Marching Simplices

Philipp Schwaha* and René Heinzl†

*Shenteq s.r.o., Záhradnícka 7, 81107 Bratislava, Slovak Republic
†Institute for Microelectronics, Technische Universität Wien, Gußhausstraße 27-29, 1040 Vienna, Austria

Abstract. A multi-dimensional algorithm for surface extraction from a structured domain is presented. Topological information is used to split the *n*-cubes into simplices which are used to extract the possible configurations of new surface elements. The algorithm is intended to complement to the existing surface extraction algorithms, which are specialized to deal with a set dimension, while this algorithm will adapt to any chosen dimension.

Keywords: C++, Meta-Programming, Computational Topology
PACS: 89.20.Ff, 02.40.Pc, 07.05.Tp

INTRODUCTION

Marching cubes [1] has been a key algorithm for the rapid extraction of surfaces. It however not only has issues with topological ambiguity, but has also been handicapped by patenting issues. Both of these issues have been addressed by the development of the marching tetrahedra [2], which suffers neither patent issues or topological ambiguity.

Where marching cubes constructs its output by examining the cuts of plane with cubes, marching tetrahedra splits the cube into tetrahedra for which intersections are then evaluated. While these methods differ in this respect, their implementations are not too dissimilar. Both are customarily implemented using look-up tables to steer the generation of new elements in a run time efficient manner. These look-up tables are usually prepared before compilation of the intended application.

We present a means utilizing generic programming methods and paradigms, which allows to forgo the use of a hard coded look-up table and instead utilize higher level representations of the underlying topological information or determine a look-up table as needed. The procedure is realized using the capabilities already present in the Generic Scientific Simulation Environment (GSSE) [3–5]. In this fashion the procedure is generalized to arbitrary dimensions. This allows to supply an algorithm for surface extraction which is suitable for data of any supplied dimension, since it will adapt to the supplied dimension.

GENERIC SURFACE EXTRACTION

The algorithm is schematically illustrated in Figure 1. The original *n*-cube cell is decomposed into simplices. It is easy to obtain all the edges from the simplices generated in this fashion. The edges are essential for the surface extraction process, since the vertices which are subsequently used to spawn the extracted surface representation are derived directly from them, by means of interpolation. While it may be tempting to discard the simplex information since vertex generation is so tightly coupled to the edges, the simplices are essential for the generation of surface elements as they eliminate ambiguity. For, while the generation of surface elements within any given simplex can be done consistently, the lone set of vertices in an *n*-cube leaves too many degrees of freedom undetermined, which is also the source for ambiguity when using the marching cubes algorithm [6]. While this situation is easily containable in the three dimensional case, the effort required in higher dimensions increases dramatically [7].

The task of determining the simplices filling an *n*-cube does not have a unique solution. As is already apparent in the case of a three dimensional cube, which admits a filling consisting of either five or six tetrahedra. While the generation of a minimal number of elements is desirable, the minimization of the element count is not a driving motivation, but symmetry needs to be considered in order to enable a seamless repetition of neighboring cells. In the particular case of the three dimensional cube this condition singles out the decomposition into six tetrahedra. In more general terms the composition in which all tetrahedra share the space diagonal which crosses the *n*-cube cell meets the required symmetry conditions.

CP1281, *ICNAAM, Numerical Analysis and Applied Mathematics, International Conference 2010, Vol. III*
edited by T. E. Simos, G. Psihoyios, and Ch. Tsitouras

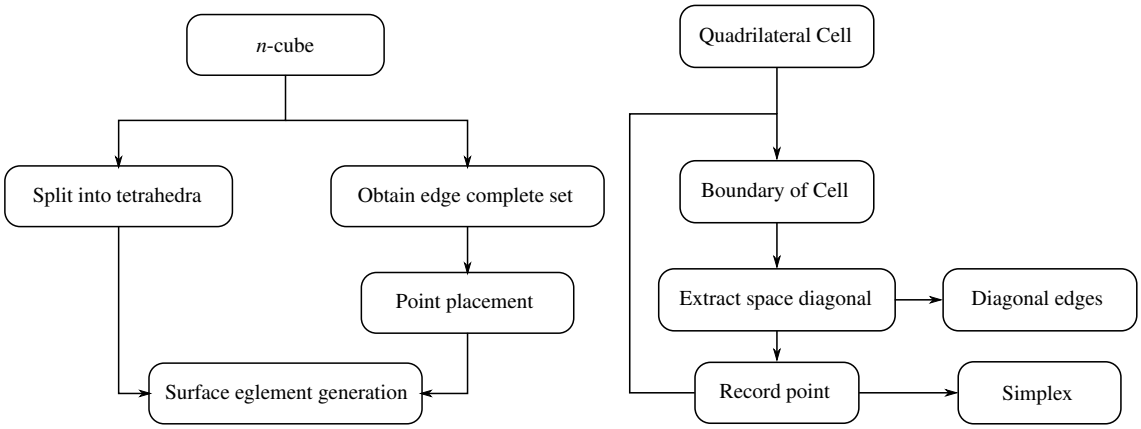

FIGURE 1. Schematic overview of the marching simplices algorithm. The overall algorithm is sketched to the left, while the right part shows the extraction of diagonal edges and simplices.

