A Computer-Assisted Proof of Correctness of a Marching Cubes Algorithm

Andrey N. Chernikov and Jing Xu

Department of Computer Science, Old Dominion University, Norfolk, VA, USA
{achernik,jxu}@cs.odu.edu

Summary. The Marching Cubes algorithm is a well known and widely used approach for extracting a triangulated isosurface from a three-dimensional rectilinear grid of uniformly sampled data values. The algorithm relies on a large manually constructed table which exhaustively enumerates all possible patterns in which the isosurface can intersect a cubical cell of the grid. For each pattern the table contains the local connectivity of the triangles. The construction of this table is labor intensive and error prone. Indeed, the original paper allowed for topological holes in the surface. This problem was later fixed by several authors, however a formal proof of correctness to our knowledge was never presented. In our opinion the most reliable approach to constructing a formal proof for this algorithm is to write a computer program which checks that all the entries in the table satisfy some sufficient condition of correctness. In this paper we present our formal proof which follows this approach, developed with the Coq proof assistant software. The script of our proof can be executed by Coq which verifies that the proof is logically correct, in the sense that its conclusions indeed logically follow from the assumptions. Coq offers a number of helpful features that automate proof development. However, Coq cannot check that the development corresponds to the problem we wish to solve, therefore, this correspondence is elaborated upon in this paper. Our complete Coq development is available online.[1]

1 Introduction

The Marching Cubes algorithm is used in a large number of applications for three-dimensional surface representation and visualization in graphics, finite element simulations, medical image computing, and other areas [1, 2]. This algorithm was originally proposed by Lorensen and Cline [3] in 1987. The algorithm processes each cubical cell of the sampled rectilinear grid one by

[1] http://sourceforge.net/projects/coq-mc

J. Sarrate & M. Staten (eds.), *Proceedings of the 22nd International Meshing Roundtable*,
DOI: 10.1007/978-3-319-02335-9_28,

one, and, by examining the values in its corners, identifies the edges of the cell that intersect the isosurface. Then the algorithm uses a predetermined table of triangle connectivity to construct the local patches of the isosurface inside every cubical cell. The union of the isosurface patches from all cubical cells makes up the final result. The exact coordinates of the intersections of the isosurface with grid edges are determined by linear interpolation.

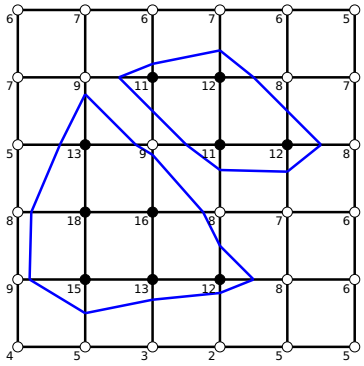

Fig. 1 A two-dimensional example of the use of a Marching Squares algorithm. The resulting isocontour (blue) corresponds to an isovalue $\xi = 10$. The nodes of the rectilinear grid corresponding to values less than ξ are shown with white circles, and the nodes corresponding to values greater than ξ are shown with black circles.

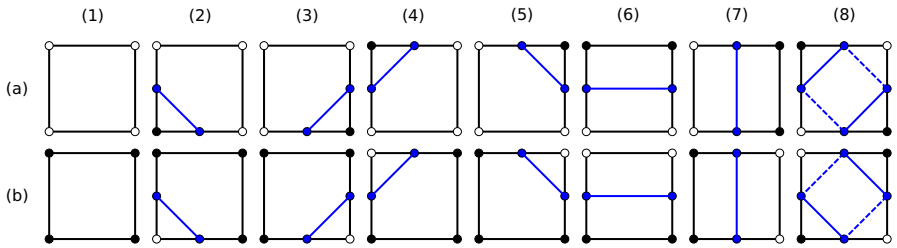

Fig. 2 Two-dimensional rules for creating intersection vertices and intersection edges. Blue circles and segments show the intersection vertices and intersection edges created by the algorithm. Solid edges correspond to the set of rules analyzed in this paper, while dashed edges show another feasible set of rules.

In Fig. 1 we show an example of the application of a two-dimensional Marching Squares algorithm to a simple data grid. The table we used to create the segments of the isocontour was published previously [1] and is shown in Fig. 2. Each cell has four corners, therefore the total number of distinct cases is $2^4 = 16$. In a three-dimensional grid each cube has eight

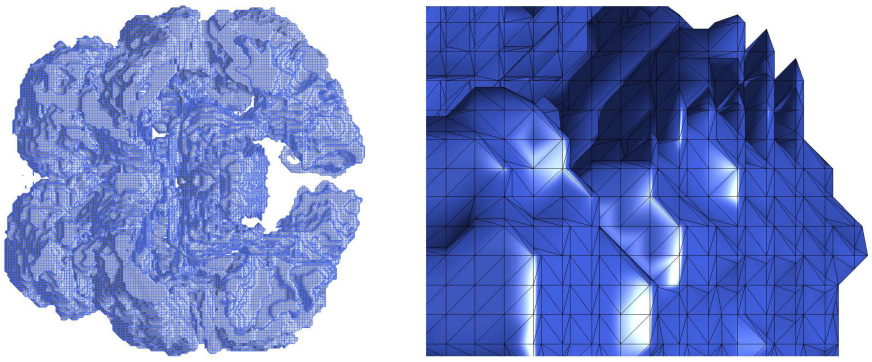

Fig. 3 A three-dimensional example of the use of a Marching Cubes algorithm: an isosurface computed using a Matlab Marching Cubes implementation [4] with a part of the SPL-PNL brain atlas [5]. **Left:** A zoomed out view. **Right:** A zoomed in view.

