Distance Mesh

Frederik Gottlieb
gottlieb@diku.dk
Distance Mesh\textsuperscript{1}

Frederik Gottlieb
gottlieb@diku.dk

Abstract

In this work, a code contribution to the OpenTissue library is presented. OpenTissue is an open-source low level API for use in middleware physics and surgical simulation. This report describes both the algorithm and implementation details towards OpenTissue, where the primary focus is on library development. A simple demonstration program has been devised, to suggest that the implementation works as expected and highlight any difficulties using the developed library code.

\textsuperscript{1}This work is done as a continuation of a course in Generic Programming and Library Development – see http://isis.ku.dk/kurser/kursus23538.htm for details on the course held at Department of Computer Science, University of Copenhagen, Spring 2006.
Contents

1 Introduction 3
2 Distance Mesh Function 3
3 Background for Tetrahedra Meshes 4
4 An Evolution of Programming Paradigms 4
   4.1 Imperative Programming 5
   4.2 Data Abstraction 5
   4.3 Object-Oriented Programming 5
   4.4 Generic Programming 6
   4.5 Programming in OpenTissue 6
5 Analyzing the Previous Version 7
   5.1 Overview 7
   5.2 Conclusion 9
6 Design 10
   6.1 Policies and Traits 10
   6.2 The Application Programming Interface 12
7 Code Review 15
   7.1 The Inspection Process 16
   7.2 The Result 16
   7.3 Conclusion 17
8 Testing 17
   8.1 Unit Testing 18
   8.2 The Edge Length Function 18
   8.3 The Demonstration Program 18
   8.4 Conclusion 24

References 27
1 Introduction

The purpose of the work presented in this paper, is to provide the OpenTissue project [10] with a new implementation of the distance mesh algorithm, first presented by Persson and Strang [11]. The algorithm takes a distance function and from this calculates a near optimal mesh of the appropriate dimension. OpenTissue already has an existing implementation of the algorithm, but this previous implementation is not ideal, as the implementation is not complete, and because the programming interface to the function, is very difficult.

To address these limitations, I establish two objectives: Firstly, to develop and document a new version of the library code, and secondly, to suggest a design for the algorithm appropriate for the C++ programming language.

A note on the status of the code. Before the code can be submitted to the OpenTissue library, several quality measures must be met. At the time of this writing, not all of these have been met, but it is the intention to meet these measures. This report mostly describes the intentions, and the how to meet the quality measures. In some areas, like testing and documentation, this has not been fulfilled yet. Hopefully, by the time this article is read, the code has been submitted and all that is proposed in the report, have been done. At this point in time, the code works as intended, the API is almost finished (only very small details lacking), and the demonstration program is running.

2 Distance Mesh Function

First a brief summary of the chosen distance mesh function is presented. The function is the same as presented by Persson and Strang in [11], and for a more detailed explanation on the function, please read the article. The function is generally applicable to any dimension, but for the work presented here only the 3D case is examined.

The premisses are to tessellate an object represented by a distance function or map into a mesh of tetrahedra. The distance function is a function, that given a point in space, returns the distance to the object. If the value returned is positive, then the point is outside the object, if the value is zero, the point is on the surface, and when it is negative, the point is inside the object. From the distance function, a gradient field can be calculated, which gives the direction for a given point to the surface of the object. Problems can occur in the gradient field, where the distance to two different point on the surface is the same.

Initially a point distribution is created, and these points will make the corners of the tetrahedra. These points are then tetrahedralized, for instance by using a Delaunay triangulation. The edges in the resulting tetrahedra mesh, will then be used in a simplified spring simulation (using only repulsive forces) to push the points into place. The reason for doing the simulation is to optimize the quality of the tetrahedra in the mesh. For the purpose of this discussion, an optimal tetrahedra is one where all angles are at 60°.
If points leave the object during the simulation, they are projected back inside using the information in the gradient field. A simplified algorithm detects if any tetrahedra cuts the surface, by simply checking if the midpoint in the tetrahedra lies outside the object. This is repeated until all the points move less than a given threshold value, and points along with the resulting tetrahedra mesh is returned as the result.

3 Background for Tetrahedra Meshes

Persson and Stang [11] first described the chosen distance mesh function, and how they implemented their ideas in Matlab code. Their algorithm was based on a simulation step, to determine near optimal locations for the mesh points. It was shown that the solution found by the distance mesh algorithm can be superior to Laplacian smoothing [6]. This algorithm was ported to the OpenTissue library, and has now been chosen for improvement.

There are other methods for generating a tetrahedra mesh. A classic approach is triangulation and tetrahedralization using Delaunay triangulation and Steiner points [2]. Using this approach, the generated mesh can be optimized, using face-swapping and edge-flipping techniques.

Another approach was taken by Shimada [13, 14], where bubbles were used to determine the mesh-point locations. The bubbles could push on each other using a physical based simulation, much like the approach taken by Persson and Stang in the distance mesh function. Bossen and Heckbert [4] developed an improved method for smoothing, which they compared to Laplacian smoothing [6], and the smoothing technique used by Shimada [13, 14].

4 An Evolution of Programming Paradigms

In this section a brief overview of the evolution of programming paradigms, and how they are related to the programming language C++, will be presented. Explained are the benefits of the various paradigms, why generic programming is a good option for library development, and why C++ is an obvious choice as a programming language for this task. Lastly, the relationship between individual paradigms and library development is described.