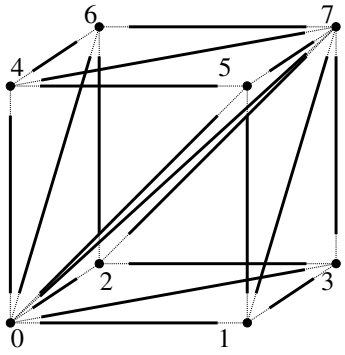

FIGURE 2. Vertices forming a 3-cube. All of the shown edges belong to tetrahedra obtained by the described algoirthm.

DECOMPOSITION ALGORITHM

The proposed algorithm utilizes information available in from the n-cube. When utilizing the topological mechanisms available in the GSSE, very detailed information regarding the arrangement of lower dimensional components is already available. The proposed algorithm therefore reuses as much of this information as possible.

Starting with a n-cube cell $\mathbf{Q}_0$. First the space diagonal $\mathbf{D}_0$ is extracted and its vertices stored for simplex construction, before the boundary elements of the n-cube $\partial\mathbf{Q}_0$, which are themselves n-cubes $\mathbf{Q}_i \in \partial\mathbf{Q}_0$ again, are queried. From each of the boundary elements $\mathbf{Q}_i$, which are of course of lower dimension than before $\dim\mathbf{Q}_0 = \dim\mathbf{Q}_i + 1$, the space diagonal $\mathbf{D}_i$ is extracted and its vertices added to the simplex again before recursing to again determining the boundary elements $\mathbf{Q}_{ij} \in \partial\mathbf{Q}_i$ of the current cell $\mathbf{Q}_i$. The procedure is applied recursively until the cells are themselves edges and thus cannot support a distinct diagonal. The space diagonals, thusly generated in conjunction with the edge boundary of the n-cube form the complete set of edges required to extract a surface representation from the n-cube. Simplices are determined while selecting the space diagonals and aggregating the encountered vertices.

An example using a 3-cube is provided to further illustrate the procedure of the algorithm. Considering an n-cube cell formed by vertices as illustrated in Figure 2. The n-cube cell is given as $\mathbf{Q}_0 = \{0,1,2,3,4,5,6,7\}$. From this the diagonal $\mathbf{D}_0 = \{0,7\}$ is extracted. The vertices $\{0\}$ and $\{7\}$ are two initial vertices of all the tetrahedra, which will be constructed, thus this edge is shared among all the generated tetrahedra. The boundary elements $\partial\mathbf{Q}_0 = \cup\mathbf{Q}_i = \{\{0,1,2,3\}, \{4,5,6,7\}, \{0,2,4,6\}, \{1,3,5,7\}, \{0,1,4,5\}, \{2,3,6,7\}\}$ are then each considered in turn, giving the diagonals $\{0,3\}$, $\{4,7\}$, $\{0,6\}$, $\{1,7\}$, $\{0,5\}$, $\{2,7\}$ respectively. As can be seen every one of the additional diagonals contains a new vertex to build a tetrahedron, when comparing to the already selected $\{0,7\}$ vertices. Adding this vertex the list of partial simplices is $\{0,3,7\}$, $\{0,4,7\}$, $\{0,6,7\}$, $\{0,1,7\}$, $\{0,5,7\}$, $\{0,2,7\}$, which are now missing a single vertex for completion. As no further diagonals can be generated as the boundary

operation on the cells $\mathbf{Q}_i$, thus no further recursion is initiated. However, the vertices of the edges are examined to complete the forming simplices. This finally gives the tetrahedra to be $\{0,1,3,7\}$, $\{0,2,3,7\}$, $\{0,4,5,7\}$, $\{0,4,6,7\}$, $\{0,2,6,7\}$, $\{0,1,5,7\}$.

The simplices are traversed letting the GSSE decompose them into their edge components. Thereby the membership relations between the simplices and the edges are available naturally. The overall complexity of the algorithm is broken down due to the reuse of the GSSE's topological operations. However, edges may be evaluated several times, as they belong to several different simplices, which comes as overhead if left unaddressed.

As each of the generated simplices is traversed the vertices originating from the edges are used for the formation of surface elements immediately, thus once all simplices have been traversed the surface has been extracted.

IMPLEMENTATION CONSIDERATIONS

The problem of splitting an n-cube into simplices is relatively expensive in terms of computational effort and grows considerable as dimension increases. It is therefore prudent to reuse the extracted topological structures as much as possible. Fortunately, the topological decomposition of an n-cube into simplices is required to be performed only once and can be reused multiple times once the information has been initialized. Furthermore, traversal of the individual simplices for every cell can be omitted when the information is condensed into an associative container, which mimics the hard coded look-up tables already in use. However, instead of only guiding subsequent formation of elements one by one, the generic approach may be used to provide all of the resulting surface elements in its own container. Since the GSSE can provide topological information it is feasible to implement the entire algorithm using meta programming techniques [8, 9].

