Distributed Adaptive Meshes
Technical Milestone Report

Matt Galloway - Pembroke College - mjg91@cam.ac.uk

January 17, 2007

1 Summary

The aim of this project is to develop a software library for use in computerised simulation and animation where triangulations are used to describe complex shaped objects. These triangulations may contain many millions of vertices connected together to form elements such as triangles and tetrahedra which describe the topology of the object being simulated or animated.

There are many applications for these triangulations such as simulation of physical objects or animation for use in films and computer games. In the aspect of simulating physical objects, it is common that the programmer of the physical equations for simulation will want to fracture the object along a given plane. This is not a trivial operation as updating the connectivity between elements involves many operations.

When triangulations begin to involve many millions of vertices and elements it is desirable to parallelise the processing in order to speed up the computation process. This project investigates designing an optimum data structure and the algorithms associated with such triangulations and their fracture and also making this operate in a parallel manner.

Essentially the software in this project provides the geometry functions necessary for simulation and animation on top of which a programmer can build the physics they want to simulate. This has not been investigated before with respect given to both fragmentation and parallelisation.

The software development philosophy applied to the project will be an incremental one, whereby solutions will be presented and analysed at various stages throughout the project. The first part of the project will involve designing the data structures, finding the best way that the connectivity of the mesh can be stored and making sure that there is as little duplication of data as possible. Then the fracture algorithm will be implemented before extending the algorithms to work in a parallel manner.

This report outlines the problem then explains the work completed to date and the plans for future work.

2 Introduction

Triangulations (or meshes) are usually unstructured when used for simulation and animation purposes meaning that the vertices can be arbitrarily connected together to form elements of different geometry (size and shape) as opposed to a structured mesh in which the elements are of uniform geometry. An example of such an unstructured mesh can be seen in **Figure 1**.

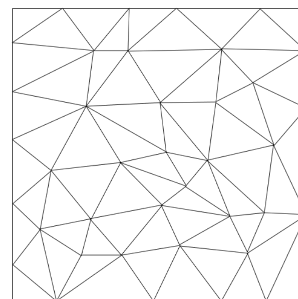

Figure 1: Unstructured Mesh

Using an unstructured mesh is particularly useful in simulation and animation because it makes it possible to describe arbitrarily shaped objects and it is possible to make the mesh more detailed in certain areas, for example when one area is of higher interest thus requiring higher precision and therefore smaller elements.

As was stated, meshes are used in computer simulation and animation. For animation purposes, the accuracy usually isn't important, for example in films or games it is not vital that objects act exactly as they would in real life. However there is evidence of a move towards life-like simulation (as in [6]) most likely due to the increasing power of computers therefore being able to provide life-like simulation in real time.

Computer simulation is the other area of interest which is involved with simulating the physical interaction of real objects using numerical techniques such as the Finite Element Method (FEM). These simulations can be of large scale such as crashing a car or of small scale such as a 3-point bend test to obtain material properties. One such small scale application is shown in [4] where simulation of the fragmentation of artificial kidney stones is achieved. Using a computer for simulation is desirable as the physical test doesn't have to be conducted which can be time consuming and expensive.

During such simulations it is quite possible that local adaptation of the mesh is required to obtain more detail in certain areas or to change the topology of the object. One such example is during a simulation the physical equations might state that a certain part of the object is to fracture. An example of how the mesh might be required to fracture is shown in **Figure 2**, using 2D triangular elements for easier understanding. The mesh must be able to cope with this fracture by maintaining connectivity and providing a means of describing the fractured faces. This process is non-trivial when the number of elements in the mesh gets as large as the many millions in today's simulations.

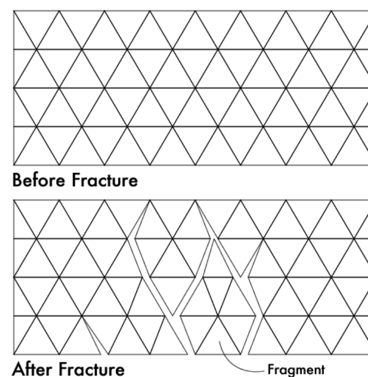

Figure 2: Mesh Fracture

This project aims to initially develop a data structure to hold mesh connectivity information in an efficient manner which will be able to maintain connectivity in the correct fashion during fracture. It also investigates the extending of the data structures and algorithms to making them run in parallel, as a parallel fragmentation algorithm has not been investigated before and it could be useful in very large simulations where the mesh is distributed amongst many processors.

3 Work Completed

3.1 Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm whereby the program is treated as *objects* which interact by passing messages between one another. It is different to traditional function-oriented programming in which the program is treated as *functions* which interact with one another. In OOP, each object is called a *class* which contain *data* and *member functions* which act upon the data of themselves and call member functions of other objects.