corners, and the total number of cases is $2^8 = 256$.[2] In Fig. 3 we show an example of a three-dimensional Marching Cubes surface constructed using a Matlab implementation [4] which we analyze in this paper.

The resemblance of the resulting isosurface to the true surface is usually measured in terms of their geometric and/or topological proximity. The stronger properties of the true surface are known, the tighter proximity conditions can be proven. In the absence of any information of the true surface, a minimal correctness requirement we can expect of a Marching Squares/Cubes algorithm is the following: all the nodes of the rectilinear grid with values less than ξ be separated by the resulting isocontours/isosurfaces from all the nodes with values greater than ξ. In this paper, for the three-dimensional algorithm, we refine this requirement down to two components, based on dimensionality:

1. *Two- and three-dimensional cohesion*, i.e., every axis-aligned plane of the three-dimensional rectilinear grid (passing through the nodes) contains zero or more isocontours from the three-dimensional Marching Cubes isosurfaces that are a correct output of a two-dimensional Marching Squares algorithm.[3]
2. *Water-tightness*, i.e., the absence of holes. The output of the Marching Cubes algorithm is required to consists of zero or more water-tight triangulated isosurfaces.

[2] In this analysis we treat the case of the sampled value being equal to the isovalue together with the case when it is greater than the isovalue. In an implementation this simplifying assumption can lead to some triangles being squeezed to an edge or a point. Such triangles can be easily pruned out by a post-processing step.

[3] The correctness of the two-dimensional Marching Squares algorithm based on the rules shown in Fig. 2 easily follows through observation and a dimensional regression of these requirements.

The first requirement ensures the separation in the directions of the axes, and the second extends it to all other directions.

One way to reason about the properties of the resulting isosurfaces is through manual examination of each of the 256 templates in the lookup table, which is labor intensive and error prone. The authors of the original paper [3] reduce the complexity of the algorithm through exploring two types of symmetry: grouping two cases with the opposite relations to the isovalue in all corners into one case, and also grouping rotationally symmetric cases. Unfortunately, as it was later pointed out [6–11], some symmetric cases cannot be treated as one case, as we show in Fig. 4. The authors [6–9, 11] state that they solved this problem, each by their own extension of the lookup table, however they do not provide formal proofs. Overall, we argue that in order to gain a high level of confidence in the correctness of a Marching Cubes algorithm, all cases need to be examined disregarding the perceived symmetry. Furthermore, considering the possibility of human error, such a proof is more reliable if based on an exhaustive verification performed by a computer.

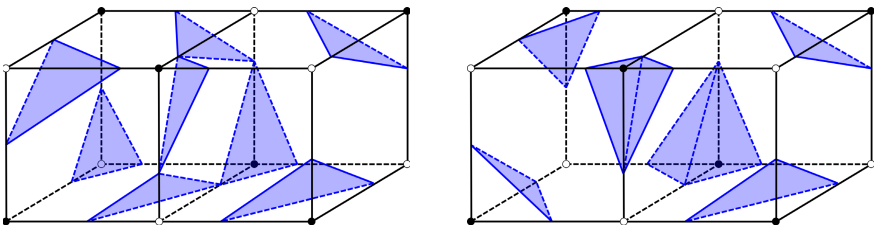

Fig. 4 Two cubes of a sampled grid sharing a common face. The corresponding corners of the cubes have pairwise opposite relations to the isovalue. **Left:** The sets of triangles created by the application of the same triangulation pattern to both cubes form a hole in the surface at the shared face. Both cubes correspond to a single case (13) in the original paper [3] which combines these two cubes into one case due to symmetry. **Right:** The triangulations are consistent in the shared face.

Since this proof requires an enumeration and a verification of a large number of cases, we decided that we need to delegate this task to a computer. One option was to write a program in a conventional imperative language like C/C++, Python, or any other. However, there is a possibility that such a program may introduce mistakes of its own. A much higher degree of assurances is offered by proof assistant software programs, which are designed specifically for proof purposes. From a number of proof assistants [12–22] we chose Coq since it appealed to us with its user friendly interface, a solid supporting library, and comprehensive documentation.

