Introduction to Integer-Coordinate Crystalline Meshes

Vasiliki Chatzi and Franco P. Preparata

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-99-01
February 1999

Introduction to Integer-Coordinate Crystalline Meshes *

Vasiliki Chatzi and Franco P. Preparata †

February 18, 1999

1 Introduction

Problems in physics and engineering defined on a given physical domain are often modeled by a system of partial differential equations, together with appropriate boundary conditions. This data completely models a physical phenomenon, and if we were to solve the differential equations we would obtain the values of the physical quantities of interest in the defined region. However, obtaining a closed form solution of the equations is often impossible and in these cases we have to settle for using a numerical method to obtain approximate results.

The finite-element method is one of the tools commonly used today to approximately solve partial differential equations associated with a physical region. The first step consists of discretizing the region by partitioning it into a set of simple shapes called *elements*. The resulting collection of elements is called a *finite element mesh*. Then, nodal points are defined on each of the elements, and a basis function is associated with each nodal point. The basis functions are used to construct a system of linear equations and finally, the system is solved to obtain an approximation to the solution of the partial differential equation. The mesh used plays an important role during the entire computation. A high-quality mesh ensures both that the resulting system of linear equations is well-conditioned, as well as that the approximation we obtain will be close to the real solution.

Although this is still an active research area, there is general agreement on some properties a high-quality mesh should have. First of all, since the problem is associated with a physical domain, the union of all the elements in the mesh should closely approximate the domain. We also would like to

*Research supported in part by the National Science Foundation under grand CCR 9400232 and by the U.S. Army research Office under grand DAAH04-96-1-0013.

†Department of Computer Science, Brown University, Box 1910, Providence, RI 02912. Tel: (401) 863-7668, Fax: (401) 863-7657, E-mail: {vc,franco}@cs.brown.edu

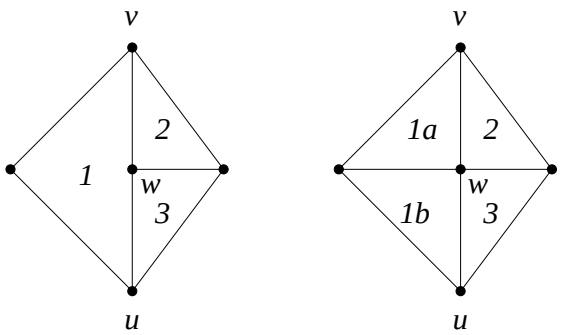


Figure 1: Two-dimensional illustration pertaining to the notion of conformity. (a) This mesh is non-conformal, because the intersection of the element marked 1 with the element marked 2 contains only part of the edge (u, v). (b) Adding one more edge makes the mesh conformal.

be able to cover different regions of the domain with elements of different size. In general, smaller elements give us a better approximation of the solution. However, using smaller elements also implies that we need more elements to cover the same region of the domain, which results in a larger and harder to solve system of equations. So, we would like to cover regions of the domain where the solution changes rapidly ("regions of interest") with smaller elements, while using bigger elements for regions where the phenomenon is relatively stable. Moreover, the transition between elements of different sizes should be *smooth* [12], that is, the ratio of areas (or volumes) of two adjacent elements should be bounded by a small constant. Finally, it is important that the resulting system of equations be well-conditioned, and therefore easy to solve. It is not entirely clear what properties of the mesh will result in a well-conditioned system, however, several theoretical and experimental results show that elements with small angles have to be avoided.

Another highly desirable but not necessary property is conformity. An element is said to be *conformal* if its intersection with any other element in the mesh is empty, or a vertex, or an edge, or a face (Figure 1). A mesh is conformal if all its elements are conformal [6, 24]. A point is called *non-conformal* if it is the vertex of one element, but is positioned in the interior of an edge of an other element (for example, the point marked w in Figure 1(a)). Non-conformal points create numerical problems which can only be solved by using complicated specialized techniques, and therefore

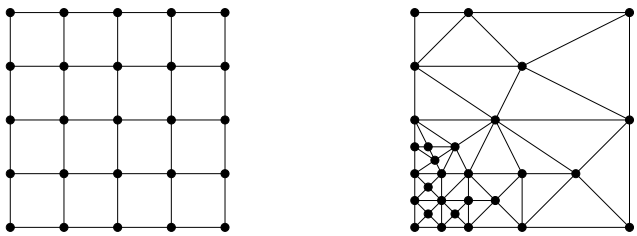


Figure 2: (a) A structured mesh. (b) An unstructured mesh on the same domain.Notice that the element density is much higher at the lower left corner. Both meshes are conformal.

should be avoided whenever possible.

Currently used meshes fall into two broad categories: *structured* meshes (Figure 2(a)), consisting of virtually identical elements, and *unstructured* meshes (Figure 2(b)), whose geometry is arbitrary within some broad topological constraints. Structured meshes are easy to construct and result in systems of equations that are easy to generate and to solve. However, they are inappropriate for problems defined over a complex region and for problems exhibiting regions of interest. Unstructured meshes are more flexible, and are therefore used to model these problems. However, the resulting systems of equations are harder to generate and solve. Moreover, the numerical errors accumulating during the generation and adaptation phases can yield meshes that are in practice non-conformal, forcing one to use complicated and time-consuming methods to compensate for this behavior.

In this paper we present *integer-coordinate crystalline meshes* which were designed to combine the advantages of both structured and unstructured meshes. A mesh is called *crystalline* if it consists of regular elements whose vertices are positioned on a regular grid. The regular elements we use in our meshes are squares in 2D and cubes in 3D. Cubes were chosen because they outperform tetrahedra for many types of problems [1]. Crystalline meshes allow us to use elements of different sizes in different regions of the domain, which makes them suitable for problems with regions of interest. Conformal transitions between regions containing d-cubes of different sizes are realized using elements belonging to a small set of distinct, non-cubical shapes. This allows us to precompute the element matrices used in the computation and reduces the cost of creating the system of linear equations to the cost of its assembly. Moreover, it guarantees that the angles of all the elements will be bounded comfortably away from zero, resulting in a system of linear

equations that is easier to solve.

If all the vertices in a crystalline mesh are placed on integer coordinates, the mesh is an *integer-coordinate crystalline mesh*. Using only integer coordinates allows us to use integer arithmetic during the mesh generation and adaptation phases, which alleviates the problem of accumulating numerical errors, that is common in other types of meshes. Notice that if the given domain is such that integer coordinates cannot be used, we can always scale it appropriately.

Another important advantage of crystalline meshes is that they are suitable for adaptive computations, where the simulated system evolves over time. Creating meshes that facilitate adaptive computation is one of our main goals, and we will spend a large part of this paper describing how mesh adaptation for crystalline meshes can be done efficiently.

Integer coordinate crystalline meshes allow for different element sizes in different regions of the domain, are conformal and only use elements belonging to a small set of distinct shapes, which allows us to construct the system of linear equations both fast and accurately. Their use minimizes numerical errors during the mesh generation and adaptation phases and simplifies adaptive computations. All these advantages, as well as empirical evidence that the systems of equations generated from crystalline meshes are well conditioned, suggest that crystalline meshes are appropriate for problems where structured meshes cannot be used.

While isolated examples of such meshes have been occasionally reported in the literature for 2D problems (for example, the cover of [5] shows a 2D crystalline mesh), our emphasis is on the far more complex and significant 3D setting. We will focus on the characteristics of crystalline meshes and on the procedures intended to adapt the element density as required to achieve high-quality solutions.

There is some similarity between crystalline meshes and meshes generated using quadtree or octree based methods [26, 4, 12, 23, 25] in that both meshes consist mainly of squares or cubes (at least at some point during the mesh generation phase). However, octree meshes do not use integer coordinates for their vertices, which results in points that are non-conformal in practice. Moreover, the last stages of the octree method are typically a triangulation phase, where the obtained cubes are cut into tetrahedra, and a smoothing phase, where the vertices of the tetrahedra are moved to obtain elements with larger angles. So the output meshes consist of irregular triangles or tetrahedra. This makes the assembling of the system of linear equations both computationally expensive and inaccurate.

Cartesian meshes (see for example [9]) are also somewhat similar to

crystalline meshes in that they also use points with integer coordinates and the elements of the mesh are squares in 2D and cubes in 3D. However, the exclusive use of d-cubes results in non-conformal points in the transition regions between elements of different sizes. This forces researchers using Cartesian meshes to develop complicated and computationally expensive techniques to deal with the singularities arising from the introduction of non-conformal points [2].

