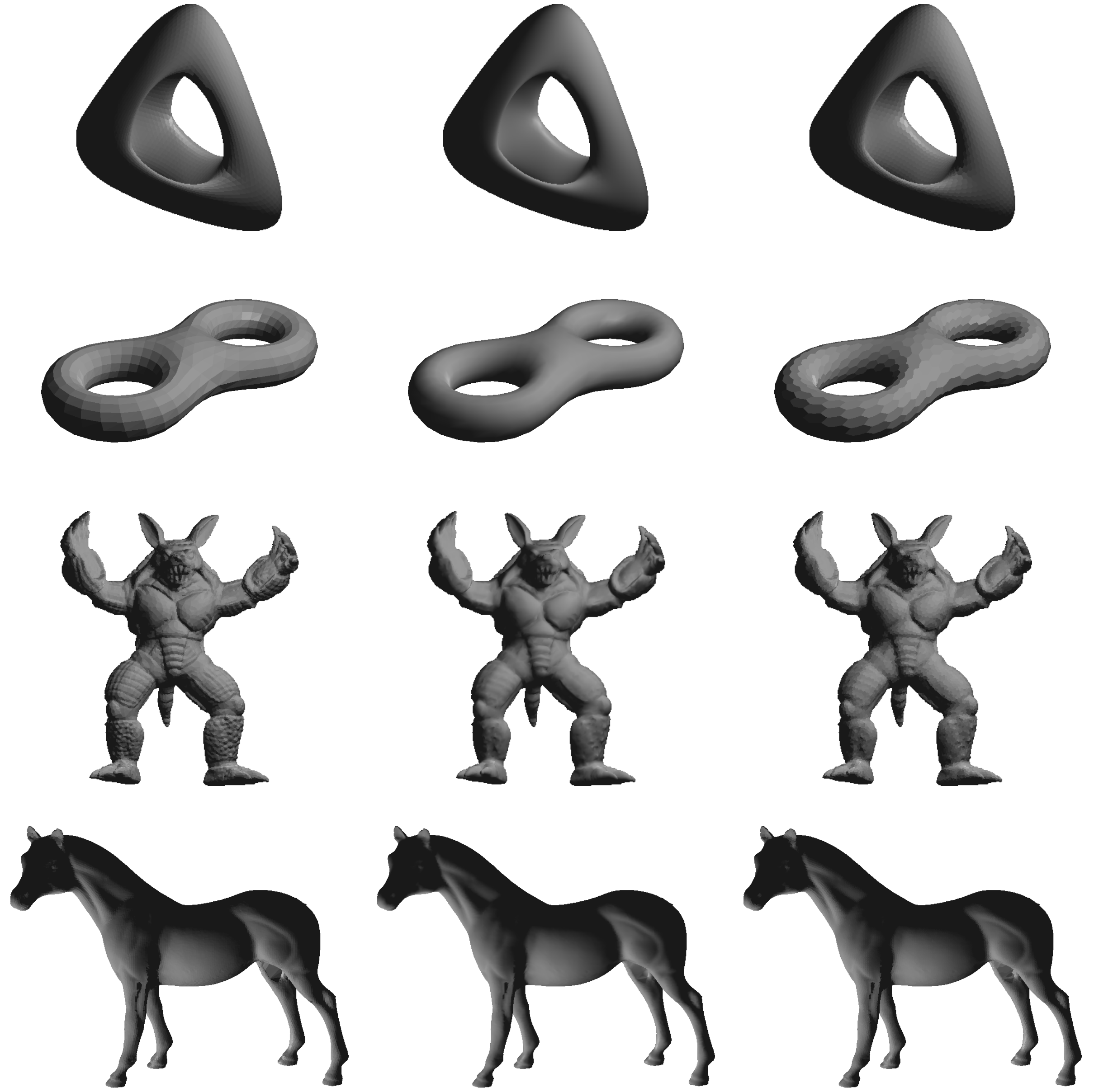

Figure 12: On the left: Triangle flat shading. On the center: Triangle Gouraud shading. On the right: Vertex flat shading.

which results in a loss of small details. This is particularly evident in armadillo's legs mesh. Instead, *constant Gaussian curvature* generates sharper edges with piecewise-flat regions which slightly degrades the 3-dimensional perception of the model. On the other hand, *Constant Gaussian curvature* preserves the details of the given geometry, which can be particularly useful for data visualization purposes.