Below we describe our Coq [23] script written in a functional programming language Gallina based on a formal language Calculus of Inductive Constructions [14]. In a number of respects this language is more restrictive than the

conventional imperative programming languages, and even some other functional languages, and therefore allows for sound logical reasoning over its constructs. For example, arguments to functions are passed only by value, case analysis must always be exhaustive, and the termination of recursive functions has to be protected by guard conditions. The script of our proof can be executed by Coq interpreter which verifies that the proof is logically consistent, in the sense that its conclusions indeed logically follow from the assumptions. Coq offers a number of helpful features that facilitate proof development, such as support of data types, a comprehensive set of tactics for proving theorems, and an integrated development environment. However, Coq cannot check if the development corresponds to the problem we wish to solve; therefore, this correspondence is elaborated upon in this paper. We show and describe the key parts of our development. We introduce some important features of the language on the as-needed basis. For further treatment of Coq the reader can consult a number of excellent tutorials [14, 24, 25]. Due to the limited space, we could not include our entire script in this paper. The reader is welcome to download and execute the complete script.

Computer-assisted proofs in Coq have been used previously to support solutions of mesh generation and other geometric problems. Dufourd and Bertot [26] presented a proof of correctness of a planar Delaunay triangulation algorithm. Gonthier proved the Four-Color Theorem [27]. Dufourd [28] developed a hypermap framework for computer-aided proofs in surface subdivisions. He uses this framework to prove the genus theorem and the Euler's formula as its corollary. Brun et al. [29] designed a two-dimensional convex hull algorithm based on hypermaps and proved its correctness. A computer-assisted proof of dihedral angle bounds for a three-dimensional tetrahedral meshing algorithm was performed by Labelle and Shewchuk [30], although the programming language was not specified.

The rest of the paper is organized as follows. In Section 2 we briefly describe the classical Marching Cubes algorithm. In Section 3 we introduce the naming conventions and the basic Coq definitions required for the proof. In Section 4 we describe our Coq proof of two- and three-dimensional cohesion. Section 5 presents our Coq proof of surface water-tightness. Section 6 concludes the paper.

2 Classical Algorithm

The main steps of the classical Marching Cubes algorithm [3, 6–9, 11] are shown in pseudocode in Fig. 5. The function TableLookup(*index*) queries a manually constructed table with a key composed of eight bits, each bit corresponding to the result of the test, $F(x) \geq \xi$ or $F(x) < \xi$, in one of the eight corners of cube b. Consider the notation shown in Fig. 6 which is used in the implementation we study here. Let c_j, $j = 0, \ldots, 7$, be the corners of the cube. Let $i_j \in \{0, 1\}$ be the corresponding colors: black (or 1) for $F(x) \geq \xi$,

Algorithm MarchingCubes(I, ξ)
Input: I is a three-dimensional image, i.e., a rectilinear grid of points $G \subset \mathbb{R}^3$ along with a mapping $F : G \to \mathbb{R}$, and an isovalue $\xi \in \mathbb{R}$
Output: A triangular surface M embedded in $\mathbb{R}^3$ that interpolates the set $\{x \in \mathbb{R}^3 \mid F(x) = \xi\}$

$M \longleftarrow \emptyset$
For each point x in G, determine whether $F(x) \geq \xi$ or $F(x) < \xi$
Compute the set B of cubes by connecting adjacent points in G
for each $b \in B$
    $index \longleftarrow$ Index(b)
    $M \longleftarrow M \cup$ TableLookup($index$)
endfor
Compute vertex coordinates in M by interpolation
return M

Fig. 5 A high level description of the Marching Cubes algorithm

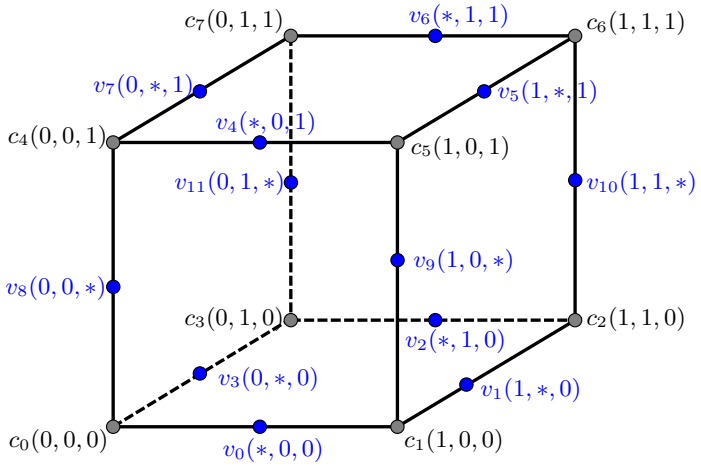

Fig. 6 Ordering and naming conventions for cube corners and cut-vertices

and white (or 0) for $F(x) < \xi$. Then the index of the table entry for a cube can be computed as

$$index = \sum_{j=0}^{7} i_j 2^j. \tag{1}$$

For the example shown in Fig. 7, $index = 213$. The return value of the function call TableLookup(213) is shown in the figure. It is a set of five triangles defined by their vertices, where each vertex is located on one edge of the cube according to the convention.