Some work has also been done on meshes that have guaranteed angle bounds [3, 15]. However, these results concentrate on triangular and tetrahedral meshes. Moreover, they seem to have more theoretical then practical value, since the meshes generated using the proposed algorithms have a much higher number of elements than needed to achieve the desired accuracy. We are not aware of any work concentrating on hexahedral or mixed meshes.

The rest of the paper is organized as follows. Section 2 gives the characterization of integer-coordinate crystalline meshes. Section 3 discusses the adaptive features (concentrating on the refinement). Section 4 discusses adaptivity in detail for 2D meshes, and Section 5 presents its generalization to the more complex 3D case. Section 6 addresses some implementation issues and gives some experimental results. Section 7 contains a summary of results and some open problems.

2 Integer-Coordinate Crystalline Meshes

An *integer-coordinate crystalline mesh* consists of a small variety of regular elements whose vertices are a subset of the points of a regular grid spanning the domain (with integer coordinates). The mesh is conformal and the element density can vary in different regions of the domain. The elements are of two types: *standard* and *transitional*. Standard elements are squares in 2D meshes and cubes in 3D meshes. Transitional elements are used to realize conformal transitions between regions where the corresponding standard elements have different sizes. In a typical crystalline mesh, the vast majority of elements are standard elements. All elements in the mesh are homothetic (modulo a rotation belonging to the group of the cube) to a few basic shapes, the factor of homothety being a power of 2. We will now present some definitions that will be used in the paper.

Definition 2.1 *A* mesh cube *is a d-cube whose boundary edges belong to the mesh.*

Definition 2.2 *The* capsule $c(x)$ *of an element x is the smallest mesh cube containing x (Figure 3).*

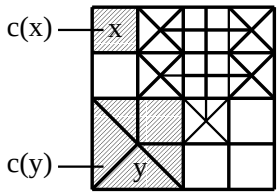


Figure 3: 2D illustration of the notion of capsules.The capsules of the elements x and y are shown shaded. x is a standard element, so it coincides with its capsule. y is a transitional element, so its capsule is the smallest enclosing mesh cube.

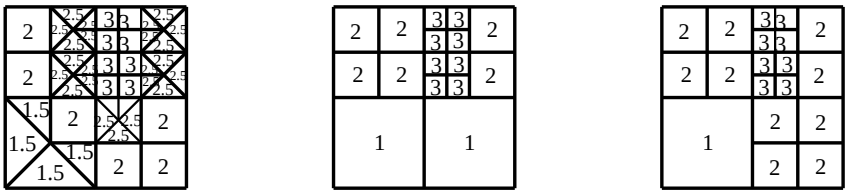


Figure 4: 2D illustration of the different types of meshes defined in this section. (a) Legal mesh (b) Rectilinear mesh. Note that one of the level-1 squares has a level-3 neighbor, therefore the mesh is not conformable. (c) A conformable mesh. Numbers denote the levels of the elements.

Since standard elements are d-cubes whose boundary belongs to the mesh, every standard element coincides with its capsule. The capsule of each transitional element is a d-cube, strictly larger than the element, that contains it. We will call capsules containing transitional elements *transitional capsules.*

Let L be the length of the side of the largest cube appearing in the mesh. If l denotes the length of the side of a generic mesh cube, the *level* of this cube is $\log_2(L/l) \geq 0$. We define the level of a standard element to be the level of its capsule, whereas the level of a transitional element is conventionally the level of its capsule $+1/2$. Note that the level uniquely identifies the size of a standard element.

Definition 2.3 *We say that a crystalline mesh is* legal *(Figure 4(a)) if:*

1. *it is conformal,*
2. *the level of a standard element differs by at most* $1/2$ *from the levels of its neighbors, and*

3. *the levels of two transitional elements belonging to adjacent capsules differ by at most 1.*

It is easy to see that the second and third condition ensure that neighboring elements will not have sizes that differ by too much. A legal crystalline mesh is conformal and exhibits smooth transitions in element sizes. The meshes constructed during the mesh generation step, as well as the meshes resulting from each adaptation phase are legal, and are the ones used in the actual finite element computations.

Definition 2.4 *A* rectilinear *mesh (Figure 4(b)) is a (not necessarily conformal) crystalline mesh inductively defined as follows:*

1. *The coarsest crystalline mesh (consisting of all standard elements of level 0) is rectilinear.*
2. *The mesh resulting from replacing one element of a rectilinear mesh with* 2^d *standard elements (*4 *squares in 2D or* 8 *cubes in 3D) is rectilinear.*

Definition 2.5 *A conformable mesh (Figure 4(c)) is a rectilinear mesh such that the level of each element in the mesh differs by at most* 1 *from the levels of its neighbors.*

The most important characteristic of conformable meshes is that they can be made conformal (resulting in a legal mesh) by using a simple, one-pass procedure. Since the mesh is conformable, we know that if an element x is non-conformal, it is because it has one or more neighbors that have a level value of $level(x)+1$. So, there is only a small number of possible non-conformal neighboring configurations. With each one of these configurations we can associate a refinement that will make the element conformal without creating any additional non-conformal points. This means that we can use one pass to examine all elements in the mesh, locate the non-conformal ones and refine them appropriately, to obtain a conformal mesh. We will talk more about exactly how this refinement can be done in the sections discussing 2D and 3D meshes.

3 Crystalline Mesh Adaptation

To obtain a high-quality solution, we need a mesh with a high element density within the regions of interest. However, given the description of a physical problem and its domain, we generally cannot establish *a priori* the regions of interest. In such cases, the finite element computation proceeds

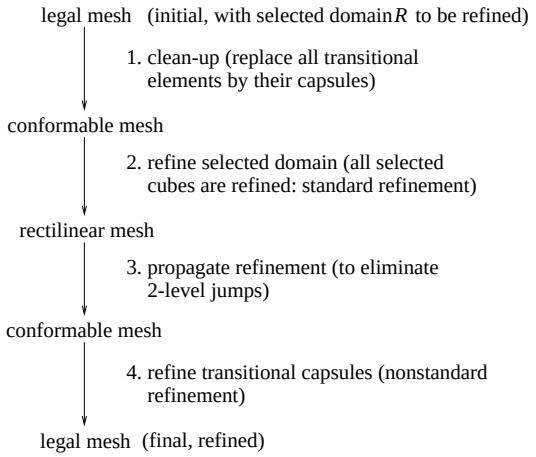


Figure 5: Flow diagram of the refinement process.

as follows. Given an initial mesh over the domain, a system of equations is generated and solved. Then, bounds on the errors associated with each element are computed and used to determine in which subregions the mesh density must be modified. Finally, a new mesh is created based on this information. The procedure is applied iteratively until a satisfactory error bound is achieved over the whole domain [6, 14, 16]. This process can be significantly facilitated and accelerated by mechanisms that automatically modify the existing mesh. Such modification is called *mesh adaptation*, and involves two reciprocal processes, *refinement* and *coarsening*. We confine ourselves to the process of refinement, since the ideas and techniques used for coarsening are analogous.

We are given a legal crystalline mesh and a region to be refined. We should refine the elements in the region in such a way that the resulting mesh is also legal. We can achieve this by doing refinement in a very controlled way. Figure 5 is a flow diagram (not necessarily the proposed algorithm) of a conceptually correct way to carry out a refinement. In the first step (clean-up phase) we replace all transitional, as well as standard elements contained in a transitional capsule, by their capsule, thereby obtaining a conformable mesh. Then, all the elements of the resulting mesh that fall completely within the selected region are refined. Standard elements with edge length l are partitioned into standard elements with edge length $l/2$, so that a standard element of level j will be replaced by 2^d elements of level $j+1$. Such refinement is called *standard*.

At the end of this step, the mesh may contain neighboring[1] elements that have level values that differ by 2. In such case, the resulting mesh is no longer conformable, so the refinement is propagated in order to obtain a conformable mesh. The propagation step also identifies the elements that will be transitional capsules in the output mesh. In the final step we refine each transitional capsule appropriately. This refinement depends upon the levels of the capsule's neighbors, and is referred to as *nonstandard refinement.*

We now analyze the refinement propagation step (step 3). At an abstract level, we identify each element of the rectilinear mesh under consideration with the node of a node- and edge-weighted directed graph G. The weight of a node is its level. We establish an arc (v_x, v_y) from node v_x to node v_y, if the corresponding elements x, y are neighbors and $level(x) > level(y)$. We assign to (v_x, v_y) the weight $w(v_x, v_y) = level(x) - level(y)$ (Figure 6).

It is clear that G is acyclic, and that if l is the maximum level of an element in the mesh, the maximum length of a path in G will be l. A path in G is said to be *smooth* if all the edges in the path have weight 1. The mesh is conformable if and only if all paths of G are smooth.