Many programming languages exist, and some languages support several paradigms, an example language is C++. If a language supports a certain paradigm, it is not only expected that language constructs to support the paradigm are provided, but also that the compiler performs satisfactory compile- and/or run-time checks to ensure correct use of the paradigm. All language features must be cleanly and elegantly integrated into the language [15]. C++ was designed to support different programming paradigms, allowing the programmer to choose which paradigms to use in a certain application, while unused paradigms do not impose an overhead on performance.
The following paradigms, which are all supported by C++, are generally thought of as being an evolution of procedural programming, data abstraction, object-oriented programming, and generic programming. Generic programming is not only an extension to object-oriented programming, but can be applied to many other paradigms, including procedural programming. In the following sections, the characteristics of each of these paradigms will be examined.

4.1 Imperative Programming

Imperative programming, in contrast to declarative programming, uses statements to alter the state of the computer. This is commonly used in conjunction with procedural programming, where the focus is on the procedures and/or algorithms needed to perform a computation. Programs written using solely this paradigm often consist of an incomprehensible maze of functions that call each other. A program using this paradigm, does not need to grow very large before it becomes difficult to maintain and develop.

4.2 Data Abstraction

Abstraction is a fundamental aspect of programming. The more that can be abstracted, the more a programmer can grasp, and thus more complex programs can be constructed. A simple way to achieve this, is to partition the program into modules, so that each module hides its data from other modules. The programmer can thus hide the data, and provide procedures as an interface to interact with the data.

This also eliminates problems associated with global state data often found in pure imperative programs. When data are globally accessible, it can be very hard to figure out which part of the code access the data at a given point, thus leading to bugs that are hard to locate and fix.

4.3 Object-Oriented Programming

Object-oriented programming and design focuses on three primary principles: encapsulation, inheritance, and polymorphism. It is a strongly modular paradigm, that separates logic into objects which can be viewed as tiny machines, each with their own data and responsibilities. Objects can receive and handle messages from other objects, and a computation is made by parsing messages along through objects. The encapsulation part of object-oriented programming is just an application of the data abstraction paradigm.

Good object-oriented designs are easy to understand, and the modular nature of object-oriented design makes it easy for a programmer to focus on a specific area of an application, without having to understand the details of the rest.

The polymorphic behavior of object-oriented programming, along with inheritance, means that programs written using this paradigm are easy to extend with additional types and/or functionality. The exact types used in the polymorphism of object-oriented programming, is determined during the runtime of the application.
4.4 Generic Programming

Generic programming is about focusing on the central aspects of an algorithm or a data structure, and representing it independently of representation details. This is done by parameterizing the generic algorithms and data structure so they can be used for a variety of types.

In C++ the notion of generic programming is taken even this [1]. Here it is also discussed how to parameterize common functionality, like memory management, and supply a suitable memory management algorithm (which itself can be parameterized) to a data structure or algorithm that needs memory management. This way of parameterizing isolated concepts within an algorithm, is called policies. The functionality is given to the data structure or algorithm at compile time, just as you would supply a type argument.

Generic programming is not a paradigm which works on it's own. It is a way to further improve the design using object-oriented programming, imperative programming, or another paradigm. The paradigm uses, unlike object-oriented programming, static polymorphism to work out the exact types used, whereas object-oriented programming uses dynamic or run-time polymorphism. This gives type safety at compile time, instead of introducing the dreaded type-cast errors of object-oriented programming. In C++, the language construct is called a template, which is also a good indication of what generic programming can be used for. It gives a template of code to the compiler, which can be customized by a programmer to suit their needs.

4.4.1 Policies and Traits

Programmatically, policies and traits are the same thing, but they differ in concept, and how and when they are are used. Policies are used in situations, where functionality can be parameterized. This could be to provide implementations for common functionality, like memory management, or it can be to separate an algorithm into sizable chunks.

Traits on the other hand captures non-behavioral aspects, that can be parameterized, like types and constant values, which are also called type-traits and value-traits respectively.

4.5 Programming in OpenTissue

Library development must be robust, extensible, and flexible. Object-oriented programming has several interesting ways, or design patterns, where the dependency between objects (or models, data etc.) is loosely coupled [7]. An object-oriented approach would then seem obvious, but when combining object-oriented programming with generic programming, you get some interesting new options [1]. For example, new strategies on how to change the classic object-oriented design patterns into generic design patterns, and even how to extend these patterns further.

With these points aside, the Standard Template Library also supports generic programming in a wide range of different algorithms and data structures. This is expanded even

\[^{2}\text{See [12] for more information on STL.}\]
further by the Boost library [3, 8], which is also extensively used in OpenTissue.

To summarize, OpenTissue uses generic programming as its fundamental paradigm, but in combination with both object-oriented programming and imperative programming. Not all parts of OpenTissue are object-oriented; sometimes it is more obvious to simply provide a function instead of encapsulating it in an object.

5 Analyzing the Previous Version

Before the design of the new implementation is undertaken, the previous version of the implementation is analyzed, along with a discussion of which parts works, and what needs improvement. This version was in OpenTissue, called PerssonStrangT4MeshGenerator\(^3\). The implementation is closely related to the Matlab version presented by Persson and Strang [11].