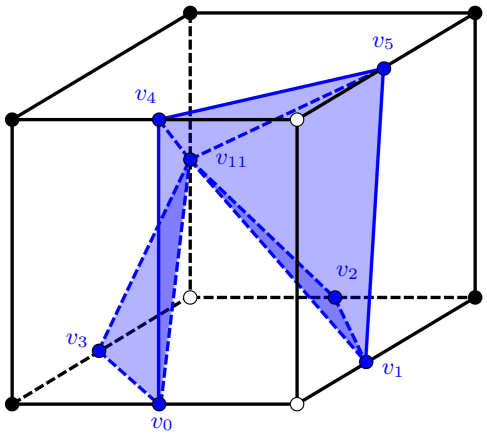

Fig. 7 An example of an intersection pattern and the corresponding triangulation: (v_{11}, v_3, v_0), (v_{11}, v_0, v_4), (v_{11}, v_4, v_5), (v_{11}, v_5, v_1), (v_{11}, v_1, v_2)

3 Definitions and Setup

In our entire development we work with a single cube which represents any cube of the sampled grid. We call it a *generic cube* because our proofs are valid for any combination of the sampled values in the corners of this cube and any isovalue. We call the points in the corners of the generic cube *cube corners*, and the points in the intersection of the resulting isosurface with the edges of the generic cube *cut-vertices* (or simply *vertices*). A *cut-edge* is an edge in one of the faces of the generic cube that connects two cut-vertices. Both cut-vertices and cut-edges can be either *prospective* if we do not know whether or not they are created, or *created* if we do have an affirmative answer.

In our Coq script we start with importing some standard libraries:

```
Require Import Bool Arith List.
```

These and other standard libraries in Coq provide proven data types and functions that we can use in our own developments, similar to other programming languages.

We then introduce our own data type *Dimension* that enumerates the names of the coordinate axes of the three-dimensional grid:

```
Inductive Dimension := Dimension_X | Dimension_Y | Dimension_Z.
```

The keyword `Inductive` indicates that here we define a finite number of ways (constructors) to build a data item of type *Dimension*. Furthermore, no other ways to build a value of type *Dimension* are allowed. In this case the constructors are simply constants naming each dimension, however they could be functions accepting parameters in more involved definitions.

Similarly, we define a data type named *Insideness* to communicate the two possible results of the test $F(c) \geq \xi$ or $F(c) < \xi$ for a grid corner c:

```
Inductive Insideness : Set := Insideness_Inside | Insideness_Outside.
```

The following function converts a list of *Insideness* values to the corresponding natural index using formula (1):

```
Fixpoint Insideness_2Index (I : list Insideness) : nat :=
  match I with
  | nil ⇒ 0
  | i :: I' ⇒
    match i with
    | Insideness_Inside ⇒ 1
    | Insideness_Outside ⇒ 0
    end +
    2 × (Insideness_2Index I')
  end.
```

Here the `Fixpoint` keyword indicates that the function is recursive. Since Coq does not provide the conventional loop constructs, the use of recursion is the standard way to iterate over a list. Coq always checks that a recursive function satisfies a *guard condition* which ensures that the function returns after a finite number of recursive calls. In this example the guard condition is satisfied by the fact that list I decreases in length at every recursive call. This reduction in size of I is performed by the `match` construct which decomposes the list into its head i and tail I'. Coq also verifies that case analysis in a `match` expression is exhaustive, and therefore allows to argue about all possible ways to construct a value of a particular inductive type.

We also define a data type for the coordinates of the corners of the generic cube. We only need two values:

```
Inductive Coord : Set := Coord_Zero | Coord_One.
```

The following function compares two values of type *Coord* for equality and returns *true* or *false* accordingly:

```
Definition Coord_IsEqual (a b : Coord) : bool :=
  match a, b with
  | Coord_Zero, Coord_Zero
  | Coord_One , Coord_One ⇒ true
  | _ , _ ⇒ false
  end.
```

A standalone underscore in a `match` expression matches any value and is used as a wildcard. The `match` clauses are evaluated in the order written, and a value is returned as soon as a fitting expression is found.

The *option* data type is standard in Coq and allows us to add a convenient constructor *None* to any type A, which can be used as a return value for search-like functions:

```
Inductive option (A : Type) : Type :=
  | Some : A → option A
  | None : option A.
```

In this case the first constructor (*Some*) is a function which takes a value of type A and returns a value of type *option A* with the name *Some* embedded in the value. The second constructor (*None*) does not receive any parameters and returns a value of type *option A* which simply consists of its name *None*. An analogy from imperative programming languages is the special treatment of pointer NULL in C/C++. For example, if the parameter data type A is specified as *Coord*, then we have a data type *option Coord* which admits three values: *Some Coord_Zero*, *Some Coord_One*, and *None*. This is convenient for our manipulations with corners, edges, and faces of the generic cube, as can be seen below.