By the action of the standard refinement (step 2), the weights of a subset of arcs of the (conformable) mesh are increased by 1, so that 2 is the largest weight of arcs of the resulting rectilinear mesh M. Consider any path $p = (v_1, v_2, \ldots, v_s)$ in the graph $G(M)$ of M. We claim:

Lemma 3.1 *In any maximal nonsmooth path p in $G(M)$, there is at most one arc of weight 2.*

Proof: We will start with a case analysis. Consider two arbitrarily chosen neighboring elements x and y in the original conformable mesh (the mesh obtained after the clean-up phase). By the definition of a conformable mesh we know that either the two elements have the same level value, or their level values differ by 1. In the second case we can assume, without loss of generality, that $level(x) = level(y) + 1$ (see Figure 7, case 2). During the refinement phase, none, or one, or both of the elements are to be refined. This gives us eight distinct cases, shown in Figure 7. In the same figure we also show the subgraph of G that is obtained in each of the cases. Notice that if an arc (v, w) has weight 2, we can conclude that the element associated with v was created during the last refinement step, while the element associated with w was not refined (Case 2,(c)).

Using this case analysis, we can prove the Lemma by contradiction. Suppose that in p there are two arcs (v_i, v_{i+1}) and (v_j, v_{j+1}) with $(1 \leq i <$

[1] The neighbors of an element x are a subset of the elements adjacent to x, and will be defined separately for the 2D and 3D case.

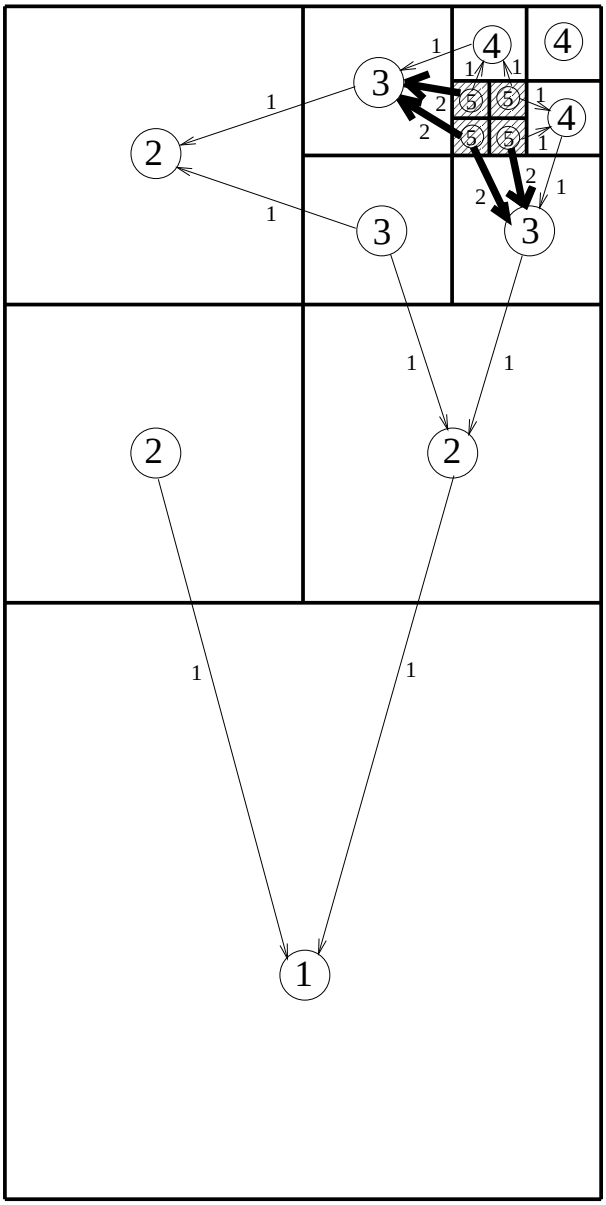


Figure 6: A rectilinear mesh and its associated graph. The heavy arcs have weight larger than 1. The region that was refined during the last step is shown shaded.

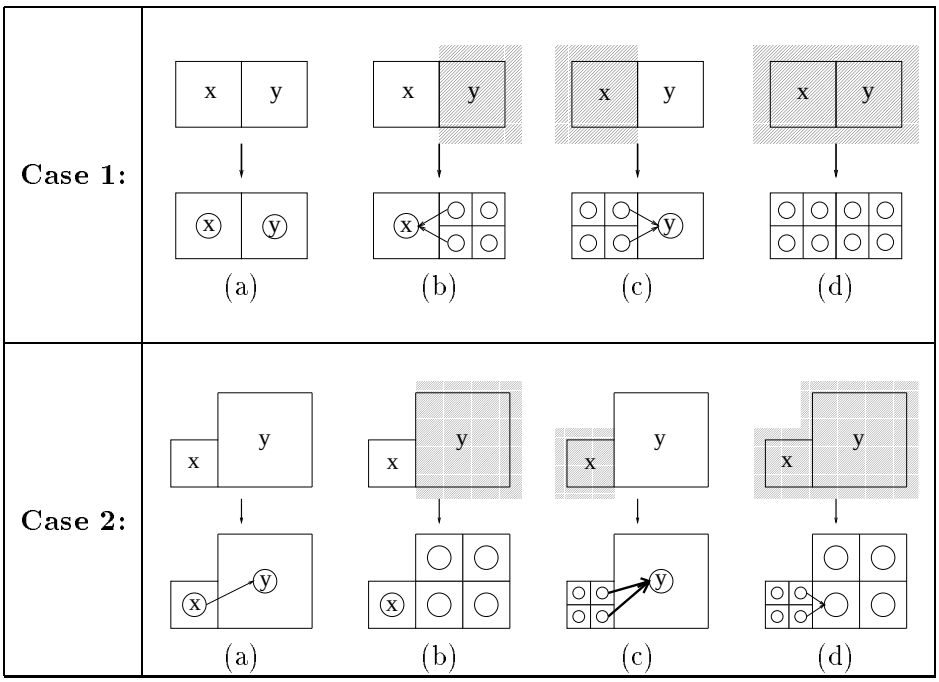


Figure 7: 2D illustration of the possible neighboring arrangements in the original conformable mesh and in the mesh resulting after the refinement. Elements selected to be refined are shown shaded. Together with the elements resulting from the refinement we also show the subgraph associated with them. Case 1: The neighbors originally have the same level value. Case 2: The level values of the neighbors differ by 1. Notice that Case 2(c) is the only one resulting in arcs with weight 2 (shown heavier).

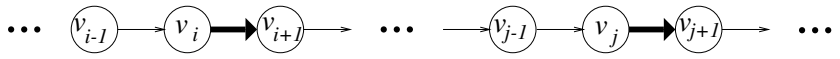


Figure 8: The path used in the proof. The arcs (v_i, v_{i+1}) and (v_j, v_{j+1}) have weight 2. All other arcs shown have weight 1.

$j \leq s-1$) of weight 2, with no intermediate weight-2 arc (Figure 8).

From the case analysis we know that since v_j has an outgoing weight-2 arc, the element associated with it has been created during the last refinement step. We also know that v_j has an incoming weight-1 arc (v_{j-1}, v_j). We see that the only case where an element that was refined during the last step has an incoming weight-1 arc is case 2(d), and this implies that the ele-

ment corresponding to v_{j-1} was also created during the last refinement. We can follow the path from v_{i-1} to v_j backwards, applying the same reasoning at each arc, concluding that all nodes on the path from v_{i+1} to v_j have been created during the last refinement step. In particular, the element corresponding to v_{i+1} has been created during the last refinement step. However, from the case analysis we know that no element created during the last refinement step can have a node associated with it with an incoming weight-2 arc, a contradiction which proves the Lemma. □

A *refinement path* of M is a path whose first arc has weight 2 and is followed by a run of weight-1 arcs. Refinement propagation occurs exactly on refinement paths. Lemma 3.1 proves that refinement propagation can be done in one pass. Assume that x, y are two neighboring elements in the mesh we obtained after step 2, v, w are the two nodes associated with them and (v, w) is a weight-2 arc. We start by refining y, thereby eliminating the level-2 jump between x and y. If w has outgoing arcs, we follow them and refine the elements associated with them. We continue following outgoing arcs and refining the corresponding elements until we reach the end of the refinement path. Lemma 3.1 proves that once an element is refined, the resulting elements will not be refined again during the refinement propagation. Using the definition of the refinement paths we can also easily prove the following theorem:

Theorem 3.2 *If $l > 0$ is the maximum level occurring in the mesh, the depth of refinement propagation is at most l.*

After the refinement propagation, we obtain a conformable mesh. Now we can, in one pass, locate all non-conformal elements and refine them in order to obtain a conformal mesh. This happens during Step 4 of the refinement procedure, and the non-conformal elements located are (standard) elements that have neighbors with higher level value. The refinement used during this step is called *nonstandard* refinement and will result in some transitional elements, at the boundaries between regions having different element density.