5.1 Overview

The distance mesh function is quite complex, so as a good starting point, the following section provides an overview of the implementation.

5.1.1 Function Arguments

The previous implementation provided all the arguments that the Matlab version does, except \( \text{pfix} \) which represents a feature missing in the implementation.

**Distance Map** \( d \) – This argument gives the signed distance for any point to the closest point on the geometric object. This is used to test whether points are within the boundary of the object. This argument is used to calculate the gradient map in the original algorithm as described by Persson and Strang [11].

**Edge Length Function** \( h \) – For a given point, \( p \), this function returns the desired edge length for that point. This is used to determine the sizes of the resulting tetrahedrons.

**Initial Distance** \( h0 \) – The distance between points in the initial distribution of points.

**Bounding Box** \( \text{bbox} \) – The axis aligned box that contains the geometry object. This is only used while creating the initial distribution of points.

**Fixed Nodes** \( \text{pfix} \) – A list of nodes that should be fixed during the mesh generation. This argument is missing from the previous version of the implementation.

The previous implementation added the following arguments to the function:

---

\(^3\)This was included by the file, `<OpenTissue/t4mesh/util/t4mesh_persson_strang_generator.h>`
Gradient Map \( g \) – The gradient map gives the direction to the nearest border from any given point in the space. This should not be necessary, as the gradient map follows the distance map, and can be calculated from it.

Points \( \text{points} \) – This argument is used as a return argument, and holds the coordinates of the resulting tessellation.

Mesh \( \text{mesh} \) – This is also a return argument, and provides the actual tessellation.

Two features of the previous implementation, limits its application. First, all of the above parameters, along with template arguments specifying the types of the parameters, had to be provided by the programmer using the function. Second, the learning curve for using the function was steep, nearly impractical.

5.1.2 Policies

The way the previous implementation was made, follows very closely the way the Matlab implementation is presented in [11]. Parts of the implementation had been made flexible by providing the functionality as policies, yet those policies were all provided in a single version, and with hard-coded constants interchanged within the source code. A single policy is even redundant as an implementation of this has already been implemented in OpenTissue. Here follows a list of the policies in order of how they are supplied to the function.

Point Distribution Policy This policy will distribute a number of points in an axis aligned bounding box. The only implementation is a uniform distribution.

Point Rejection Policy This policy will reject certain points from the initial distribution. The implementation uses two rejection policies: First, all points outside the geometry are rejected; and secondly, points are rejected based on a probability, which is \( 1/h(p)^2 \), where \( h(p) \) is the desired edge-length at the given point \( p \).

Retriangulation Policy This policy is invoked to get a boolean value if the distribution of points has moved so much that a retriangulation is required.

Meshing Policy The policy to invoke whenever the points in the distribution needs to be retriangulated. In almost all circumstances the algorithm to be used is the Delaunay triangulation algorithm.

Filter Tetrahedra Policy The policy that rejects tetrahedrons from the solution. The implementation rejects tetrahedrons based on weather the center of the tetrahedron is outside the boundaries of the geometric object or not.

Physical Simulation Policy The policy that determines how the points move internally. This is done by providing a solution that implements a spring force, that only rejects points. That is, a spring simulation where points only are rejected not attracted.
Point Projection Policy  After the physical simulation some points might be outside
the boundaries of the geometry. This policy determines this fact, and if so, they
are projected onto the boundaries of the geometric object. There already exists an
implementation of this projection method in OpenTissue, and this will be used to
reduce redundant code.

Termination Policy  The policy that determines when the solution is acceptable and if
the iteration should be terminated. The implementation terminates the iteration
when the points have moved less than a fixed threshold.

There is no evidence that care has been taking in choosing the ordering of the template
arguments. The meshing policy for example, will most certainly always be the Delaunay
triangulation. This would normally be left as the default policy, but it is not in the end of
the template arguments. It would probably be more useful to tweak the physical simulation
policy, to get the points to move in a different way, or change how the termination policy
determines a satisfactory result.

Another problem with the implemented policies are the hard-coded constants. Take for
instance the implementation of the termination policy. This policy will end the iteration
within the main loop if the points have moved less than a certain threshold. It would
probably be useful to tweak on this threshold value, yet it is hard-coded within the imple-
mentation.

5.2 Conclusion

The previous implementation had a number of shortcomings, that needed improvement.
Foremost was readability – the name giving standard used in the implementation, was not
OpenTissue’s standard, and in several cases, the names given to identifiers didn’t inform
the reader of the intended use of the identifier.

Second, the API to the method has been crowded with too many template arguments.
Most of these had default assignments, but it was hard to use the function without having
a deeper understanding of the implementation, and all of the policies. The source code is
crowded with policy classes that are used as default template arguments, and it is easy to
loose the big picture when trying to figure out how to use the function.

The main issues that are going to be addressed in the new implementation, will be read-
ability and an easier API to the function. This also includes better documentation, better
name giving and better conformity to the OpenTissue standards.

Besides these points the code actually works (almost). It would be easier to modify the
existing code, fix the problems, rather than having to rewrite the whole function. Also the
experience gathered during the first implementation won’t be lost this way.
6 Design

The API Design must be as simple as possible – preferably a single overloaded method that can handle several different scenarios. The overloading could be done both by template specializations and/or by regular function overloading. The goal is that the method should be available to the programmer, without any need for a deeper understanding of the inner workings of the algorithm. If the programmer wants a deeper understanding, it should be a goal to make the code readable, so it would be relatively easy to understand.

