Télécom Paristech

IG3DA

Informatique Graphique 3D Avancée

Personal Project Report: Characterising Shapes using Conformal Factors

Bernard Lupiac

February 11, 2018

1 Reminders

As an introduction to this report, there will be a quick reminder of the contents of the paper.

1.1 Practical uses

The objective of this paper is to create a shape descriptor (*ie.* a simplified representation that describes a feature of the mesh). This descriptor allows to compare two meshs, and know how similar they are.

This is useful for example in a mesh repository, where is it useful to show meshes that are similar to the one the user is seeing at the moment.

1.2 Conformal Factor

The conformal factor is a unit that can be obtained by solving the following equation:

$$L\phi = K^t - K^{orig} \tag{1}$$

In this equation, ϕ is the vector with *conformal factors* of each vertex. L represents the *laplacian matrix*, with each cell being the cotangents of the vertices i and j. The diagonal is the sum of all cotagents of a vertex. K^{orig} is the vector with *gaussian curvatures* of each vertex, which is the sum of the angular defect on a vertex. The last part, K^t is the vector with the *target curvatures* of each vertex, which in a few words, distributes the total gaussian curvature between all vertices proportionately to their area.

1.3 Signature

Once the conformal factors are obtained, the actual shape descriptor can be calculated. But before that, an additional step is taken to make sure tessellation has no influence on the result. It consists on inserting many vertices randomly on the mesh, then get their conformal factor as an interpolation of nearby vertices.

Finally, the signature is a histogram made by 200 bins. Each bin represents the percentage of occurrences of vertices (originally calculated, or inserted later) in a certain interval.

2 Using the application

2.1 Inputs

The application can be built and executed without any arguments, in this case, the default mesh (*ie.* armadillo1) will be used for the calculations. The path to a .off can also be given as an argument, this file is then used for the calculations.

2.2 Outputs

1. A visual output, in the shape of an OpenGL window, that displays the mesh with the conformal factor as vertex colors. The same color scale as in the original paper (Figure 1) is used. It can be replaced by a greyscale by commenting the pre-processor constant "GRADIENT" in Main.cpp, l.32. The mesh can be rotated by moving the mouse with the left button pressed, panned by moving the mouse with the right button pressed and zoomed in/out using the mouse wheel.
2. A .csv file containing the signature of the mesh in the output folder: this file can be used as as input for different applications that use the conformal factor.

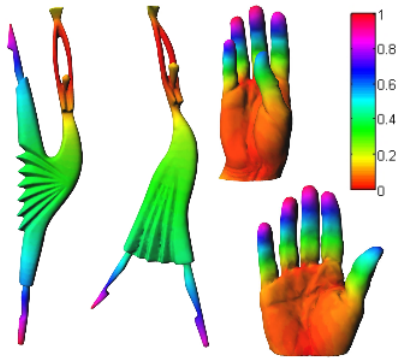

Figure 1: Figure of the original paper, showing the color scale used

2.3 Additional features

- By turning on the TIMER macro (mesh.cpp l.22, mesh.h l.29), the computation time of different parts of the application are logged to the console.
- By turning on the DEBUG macro (mesh.cpp l.21, mesh.h l.28, Main.cpp l.33), the visual output is the gaussian curvature instead of the conformal factor.
- The PRUNEFACT macro (mesh.cpp l.28) determines the percentage of values pruned from the conformal factors vector. Using 0 disabled pruning, and 0.01 for example removes 0.5% of the highest and lowest values from the vector. The need to use this feature is explained in section 4.2.
- The NOISEFACT macro (mesh.cpp l.29) determines the intensity of noise added to the mesh. Noise added is calculated by:

$$P_{new} = P + rand \cdot N \cdot noiseFactor \tag{2}$$

  Where P_{new} is the new position and P is the old. $rand$ is a random number between -1 and 1, N is the normal of the vertex and $noiseFactor$ is the noise factor given as an input. The need to use this feature is explained in section 4.3.

3 Implementation

3.1 Conformal Factor

Last year's IGR202 project was used as a basis for this one. It provided many useful data structures, such as a vector with all triangles a vertex is contained in. Some relevant methods were also already implemented, like the parsing of .off files, calculating the cotangent of a vertex in a triangle and giving all vertices in a 1-neighbourhood of a vertex. Also very useful is the visual output the application gives, making it possible to debug the application at different levels.