3.2 Initial Project Thoughts

The project will be written in C++ which is an OOP language. It was decided that this would be used because it will make it easier for future development because classes can be derived from the base classes which will be provided thus allowing the easier implementation of different types of elements (e.g. rectangular) later on. It also makes sense to use an OOP approach because the components of a mesh (i.e. vertices, elements) are naturally best described as individual objects. Using an object-oriented approach also allows the use of the C++ Standard Template Library (STL) and Boost [1] which contain many optimised objects which may be of use.

Initially in this project, time was spent reading the literature to learn about how the C++ STL and Boost works and how they might be used in the project ([1, 3, 11]). This involved learning the C++ technique of templates which enables the programmer to write a class or function but leave the types of variables unspecified until the code is compiled. This allows the programmer to write a generic algorithm for many different types without having to 'reinvent the wheel' every time the algorithm is required for a different type. This will be used extensively throughout the project including a programming technique learned from [11, chap. 15] known as *traits* which enables the specialisation of a class at compile time using a traits template parameter.

Other literature was also read ([2, 7, 9, 10]) from which current data structures were analysed and evaluated. Specifically, [7, sect. 4.3.1] gives a simple data structure for representing a mesh using vertex, edge and face definitions. The author creates their own *linked lists*[1], something which using the STL in this project will be able to improve upon as the STL defines containers which perform the list management internally at an improved efficiency. These simple data structures for vertex, edge and face are built upon in [7, sect. 4.4] where three very common data structures for representing a polyhedron are evaluated. The three data structures which are described are the 'winged-edge', 'twin-edge' and 'quad-edge' data structures. These are different ways of representing the connectivity information between the vertex, edge and face structures.

[1] A linked list is a technique for linking together groups of object which are related, using *pointers* to the adjacent objects in the list

3.3 Developing the Data Structure

The first iteration of the data structure which was analysed was taken directly from [10] where the connectivity between the objects (vertices, edges, faces and tetrahedra) is achieved by each edge, face and tetrahedron knowing which other objects it is connected to. This seemed far too much information storage as some of this information can be left out and then generated at run-time if needed. It would also be very clumsy to update all this information during fracture and therefore another solution was needed.

Secondly the ideas from [5] were used to gain some insight into how best to cut down the amount of information storage. The author describes how using a minimal set of connectivity information between objects can still be used to maintain connectivity under mesh adaptation.

Another data structure which was analysed was from [2] which stores just vertices and elements where each element knows which elements are adjacent to it and which vertices it is made from. This is a massive cut down from that proposed in [10] and therefore reduces the memory usage greatly and so this was the starting point for my own data structure. The data structure finally implemented is described in **Figure 3** where the tetrahedron at the centre has its nodes labelled to illustrate how each face is referenced (i.e. face 1 is opposite node 1). The idea is that any information about edges or faces which are required by the various algorithms can be constructed at run-time. This will lead to added complexity and thus more processing, but it also means less information is needed to be updated when the mesh is adapted, thus less processing during adaptation. Also the memory usage is reduced which is useful as then larger meshes can be processed with the same amount of physical memory.

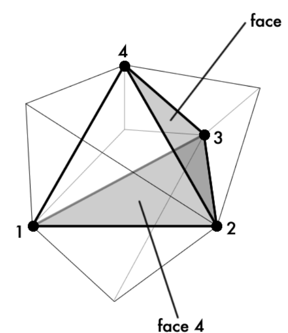

Figure 3: Tetrahedron Data Structure

A function was also written to read in a mesh from an ASCII file in the format shown in **Figure 4**. This is useful so that a mesh can be generated in an external program and then read into this software easily.

It is possible for the user to request an output from the software at any time which will be a file in the same format. This enables the user to store details about the mesh at relevant stages. For instance in a simulation one might want to store the resulting mesh after adaptation for further analysis in other programs.

NODES:
$\mathtt{ID}_0$ x_0, y_0, z_0
$\mathtt{ID}_1$ x_1, y_1, z_1
...
$\mathtt{ID}_n$ x_n, y_n, z_n

ELEMENTS:
$\mathtt{ID}_0$ i_0, j_0, k_0, l_0
$\mathtt{ID}_1$ i_1, j_1, k_1, l_1
...
$\mathtt{ID}_n$ i_n, j_n, k_n, l_n

Figure 4: Input file

3.4 Fracture Algorithm

In simulations where the mesh might want to fracture along a given path, the mesh data structure must provide the means of doing so whilst maintaining connectivity.

There are two ways of achieving this with the first being to insert special elements at faces where fracture might occur and then involving these in the calculations to check at each iteration whether or not fracture actually has occurred yet. This is cumbersome because we have to work out and program in where fracture might occur.