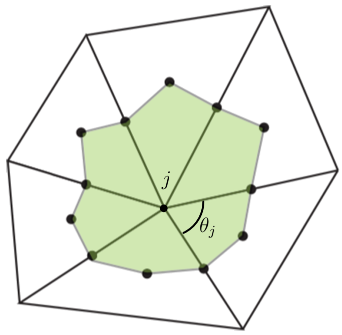

Figure 13: On the left: angle defect is denoted with θ_j.

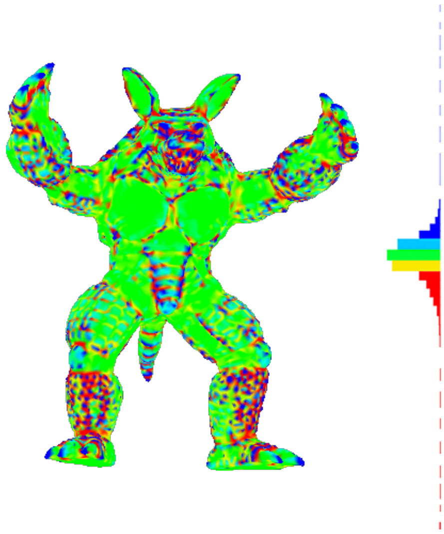

Figure 14: Color bar showing respective colors for negative, flat and positive curvatures. Negative curvatures are mapped to red, flat curvatures to green and positive curvatures to blue.

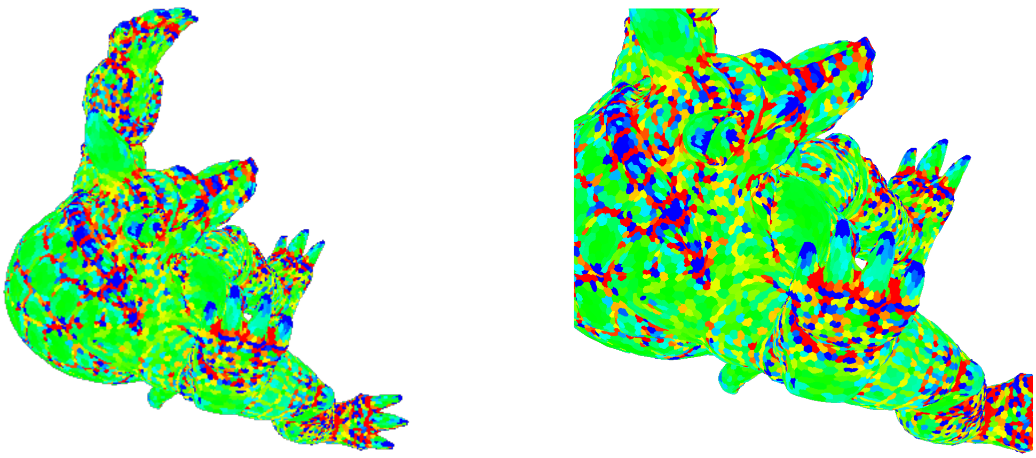

Figure 15: Constant Gaussian curvature

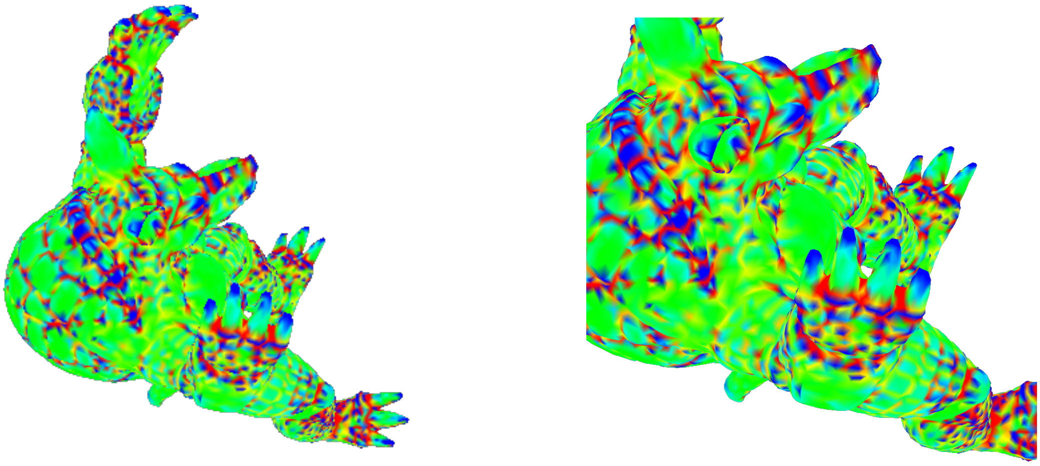

Figure 16: Gouraud Gaussian curvature

5 Edge area based

As the section before, we want now to show extensions where we split the surface of the triangle mesh likewise into regions around edges and we draw all pixels in these regions with the same color (see Figure 19). This results in rhombus-shaped areas with constant color around each edge.