To compare for equality two values of type *option Coord* we use the following function:

```
Definition CoordOption_IsEqual (A B : option Coord) : bool :=
  match A, B with
  | Some a, Some b ⇒ Coord_IsEqual a b
  | None, None ⇒ true
  | _ , _ ⇒ false
  end.
```

Now we can define a data type for three-dimensional coordinates:

```
Inductive CoordOption3 : Set :=
  CoordOption3_Cons :
    option Coord → option Coord → option Coord → CoordOption3.
```

Here the constructor is a function which takes three values of type *Coord* as parameters and builds a value of type *CoordOption3*. Any of the three components of a value of this data type can assume a value of *None*, represented as $*$ in Fig. 6. For example, the edge of the cube between corners $c_0(0,0,0)$ and $c_1(1,0,0)$ can be considered having coordinate $(*,0,0)$. The face defined by corners $c_0(0,0,0)$, $c_1(1,0,0)$, $c_2(1,1,0)$, and $c_3(0,1,0)$ can be considered as having coordinate $(*,*,0)$. The whole cube can be thought of as having coordinate $(*,*,*)$. Depending on the point of view, the value of *None* in this context can be also though of as "Any".

The definition of the function deciding equality of two *CoordOption3* values is based on the component-wise comparison:

```
Definition CoordOption3_IsEqual (A B : CoordOption3) : bool :=
  match A, B with
    CoordOption3_Cons x1 y1 z1,
    CoordOption3_Cons x2 y2 z2 ⇒
      (CoordOption_IsEqual x1 x2) &&
      (CoordOption_IsEqual y1 y2) &&
      (CoordOption_IsEqual z1 z2)
  end.
```

We will also be using data type *Location* which is a convenient structure for representing a face of the generic cube:

```
Inductive Location : Set :=
  Location_Cons : Dimension → Coord → Location.
```

The following function finds the common location (i.e., a face of the cube), if any, of two *CoordOption3* (i.e., edges) parameters:

```
Definition Location_GetCommon
           (A B : CoordOption3) : option Location :=
  let f a b d :=
    match a, b with
    | Some u, Some v ⇒
        if Coord_IsEqual u v
        then Some (Location_Cons d u)
        else None
    | _ , _ ⇒ None
    end
  in
  match A, B with CoordOption3_Cons x1 y1 z1,
                  CoordOption3_Cons x2 y2 z2 ⇒
    let fx := f x1 x2 Dimension_X in
    let fy := f y1 y2 Dimension_Y in
    let fz := f z1 z2 Dimension_Z in
    match fx, fy, fz with
    | Some _, None , None ⇒ fx
    | None , Some _, None ⇒ fy
    | None , None , Some _ ⇒ fz
    | _ , _ , _ ⇒ None
    end
  end.
```

For compactness and esthetics we substitute long and/or repetitive terms using local binding to short names with the `let-in` construct.

By now we have built the preliminaries for working with the triangles of the resulting surface. First we define an edge of a triangle as a pair of values of type *CoordOption3*:

```
Inductive TriangleEdge : Set :=
  TriangleEdge_Cons : CoordOption3 → CoordOption3 →
TriangleEdge.
```

The following function checks two *TriangleEdge*s for equality:

```
Definition TriangleEdge_IsEqual (E1 E2 : TriangleEdge) : bool :=
  match E1, E2 with TriangleEdge_Cons A1 B1,
                    TriangleEdge_Cons A2 B2 ⇒
    let f x y := (CoordOption3_IsEqual A1 x) &&
                 (CoordOption3_IsEqual B1 y)
    in (f A2 B2) || (f B2 A2)
  end.
```

We define a *Triangle* as a triple of the coordinates of its vertices. The vertices of the triangles in our problem are cut-vertices which are defined by the corresponding edges of the generic cube.

```
Inductive Triangle : Set :=
  Triangle_Cons :
    CoordOption3 → CoordOption3 → CoordOption3 → Triangle.
```

At this point we are ready to define the table of triangle connectivity. We wrote a short Matlab routine to transform the table developed previously [4] into the following function:

```
Definition TriangleList_Get (n : nat) : list Triangle :=
  let g u v w := Triangle_Cons u v w in
  let f x y z := CoordOption3_Cons x y z in
  let a := (Some Coord_Zero) in
  let b := (Some Coord_One) in
  let c := None in
  match n with
  | 0 ⇒ [ ]
  | 1 ⇒ [ g (f c a a) (f a a c) (f a c a) ]
  | 2 ⇒ [ g (f c a a) (f b c a) (f b a c) ]
  | 3 ⇒ [ g (f b c a) (f a a c) (f a c a); g (f b a c) (f a a c) (f b c a) ]
    ...
  | 254 ⇒ [ g (f c a a) (f a c a) (f a a c) ]
  | 255 ⇒ [ ]
  | _ ⇒ [ ]
  end.
```

The dots in this and other displays are used in place of some omitted code. The parameter n to this function is the case number, computed with function *Insideness_2Index*, and the return value is the corresponding list of triangles.

4 Proof of Two- and Three-Dimensional Cohesion

Fig. 2 shows the rules for creating cut-edges and cut-vertices in any face of the generic cube. Black circles represent the corners marked with *Insideness_Inside* and white circles represent the corners marked with *Insideness_Outside*. We organized the figure in two rows: the *Insideness* values in the bottom row (b) are the opposite of the *Insideness* values in the top row (a). These rules satisfy two fundamental requirements:

- A cut-vertex is created in an edge of the grid if and only if this edge has opposite *Insideness* values at its ends.
- In every case all the *Insideness_Inside* corners are completely separated by cut-vertices and cut-edges from all the *Insideness_Outside* corners. In each column, except the last one, there is just one way to satisfy this requirement, that is followed. In the last column each of the two cases, (a) and (b), admits two solutions, and therefore we could have four different sets of two-dimensional rules. However, to avoid creating holes, we should always use either the pair of edges that intersect the diagonal between the *Insideness_Inside* corners, or the pair of edges that intersect the diagonal between the *Insideness_Outside* corners. The former pair of edges (used in the implementation we study here) is shown with the solid lines, and the latter is shown with the dashed lines.