Thus it is possible to concisely access a surface extraction in the following manner:

```
tempalte<typename InputDomainType, typename SurfaceDomainType,
         typename ParameterPackageType>
void surface_extractor(InputDomainType& input_domain,
                       SurfaceDomainType& output_surface,
                       const ParameterPackageType& parameters)
{
  typedef typename get_cell_type<InputDomainType>::type    cell_type;
  typedef typename get_cell_type<SurfaceDomainType>::type simplex_type;
  marching_simplices<cell_type, simplex_type, ParameterPackageType>::type
     march(parameters);

  gsse::traverse<at_cl>
  [
     insert(output_surface, march(_1) )
  ] ( domain ) ;
}
```

It utilizes the functional paradigm in the spirit of Boost Phoenix [10, 11] which is enabled by GSSE facilities for traversal and extends the GSSE by a surface extraction mechanism. The parameter pack can be used to supply a prescription for interpolation, so as to provide an additional degree of adaptivity.

```
template<typename ContainerType>
struct generator
{
   ContainerType return_template;
   template<typename InitialisationType>
   generator(InitialisationType*)
   {  ...  }
   template<typename QuadCellType>
   ContainerType operator()(const QuadCellType& quad) const
   {  ...  }
};
```

The `InitialisationType` encodes the instructions to initialize the `return_template` at construction time. Subsequent calls to the `operator()` uses this information to assemble and return the correct surface elements by remapping the representation, which is local to the cell, to the global indices. By initializing the member variable in this fashion in a templated constructor, where the type is utilized for control enables the structs to share the same type and thus be easily stored in the associative container, while at the same time returning different values at evaluation time.

Since it is not possible to invoke the templated constructor explicitly by the compiler needs to deduce the type automatically, a dummy pointer argument is used, whereby any instantiation of the passed type can be avoided by using

```
generator( (type*) 0 )
```

For compatibility with the containers in the STL, a default constructor is also required.

It should go unnoted that the size of the associative container increases quickly with increasing dimension if symmetry is not utilized to reduce the number of different cases. This can be accomplished by establishing a canonical ordering which is then mapped to the rest by permutations corresponding to the symmetries.

CONCLUSION

An algorithm of decomposing an n-cube into simplices in arbitrary dimension has been discussed. It builds upon topological information already available and extends the capabilities of the capabilities of the GSSE to allow for surface extractions of arbitrary dimensions. The chosen approach evaluates the expensive combinatorial components of the decomposition and element generation at compile time in order to ease run time evaluations. The proposed algorithm is not intended to be a replacement of already established surface extraction methodologies of marching cubes or marching tetrahedra, but rather to complement them, since it provides a means for surface extraction which adapts to any required dimension, instead of the implementations specific for a single dimension. The used approach to implementation makes it easy to select more specific algorithms or implementations where required.

ACKNOWLEDGMENTS

We are greatful to Franz Stimpfl and Josef Weinbub for their invaluable support and discussions. This work has been supported by the Austrian Science Fund FWF, project P19532-N13.

REFERENCES

1. W. E. Lorensen, and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," ACM, New York, NY, USA, 1987, vol. 21, pp. 163–169, ISSN 0097-8930.
2. P. Shirley, and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," ACM, New York, NY, USA, 1990, vol. 24, pp. 63–70, ISSN 0097-8930.
3. R. Heinzl, *Concepts for Scientific Computing*, Dissertation, Technische Universität Wien, Austria (2007).
4. R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr, "Parallel Library-Centric Application Design by a Generic Scientific Simulation Environment," in *Proc. of the POOSC*, Paphos, Cyprus, 2008.
5. R. Heinzl, and P. Schwaha, "A generic topology library," 2009, ISSN 0167-6423, URL `http://www.sciencedirect.com/science/InProceedings/B6V17-4XBR4GX-1/2/339efd429eaae7140a250e3b8a0c71b8`.
6. G. M. Nielson, and B. Hamann, "The asymptotic decider: resolving the ambiguity in marching cubes," in *VIS '91: Proceedings of the 2nd conference on Visualization '91*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pp. 83–91, ISBN 0-8186-2245-8.
7. A. J. Zomorodian, "Topology for Computing," in *Cambridge Monographs on Applied and Computational Mathematics*, 2005.
8. T. L. Veldhuizen, "Using C++ Template Metaprograms," 1995, vol. 7, pp. 36–43, ISSN 1040-6042, reprinted in C++ Gems, ed. Stanley Lippman.
9. D. Abrahams, and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*, Addison-Wesley, 2004, ISBN 0321227255.
10. Boost C++ Libraries, *1.43*, URL `http://www.boost.org`, http://www.boost.org.
11. Boost, *Boost Phoenix 2.0*, URL `http://www.boost.org/libs/spirit/phoenix`, http://www.boost.org/libs/spirit/phoenix.