SOFTWARE TESTING

for

Mesh Smoothing Tool

Release 1.0

Version 1.0 approved

Prepared by Jan Jose Hurtado Jauregui
Pontifícia Universidade Católica do Rio de Janeiro

Contents

1 Test Plan

The implementation consists of a lot of calls of functions contained in third party libraries. We will not include adaptations and managers of these functions for software testing. Also we will not consider the visualization module because it works as a manager of the OpenGL API and visualization is not the main objective of this software. We will focus on mesh processing functions adopting two different test methodologies. The first one is the well know unit testing, where we will use sampled functions evaluating a set of test cases. In the second one, we want to evaluate if the behavior of the smoothing algorithms is correct. So we will evaluate them using metrics to obtain numerical results and visualize the resulting meshes to detect possible problems.

2 Unit testing

In this test, we use five sampled functions which are part of the core of mesh processing module and are critical in the software behavior:

1. Update vertex positions (updateVertexPositions). We compare the difference between obtained vertex positions and normals, and the expected ones.
2. Compute average edge length (averageEdgeLength).
3. Compute mesh area (getArea).
4. Compute average radius regarding face centroid distances (getRadius).
5. Compute barycentric area of a single vertex (getVertexArea).

For each function we have an expected value and a tolerance depending on the test case. The test cases we considered for testing are the following:

1. Test case 0: a mesh without vertices and faces.
2. Test case 1: a single vertex.
3. Test case 2: a single triangle (three vertices).
4. Test case 3: a plane conformed by two triangles (four vertices).

For more details see Figure 1. The coordinates are described between brackets and the indices in red. The triangle of test case 2 is defined by (0,1,2) and the triangles of test case 3 by (0,1,2) and (0,2,3). The normals for both triangles can be derived as (0,1,0). As we said before, we define for each test case an expected value and a tolerance. In Table 1 we present the expected values:

Due to the same scale used for all test cases and the nature of the evaluated functions, we give a tolerance of 0.001 for all of them.

We tested these functions without using and external tool. We implemented our own code which is shown as follows:

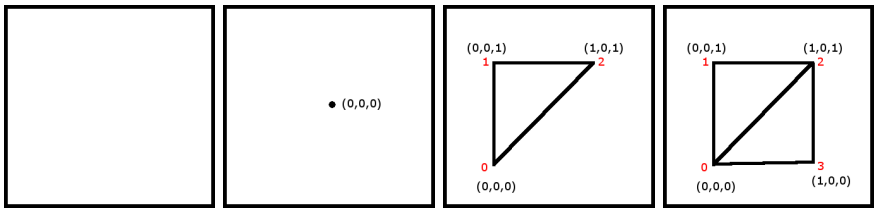

Figure 1: Triangular meshes used for unit testing of mesh processing module (sampled functions). From left to right: test case 0, test case 1, test case 2 and test case 3. Coordinates between brackets and indices in red.