The first step on the calculation of the conformal factor is filling a vector with the gaussian curvatures, iterating over the mesh's vertices. Then, for the target curvature, there is also an iteration on the mesh's vertex. The same vector is used to save space and time: later only a vector with the difference of the target and the gaussian curvature will be needed.

For the laplacian matrix, a sparse matrix is used, since each row will only have a handful of non-zero elements (number of neighbours of vertex plus the diagonal). There is once again an iteration over the mesh's vertex to fill the matrix.

Once the laplacian matrix and the curvature difference vector have been obtained, an Eigen solver is used to get the vector with the conformal factors. Since this calculation is by far the biggest bottleneck in calculation time, two different Eigen solvers were compared: SparseLU, which works with sparse matrices, and ConjugateGradient, which is adapted to solve Poisson 3D equations such as the conformal factor one. The SparseLU solver was superior in every test performed, and for this reason was kept for the rest of the project. The method that uses the ConjugateGradient is still in the source code (mesh.cpp at l.506-523).

3.2 Signature

As explained in the reminder, before calculating the signature, many vertices (5 times the number of original vertices) are introduced in the mesh to make the result more robust to tessellation. This is done in a few steps:

1. Find a random triangle in the mesh
   This is much more complicated than it sounds: the probability of the triangle being chosen has to be proportional to its area.
   During preprocessing, a vector with the area of each triangle has to be calculated, along with the total area of the mesh. The next step is a huge optimisation of this part of the calculation (this part was previously a bottleneck of the application), that reduces the time complexity from $O(n^2)$ to $O(n \cdot log(n))$: using an accumulated area vector instead of a simple area vector. This way, with a number between 0 and the total area, it is possible to do a binary search for the lookup instead of iterating over the area vector.

2. Put the vertex randomly in the triangle
   Done by implementing the equation (1) in the following paper:
   https://www.cs.princeton.edu/~funk/tog02.pdf

3. Getting the conformal factor of the vertex Finds barycentric coordinates of the vertex inside the triangle, and uses them as weights to get an interpolated value from the conformal factor of the three vertices of the triangle.

Finally, the signature is generated: it is a vector of "bins", going from -99 to 100. The conformal factor vector is normalised, and there is a last iteration over all the points, incrementing the corresponding bin. The bins are then normalised to make reading and interpretation easier.

4 Results

As a foreword, the original paper gives very little quantitative results. They are mainly comparisons with other existing algorithms at the time, so it is hard to compare directly this implementation with the original. The fact that this is a 10 year old paper would also make comparing computation times not very productive.

Another important observation is that the range of meshes that can be used with this algorithm is very reduced: it is only applicable to watertight shapes.

4.1 Similar meshs

The main use of this application is to determine which mesh are similar to other, and works especially well for isometric transformations of a mesh.

The test with three different armadillos is very satisfactory, since the histograms are very similar.