Knowing the length of the refinement path allows us to compute the complexity of the refinement algorithm. It is given in the following Corollary of Theorem 3.2.

Corollary 3.3 *Let R be the region to be refined, and let $m(R)$ and $m(\delta(R))$ denote respectively the numbers of elements in R and on its boundary. Then the complexity of the refinement algorithm is $\Theta(m(R) + l \cdot m(\delta(R)))$. If $m(\delta(R)) = o(m(R))$, the complexity of the refinement is $\Theta(m(R))$.*

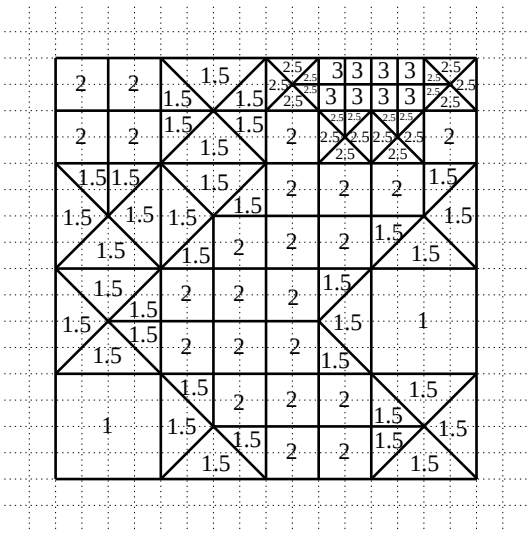


Figure 9: A 2D crystalline mesh, shown in solid lines. The dotted lines represent the regular grid supporting the crystalline mesh. The numbers indicate the level of each finite element.

Note that condition $m(\delta(R)) = o(m(R))$ is commonly met, and in this case, the complexity of refining the region R dominates the complexity of the refinement propagation. In the following sections we present efficient refinement techniques for crystalline meshes in 2D and 3D.

4 2D Crystalline Meshes

4.1 Description and Properties

The two-dimensional case, inherently very simple, is handled by a procedure analogous to that of the more complex three-dimensional case (Section 5), and is therefore presented here for pedagogical reasons. In 2D crystalline meshes the standard element is a square. The transitional elements are right-angle isosceles triangles. We define two elements to be *neighbors* if they share an edge. Figure 9 shows a 2D integer-coordinate crystalline mesh.

If we are careful during the mesh generation and refinement steps, we can maintain all the elements so that their vertices are presented in counter-clockwise order. In this case the stiffness matrices [27] of all square elements are exactly the same. If we give a consistent local index to each vertex of

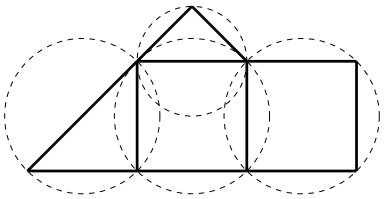


Figure 10: A square and all the different neighbors it can have in a crystalline mesh.The circumcircles are shown in dashed lines. Notice that the Delaunay property always holds.

the triangle (for example, if the vertex at the right angle always has local index 0), the stiffness matrices of all the triangles will also be identical (regardless of size and orientation). In order to create the system of equations, we precompute the matrices of the two distinct shapes, and then assemble the local matrices to construct the matrix of the system using the standard finite element procedure. This reduces the cost of creating the system to the cost of its assembly.

The smallest angle present in any 2D crystalline mesh is 45° (and occurs in the transitional elements). Moreover, any 2D crystalline mesh has the Delaunay property: the interior of the circumcircle of any element does not contain any vertices of the mesh [10, 17]. The Delaunay property is very important and 2D meshes that exhibit it are considered ideal, because the system of equations that results from them is easy to solve. Bellow we will prove the claim.

Lemma 4.1 *Two-dimensional crystalline meshes have the Delaunay property.*

Proof: We can prove the lemma by exhaustively examining all possible neighboring configurations. We have already seen that the mesh will consist only of squares and right-angle isosceles triangles, and that it is conformal. We consider each of the shapes and all possible neighbors that the shape could have. If all possible neighboring arrangements have the Delaunay property, the whole mesh will also have the Delaunay property.

Let us first look at the square elements. The only possible neighbors a square could have in a crystalline mesh is either a square of the same size or triangles of two different sizes. Figure 10 shows all possible neighbors of the square and the circumcircles of all the elements. It is easy to verify that the Delaunay property holds.

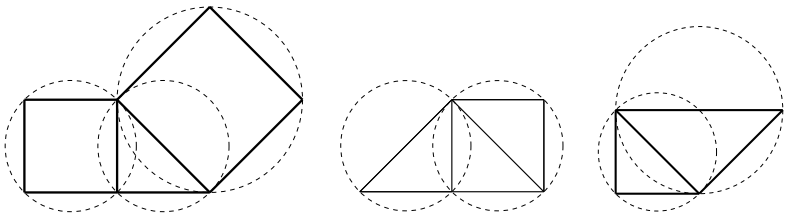


Figure 11: All possible neighbors of a right angle isosceles triangle in a crystalline mesh. (a) Squares of two different sizes. (b) Triangles of the same size sharing either the hypotenuse or one of the other sides. (c) Triangles of two different sizes in a side-hypotenuse adjacency. Dashed lines denote the circumcircles. The Delaunay property holds in all cases.

An element that is a right-angle isosceles triangle can have squares of two different sizes as neighbors (Figure 11(a)). It can also have triangles of the same size sharing either the hypotenuse or one of the two other sides, or a triangle of different size with a side-hypotenuse adjacency. All possible configurations are shown in Figure 11(b) and (c), and we can verify that the Delaunay property holds in all cases. □

4.2 Refinement of 2D Crystalline Meshes

The general refinement process discussed in Section 3 is illustrated in Figure 12 for a 2D mesh. We start with a legal crystalline mesh and a region R that needs to be refined (shown shaded in Figure 12(a)). During the clean-up phase, we coarsen all transitional elements, thereby obtaining a non-conformal mesh that consists only of squares (Figure 12(b)). Clearly, the resulting mesh is conformable. We then use standard refinement to refine all elements within the selected region (Figure 12(c)). The resulting mesh is rectilinear but might not be conformable. In order to make it conformable we propagate the refinement to eliminate all two-level jumps (Figure 12(d)). Finally, we use non-standard refinement to refine all non-conformal elements, in order to obtain a legal mesh (Figure 12(e)), which will be used in the computation. All shaded elements in Figure 12(e) are those altered by the refinement process.

Refinement propagation ensures that the resulting mesh is conformable. Each element in the conformable mesh will have edges that are either conformal or have a vertex in the middle. This means that there are only five

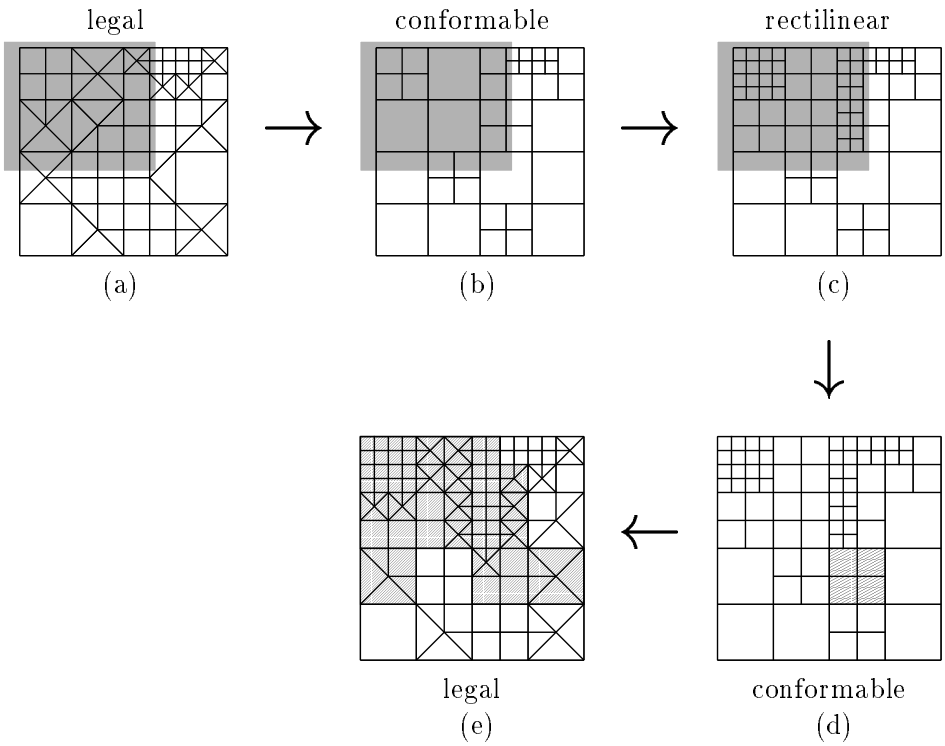


Figure 12: (a) A region R has been selected for refinement (shown shaded). (b) The non-standard elements are removed. (c) Squares that fall completely within R are regularly refined. (d) The refinement is propagated to neighboring elements (striped element). (e) Non-conformal elements are refined. The striped region contains all the elements that were modified by the refinement. Notice that it almost coincides with R.