Table 1: Expected values

	test case 0	test case 1	test case 2	test case 3
update vertex positions	0	0	0	0
average edge length	0	0	1.138	1.0828
get area	0	0	0.5	1
get radius	0	0	0	0.4714
vertex area	0	0	0.1667	0.3333

```
#include "mesh/util.h"
#include "mesh/iomesh.h"
#include "mesh/denoising.h"

typedef float (*t_testFunctionPtr)(TriMesh &);

float comparePositions(TriMesh & m1, TriMesh & m2)
{
        float error = 0;
        for (TriMesh::VertexIter v_it = m1.vertices_begin(); v_it != m1.vertices_end(); v_it++)
        {
                int index = v_it->idx();
                error += (m1.point(TriMesh::VertexHandle(index)) - m2.point(TriMesh::VertexHandle(index))).
                      length();
        }
        return error;
}

float compareNormals(TriMesh & m1, TriMesh & m2)
{
        float error = 0;
        for (TriMesh::VertexIter v_it = m1.vertices_begin(); v_it != m1.vertices_end(); v_it++)
        {
                int index = v_it->idx();
                error += (m1.normal(TriMesh::VertexHandle(index)) - m2.normal(TriMesh::VertexHandle(index)))
                      .length();
        }
        return error;
}

float testUpdateVertexPositions(TriMesh & mesh)
{
        TriMesh res = mesh;
        vector<TriMesh::Normal> normals;
        getAllFaceNormals(res, normals);
        updateVertexPositions(res,normals,10,false);
        return comparePositions(res, mesh) + compareNormals(res, mesh);
}

float testGetAverageEdgeLength(TriMesh & mesh)
{
        return getAverageEdgeLength(mesh);
}

float testGetArea(TriMesh & mesh)
{
        return getArea(mesh);
}
```

```
float testGetRadius(TriMesh & mesh)
{
        return getRadius(mesh,1.0);
}

float testGetVertexArea(TriMesh & mesh)
{
        vector<float> areas;
        getAllFaceAreas(mesh, areas);
        if (mesh.n_vertices() != 0)
                return getVertexArea(mesh, TriMesh::VertexHandle(0) , areas);
        else
                return 0.0f;
}


vector<string> testNames = { "updateVertexPositions", "getAverageEdgeLength", "getArea", "getRadius", "
      getVertexArea" };
vector<t_testFunctionPtr> testFunctions = { &testUpdateVertexPositions, &testGetAverageEdgeLength,&
      testGetArea, &testGetRadius, &testGetVertexArea};
vector<float> tolerance = {0.001f , 0.001f, 0.001f, 0.001f, 0.001f};
vector<float> expectedValuesTestCase0 = { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
vector<float> expectedValuesTestCase1 = { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
vector<float> expectedValuesTestCase2 = { 0.0f, 1.138f, 0.5f, 0.0f, 0.1667f };
vector<float> expectedValuesTestCase3 = { 0.0f, 1.0828f, 1.0f, 0.4714f, 0.3333f };

TriMesh testCase0; //empty
TriMesh testCase1; //single point
TriMesh testCase2; //single trianle
TriMesh testCase3; //plane (two triangles)

void initializeTestCases()
{
        testCase0.clear();
        testCase1.clear();
        importMesh(testCase1, "TestCase1.off");
        testCase1.request_face_normals();
        testCase1.update_normals();
        importMesh(testCase2, "TestCase2.off");
        testCase2.request_face_normals();
        testCase2.update_normals();
        importMesh(testCase3, "TestCase3.off");
        testCase3.request_face_normals();
        testCase3.update_normals();
}

bool OK(float value, float expected_value, float tolerance)
{
        return (value >= (expected_value - tolerance) && value <= (expected_value + tolerance));
}

void runTests()
{
        for (int i = 0; i < testNames.size(); i++)
        {
                cout << testNames[i] << endl;
                cout << "TestCase0 -- " << "value: " << (*(testFunctions[i]))(testCase0) << " / expected
                      value: " << expectedValuesTestCase0[i] << " / OK: " << OK((*(testFunctions[i]))(
                      testCase0), expectedValuesTestCase0[i], tolerance[i]) << endl;
                cout << "TestCase1 -- " << "value: " << (*(testFunctions[i]))(testCase1) << " / expected
                      value: " << expectedValuesTestCase1[i] << " / OK: " << OK((*(testFunctions[i]))(
                      testCase1), expectedValuesTestCase1[i], tolerance[i]) << endl;
                cout << "TestCase2 -- " << "value: " << (*(testFunctions[i]))(testCase2) << " / expected
                      value: " << expectedValuesTestCase2[i] << " / OK: " << OK((*(testFunctions[i]))(
                      testCase2), expectedValuesTestCase2[i], tolerance[i]) << endl;
                cout << "TestCase3 -- " << "value: " << (*(testFunctions[i]))(testCase3) << " / expected
                      value: " << expectedValuesTestCase3[i] << " / OK: " << OK((*(testFunctions[i]))(
                      testCase3), expectedValuesTestCase3[i], tolerance[i]) << endl;
                cout << endl;
        }
}
```

2.1 Results 1

These results are from the initial implementation. We can see some not defined numbers.

```
updateVertexPositions
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase2 -- value: 0 / expected value: 0 / OK: 1
TestCase3 -- value: 0 / expected value: 0 / OK: 1
```

```
getAverageEdgeLength
TestCase0 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase1 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase2 -- value: 1.13807 / expected value: 1.138 / OK: 1
TestCase3 -- value: 1.08284 / expected value: 1.0828 / OK: 1

getArea
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.5 / expected value: 0.5 / OK: 1
TestCase3 -- value: 1 / expected value: 1 / OK: 1

getRadius
TestCase0 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase1 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase2 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase3 -- value: 0.471405 / expected value: 0.4714 / OK: 1

getVertexArea
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.166667 / expected value: 0.1667 / OK: 1
TestCase3 -- value: 0.333333 / expected value: 0.3333 / OK: 1
```