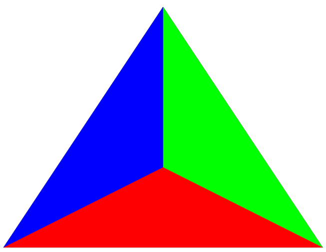

Figure 18: Min diagram

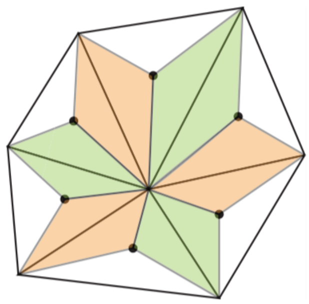

Figure 19: Region around an edge

5.1 Min diagram - Edge area

For each point in a triangle, we can easily determine its closest edge, which we use as a cue for coloring. A different approach from interpolating, can be found coloring vertex areas based on the minimum barycentric coordinate. The color is given by the region farthest from a vertex (Fig. 18, Pseudocode 2).

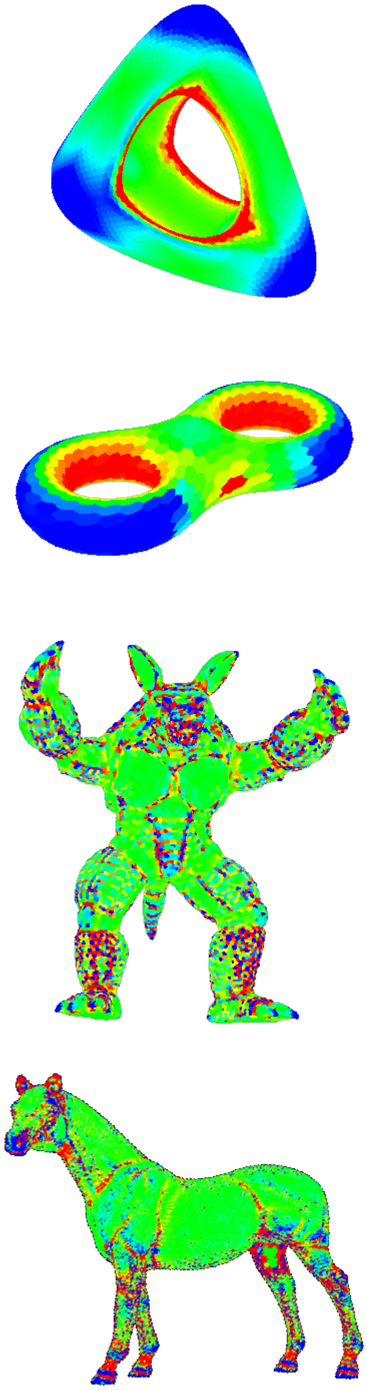

Figure 17: On the left: Constant Gaussian curvature. On the right: Gouraud Gaussian curvature.

5.2 Mean Curvature

Access to mesh edges requires to set up a list of edges over triangles. To avoid redundant datas we have apply the convention that an edge would be count only if it goes from a lower to higher vertex. An edge structure looks like the one in the pseudocode A, where index_v1 and index_v2 are the indexes vertices, norm_edge is the length of edge, n1 and n2 are the normals of triangles, cot_alpha and cot_beta are the cotangents of opposite angles to the edge, area_t1 and area_t2 are the areas of triangles.

5.3 Constant Mean Curvature

Constant mean Curvature returns a constant color around each edge (See Fig. 21). It first calculate the mean curvature for each edge:

$$H(E) = ||E||(\theta_E/2)$$

where θ_E is the angle between the two normals of T_1 and T_2 (See Fig. 20).

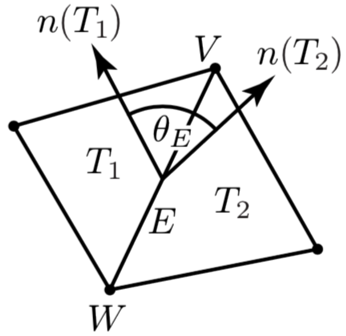

Figure 20: The dihedral angle θ_E at a mesh edge E is the angle between the normals of the adjacent triangles. [6]

The mean curvature for each vertex V of a mesh is defined as:

$$H(E) = \frac{1}{2\mathcal{A}_{Barycentre}} \sum_{i=1}^{n} ||E_i||(\theta_E/2)$$

This value is then normalized since it is an integral value. Every value is then mapped to positive or negative curvature, depending if the mesh at this edge is convex or concave, this can be known testing the $3D$ determinant of the 3-by-3 matrix $M = [e, n_1, n_2]$ with those three vectors as columns (e is the edge $[W, V]$, n_1 is the normal of T_1 and n_2 is the normal of T_2). If $det(M) > 0$ then the mesh is convex at $[W, V]$ and the mean curvature would be positive, else the mesh is concave and the mean curvature would be negative. This technique of edge flat shading represents an alternative to the classic triangle flat shading.