Take the average programmer who just needs a method to convert a distance map into manageable tetrahedra. All that the programmer has, is a distance function of the geometric object, and all that is expected, is that a tessellation is returned. With the previous implementation it was also needed to calculate the gradient map of the distance map, and provide the edge length function. It is possible to calculate a default gradient map given the distance map, and this is exactly one of the improvements, that will be looked upon. The edge length function can be provided just as a constant.

The implementation should still be of a generic nature, such that central parts of the algorithm may be easily exchanged. Some of the existing policies will be kept, but will be made more flexible and configurable. The goal here is to provide a basic policy library that users of the function can use. It is beyond this implementation to provide an actual library of policies, only the default ones will be implemented. There will be room though, to extend with more policies if the need arise.

A feature of the implementation presented in the original article [11] is the ability to fix certain points in the mesh. This feature was lacking in the previous implementation, but will be included in the new.

6.1 Policies and Traits

A goal of the new implementation is also to provide better access to policies in the algorithm. The policies should be configurable, easier to understand and easier to extend with different functionality. Also, the gradient map of the distance function could be viewed as a trait and, as such, just be indirectly supplied along with the distance map.

The policies define the central parts of the mesh-generating algorithm – it’s important that they serve a specific role, but how they do it is up to the actual implementation. In no particular order, the following presents a list of the central aspects of the algorithm that could become policies. The names given to these new policies should not be confused with the ones in the previous implementation.

**Point Distribution Policy** This policy will – given a bounding box – place a number of points within the box. These points serve as the starting point for the tetrahedrons. How the points are distributed within the box is decided upon by the policy. In the
previous implementation this step was divided into two separate policies, the Point Distribution and the Point Rejection Policies. Actually what these two policies do is to collaborate to give a initial distribution of points.

**Triangulation Policy** This policy will take a set of points in space, and provide a triangulation (into tetrahedrons) from the set. This policy might take into account the relative movement of points since last invocation, and simply return a previously calculated result. Compared to the previous implementation, this policy combines three policies, the Retriangulation, the Meshing and the Filter Tetrahedron Policies. The reason, again, is not to capture lines of code from the original program, but instead capture an aspect and provide it as a policy for the function.

**Simulation Policy** This policy will move the points within the boundary of the geometric object based on rules defined within the policy. This policy guarantees that the points move towards an equilibrium and that no points are moved outside of the boundary after the policy has run. This policy captures two policies from the previous implementation, the Physical Simulation and the Point Projection Policies.

**Termination Policy** A policy that determines when the mesh generator has a result that is good enough. Different methods to determine this is suggested by Persson and Stang [11], amongst these a simple method that simply looks at if all nodes moved less than a threshold.

As is evident in the above listing, the library should provide functions to create the policies of choice. These could be build using building blocks matching the policies in the previous implementation, or in a different size. In any case, the functions provided should give a great deal of flexibility when creating a custom policy.

From the above policies a pseudo code example could be set up to show how the policies interact with each other:

1. Calculate initial point distribution using the Point Distribution Policy.
2. While true
   (a) Calculate the triangulation (if necessary) from the point distribution using the Triangulation Policy.
   (b) Move the points within the geometric object using the Simulation Policy.
   (c) If the solution is satisfactory according to the Termination Policy, terminate the function and return the result.

This is a very simple description of the algorithm, but it is easy to understand, and it is easy for a programmer using the function to identify which parts of the function that might need to be tweaked using customary policies.
6.2 The Application Programming Interface

The new implementation has two goals towards the interface of the function. First, it must be simple for the programmer, who just wishes to use it with default arguments without understanding the inner workings of the function. Second, the interface must give the programmer the possibilities to tweak the policies used in the function.

The way this will be implemented is to provide functions that will help configure the generator, provide the function in different overloaded cases, and provide default policies.

6.2.1 Policy Structure

The policies provide functionality, but must be supplied at compile time to the function or class that will make use of them. For this reason, the policy cannot be a dynamic class. The best option here is to provide policies as classes, with a static member function that provides the functionality. The following shows the basic way in which the policies are programmed:

```cpp
template <typename real_type, ...>
struct PolicyName
{
    static real_type do_something( args ) {
        ...
    }
}
```

This makes it possible to provide the policy through a template argument, and call the function contained within.

The drawback of this approach is that we need to call the method through the policy, something like `policy_type::do_something(...)`, instead of just writing `policy_type(...)`. With the second approach, we could pass functors (classes with the `()`-operator), function pointers, boost functions, or something similar. The reason for not doing this is to make the code more explicit – it is the intention that the passed template, or policy, has a member function that can be called. This ensures a reasonable compilation error, if a programmer supplies a bad template argument.

It would be nice, if the policies were supplied with semi-constant values, for use as threshold parameters or something similar. This could either be a field in the struct, or perhaps a template argument. The problem with this approach, is that the C++ standard don't allow real type constants (float and double) at compile time, prohibiting real types in templates and as `static const`. A solution could be to supply the constants as preprocessor macros. Macros, however, should be avoided, just think of name conflicts etc. Another solution might be to supply two integers and represent the real number as a fraction. While feasible, this is not a particular elegant programming solution. A different approach could be used,
namely value-traits, which will be examined in the following section. The problem with value traits, is that the values now will be separated away from the policy that uses them. In the scope of things, this is just a minor annoyance, that can easily be accepted when considering the gains in flexibility.