2.2 Results 2

The problem with updateVertexPositions function was a division by 0. We fixed it and obtained the following results:

```
updateVertexPositions
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0 / expected value: 0 / OK: 1
TestCase3 -- value: 0 / expected value: 0 / OK: 1

getAverageEdgeLength
TestCase0 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase1 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase2 -- value: 1.13807 / expected value: 1.138 / OK: 1
TestCase3 -- value: 1.08284 / expected value: 1.0828 / OK: 1

getArea
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.5 / expected value: 0.5 / OK: 1
TestCase3 -- value: 1 / expected value: 1 / OK: 1

getRadius
TestCase0 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase1 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase2 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase3 -- value: 0.471405 / expected value: 0.4714 / OK: 1

getVertexArea
TestCase0 -- value: 0 / expected value: 0 / OK: 1
```

```
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.166667 / expected value: 0.1667 / OK: 1
TestCase3 -- value: 0.333333 / expected value: 0.3333 / OK: 1
```

2.3 Results 3

The problem with getAverageEdgeLength function was a division by 0 too:

```
updateVertexPositions
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0 / expected value: 0 / OK: 1
TestCase3 -- value: 0 / expected value: 0 / OK: 1

getAverageEdgeLength
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 1.13807 / expected value: 1.138 / OK: 1
TestCase3 -- value: 1.08284 / expected value: 1.0828 / OK: 1

getArea
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.5 / expected value: 0.5 / OK: 1
TestCase3 -- value: 1 / expected value: 1 / OK: 1

getRadius
TestCase0 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase1 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase2 -- value: -1.#IND / expected value: 0 / OK: 0
TestCase3 -- value: 0.471405 / expected value: 0.4714 / OK: 1

getVertexArea
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.166667 / expected value: 0.1667 / OK: 1
TestCase3 -- value: 0.333333 / expected value: 0.3333 / OK: 1
```

2.4 Results 4

And also the problem with getVertexArea was a division by 0.

```
updateVertexPositions
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0 / expected value: 0 / OK: 1
TestCase3 -- value: 0 / expected value: 0 / OK: 1

getAverageEdgeLength
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 1.13807 / expected value: 1.138 / OK: 1
TestCase3 -- value: 1.08284 / expected value: 1.0828 / OK: 1

getArea
```

```
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.5 / expected value: 0.5 / OK: 1
TestCase3 -- value: 1 / expected value: 1 / OK: 1

getRadius
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0 / expected value: 0 / OK: 1
TestCase3 -- value: 0.471405 / expected value: 0.4714 / OK: 1

getVertexArea
TestCase0 -- value: 0 / expected value: 0 / OK: 1
TestCase1 -- value: 0 / expected value: 0 / OK: 1
TestCase2 -- value: 0.166667 / expected value: 0.1667 / OK: 1
TestCase3 -- value: 0.333333 / expected value: 0.3333 / OK: 1
```

3 Algorithm evaluation

In this section we will compare and evaluate the behavior of mesh smoothing algorithms: Uniform Laplacian, Bilateral Normal Filtering and Guided Mesh Denoising. To compare our results we rendered the resulting meshes with flat shading. In the case of numerical comparisons we used two metrics. The first one is called mean square angular error (MSAE) which measures the difference between normals. It is defined as $MSAE = E[\angle(\mathbf{n}_d, \mathbf{n})]$, where $\mathbf{n}_d$ is the desired normal. The second one measures volume preservation. We obtain the volume difference rate regarding the desired volume. It is defined as follows: $|vol_d - vol|/vol_d$, where vol_d is the desired volume. As a first experiment we applied the algorithms to a set of meshes which are typically used in anisotropic filtering papers. We added some gaussian noise with intensity 0.2 to the original meshes and executed the algorithms over the noisy meshes. The name of the meshes are: block, fandisk, sharp sphere, twelve, bunny, julius and nicolo. The first four are meshes with sharp features and the others are smoother. Following this classification we will call set A the set of meshes with sharp features and set B the set of smoother meshes. For the set A we used the following parameters. Uniform laplacian filtering: 5 iterations. Bilateral normal filtering: 40 normal iterations, 20 vertex iterations. Guided mesh normal filtering: 40 normal iterations, 20 vertex iterations. For the set B we used the following parameters. Uniform laplacian filtering: 5 iterations. Bilateral normal filtering: 6 normal iterations, 6 vertex iterations. Guided mesh normal filtering: 6 normal iterations, 6 vertex iterations. The other parameters where computed using for example the average edge length. In Figure 2 we show the original mesh (first column), the noisy mesh (second column) and the resulting meshes after executing each algorithm. In Table 2 we show the resulting MSAE for each algorithm. Also, in Table 3 we show the volume difference rate.