5.4 Gouraud Mean Curvature

Gouraud mean Curvature returns an interpolated color around each vertex. The main idea is to calculate the mean curvature $H(V)$ for each vertex V. In a mesh, every edge

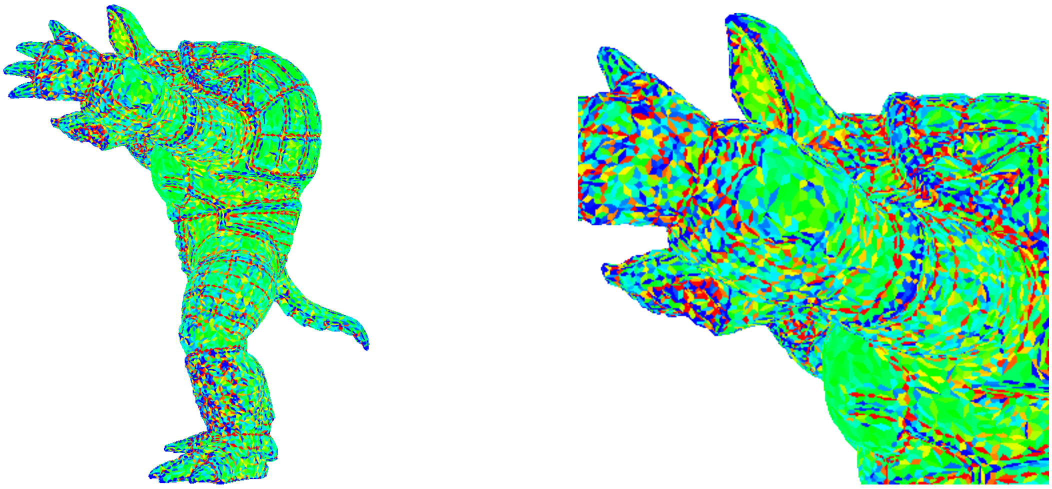

Figure 21: Constant mean curvature

has two opposite angles (let us denominate these with α and β), the mean curvature per vertex is defined as:

$$H(V) = \frac{1}{2\mathcal{A}_{Mixed}} \sum_{i=1}^{n} (\cot \alpha_i + \cot \beta_i)(V - V_i)$$

where V_i is one of the endpoints of the edge E_i (see Fig. 22).

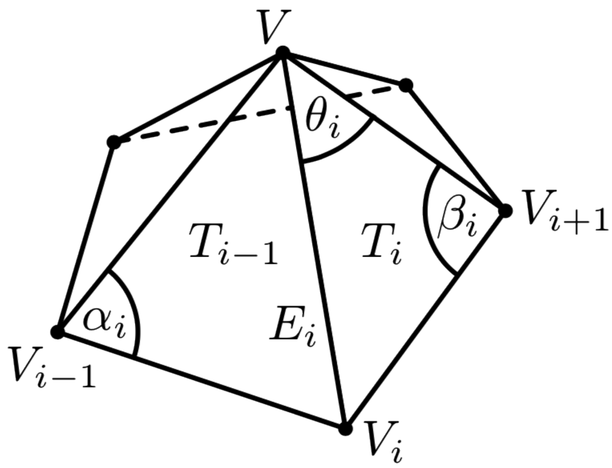

Figure 22: A vertex V of a triangle mesh with neighbouring vertices V_i and adjacent triangles T_i. The angle of T_i at V is denoted by θ_i and the angles opposite the edge E_i by α_i and β_i. [6]

These values are then interpolated using the automatic OpenGL interpolation resembling the classic Gouraud shading (see Fig. 23).