We encoded the two-dimensional rules shown in Fig. 2 with the following function:

```
Definition CutEdge2D_Exists
           (U V : CoordOption3)
           (I : list Insideness) : bool :=   ...
```

Given two potential cut-vertices, defined by their respective coordinates U and V, and an *Insideness* configuration I, this function returns a boolean value indicating whether or not the cut-edge between U and V is created according to the two-dimensional rules.

Below we are going to prove that the set of cut-edges returned by the function *CutEdge2D_Exists* is exactly the same as the set of cut-edges created by the triangulation returned by the function *TriangleList_Get*, for each of the 256 lists of eight *Insideness* values. We are going to break this goal into two subgoals, each verifying that one set is a subset (including set equality) of the other.

First, we define the function which checks that the three-dimensional polygons (edges of the triangles lying in the faces of the generic cube) are a subset

of the cut-edges specified by the two-dimensional rules, for a given list I of *Insideness* values:

```
Definition Polygons3D_Subset_CutEdges2D
           (I : list Insideness) : bool :=
  let f x y :=
    match Location_GetCommon x y with
    | Some _ ⇒ CutEdge2D_Exists x y I
    | None ⇒ true
    end
  in
  let fix g T :=
    match T with
    | nil ⇒ true
    | (Triangle_Cons A B C) :: T' ⇒
      (f A B) && (f B C) && (f C A) && (g T')
    end
  in
  g (TriangleList_Get (Insideness_2Index I)).
```

Here the local function f checks if the edge (x, y) lies in a face of the cube, and, if affirmative, if this edge is specified by the function *CutEdge2D_Exists*. The second local function g (recursive as indicated by the keyword `fix`) iterates over the list of triangles returned by the function *TriangleList_Get* and calls function f for each edge of each triangle.

Now we are going to define the other part of the subset verification. We start with listing the coordinates of all the potential cut-vertices:

```
Definition AllVertices : list CoordOption3 := [
  CoordOption3_Cons None (Some Coord_Zero) (Some Coord_Zero);
  CoordOption3_Cons (Some Coord_One) None (Some Coord_Zero);
    ...
  CoordOption3_Cons (Some Coord_Zero) (Some Coord_One) None
].
```

Then we define a function which checks that the cut-edges specified by the two-dimensional rules are a subset of the edges of the triangles lying in the faces of the cube:

```
Definition CutEdges2D_Subset_Polygons3D
           (I : list Insideness) : bool :=
  let T := TriangleList_Get (Insideness_2Index I) in
  let fix f u V :=
    match V with
    | nil ⇒ true
```

```
    | v :: V' ⇒
      let b := f u V' in
      if CutEdge2D_Exists u v I
      then
        let E := TriangleEdge_Cons u v in
        (beq_nat (TriangleEdge_Count T E) 1) && b
      else b
    end
  in
  let fix g V :=
    match V with
    | nil ⇒ true
    | v :: V' ⇒ (f v V') && (g V')
    end
  in
  g AllVertices.
```

Again, we have two local functions. The first recursive function *f*, given a potential cut-vertex *u* and a list of potential cut-vertices *V*, for each *v* from *V* checks if edge (*u*,*v*) is defined by the function *CutEdge2D_Exists*. If affirmative, it also checks if (*u*,*v*) appears exactly once on the list of triangles returned by the function *TriangleList_Get*. The second recursive function *g* calls *f* for each vertex from the list *AllVertices* and the sublist of *AllVertices* after this vertex. In other words, we check all pairs of potential cut-vertices.

The following theorem formally proves that both of the above functions return *true* for all possible lists of eight *Insideness* values:

```
Theorem CutEdges2D_SetEqual_Polygons3D :
∀ i0 i1 i2 i3 i4 i5 i6 i7 : Insideness,
let I := [i0;i1;i2;i3;i4;i5;i6;i7] in
(Polygons3D_Subset_CutEdges2D I) &&
(CutEdges2D_Subset_Polygons3D I) = true.
Proof.
  intros;
  destruct i0, i1, i2, i3, i4, i5, i6, i7;
  vm_compute;
  reflexivity.
Qed.
```

Once a theorem is stated, Coq enters into a proof mode. The statement of the theorem becomes a goal which needs to be discharged through some sequence of predefined transformations. These transformations are called *tactics*. The user is responsible for choosing the appropriate tactics. Coq automatically performs the transformations of the goal according to the tactics, and enforces that these transformations are performed in a logically sound manner. In the

case of our theorem *CutEdges2D_SetEqual_Polygons3D*, we prove it by using the following four tactics:

- `intros` moves the universally quantified variables into the local context, which is similar to assuming a view of the goal as a function of these variables.
- `destruct` *i0*, *i1*, *i2*, *i3*, *i4*, *i5*, *i6*, *i7* considers all possible values for all of the variables in the list. Since each of the eight variables can assume one of two values (*Insideness_Inside* or *Insideness_Outside*) this tactic generates $2^8 = 256$ different lists of these values.
- `vm_compute` computes the goal for a given local context. Since we concatenated the tactics with a semi-colon, every tactic in the sequence applies to *all* subgoals produced by the previous tactic.
- `reflexivity` is used to finish up the proof when it is a simple equality, such as *true* = *true* in this case.