As we can see, the uniform laplacian filtering does not preserve the details; it is clear in meshes of set A. Due to the shrinkage effect, volume is not preserved. Bilateral normal filtering and guided mesh denoising generate much

Table 2: MSAE

mesh	Laplacian	Bil. Normal	Guided
block	11.7668	2.44902	**2.19461**
fandisk	11.0744	2.56373	**2.20445**
sharp sphere	11.3231	9.34998	**6.2686**
twelve	8.00568	9.36095	**2.07653**
bunny	7.01866	**5.32788**	6.84292
julius	6.27873	7.2667	**7.1408**
nicolo	7.14872	**6.38959**	7.62002

Table 3: Volume difference rate

mesh	Laplacian	Bil. Normal	Guided
block	0.03506	0.00130	0.00135
fandisk	0.02450	0.00072	0.00001
sharp sphere	0.04686	0.00124	0.00115
twelve	0.02193	0.00288	0.00059
bunny	0.02032	0.00023	0.00023
julius	0.00661	0.00005	0.00013
nicolo	0.01499	0.00060	0.00682

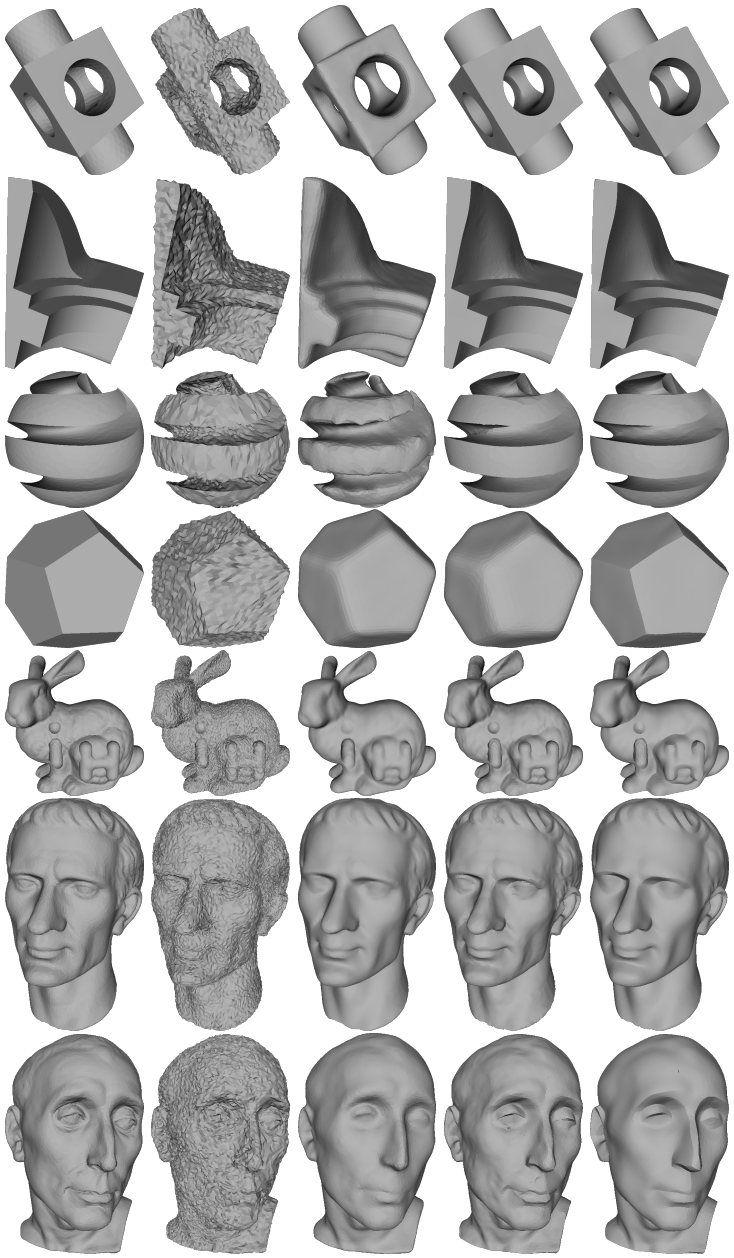

Figure 2: From top to bottom: block, fandisk, sharp sphere, twelve, bunny, julius, nicolo. First column: input mesh. Second column: noisy mesh. Third column: uniform laplacian. Fourth column: bilateral filtering. Fifth column: bilateral normal filtering. Sixth column: guided mesh denoising.

better results than the other filter, considering a visual analysis and comparing the MSAE. Also, both don't suffer shrinkage compared to the other ones, the volume difference rate is near to 0. Considering the MSAE, guided mesh normal filtering has low error than bilateral normal filtering when processing meshes with sharp features (set A), but when working with smoother meshes (set B), bilateral normal filtering is better.

Regarding the implementation, we can see that the behavior of the algorithms is correct, considering the behavior described in the corresponding papers. Numerical results have sense and visualizations works like we expected.