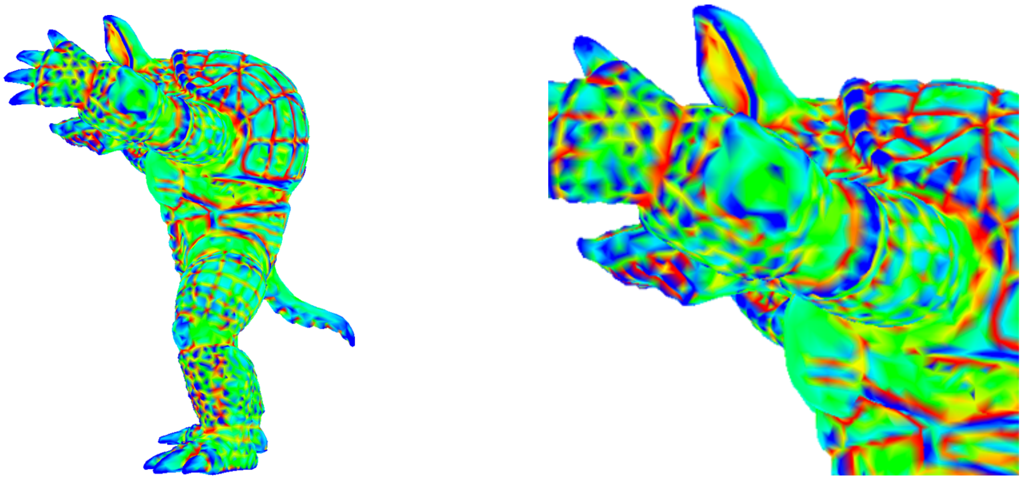

Figure 23: Gouraud mean curvature

5.5 Evaluation and Comparison between constant mean curvature and gouraud mean curvature

Constant mean curvature is a curvature per edge that return a constant color around each edge using the min diagram algorithm. The result is a noisy and sharped mesh that emphasizes edge. This visualization could be helpfull to show the quality of mesh triangulation and flows around surfaces. These last are observable because edge are directed allowing the user to better analyze curvatures (see Fig. 21). *Gouraud mean curvature*, on the other hand, return blurry meshes than the ones obtained with *Gouraud Gaussian curvature*. This smooth is caused by the fact that we are averaging per vertex, mean curvature would be calculated per edge but in this case, we had applied it on vertices opposite to edges, this result in a losing of data (see Fig. 23).