At this point Coq discharges all proof goals completely, and we enter the command `Qed` which validates the proof and includes the theorem into the environment.

5 Proof of Water-tightness

In this section we describe our proof that the lists of triangles returned by the function *TriangleList_Get* for all cubes in the grid collectively form a water-tight surface. In particular, for each edge E of each triangle, we prove that the following conditions are satisfied:

(i) if E lies in some face F of the current-case cube (E is a cut-edge):
  (a) E appears exactly once cumulatively in all triangles in this cube, and
  (b) for any possible neighbor cube incident to F, E appears exactly once cumulatively in all triangles of this neighbor cube;

(ii) if E does not lie in any of the faces of the current cube (it is not a cut-edge), then it appears exactly twice cumulatively in all triangles of the current cube.

The following function examines all possible lists of eight *Insideness* values that are partially defined by a list *OI* of eight *option Insideness* values. More specifically, the function builds lists *I* of *Insideness* values such that each member of *OI* which is *None* is represented in turn by *Insideness_Inside* and *Insideness_Outside* in *I*. The members of *OI* that have a value of *Some i* are represented only by *i*. This way, we can constrain the four *Insideness* values in the corners of a face shared by two cubes, and check all combinations of the *Insideness* values in the four corners of the neighbor cube that are not shared. This function returns *true* if and only if the edge E appears exactly once in all possible *Insideness* configurations of the neighbor cube.

```
Fixpoint TriangleEdge_IsCount1
         (OI : list (option Insideness))
         (I : list Insideness)
         (E : TriangleEdge) : bool :=
  match OI with
  | nil ⇒
      let T := TriangleList_Get (Insideness_2Index I) in
      beq_nat (TriangleEdge_Count T E) 1
  | oi :: OI' ⇒
      let f x := TriangleEdge_IsCount1 OI' (I ++ [x]) E in
      match oi with
      | Some i ⇒ f i
      | None ⇒ (f Insideness_Inside) && (f Insideness_Outside)
      end
  end.
```

Here *beq_nat* is a built-in Coq function which checks two natural numbers for equality and returns the corresponding boolean value.

The function *TriangleEdge_IsCount1_NeiCube* below builds the input parameters and calls the function *TriangleEdge_IsCount1*:

```
Definition TriangleEdge_IsCount1_NeiCube
           (L : Location)
           (A B : CoordOption3)
           (I : list Insideness) : bool :=   ...
```

In this function, L identifies the face of the current cube, A and B identify the end points of the edge, and I is the *Insideness* configuration of the current cube.

The following function checks all the conditions we specified above:

```
Definition TriangleList_NoHoles (I : list Insideness) : bool :=
  let WholeList := TriangleList_Get (Insideness_2Index I) in
  let f x y :=
    let E := TriangleEdge_Cons x y in
    let n := TriangleEdge_Count WholeList E in
    match Location_GetCommon x y with
    | None ⇒ beq_nat n 2
    | Some L ⇒
        if (beq_nat n 1)
        then TriangleEdge_IsCount1_NeiCube L x y I
        else false
    end
  in
```

```
let fix g CurrentList :=
  match CurrentList with
  | nil ⇒ true
  | (Triangle_Cons A B C) :: CurrentList' ⇒
      (f A B) && (f B C) && (f C A) && (g CurrentList')
  end
in
g WholeList.
```

This function first obtains the list of triangles *WholeList* corresponding to the given *Insideness* configuration. Then it calls the local recursive function *g* which iterates over all triangles on the list, and for each edge of each triangle calls another local recursive function *f*. Function *f* checks the number of times the given edge appears in the triangles of *WholeList*, and calls *TriangleEdge_IsCount1_NeiCube* when this number is 1.

Finally, we state and prove a theorem which ensures the absence of holes:

```
Theorem No_Holes :
∀ i0 i1 i2 i3 i4 i5 i6 i7 : Insideness,
TriangleList_NoHoles [i0;i1;i2;i3;i4;i5;i6;i7] = true.
Proof.  ...  Qed.
```

The tactics used in this proof are the same as those used in the proof of `Theorem` *CutEdges2D_SetEqual_Polygons3D*.