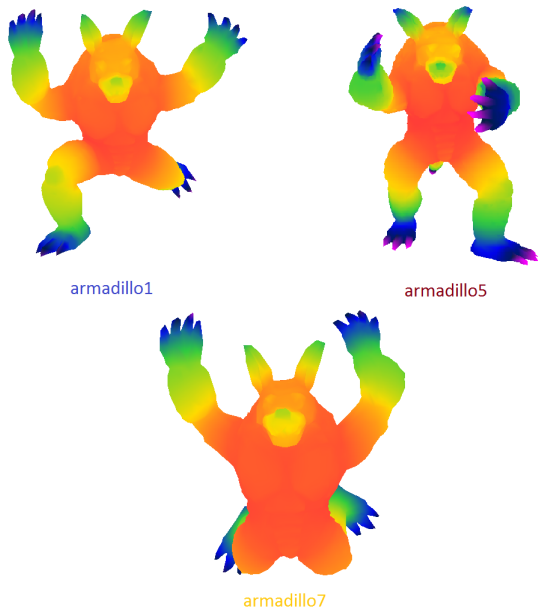

Figure 2: Shapes used in the similarity comparison

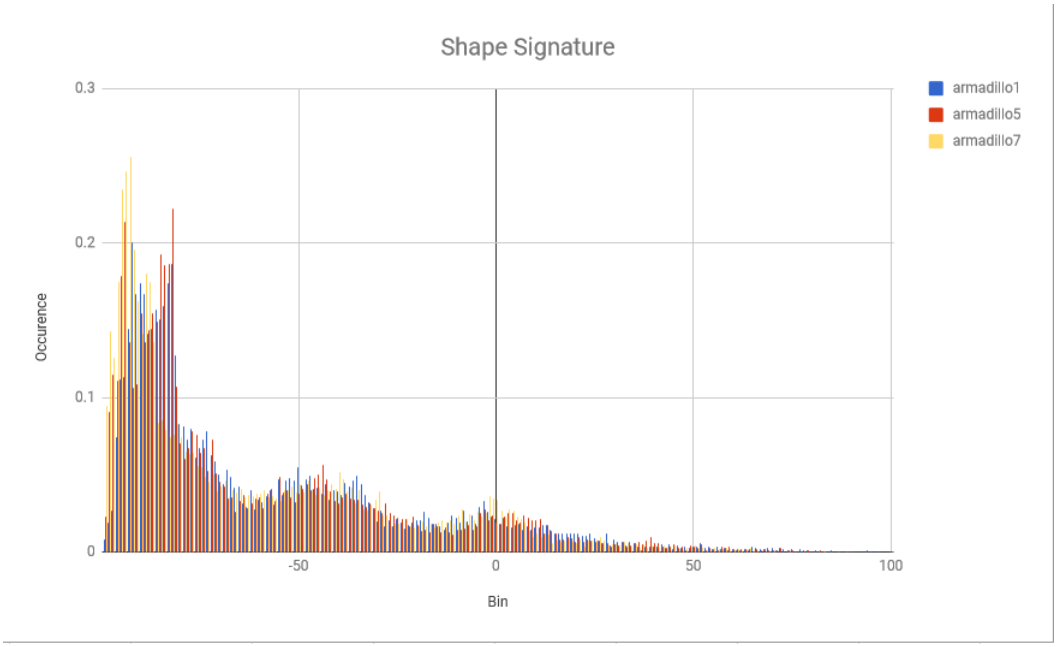

Figure 3: Comparison of the signatures of three different armadillo meshs

4.2 Robustness to tessellation

Another important point of the original paper was being robust to tessellation. The first test, performed on the same mesh with 5k triangles and 330k triangles was not very satisfactory:

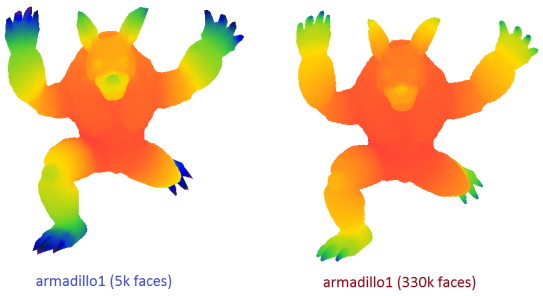

Figure 4: Shapes used in the tessellation comparison without pruning

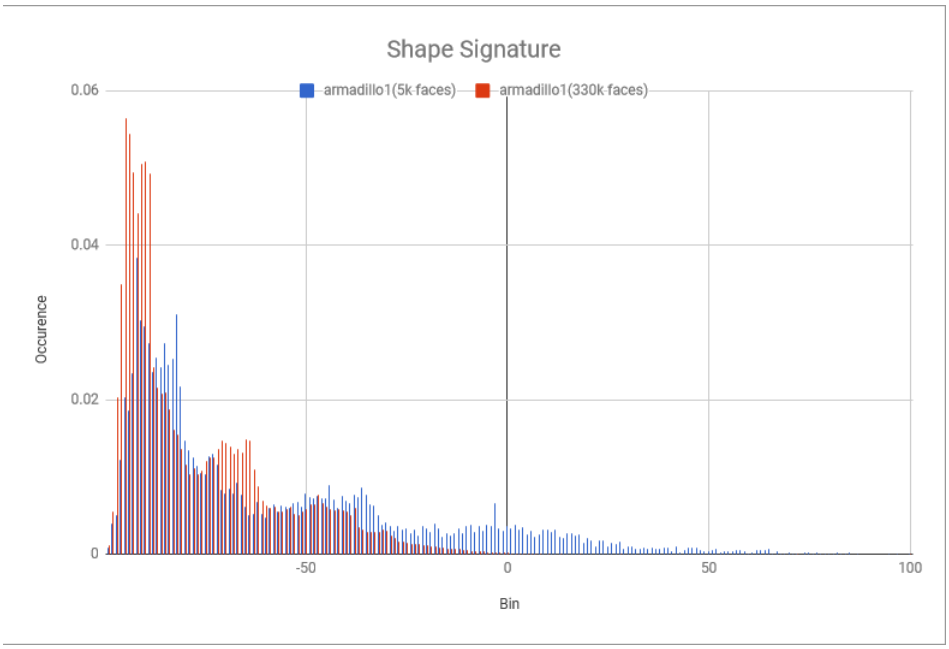

Figure 5: Comparison of the signatures of the same mesh with tessellation

However, when looking closely, the red signature looks like the blue one, but "compressed" to the left. This could be caused by a few vertices having extraordinary values. They would makes the range of possible values be very different. Then, the normalisation step causes the signatures to be completely different.

By implementing a pruning function, removing 0.5% of the highest and lowest values of the final vector, the results are much more acceptable:

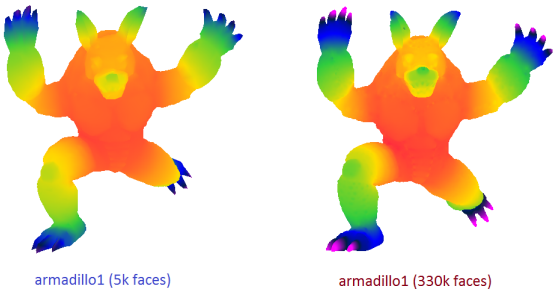

Figure 6: Shapes used in the tessellation comparison with pruning

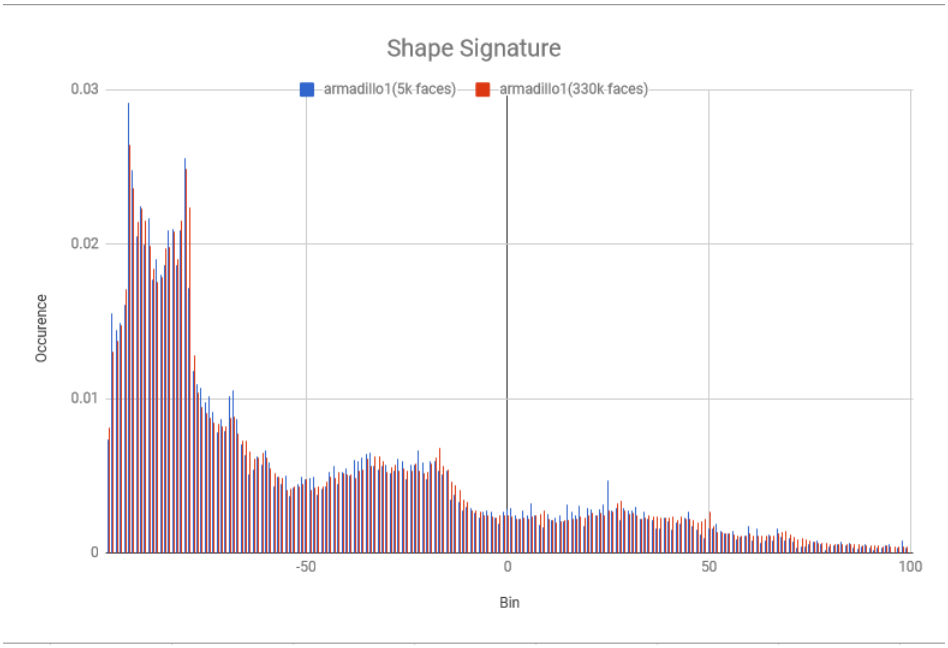

Figure 7: Comparison of the signatures of the same mesh with tessellation and pruning

4.3 Robustness to noise

The last important point of the algorithm is being robust to noise. A comparison of the same mesh with no noise, a decent amount of noise and extreme noise were compared. It's not really a problem that the version with more extreme noise isn't similar to the original since it is an extreme case.

However the other two have very similar signatures, which means the implementation is robust to noise.