Therefore an adaptive approach is desired whereby the physics can tell the mesh to fracture at a given location. The mesh handles all the processing of the fracture and inserts a *cohesive element* (as described in **Figure 5**) which is a special element to store information about the fractured face for the physics to deal with ([8] shows such a cohesive element).

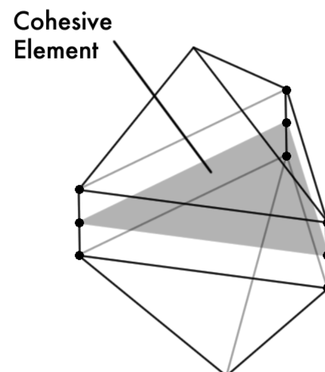

Figure 5:Cohesive Element

This approach of adding in cohesive elements is described and implemented in [10] using a C data structure and cumbersome algorithms. The approach taken in this project makes use of the STL and Boost algorithms along with a different approach to the data structure design to enable efficient fracture processing. Current testing has shown that the data structure designed works well with the currently implemented fracture algorithm.

3.5 Current Testing

Current testing has been involved with making sure that the data structure contains all necessary information to successfully store and update connectivity information during fracture and also testing the fracture algorithm. This has been achieved by creating a test program using OpenGL which can read in a mesh from a plain text file (in the format given **Figure 4**) and then fracture faces.

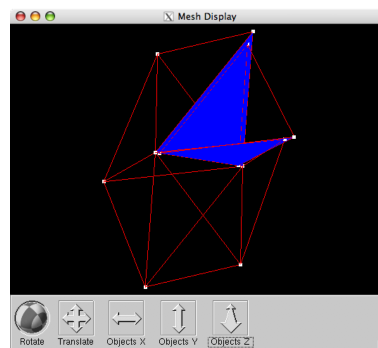

Figure 6: OpenGL Based Test Program

Elements are drawn by showing their edges drawn in red meaning a boundary edge and green meaning a non-boundary edge; cohesive elements are shown in blue. A screen-shot of this test program running on a simple mesh with 6 elements can be seen in **Figure 6**.

4 Future Work

Currently no work has been undertaken on implementing the parallelisation of the various algorithms. Work has been done in analysing current parallelisation techniques to understand the problems which will arise and thought has been put into how they might be overcome. The next part of the project will implement this parallelisation and then further test programs will be designed to show how the algorithms work in parallel. It is hoped that an actual physical simulation will be implemented, perhaps a 3-point bend

test of a material. This will prove that the algorithms work and show how the fact that an object-oriented and parallel approach taken to this project have helped make it easier for programmers of simulations.

Therefore the approach which will be taken for the following weeks will be to initially develop the way in which the algorithms will be parallelised. This will first be done using 2D triangular elements which are a lot easier to understand, then this will be extended to 3D tetrahedral elements. The resulting algorithms will be presented in the final report along with examples showing how it works.

Documentation will also be created which should be all that a programmer of a simulation needs to use this software library to help create their own software.

References

[1] Boost (2005). Boost Libraries and Documentation. http://www.boost.org/libs/.

[2] Celes, W., Paulino, G. H., and Espinha, R. (2005). A compact adjacency-based topological data structure for finite element mesh representation. *International Journal for Numerical Methods in Engineering*.

[3] Meyers, S. (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley.

[4] Mota, A., Knap, J., and Ortiz, M. (2006). Three-dimensional fracture and fragmentation of artificial kidney stones. In *Journel of Physics: Conference Series*, volume 46, pages 299–303.

[5] Nienhuys, H.-W. and van der Stappen, A. F. (2003). Maintaining mesh connectivity using a simplex-based data structure. Technical report, Utrecht University.

[6] O'Brien, J. F. and Hodgins, J. K. (1999). Graphical modeling and animation of brittle fracture. In *Computer Graphics Annual Conference Series*.

[7] O'Rourke, J. (1998). *Computational Geometry in C*. Cambridge University Press, 2nd edition.

[8] Ortiz, M. and Pandolfi, A. (1999). Finite-deformation irreversible cohesive elements for three-dimensional crack-propogation analysis. *International Journal for Numerical Methods in Engineering*, 44:1267–1282.

[9] Pandolfi, A. and Ortiz, M. (1998). Solid modelling aspects of three-dimensional fragmentation. *Engineering with Computers*, 14:287–308.

[10] Pandolfi, A. and Ortiz, M. (2002). An efficient adaptive procedure for three-dimensional fragmentation simulations. *Engineering with Computers*, 18:148–159.

[11] Vandevoorde, D. and Josuttis, N. M. (2003). *C++ Templates: The Complete Guide*. Addison-Wesley.