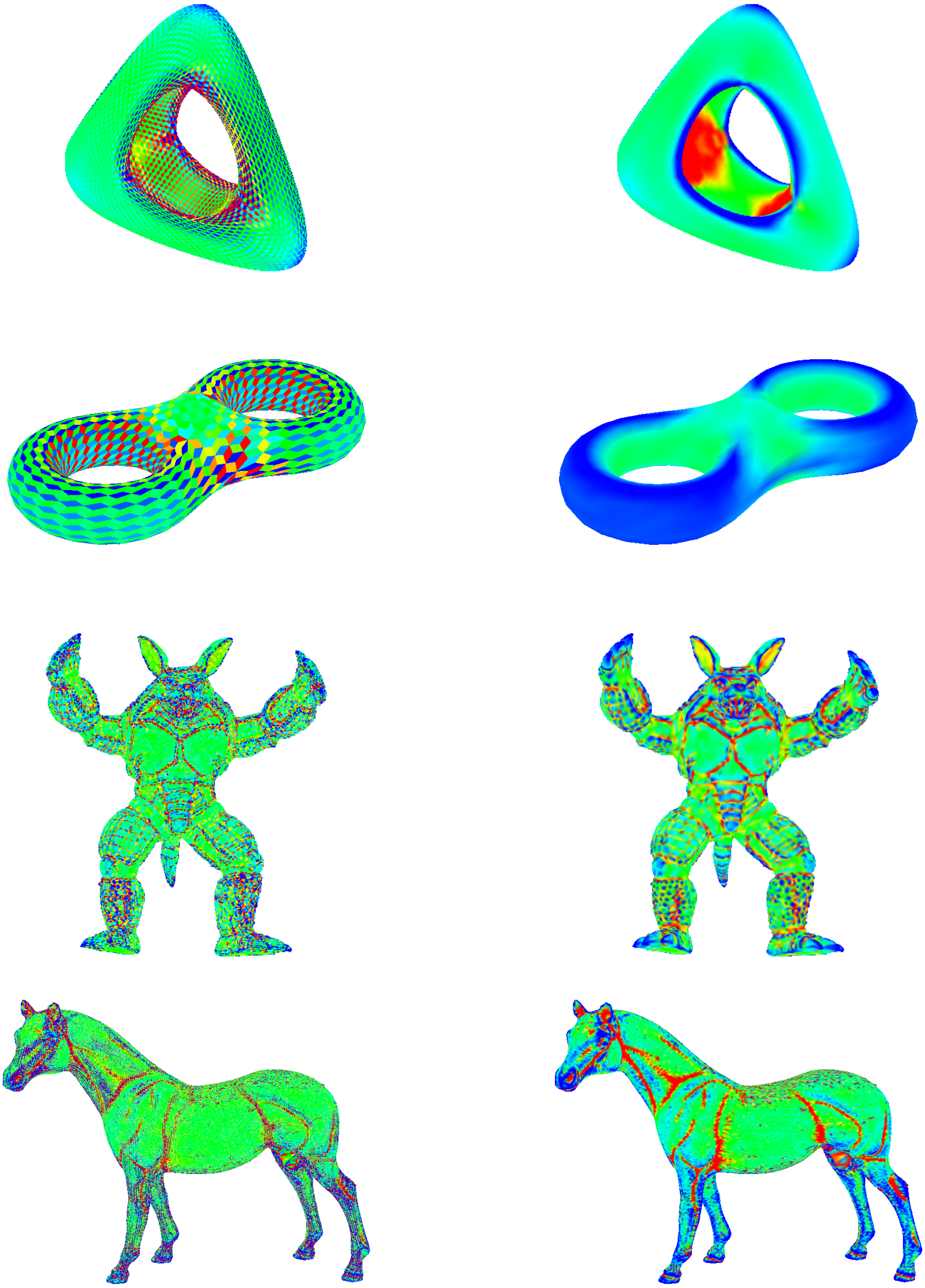

Figure 24: On the left: Mean curvature per edge. On the right: Gouraud mean curvature.

6 Conclusions

Making computation per vertex (e.g. *Flat Shading*) is more efficient because in general a models have less vertices than triangles (see Table 1). For example, the armadillo model has 15002 vertices and 30000 triangles, then make calculation per vertex instead of triangle results in half of the computations.

Model	#vertices	#triangles
Armadillo	15002	30000
Eight	766	1536
Genus3	6652	13312
Horse	48485	96966
Icosahedron_1	42	80
Icosahedron_2	162	320
Icosahedron_3	642	1280

Table 1: Comparative table: number of vertices and triangles in models.

Making computation per edge would also be more efficient, because edges are shared between 2 triangles in a mesh.

6.1 Software

A software for alternative data visualization using the power of barycentric coordinates and GPU programming. It allows the user to upload different models, choose different shaders, zoom or rotate the model. On Fig. 25, a *constant Gaussian curvature* shader is set on the model using a 90 *percentile*, on the right graphs plot Gaussian curvature values obtained for each vertex. The first graph shows the real values of Gaussian curvature without removing the outliers. The second graph shows just the values in the 90 *percentile* (then all the outliers were discarded).

6.2 Architecture

The software was developed in c++, for the real-time graphics programming (e.g. create the scene viewer, enabling the manipulation of 3D scenes) I have used OpenGL 3.3 and GLSL.

As graphical user interface I have used *Dear ImGui* since it has no external dependencies and it is designed to create content creation tools and visualization/debug tools. It is suited to integration in games engine (for tooling), real-time 3D applications, fullscreen applications, embedded applications,or any applications on consoles platforms where operating system features are non-standard.

To allow the creation of an OpenGL context, the defination of window parameters and to handle user input I have used the library *GLFW3*.

There are different versions of OpenGL drivers, to retrieve the location of the functions required and to store them in function pointers for later use, I have used the library *GLAD* that load all relevant OpenGL functions according to that version at compile-time.

Figure 25: Software

6.3 Comparison with meshlab

All the values obtained for the *Gouraud Gaussian curvature* and *Gouraud mean curvature* were compared to the result provided by the program *meshlab*[1].

[1]Meshlab is an open source system for processing and editing 3D triangular meshes. It provides a set of tools for editing, cleaning, healing, inspecting, rendering, texturing and converting meshes. It offers features for processing raw data produced by 3D digitization tools/devices and for preparing models for 3D printing. `http://www.meshlab.net/`

Model	our software	meshlab
Armadillo	[-33034.2, 90017.9]	[-33033.843750, 90019.632812]
Eight	[-116.89, 58.3357]	[-116.889397, 58.335678]
Genus3	[-1753.2, 209.183]	[-1753.197632, 209.180344]
Horse	[-321731, 1.93041e+06]	[-4177.138184, 4853.229004]
Icosahedron_1	[1.06991, 1.07979]	[1.069908, 1.079790]
Icosahedron_2	[1.01461, 1.02012]	[1.014615, 1.020121]
Icosahedron_3	[1.00291, 1.00519]	[1.002954, 1.005204]

Table 2: Comparative table: our Gouraud Gaussian curvature values ([min, max]) and meshlab Gouraud Gaussian curvature values ([min, max])