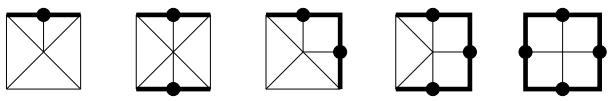


Figure 13: All possible non-conformal configurations for elements in a 2D conformable mesh. Bold sides denote boundaries shared with higher level elements.

different configurations (modulo right-angle rotations) a non-conformal element can have, depending on the subset of sides it shares with higher level elements. All possible configurations are shown in Figure 13, with empha-

sis given to the non-conformal vertices and the sides containing them. In the same figure we show an appropriate non-standard refinement for each configuration. Notice that all non-conformal edges are divided in two, while all conformal edges are not divided. Therefore, the mesh will be conformal (and legal) after the last step of the algorithm has been executed.

Going back to the example of Figure 12 we realize that the refinement procedure, as described so far, is not very efficient. For example, notice that the transitional elements in the lower right-hand corner of the mesh are removed during the clean-up phase, resulting in a standard element. During the last phase of the refinement procedure this element is refined, in order to obtain a conformal mesh. The resulting elements are exactly the same as in the original mesh. This suggests that we would obtain a more efficient algorithm if we could identify elements that are identical in the input and output meshes and avoid processing them. This becomes more important in cases where the input mesh consists of many elements and the region R to be refined contains only a small portion of them. In general, we would like the refinement procedure to run in time proportional to the number of elements in R and not to the number of elements in the entire mesh, and produce exactly the same meshes as the procedure shown in Figure 5.

In order to develop an efficient algorithm we notice that all elements in R need to be modified. Moreover, elements that are not in R need to be modified only if at least one of their neighbors is modified. So we can avoid modifying elements that are not affected by the refinement procedure by keeping track of all modified elements and checking all their neighbors to see if modifications are in order.

The algorithm starts by replacing all transitional elements, whose capsule falls completely within the selected region R, with said capsule (Figure 14(b)). This is equivalent to the clean-up phase, but is performed only on elements in R. Next, all standard elements completely within R are refined, using standard refinement (Figure 14(c)). Now we have to perform the two last steps of the procedure, refinement propagation and non-standard refinement of non-conformal elements. The refinement propagation is accomplished by using the procedure PROPAGATE_REFINEMENT, shown as pseudocode in Algorithm 1.

Since all elements within R have been refined, paths of G within R are by construction smooth. So, the refinement propagation will start from elements that are at the boundary of R and proceed outwards. We will simulate a breadth-first search of the graph G using a queue Q. Throughout the execution of the algorithm, Q contains elements that have been modified by the refinement procedure and whose neighbors need to be checked and

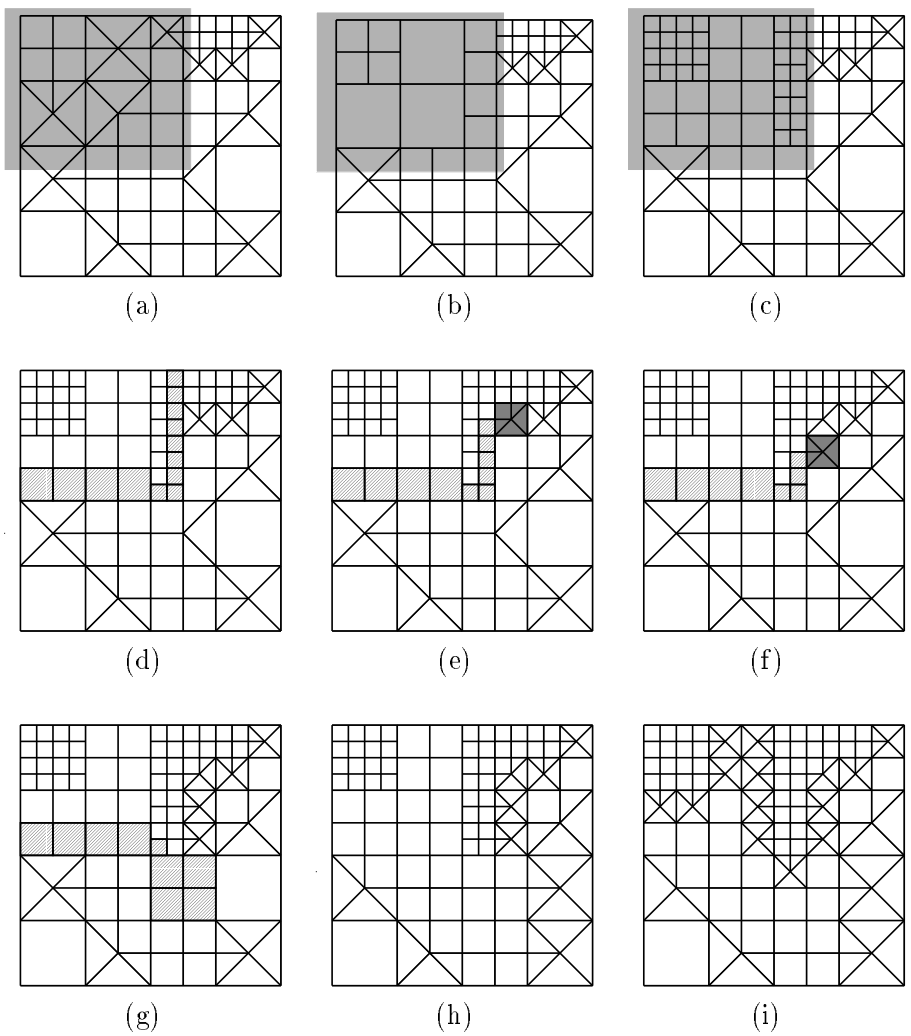


Figure 14: An example of the refinement procedure: (a) The initial mesh. The region R is shown shaded. (b) Non-standard elements in R are removed. (c) Elements in R are refined (standard refinement). (d) Elements on the boundary of R are inserted into Q (shown shaded). (e) Level difference of 1/2: the neighbor is coarsened and its capsule is refined. (f) Level difference of 1, and neighbor does not belong to a transitional capsule: the neighbor is refined using non-standard refinement. (g) The neighbor belongs to a transitional capsule: the capsule is refined using standard refinement and the resulting elements are inserted in Q. (h) State of mesh when Q is empty. (i) Non-standard refinement is used to make the mesh conformal. The resulting mesh is legal.

```
PROPAGATE-REFINEMENT(M, R)
 1  for each element x on the boundary of R
 2  do insert x into Q
 3  while Q ≠ ∅
 4  do x ← head[Q]
 5     for each neighbor y of x with level(y) < level(x) AND y ∉ Q
 6     do δ ← level(y) − level(x)
 7        switch
 8           case δ = 1/2 :
 9                z ← COARSEN(y)
10                NONSTANDARD-REFINE(z)
11           case δ = 1 AND y ∉ to a transitional capsule :
12                NONSTANDARD-REFINE(y)
13           case (δ = 1 AND y ∈ to a transitional capsule ) OR δ = 3/2 :
14                z ← COARSEN(y)
15                STANDARD-REFINE(z)
16                insert all resulting elements in Q
```

Algorithm 1: The Algorithm used to simulate a breadth-first search on the graph G. COARSEN(y) replaces all elements belonging to the transitional capsule of y with the capsule z and returns it.

maybe modified.

We start by inserting into Q all elements that are on the boundary of R (shown lightly shaded in Figure 14(d)). The algorithm then proceeds by extracting elements from the queue, examining all their neighbors, modifying them if necessary, and when appropriate inserting new standard elements that need processing into the queue. The algorithm terminates when the queue is empty.

To gain the correct insight into the operation of the proposed algorithm, we will prove some facts about it.