6.2.2 Template Arguments or Traits

The distance mesh function, along with the policy, depends on less than nine different types, not counting the policies. It would be possible to pass all nine type arguments (or a smaller subset) as template arguments to the function, but this reduces readability considerably. Many of these types are related; for instance we have the point container type, which represents a container with point types inside. When all the relations between the types are analyzed, it is boiled down to, in the default configuration, to two dependant types, which are called real_type and value_type. It might even be the case that these two types would be the same, but for the time being they will be kept.

The value_type is used when dealing with types outside the function, like the precision in points, the axis-aligned bounding box, etc., while the real_type is used in internal computations. To avoid loss in precision, the real_type should be a more precise type than value_type, but, if this is not the case, it should not result in flawed behavior.

Now back to the issue of the nine types, which depend on real_type and value_type in the default setup, but this might be changed if a programmer wishes to use a different setup. To accommodate this, the interface could either provide nine template arguments for the types, or a single template defined as a traits class, with all the types defined within. Another advantage of the traits class approach, is that if a programmer wants additional types in one of the policies, it would be simple to extend the traits class, with these new type definitions.

The traits class will have template arguments to define real_type and value_type, and from these, create the other types needed in the function. A default instantiation of the traits class will also be provided.

The traits class will also have so-called value-traits - a way to wrap constants at compile time, which normally can't be supplied at this time, such as double and float. The way to do this, is simply to wrap the constant value inside a static inline method in the traits class, which could look something like this:

```cpp
template <typename real_type, ...>
struct Traits {
    ...
    static inline real_type simulation_timestep() {
        return boost::numeric_cast<real_type>(0.02);
    }
    ...
}
```

If a programmer wishes to change something in the traits class, it is relatively easy to do. The distance mesh function will take the traits class as a template argument, thereby providing the means for the programmer to supply a custom traits class. If the changes are simple, it is simply a matter of deriving the traits class and overriding the type definitions and/or value-traits, that needs to be changed.

6.2.3 Distance Mesh Interface

The distance mesh function must have an interface, and how complex it should be and what functionality that should be exposed, will be discussed here. As for whether to supply the functionality as a pure function or a function contained in a structure, the same approach as for the policies is chosen here. If indeed a function is needed instead, one could wrap the member function inside a functional reference, using either STL or Boost.

The function will be provided in increasingly complex versions, from a version that only takes the distance map (almost) to a version where every little detail, regarding the function, can be specified. First, the most complex version will be described, as the other versions will be either forwarding their calls to this one, or mimic the behavior in an optimized way.

First the template arguments will be defined, in the way they are supplied to the function:

- The first argument is the type of the mesh that should be operated on. This is also the type of one of the return arguments. As this type is very independent of the rest, it is supplied by itself, and not part of the traits class. This type must conceptually have the same interface as the OpenTissue t4mesh data structure.

- Next follows the traits class, which contains all the other type definitions used in the function. This includes the type of the distance map, the edge map, the gradient map, the point type, etc.

- The four policy classes follows next, in the same order as described previously (6.1).

The template arguments are not supplied with a default value. Even though that would have made the design more flexible, the reason for not doing it was complexity and readability. All the policies take several template arguments, which themselves take template arguments, also called template template arguments. If we were to specify default values, it would be necessary to supply all the details of the template template arguments. Not only would that reduce readability, but it would also limit the policy design; all future policies would have to have the exact same template arguments.

There is no need to go through the arguments to the function, as these are the same as for the previous version, see 5.1.1, with the only addition of prefix. This is only in the most complex version. In simpler versions, the gradient map, for instance, will be calculated from the distance map.
The function will be supplied as described above, but simpler versions will also be supplied. These, simpler versions, will provide an interface with fewer options to choose between, and the simplest one will only take the `real_type` as a template argument. This will help programmers to understand the default values for the complex parameters, and the novice programmer can simply ignore these.

## 7 Code Review

A code inspection was performed on the developed library code to ensure the quality of the code [9]. The suggested way to make a code inspection is to involve at least three people and assign them to the following roles:

**Author** The author of the code to be inspected has the responsibility to briefly explain the code to the others, but has no other obligations than this. This role was assigned to me.

**Reviewer** Must read the code and focus on items presented in a checklist. The Reviewer can’t modify the code and should not concentrate on how to fix the code, but simply points out defects. This role was filled by Jesper Damkjaer.

**Moderator** This is the person responsible for scheduling the code review, scheduling meetings, and assigning roles. It is also the responsibility of the moderator to pace the review at a reasonable rate. Because of time constraints and limited resources, this role was taken by me, even though this is against the recommended procedure.

The quality of a given piece of code is hard to measure, and different programmers might have different opinions about how the quality should be measured. A good idea is to establish guidelines on how to perform the review, which areas of the code should receive focus, and what is considered quality. OpenTissue doesn’t have any such guidelines yet, so the project presented in this paper also functioned as a pilot project on how to perform quality assurance on OpenTissue code – this is the work of Damkjaer, I only provided the code to be reviewed. Before the review was started, we agreed upon the following list of focus areas, based on suggestions by McConnell [9] and our own priorities.