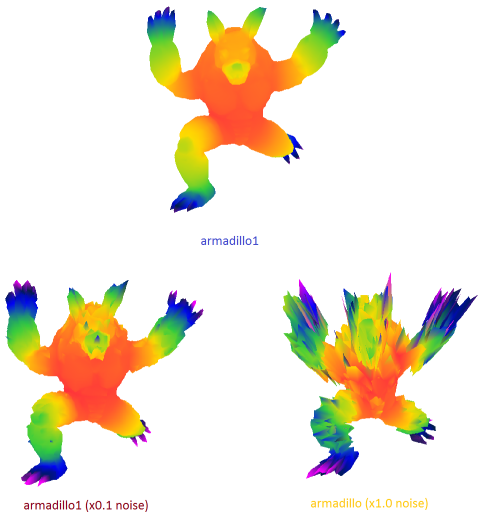

Figure 8: Shapes used in the noise comparison

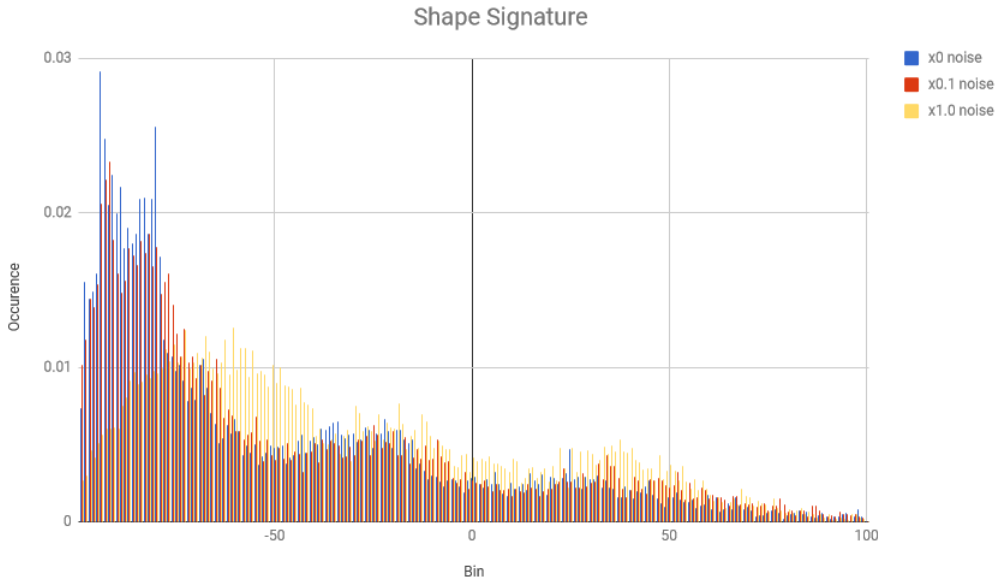

Figure 9: Comparison of the signatures of the same mesh with different levels of noise

4.4 Execution time

For most meshs used in this report, calculations take a few milliseconds. The exception is the 330k triangles armadillo, who takes about 3.5 seconds.

For this reason, the calculation time for a larger shape has been calculated: With around 170MB, the Neptune mesh has about to 4 million triangles.

Figure 10: Conformal factors of the Neptune mesh

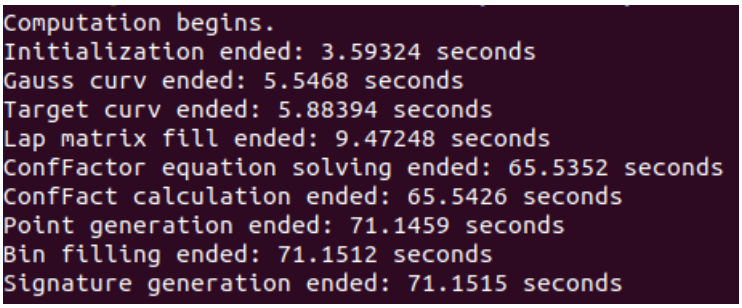

Figure 11: Calculation time of each step of the algorithm for the Neptune mesh

As the numbers show, the step that takes the most time, by far, is the Eigen solver. In Section 3.1, the efforts to reduce this time are explained. Before the optimisation described in Section 3.2 with the accumulated area vector, the point generation was the bottleneck of the algorithm.

4.5 Possible improvements

- A version that doesn't make a visual output could save memory since some vectors wouldn't need to be saved.
- Making the pre-processor constants into command line arguments would be very useful.
- The Eigen solver does not work when there is a 0 in the diagonal of the laplacian matrix, which reduces the number of meshs that can be taken as an input. Maybe there is a way to fix this?