Lemma 4.2 *During the execution of the algorithm* PROPAGATE_REFINEMENT, *Q contains only standard elements that have been created during the current refinement step.*

Proof: We initialize Q by inserting all elements on the boundary of R. During the previous two steps of the refinement procedure all transitional elements in R have been removed and then all elements in R (which are now standard) have been refined using standard refinement. So, all elements in R are now standard elements that have been created during the current refinement step. Since the elements inserted in Q are in R, this is enough to prove that the lemma is true when Q is initialized.

The only other point where elements are inserted into Q is line 16. These elements are the result of a STANDARD_REFINE operation (line 15). There-

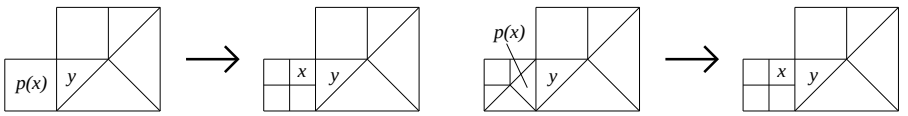


Figure 15: Examples showing the worst cases for Lemma 4.4: (a) x is created by refining a standard element. In this case, $level(x) - level(y) = \frac{3}{2}$. (b) x was created by first removing some transitional elements and then using standard refinement to refine their capsule. In this case also $level(x) - level(y) = \frac{3}{2}$.

fore, these elements, too, are standard elements created during the current refinement step. □

Lemma 4.3 *The procedure* PROPAGATE_REFINEMENT *terminates.*

Proof: The procedure terminates when the queue Q is empty. We have already proven that only standard elements that have been created during the current refinement step are inserted into Q. Moreover, Lemma 3.1 proves that once an element has been refined, none of the resulting element will be refined again during the refinement propagation step.

Combining these two facts, we conclude that once an element has been removed from the queue, it will not be refined again and no elements resulting from it can be inserted into the queue. This proves that only a finite number of elements will ever be inserted into Q, and therefore the procedure will terminate. □

Lemma 4.4 *Let x be an element that has just been extracted from the queue Q and y one of its neighbors. Then, $level(x) - level(y) \leq \frac{3}{2}$.*

Proof: First of all, notice that whenever an element is removed from the mesh by the refinement procedure, it is replaced by elements of the same or higher level value. Therefore, in order to prove the bound we can assume that y is an element that is present in the input mesh.

Since x was just removed from Q, we know that it is a standard element created during the current refinement step (Lemma 4.2). We can distinguish two cases, depending on how x was created:

- x was created by refining a standard element in the original mesh, using standard refinement (Figure 15(a)). Let $p(x)$ denote this standard element in the original mesh. Obviously, y was a neighbor of $p(x)$ in the input mesh, and since the mesh was legal:

$$level(p(x)) - level(y) \leq \frac{1}{2}.$$

Moreover,

$$level(x) = level(p(x)) + 1.$$

Combining these two facts we get:

$$level(x) - level(y) \leq \frac{3}{2}.$$

So, the lemma holds in this case.

- x was created by coarsening a transitional element, and then refining the resulting transitional capsule (Figure 15(b)). Let $p(x)$ be the transitional element in the input mesh that was a neighbor of y. Since the original mesh is legal:

  $$level(p(x)) - level(y) \leq 1.$$

  We also know that:

  $$level(x) = level(p(x)) + \frac{1}{2}.$$

  Combining these two facts we get:

  $$level(x) - level(y) \leq \frac{3}{2}.$$

This is enough to prove that the lemma always holds. □

Using the above lemma and the fact that level values are always either integers or $integer + \frac{1}{2}$ we can prove that:

Corollary 4.5 *Let x be an element that has just been extracted from the queue Q and y one of its neighbors. Let $\delta = level(x) - level(y)$. If $\delta > 0$ then $\delta \in \{\frac{1}{2}, 1, \frac{3}{2}\}$.*

During the PROPAGATE_REFINEMENT subroutine and while we are examining the neighbors of x, we are only interested in neighbors with level values lower than $level(x)$, since the refinement cannot propagate to neighbors with the same or higher level value. We will now describe and justify the actions that need to be performed in each of the cases addressed in Corollary 4.5.

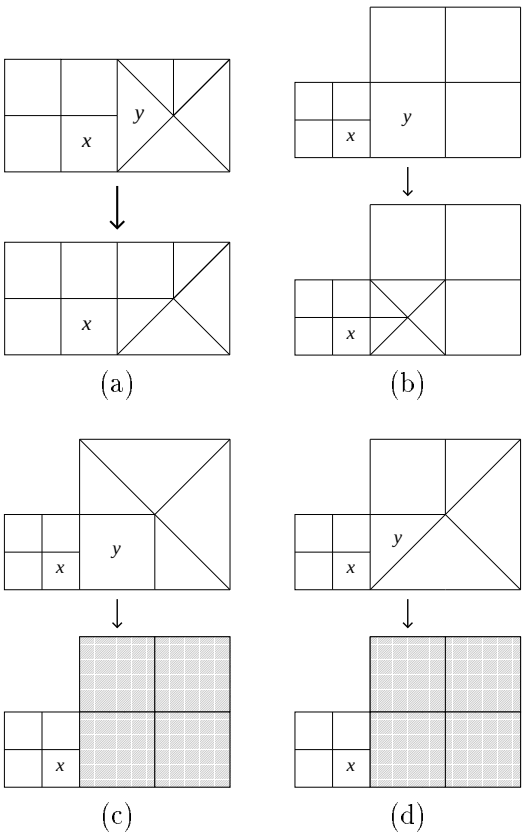


Figure 16: Examples of how neighbors of x with higher level value are processed by the algorithm. (a) $level(x) - level(y) = \frac{1}{2}$. The COARSEN subroutine is used and the obtained transitional capsule is refined using non-standard refinement. (b) $level(x) - level(y) = 1$ and y does not belong to a transitional capsule. Non-standard refinement is used to refine y. (c) $level(x) - level(y) = 1$ and y belongs to a transitional capsule. The capsule is obtained by using the COARSEN subroutine and then refined with standard refinement. The resulting standard elements, shown shaded, are inserted into the queue. (d) $level(x) - level(y) = \frac{3}{2}$. This case is handled in exactly the same way as case (c).

1. $level(x) - level(y) = \frac{1}{2}$ (Figure 16(a)).
   This means that y is a transitional element, whose capsule has a level value of $level(x) + 1$. In this case, the refinement does not have to be propagated, but y might not be conformal (because of the construction

of x). So, we use the subroutine COARSEN to replace y and all the elements in its capsule by the capsule, and refine the capsule again, using non-standard refinement. This process is guaranteed to make the boundary between x and the capsule of y conformal.

2. $level(x) - level(y) = 1$ (Figure 16(b) and 16(c)).
   In this case, y is a standard element. We can again distinguish two cases.

   In the first case (Figure 16(b)), y does not belong to a transitional capsule. This means that y has been created by a standard refinement operation. Therefore, it is enough to refine y, using non-standard refinement, in order to make the boundary between x and y conformal.

   If y belongs to a transitional capsule (Figure 16(c)), it has been created by a non-standard refinement operation. The transitional capsule to which y belongs and x have a difference of two in level values. In order to understand what the appropriate action is in this case, we will look at how the original algorithm, presented in Figure 5, would handle this case. Since we want the two refinement algorithms to produce the same mesh, we should simulate its actions. The original algorithm starts by removing all transitional and standard elements contained in transitional capsules by their capsules. Therefore, y would have been removed, and the corresponding neighbor of x would be y's transitional capsule $tc(y)$. Since $level(x) - level(tc(y)) = 2$, the refinement has to be propagated, and $tc(y)$ will be refined, using standard refinement. Finally, the refinement will be propagated, if necessary, to the neighbors of the newly created elements. We can achieve the same results by using the COARSEN subroutine to replace y and all the elements in the same transitional capsule by the capsule. Next, we refine the capsule using standard refinement. This entire process is a refinement propagation step. In order to ensure that the refinement will be propagated further, if necessary, we insert the newly created elements into the queue.

3. $level(x) - level(y) = \frac{3}{2}$ (Figure 16(d)).
   This case is similar to the second case. The element y is a transitional element and its capsule has a level difference of two with x. Using the same reasoning as above, we conclude that the refinement has to be propagated. We coarsen y and then use standard refinement to refine its capsule. Finally, we insert all elements created by this refinement into the queue.