- **Code standard** – is the code like the code in OpenTissue, are name giving standards used, and is the code up to the standards of OpenTissue?
- **Comprehensiveness** – is the code comprehensive and readable, and is the commenting sufficient?
- **Correctness** – are there any bugs in the code?

---

4 damkjaer@diku.dk
7.1 The Inspection Process

As I had no active part in the actual review – my role was to be the author – the experiences gathered during the review are presented here, as they were presented to me\(^5\). Damkjaer, who performed the inspection, also gathered experience for a formalized procedure for code reviews in OpenTissue.

The background article by McConnell [9] is presented in the context of working on programming in a large corporation, like Microsoft. Here all the roles during the inspection are performed by internal programmers, as large corporations don’t tend to publicize their work. In OpenTissue development, this is not always the case, as external programmers can submit code and use code from OpenTissue. The point is that the code submitted might not be ready for an inspection according to the above mentioned focus areas. Instead Damkjaer proposed that the review should be performed in two stages, the first stage should be a conformative review, where readability and coding standards are of highest priority. The second stage should be a deeper, more thorough, review, with the primary focus on program behavior and correctness.

During the meeting it was also discussed how important the moderator role is in a code review on OpenTissue. The conclusion was that the role should be negligible, as this role easily could be assumed by either a reviewer or the author(s).

With all of this in mind, the review that was done on the code, at the time of this writing, only consisted of the first review step, the conformative stage. A second review with at least two reviews has been proposed to be scheduled after this report is due. This is of course done to ensure high quality when the code is submitted to the OpenTissue library.

7.2 The Result

The review focused on code standards, comprehensiveness, and to some extent design choices. The goal here was to improve the readability of the code, and in some areas improve the code according to the design choices mentioned in section 6, which had been read by the reviewer. Because of this, the following presentation of the outcome of the inspection, might be very code specific in some areas.

The distance mesh function, along with utility functions and policies, is very functional in their design, and this is implemented in the code by using struct’s with static member

---

\(^5\)Presented during the inspection meeting on May 21st 2007 at DIKU, attended by Jesper Damkjaer and Frederik Gottlieb.
functions. This is not very good object-oriented design, but the point is that the policies supply a function through the type, and this was the only option known to both the author and the reviewer. This is also the primary approach taken by Alexandrescu [1]. This point is noted, but since no real alternative exists, this is not changed.

The distance mesh function, along with policies, all use a traits class, which defaults to the `distance_traits` struct. The author meant to increase readability by introducing a macro\textsuperscript{6}, which declares type definitions for the types contained in the traits class. It was agreed upon that macros in general should be avoided, and a discussion followed focused on whether the macro could somehow work in a non intended way. In the end it was agreed upon to keep the macro, as the readability concern was met, and we could not see any problems with keeping it.

In the design it is pointed out that threshold constants in the policies should be modifiable – the inspection was performed before the introduction of value-traits. It was suggested that the policies should expose these constants as fields in the struct, so the programmer could change these if needed be. As this is not possible it was solved by using value-traits instead, see 6.2.1.

It was pointed out that the commentary in the program was not sufficient to fully understand all the functional details of the program. Also, top level commentary in form of Doxygen [5] comments should be written and improved.

Include-guards and file naming conventions were also discussed, and it was agreed upon to change the policy files to include “default” in the filename to emphasize that the code contains a default policy. If a programmer were to support an alternative policy in the future, this name would make more sense.

### 7.3 Conclusion

The code is not complete, but it is close to a satisfactory result. The main point that was made during the inspection meeting was that even though correctness was neglected during the inspection, the readability is much more important in the long run. It is not a good position to assume that a correctness inspection would identify all the bugs in the code, so it’s better to deliver code that is easy to maintain by others.

Now that the algorithm is implemented and is up and running, it is the author’s intention to have the code included in OpenTissue; the second stage of the review process will thus be performed at a later stage.

### 8 Testing

To further improve the quality of the developed library code, testing was proposed to improve the correctness. The nature of the code makes it complex to perform regular

\textsuperscript{6}\texttt{DISTANCE_MESH_TYPEDEF'S}
testing, like unit testing, so instead of testing functionality, the testing was performed on
the template type instantiation level. In short, the code was instantiated using different
types, looking for correct compilation.

A demonstration program showing how the distance mesh function can be put to use, was
made. The screen shots presented in this report were all taken from this demonstration.
This program is not an actual test, but from the result it is possible to draw conclusions
of the quality of the resulting mesh structure.

8.1 Unit Testing
A simplified test setup was used to catch the most obvious type-bugs in the program. The
code includes many template arguments, and different types, making it easy to make a
mistake when deciding upon types. The purpose of the unit testing was to find and fix
these bugs, enabling the function to run, even when the basic types are changed.

There are two levels of policies. The first level includes the default policies that are used
by the distance mesh function. The second level includes the policies that are used in the
first level policies. During the test, the second level policies were tested first, then the first
level, and if all of these tests passed, then the distance mesh function itself was tested.

The tests are performed by simply instantiating the templates with different type argu-
ments, to see if it will compile without error. As mentioned, this will not find any errors
in the calculations or the logic, but will see if the template arguments works as expected.
Basically the testing was done using a mixture of the float and the double data type.