6 Conclusions

The Marching Cubes algorithm is difficult to implement and to verify manually due to a large number of cases. One way to reduce the number of cases is to exploit symmetry, however this approach can introduce mistakes of its own, as it happened in the original paper. In this paper we present our formal proof of correctness of an existing Marching Cubes implementation. Our proof checks all cases disregarding any perceived symmetry. Our development is performed with the Coq proof assistant software which provides a solid framework for proof construction and validation. The interested reader can download and execute our Coq script. Furthermore, our script can be used to verify a different table of triangle connectivity, since the problem admits many correct solutions. For example, one might wish to construct a table which improves certain features of the resulting triangulation, such as planar or dihedral angles, triangle normals, surface area, or vertex degrees.

The proof presented here works with a uniform grid only. We are currently working on extending this analysis to the more complex case of a non-uniform

grid, for example, represented by an octree. Another extension we are planning is the use of a Coq mechanism known as *program extraction*: the parts of the script that deal with the proof of correctness can be separated from the parts that actually execute the algorithm. Program extraction in Coq allows for automatic production of executable code in OCaml, while the correctness of this code is backed by the removed logical part. This logical part can be thought of as a *certificate*, a notion used in the formal methods community, that is a formal artifact which demonstrates that a program satisfies its specifications.

References

1. Schroeder, W.J., Martin, K.M.: Overview of visualization. In: Johnson, C., Hansen, C. (eds.) Visualization Handbook. Academic Press, Inc. (2004)
2. Newman, T.S., Yi, H.: A survey of the marching cubes algorithm. Computers & Graphics 30(5), 854–879 (2006)
3. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3D surface construction algorithm. SIGGRAPH Comput. Graph. 21(4), 163–169 (1987)
4. Hammer, P.: Matlab implementation of marching cubes (2011), `http://www.mathworks.us/matlabcentral/fileexchange/32506-marching-cubes`
5. Talos, I., Jakab, M., Kikinis, R., Shenton, M.: SPL-PNL brain atlas (March 2008), `http://www.spl.harvard.edu/publications/item/view/1265`
6. Natarajan, B.K.: On generating topologically consistent isosurfaces from uniform samples. The Visual Computer 11, 52–62 (1994)
7. Chernyaev, E.V.: Marching cubes 33: Construction of topologically correct isosurfaces. Technical Report CERN CN/95-17 (1995)
8. Montani, C., Scateni, R., Scopigno, R.: A modified look-up table for implicit disambiguation of marching cubes. The Visual Computer 10, 353–355 (1994)
9. Lopes, A.M.: Accuracy in Scientific Visualization. PhD thesis, The University of Leeds (1999)
10. Dürst, M.J.: Letters: Additional reference to "marching cubes". Computer Graphics 22, 72–73 (1988)
11. Heiden, W., Goetze, T., Brickmann, J.: Fast generation of molecular surfaces from 3D data fields with an enhanced "marching cube" algorithm. Journal of Computational Chemistry 14, 246–250 (1993)
12. Kaufmann, M., Moore, J.S.: An ACL2 Tutorial. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 17–21. Springer, Heidelberg (2008)
13. Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda – A Functional Language with Dependent Types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009)
14. Bertot, Y., Castéran, P.: Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
15. Harrison, J.: HOL light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009)

16. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011)
17. Megill, N.D.: Metamath: A Computer Language for Pure Mathematics. Lulu Publishing, Morrisville (2007), http://us.metamath.org/downloads/metamath.pdf
18. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. Journal of Formalized Reasoning 3(2) (2010) (Special Issue: User Tutorials I)
19. Rahli, V., Bickford, M., Anand, A.: Formal program optimization in Nuprl using computational equivalence and partial types. In: The 4th Conference on Interactive Theorem Proving, Rennes, France, pp. 261–278 (July 2013)
20. McCune, W.: Prover9 and Mace4 (2005-2010), http://www.cs.unm.edu/~mccune/prover9
21. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
22. Pfenning, F., Schürmann, C.: System description: Twelf - A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
23. INRIA France. The Coq proof assistant, version 8.4, http://coq.inria.fr
24. Chlipala, A.: Certified Programming with Dependent Types (2013), http://adam.chlipala.net/cpdt
25. Pierce, B.C., Casinghino, C., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B., et al. Software Foundations (2012), http://www.cis.upenn.edu/~bcpierce/sf
26. Dufourd, J.-F., Bertot, Y.: Formal Study of Plane Delaunay Triangulation. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 211–226. Springer, Heidelberg (2010)
27. Gonthier, G.: Formal proof – the Four-Color Theorem. Notices of the AMS 55, 1382–1393 (2008)
28. Dufourd, J.-F.: A hypermap framework for computer-aided proofs in surface subdivisions: genus theorem and Euler's formula. In: Proceedings of the 2007 ACM Symposium on Applied Computing, pp. 757–761. ACM, New York (2007)
29. Brun, C., Dufourd, J.-F., Magaud, N.: Designing and proving correct a convex hull algorithm with hypermaps in Coq. Computational Geometry: Theory and Applications 45(8), 436–457 (2012)
30. Labelle, F., Shewchuk, J.R.: Isosurface stuffing: Fast tetrahedral meshes with good dihedral angles. ACM Transactions on Graphics 26(3), 57.1 – 57.10 (2007)