Notice that during the while-loop (lines 3-16), we never modify elements

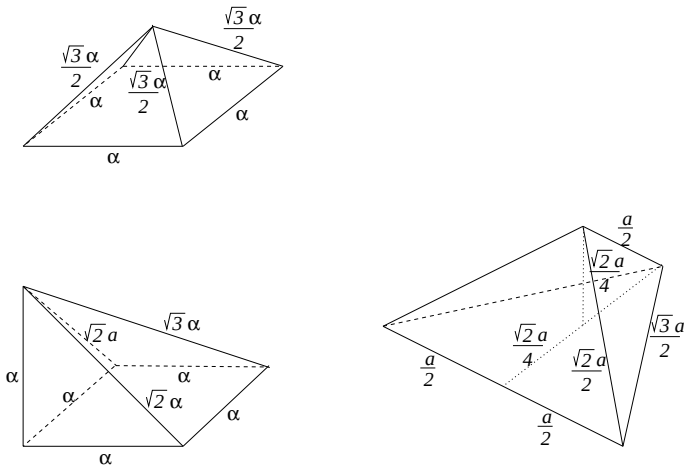


Figure 17: The two different types of pyramids and the tetrahedron used as transitional elements.

that are already in the queue. This is justified, because only elements created in this refinement step are placed in the queue (Lemma 4.2), and we know that these elements do not need to be refined again during the refinement propagation step (Lemma 3.1). So, the refinement propagation will be executed correctly. Figure 14(h) shows the mesh immediately after the queue has become empty and the execution of the subroutine PROPAGATE_REFINEMENT is about to terminate.

As the last step of the refinement procedure, the remaining non-conformal elements are refined, using non-standard refinement. The resulting mesh is legal (Figure 14(i)) and is guaranteed to be the same as the one resulting from applying the procedure shown in Figure 5 to the same initial mesh with the same region R.

As was our objective, the new refinement procedure only takes time proportional to the number of elements that need to be modified. If the number of elements on the boundary of R is asymptotically smaller than the number of elements in R, the entire procedure runs in time proportional to the number of elements in R.

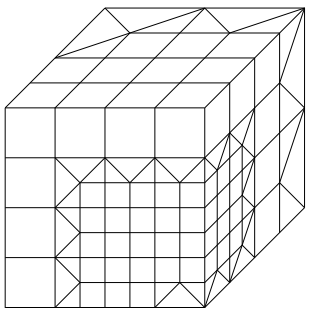


Figure 18: The boundary of a 3D crystalline mesh. Notice how different regions of the mesh have different element density.

5 3D Crystalline Meshes

5.1 Description and Properties

In three-dimensional crystalline meshes, the standard elements are cubes. Transitional elements are of three types: two distinct types of pyramids and one tetrahedron. The three types of transitional elements are shown in Figure 17. 3D crystalline meshes are conformal, and can have elements of different size in different regions of the domain. Figure 18 shows the boundary of a 3D crystalline mesh. Notice that the boundary itself is a legal 2D crystalline mesh.

For the three-dimensional case we chose to slightly modify the definition of a neighbor. Specifically, we say that two elements are *neighbors* if they share at least one *vertex*. This new definition allows us to develop simpler algorithms for mesh generation and refinement. Moreover, its use greatly reduces the number of distinct shapes of transitional elements needed to realize conformal transitions between regions with different sizes. One of our earlier attempts to generate conformal 3D crystalline meshes while using the definition that neighbors are elements that share edges resulted in more than ten different transitional element shapes.

If we are careful during the mesh generation and refinement phase, we can create the elements so that each vertex has a standard local number. In this case, the stiffness matrices of all the elements that have the same shape are the same. In order to create the system of linear equations, we precompute one matrix for each shape and then assemble the system. So, the cost of creating the system is again reduced to the cost of its assembly.

The use of pyramids in 3D meshes is unusual. The main reason for this

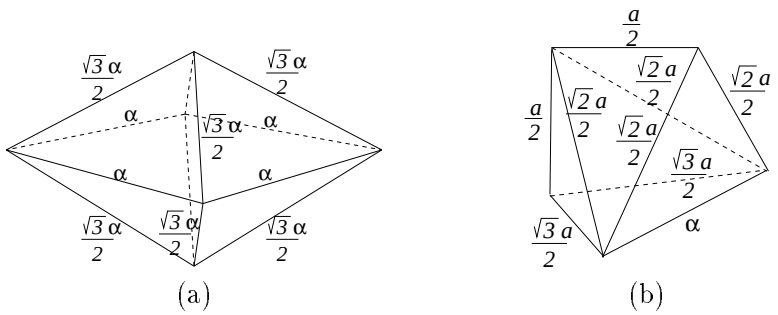


Figure 19: The two neighboring arrangements not exhibiting the Delaunay property. (a) Two pyramids sharing the square face. If α denotes the length of the side of the square face, the length of the sides shared between triangular faces is $\frac{\sqrt{3}\alpha}{2}$ and the height of the entire arrangement is α. (b) Two tetrahedra sharing the face without a right angle. We use α to denote the length of one arbitrarily chosen side. The length of the other sides of the arrangement are also shown.

is that good basis functions for pyramidal finite elements are hard to find. In order to facilitate the use of 3D crystalline meshes we have developed a method for generating Lagrangian basis functions for order k pyramids[2], for any $k > 0$ [8]. The method is inductive and can be easily automated. The resulting functions are compatible with the functions used for Lagrangian cubes and tetrahedra [27, 24].

By examining all four possible shapes in a 3D crystalline mesh we can establish that the smallest planar angle is 35° while the smallest dihedral angle is 45°. Moreover, every 3D crystalline mesh will have the Delaunay property almost everywhere: out of 15 possible neighboring arrangements, 13 are Delaunay. This result was established by exhaustively examining all possible neighboring configurations (using computer simulations), in a way similar to the proof of the corresponding claim for the 2D case. The two non-Delaunay neighboring arrangements are shown in Figure 19.

5.2 Refinement of 3D Crystalline Meshes

The refinement of 3D crystalline meshes conforms to the general philosophy of Section 3 and can be done using Algorithm 1, which was also used for 2D meshes. However, there are some modifications due to the new definition of

[2]An order k Lagrangian pyramid is a pyramid with $k+1$ nodal points placed on each of its edges and with faces that are Lagrangian squares or triangles of order k.

neighboring elements. According to this definition, a 3D mesh is conformable if elements that share a *vertex* have a level difference of at most one. Since we want the refinement propagation step to result in a conformable mesh, we have to propagate the refinement from an element x to an element y even if the two elements share only a vertex. This means that, in general, more elements will be affected by the refinement, but the difference will not be significant. For example, if the region R to be refined has the shape of a parallelepiped, the number of additional elements that will be affected by the refinement is bounded from above by $8l$ (where l is the maximum level occurring in the mesh). Remember that according to Corollary 3.3 the number of elements affected by the refinement is $\Theta(m(R) + l \cdot m(\delta(R)))$.

We also have to specify how the last step of the algorithm (using non-standard refinement to refine non-conformal elements in order to obtain a conformal mesh) can be performed. In the 3D case, the operation is much more complicated than in the 2D case, since now an element can be non-conformal because of non-conformal faces, non-conformal edges or a combination of the two. This results in a very large number of possible con-conformal configurations, each one of which has to be treated separately. An added source of complexity comes from the requirement to keep faces shared between elements conformal, while one or both of the elements are refined. Finally, we would like the procedure for making the mesh conformal to work in one pass.

In order to deal with the complexity of the operation and ensure that the resulting mesh is conformal, we chose a different approach. Elements in the conformable mesh are examined and are chosen for refinement if they share at least one vertex with a higher level element. The type of non-standard refinement used for each element is determined by the number of vertices shared with higher level elements as well as their relative positions (modulo reflections and rotations). This results in 22 different patterns, as many as there are equivalence classes of 3 variable boolean functions (Harvard classes [13]). Associated with each pattern is a specific refinement.

Notice that some of the elements chosen for refinement by the method proposed above are conformal in the original conformable mesh, which means that refining them is not necessary. This implies that the proposed method may result in meshes that consist of more elements than absolutely necessary. In practice, however, the number of conformal elements refined is small, while the added benefits from the simplicity of both the algorithm and the implementation are enormous. For example, if the region R to be refined has the shape of a parallelepiped, only eight conformal elements will be refined (one corresponding to each of the eight vertices of R).