8.2 The Edge Length Function
One of the parameters in the distance mesh function is the desired edge length function.
The purpose of this variable eluded me at first, but after rigorously testing and analyzing
the meaning of the variable was clearer. It is a desired relative edge length that the function
specifies. What this means is that if the function returns a constant – regardless (almost)
of the value – all tetrahedra will have the same desired edge length. The point here is that
the value returned by the edge length function is relative and not absolute.

Even though it would be fun to test the function with something other than a constant,
time wouldn’t permit such a test setup.

8.3 The Demonstration Program
The demonstration program was made for several reasons, one being to demonstrate how
to call the distance mesh function API, another being to provide this work with screen
shots and finally, to test the quality and performance of the function.

This section is chronological in order, meaning that as problems was discovered and fixed,
this section was written. This means that the screen shots and the problems presented,
Figure 1: A sphere made using the distance mesh function in the demonstration program. The two figures use the same view, the left showing a cross section of the sphere.

are from an earlier stage in the program. Towards the end of the section, the current state of the program is presented.

The demonstration program displays the tessellation of a given figure. In figure 1, a sphere has been tessellated, and shown in full and with a cross section through the yz-plane. The figure contains 5149 points and 34.180 tetrahedra. From the cut view it is easy to see the many tetrahedra, but in the right hand figure, the tessellation level, doesn’t seem to be as high, and many flat or planar areas appear on the surface. Another problem (which is probably related) can be seen on the left figure. On the edges of the sphere, degenerated tetrahedrons appear. These are almost planar and larger than the tetrahedra in the inner of the sphere.

Several reasons for the degenerated tetrahedra might exist: First, the points on the edges are made from uniform points, which means that in some places along the edges, more than three points could be planar. When this happens, the Delaunay triangulation might perform unexpected, and generate degenerated tetrahedra. The other plausible cause, could be that points are projected back inside the edge of the sphere, and that might contribute to the problem.

8.3.1 Point Distribution

To investigate the problem along the edges of the sphere, a random placement of points could be tried. Instead of using the default point distribution policy, which gives a uniform distribution, a policy that places the points randomly within the bounding box is used. The result is displayed in figure 2. It is clear that the sphere is more sphere-like, no strange flatness on the outside, and when the mesh is cut in yz-plane, it is clear that the degenerated tetrahedra have disappeared.
8.3.2 Springs Gone Missing

Another problem that appears, is that some of the points seem to attract each other, instead of rejecting each other. To further investigate why this happens, an experiment was set up, where the tetrahedra count was reduced. A feature implemented in the function, makes it possible to pass along the points from a previous calculation, to get a new iteration in the points. Now if you repeat this enough times, the tessellation will converge on an equilibrium. While this is done, the program reports how many tetrahedra there is in the simulation, and how many edges that are found in the spring simulation policy.

In figure 3 the four pictures display the resulting sphere after 90, 100, 110, and 120 calls to the distance mesh function. What happens is that it seems that points converge on each other. This at first puzzled me, but after examining how many springs (edges) that are found during the simulation step, this makes sense. When the calculation has reached it’s equilibrium, after more than 200 calls, there are no springs found in the mesh! This might point to an irregularity in the simulation policy, that not all edges are used in the simulation.

Figure 2: The figure shows the sphere, but now generated with a random point distribution policy. The right figure shows the sphere, where only tetrahedra where the midpoint has $x \geq 0$.

Figure 3: A sphere made using the distance mesh function in the demonstration program. The two figures use the same view, the left showing a cross section of the sphere.
8.3.3 Number of Iterations

When using the uniform point distribution policy, the tetrahedra generated will always be of a decent quality, but with the random distribution, it is visible that not all tetrahedra are nice. This is clear on figure 2 where the tetrahedra on the left, are not all well-formed. A perfect tetrahedra is one, where all edges have the same length, and the angles are all $60^\circ$. This should not be an issue with the algorithm used, so further investigation into this was performed.

The investigation disclosed a bug in the termination policy. This made the policy more agreeable than intended. The effect of the bug was that only edge points was examined for movement, instead of all internal points. Fixing this bug drastically increased the running time, but now the simulation actually works. The images displayed in figure 4 shows the sphere, a cut-through, and the sphere surrounded by an iso-mesh extracted from the same distance field. What is also apparent, is that the distance mesh function, results in the same sphere (almost) as the iso-mesh extraction function.

8.3.4 Tweaks

Running the distance mesh function on different geometric objects, varies very much in the quality produced. To get a decent result, many of the application parameters must be
tweaked to suit the geometric object.

The table in figure 5 displays the parameters used in the screen shots on pages 25-26, where the images on the left are made solely from the distance mesh function and the images on the right is the same, but with the iso-surface shown as a wire-mesh. The reason for this, is to be able to visual verify whether the distance mesh algorithm produces results inside the boundaries of the geometric object.

Only the most commonly changed parameters are shown in figure 5. All distance map data was taken from DataTissue, a part of the OpenTissue distribution [10].

All the figures was made using the uniform distribution policy. In the table figure 5, $h_0$ is the constant used in the initial distribution, $\text{dptol}$ is the threshold value used in the termination policy. $\text{mvtol}$ is the threshold value used in the triangulation policy when deciding whether or not to retriangulate. $\text{simstep}$ is the size of the time step used in the simulation policy. Points and tetrahedra count is an average on the number of time each figure has been run, which is between 5 and 10 times for each figure. Running time is provided as two values, the first indicates the average running time of the first iteration, and the second running time indicates the time for subsequent iterations.