Model	our software	meshlab
Armadillo	[-289.74, 392.54]	[-289.739105, 392.536896]
Eight	[1.34736, 10.9581]	[1.347356, 10.958117]
Genus3	[-7.02334, 117.345]	[-7.023277, 117.344765]
Horse	[-500.961, 1202.51]	[-500.956573, 1202.507446]
Icosahedron_1	[0.999998, 1]	[0.999998, 1.000002]
Icosahedron_2	[0.999989, 1.00009]	[0.999989, 1.000094]
Icosahedron_3	[0.99993, 1.00007]	[0.999930, 1.000075]

Table 3: Comparative table: our Gouraud mean curvature values ([min, max]) and meshlab Gouraud mean curvature values ([min, max])

7 References

[1] Learn Opengl `https://learnopengl.com`.

[2] Discrete differential geometry. `http://brickisland.net/cs177/?p=144`.

[3] Maths for computer graphics. `https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf`.

[4] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Lèvy. *Polygon Mesh Processing*. A K Peters, Ltd., 2010.

[5] Harsha. The world of shaders. goHarsha `https://goharsha.com/lwjgl-tutorial-series/world-of-shaders/`.

[6] K. Hormann. *Encyclopedia of Applied and Computational Mathematics*, chapter Geometry Processing, pages 593–606. Springer, Berlin, Heidelberg, 2015.

[7] K. Hormann. Slide of course: Computer graphics. iCorsi3.

[8] M. Meyer, M. Desbrun, P. Schrder, and A.H. Barr. *Discrete Differential-Geometry Operators for Triangulated 2-Manifolds*, pages 35–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[9] A. Patel. Red blob games. https://www.redblobgames.com/x/1730-terrain-shader-experiments, July 2017.

[10] J. Zhang, B. Deng, Z. Liu, G. Patanè, S. Bouaziz, K. Hormann, and L. Liu. Local barycentric coordinates, 2014.

A Pseudocodes

Listing 1: Max diagram - Vertex area (Section: 4.1)

```
#version 330 core

in vec3 coords;
in vec4 wedge_color[3]; // an array of 3 vectors of size 4 (since it is a triangle)
out vec4 fragColor;

void main()
{
    if (coords[0] > coords[1]) {
      if (coords[0] > coords[2]) { // 0 > 1 && 0 > 2
        fragColor = wedge_color[0];
      }
      else { // 0 > 1 && 2 > 0 --> 2 > 0 > 1
        fragColor = wedge_color[2];
      }
    }
    else {
      if (coords[1] > coords[2]) { // 1 > 0 && 1 > 2
        fragColor = wedge_color[1];
      }
      else { // 1 > 0 && 1 < 2 --> 2 > 1 > 0
        fragColor = wedge_color[2];
      }
    }
}
```

Listing 2: Min diagram - Edge area (Section: 5.1)

```
#version 330 core
in vec3 coords;
in vec4 wedge_color[3]; // an array of 3 vectors of size 4 (since it is a triangle)
out vec4 fragColor;

void main()
{
  if (coords[0] < coords[1]) {
    if (coords[0] < coords[2]) {
      fragColor = wedge_color[0];
```

```
            }
            else {
                fragColor = wedge_color[2];
            }
        }
        else {
            if (coords[1] < coords[2]) {
                fragColor = wedge_color[1];
            }
            else {
                fragColor = wedge_color[2];
            }
        }
    }
```

Listing 3: Edge structure (Section 5.2)

```
    struct edge
    {
        float norm_edge;
        int index_v1;
        int index_v2;
        Point3d n1;
        Point3d n2;
        float value_mean_curvature;
        float cot_alpha;
        float cot_beta;
        float area_t1;
        float area_t2;
    };
```

Listing 4: Region $\mathcal{A}_{Mixed}$ on an arbitrary mesh. [8] (Section: 2.2)

```
    A_Mixed = 0
    For each triangle T from the 1-ring neighborhood of x
    If T is non-obtuse (Voronoi safe)
        A_Mixed += Voronoi region of x in T
    Else
        If x is obtuse
            A_Mixed += area(T)/2
        Else
            A_Mixed += area(T)/4
```

Listing 5: Vertex Shader for vertex/triangle flat shading and triangle Gouraud shading using lighting (Section: 4.2)

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 5) in vec3 aNormalTriangle;

struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

out vec4 color;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 view_position;
uniform Light light;
uniform float shininess;

uniform bool isFlat;


// get specular color at current Pos
vec3 get_specular(vec3 pos, vec3 normal, vec3 light_direction) {

    // get directional vector to the camera from pos
    vec3 view_direction = normalize(view_position - pos);