Lets examine these images and the table above. The two pictures of the bunny shows a weakness of the algorithm. If the figure is concave, the algorithm have a hard time not making tetrahedra that crosses the boundaries of the geometric object. This is especially visible in Bunny 1, where tetrahedra are spotted between the ear and the head outside the boundary. In both pictures the tetrahedra falls outside the boundary between the hip and the body of the bunny. By increasing the number of points, we get a somewhat better result. In this case we also had to increase the movement tolerance for the termination policy, or the running time would be far greater.

Another way to solve the problem of tetrahedra extending across the boundaries of the object, is to change the policy in charge of detecting this. The policy used now is a simple test if the midpoint of the tetrahedra is outside the boundaries. As can clearly be seen several places on the bunny, this test is not sufficient on maps like the bunny.

In the lamp images, we see that as we increase the number of points, we get more details on

<table>
<thead>
<tr>
<th>Name</th>
<th>$h_0$</th>
<th>$\text{dptol}$</th>
<th>$\text{mvtol}$</th>
<th>$\text{simstep}$</th>
<th>Points</th>
<th>Tetrahedra</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bunny 1</td>
<td>0.03</td>
<td>0.001</td>
<td>0.6</td>
<td>0.02</td>
<td>7379</td>
<td>50557</td>
<td>3.11s / 2.95s</td>
</tr>
<tr>
<td>Bunny 2</td>
<td>0.025</td>
<td>0.00205</td>
<td>0.6</td>
<td>0.02</td>
<td>12778</td>
<td>85029</td>
<td>84.3s / 5.17s</td>
</tr>
<tr>
<td>Lamp 1</td>
<td>0.015</td>
<td>0.01</td>
<td>0.6</td>
<td>0.02</td>
<td>1780</td>
<td>11545</td>
<td>0.76s / 0.73s</td>
</tr>
<tr>
<td>Lamp 2</td>
<td>0.01</td>
<td>0.01</td>
<td>0.6</td>
<td>0.02</td>
<td>6271</td>
<td>41959</td>
<td>3.77s / 3.06s</td>
</tr>
<tr>
<td>Lamp 3</td>
<td>0.0075</td>
<td>0.01</td>
<td>0.6</td>
<td>0.02</td>
<td>14759</td>
<td>107884</td>
<td>12.0s / 6.86s</td>
</tr>
<tr>
<td>Dragon</td>
<td>0.015</td>
<td>0.01</td>
<td>0.6</td>
<td>0.02</td>
<td>9556</td>
<td>65165</td>
<td>7.84s / 4.0s</td>
</tr>
<tr>
<td>Cow</td>
<td>0.04</td>
<td>0.01</td>
<td>0.6</td>
<td>0.02</td>
<td>682</td>
<td>4387</td>
<td>0.27 / 0.25</td>
</tr>
</tbody>
</table>

Figure 5: Parameter data and performance measures of the distance mesh function.
the bars in the lamp. From the isometric surface of the lamp, it is clear the geometric object is disconnected – there are parts of the geometric object that are not connected to others. When the point count is low we loose this information, but as the count increases we get almost disconnected chunks in the tetrahedra mesh too – but then, the algorithm and the tetrahedra mesh structure is not geared towards disconnected objects. In the cow image, we clearly see, that with a low count, we loose all disconnected pieces.

The algorithm is relatively fast when the point count is kept low. The lamp 1 and the cow images are examples of this. Now nothing has been done to test the desired relative edge length function parameter, but if done correctly, the details on the images could be increased without increasing the point count. This is due to the fact that points would be attracted towards areas where more detail is needed, if done correctly.

Analyzing the dependence between point/tetrahedra count and running times on subsequent iterations, we can see a linear tendency, though the number of tests and different objects are too low to draw any definite conclusion on the running time, see figure 6. One thing we can conclude, is that the first run of the algorithm is very hard to predict. This depends on several parameters, for instance, the movement tolerances used in both the termination policy and the retriangulation policy can increase the number of iterations performed before the function terminates. This of course has a direct impact on running times, regardless of the number of points and tetrahedra.
8.4 Conclusion

As shown from the tests the distance mesh implementation works, but it is still hard to use. The novice programmer might still call the function and get a result, but the result that is returned is not likely to be satisfactory. The problem is that a lot of tweaking and testing is necessary for each geometric object used, and time and care should be taken in providing correct values. No guidelines can be given on the values, as this depends on a lot of factors – the size of the object, the amount of space it occupies in its bounding box, if the object is concave or convex, if it is disconnected, etc.

On the other hand, the quality of the code have increased significantly. The code is now tested and documented, and further work using the distance mesh function should be easier than before. Many of the experiences gathered during this project, will be provided as doxygen comments on the distance mesh code.

It is also shown, that the default policies are not sufficient. Among these are the uniform versus the random distribution policies. Which is better depends very much on the geometric object at hand. For the sphere, the random distribution proved superior, but as this distribution is slower to converge on an equilibrium, the uniform distribution should be tried first. Another policy that should have an alternative implementation is the filter tetrahedra policy. An alternative to testing midpoints should be implemented, and used in some cases – like the bunny.
References