    // specular shading
    vec3 reflect_direction = - normalize(reflect(light_direction, normal));
    float specular_intensity = pow(max(dot(reflect_direction, view_direction), 0.0), shininess);

    // get resulting color
    return light.specular * specular_intensity;
}

```

```

    // return diffuse at current Pos
    vec3 get_diffuse(float lambert_term) {
        return light.diffuse * lambert_term;
    }

    vec4 get_result_color_lighting(vec3 pos, vec3 normal, vec3 light_position) {
        vec3 light_direction = normalize(light_position - pos);
        float diffuse_intensity = max(dot(light_direction, normal), 0.0);

        vec3 ambient = light.ambient;
        vec3 diffuse = get_diffuse(diffuse_intensity);

        if(diffuse_intensity > 0.0001){
            vec3 specular = get_specular(pos, normal, light_direction);
            return vec4((ambient + diffuse + specular), 1.0);
        }

        return vec4((ambient + diffuse) , 1.0);
    }


    void main() {

        vec3 world_position = vec3(model * vec4(aPos, 1.0));
        vec3 world_normal;

        if(isFlat){ // triangle normal
            world_normal = mat3(transpose(inverse(model))) * aNormalTriangle;
        } else { // vertex normal
            world_normal = mat3(transpose(inverse(model))) * aNormal;
        }

        vec3 light_pos = vec3(projection * vec4(light.position, 1.0));

        color = get_result_color_lighting(world_position, world_normal, light_pos); // color obtained with lighting calculations
```

```

        gl_Position = projection * view * model * vec4(aPos,
           1.0);
    }
```

Listing 6: Vertex Shader for constant/Gouraud Gaussian curvature and constant/-Gouraud mean curvature (Sections: 4.5 and 5.2)

```
    #version 330 core
    layout (location = 0) in vec3 aPos;
    layout (location = 2) in vec3 gaussian_curvature;
    layout (location = 3) in vec3 mean_curvature_edge;
    layout (location = 4) in vec3 mean_curvature_vertex;

    out vec4 color;

    uniform mat4 model;
    uniform mat4 view;
    uniform mat4 projection;

    uniform float min_curvature;
    uniform float max_curvature;

    uniform bool isGaussian;
    uniform bool isMeanCurvatureEdge;

    vec3 interpolation(vec3 v0, vec3 v1, float t) {
        return (1 - t) * v0 + t * v1;
    }

    vec3 hsv2rgb(vec3 c)
    {
        vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
        vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
        return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c
           .y);
    }


    vec4 get_result_color(){
        float val = gaussian_curvature[0]; //
           gaussian_curvature is a vec3 composed by same
           value
        if(!isGaussian && !isMeanCurvatureEdge){
            val = mean_curvature_vertex[0]; // mean curvature
```

```
            is a vec3 composed by same value
        } else if(!isGaussian && isMeanCurvatureEdge){
            val = mean_curvature_edge[0]; // mean curvature is a vec3 composed by same value
        }

        // colors in HSV
       vec3 red = vec3(0.0, 1.0, 1.0); //h s v
       vec3 green = vec3(0.333, 1.0, 1.0);
        vec3 blue = vec3(0.6667, 1.0, 1.0);

        if (val < 0) { //negative numbers until 0
            return vec4(hsv2rgb(interpolation(green, red, min(val/min_curvature, 1.0))), 1.0);
        } else { //from 0 to positive
            return vec4(hsv2rgb(interpolation(green, blue, min(val/max_curvature, 1.0))), 1.0);
        }
    }


    void main() {
        vec3 pos = vec3(model * vec4(aPos, 1.0));

        color = get_result_color();

        gl_Position = projection * view * model * vec4(aPos, 1.0);
    }
```

Listing 7: Geometry Shader for triangle flat shading, vertex flat shading, constant gaussian curvature, constant mean curvature (Sections: ?? and 5)

```
#version 330 core

layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

in vec4 color[3]; // an array of 3 vectors of size 4 (since it is a triangle)
out vec3 coords;
out vec4 wedge_color[3]; // an array of 3 vectors of size 4 (since it is a triangle)

void main()
{
```

```
        wedge_color[0] = color[0];
        wedge_color[1] = color[1];
        wedge_color[2] = color[2];

        coords = vec3(1.0, 0.0, 0.0);
        gl_Position = gl_in[0].gl_Position;
        EmitVertex();

        coords = vec3(0.0, 1.0, 0.0);
        gl_Position = gl_in[1].gl_Position;
        EmitVertex();

        coords = vec3(0.0, 0.0, 1.0);
        gl_Position = gl_in[2].gl_Position;
        EmitVertex();

        EndPrimitive();
    }
```