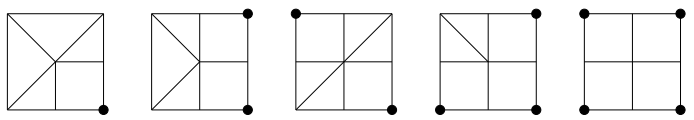


Figure 20: Patterns appearing on the faces of non-conformal cubes to be refined (standard patterns). Vertices shared with higher level cubes are shown as solid circles. Faces that do not share any vertices with higher level cubes are not partitioned

Now we also have to describe how to refine each one of the resulting 22 configurations, so that we can guarantee that the resulting mesh is conformal. To further simplify the problem, we enforce the condition that the cuts on the faces of the cubes that were chosen to be refined follow specific patterns, depending on which vertices are shared with higher level cubes (Figure 20). Faces that do not share any vertices with higher-order cubes should not be partitioned. Notice than an edge is partitioned in two edges if and only if one of its endpoints is shared with a higher level cube. Also notice that if a vertex x is shared with a higher level cube, the face is refined so that a square is formed, having x as one of its vertices. We claim that if all elements that share vertices with higher level cubes are refined so that their faces are cut according to these patterns, the resulting mesh is conformal. This claim is formalized as follows:

Lemma 5.1 *Given a conformable 3D crystalline mesh, if all cubes in the mesh that share vertices with higher-level cubes are refined so that their faces bear a standard pattern, the resulting mesh will be conformal.*

Proof: In the 3D case, non-conformity can arise from either non-conformal edges or non-conformal faces.

We will first consider the edges of the elements, and will prove that after the refinement, all edges shared between elements will be conformal. Let x be an element in the conformable mesh and (v, w) one of its edges. The edge can be shared by up to four cubes in the mesh. We can distinguish two cases.

1. All elements sharing (v, w) have the same level value (Figure 21(a)).

   In this case, if either v or w also belong to an element with higher level value, x will be refined so that a new vertex will be created in the middle of (v, w). The same is true for all other elements sharing

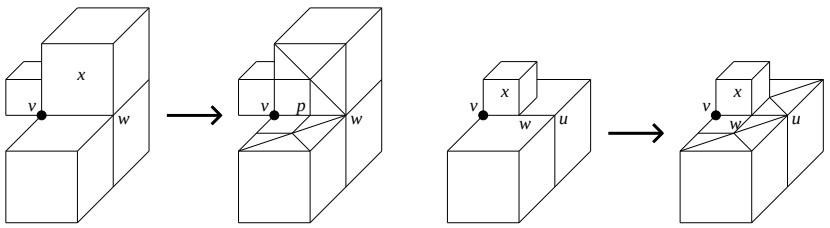


Figure 21: Examples, illustrating why edges will be conformal after the non-standard refinement step. (a) x shares (v, w) with elements having the same level value. Since v also belongs to an element with higher level value, all elements sharing (v, w) are refined. The resulting edges are conformal. (b) x shares (v, w) with elements having lower level values. These elements will be refined, resulting in two conformal edges.

(v, w). Therefore, after the refinement, the two edges resulting from splitting (v, w) will be conformal.

In the example shown, x shares the edge (v, w) with two other elements that have the same level value as x. However, vertex v also belongs to a higher level element. This means that all elements sharing (v, w) will be refined. By looking at the standard refinement patterns in Figure 20, we see that all elements will be refined so that a vertex p will be inserted at the midpoint of (v, w). The resulting edges, (v, p) and (p, w) will be conformal.

2. The elements sharing (v, w) have different level values (Figure 21(b)).

   Since we assume that the elements share the entire edge, this can only mean that the elements that have a different level value than x will be larger, or equivalently, have a lower level value. The mesh is conformable, so the only valid level value for these elements is $level(x) - 1$. This implies that the edges of the larger cubes will have a length which is twice the length of the edges of x. Moreover, by construction, either v or w is a vertex of these larger elements. Without loss of generality, we can assume that it is v. Let (v, u) be the edge of the larger cubes that contains w. Notice that w is the midpoint of (u, v).

   When the lower-level elements are examined by the algorithm, the fact that they share a vertex with a higher level element will be discovered, and they will be refined. Regardless of which other vertices of the elements are shared with higher-level elements, (v, u) will be divided into two edges, by inserting a vertex at its midpoint, which

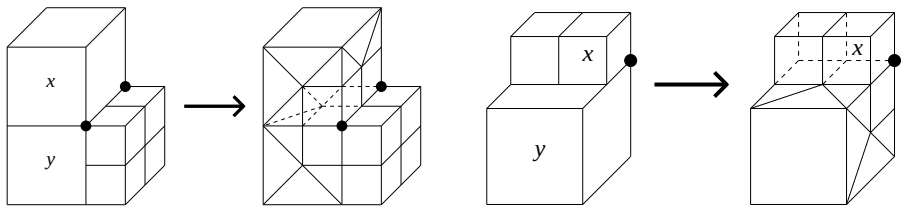


Figure 22: Examples, illustrating why faces will be conformal after the nonstandard refinement step. (a) x and y share a face, that have two vertices belonging to higher level elements. Both x and y are refined, and the shared face is refined according to the same pattern. The dashed lines show how the shared face will be refined. (b) x shares f with the lower-level element y. y is refined so that one of the faces of the resulting elements exactly matches f.

coincides with w. So, after all elements have been refined, both edges resulting from (v, u) will be conformal.

In the example shown, x is sharing (v, w) with two elements with level value $level(x) - 1$. Both of the larger cubes have v as one of their vertices. During the last step of the algorithm both larger elements will be refined. The refinement will divide (v, u) into two edges of equal size, by inserting a vertex at its midpoint. This vertex coincides with w. Both edges resulting from the refinement are conformal.

This proves that after the nonstandard refinement step, all edges in the mesh will be conformal.

Let us now consider the faces of the elements in the mesh. We will prove that after the nonstandard refinement, all faces in the mesh will also be conformal. Since non-conformity can only arise from non-conformal edges or non-conformal faces, this will prove that the resulting mesh will be conformal.

Again, let x be an element in the conformable mesh and f be one of its faces. We have two separate cases, depending on the level of the element x shares f with.

1. Both elements have the same level value (Figure 22(a)).

   In this case all four vertices of the face are shared by both elements. If any of the four vertices is shared with a higher level element, both elements sharing f will be refined and f will be partitioned during the refinement. However, both elements will partition f according to the

same pattern, and therefore the resulting faces will be conformal. If none of the four vertices of f belongs to a higher level element, the element might still be refined (caused by other vertices that are not in f), but in any case f will not be partitioned and will be conformal in the resulting mesh.

In the example shown, elements x and y share a face. Two of the vertices of the shared face are also shared by higher level elements. Therefore, both elements are refined. In this case, the shared face is refined according to standard pattern 2a, and all five resulting faces are conformal.

2. The element with which x shares f has a different level value (Figure 22(b)).

   Let this element be y. Using the same reasoning as in the case of the edges, we conclude that y should have a lower level value, and since the mesh is conformable, its level value is $level(x) - 1$. In this case, the two elements also share exactly one of their vertices. Therefore, y is chosen to be refined, since it shares a vertex with x which has a higher level value. The face of y that contains f will be refined according to the appropriate standard pattern. By looking at the standard patterns in Figure 20, we see that a square will be formed, that matches f. So, after the refinement f will be conformal.

   In the example shown, x is sharing f with a lower level element y. During the nonstandard refinement phase, y is refined. In this specific instance, the face of y that contains f will be refined according to standard pattern 2a, since it contains two vertices that also belong to higher level elements, and both of these vertices are the endpoints of the same edge. One of the faces of the resulting elements matches f exactly.

Notice that for both the edge conformity and the vertex conformity proof we used the fact that the original mesh is conformable, since we assumed that elements sharing a vertex will not have a level difference of more than one. □

Lemma 5.1 above proves that *any* refinement that cuts the faces of the cubes in a way consistent with Figure 20 will result in a conformal mesh. This simplifies the problem, but doesn't solve it completely. For each of the 22 possible neighboring arrangements we have to describe how exactly the refinement should be performed. We will devote the next subsection to a discussion of this problem.

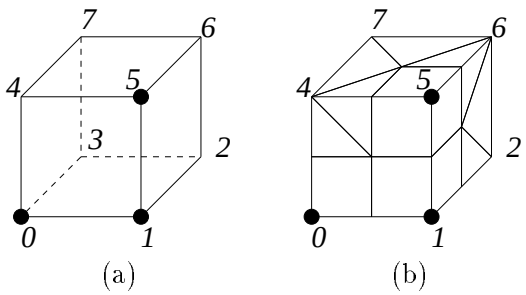


Figure 23: (a) A non-conformal cube that shares three vertices with higher level cubes. These vertices appear as solid circles. Also shown is the standard numbering of the vertices. (b) The faces of the cube have been marked by using the appropriate standard pattern.

5.3 Non-standard Refinement

For each one of the 22 possible neighboring configurations, there are several ways to refine the element so that the standard patterns appear on the element's faces, and the elements generated from the refinement are conformal. All of these refinements would result in conformal meshes and are acceptable.

However, while choosing specific refinements for each of the 22 neighboring configurations we have two more goals in mind. First, we would like to minimize the number of elements generated by each non-standard refinement, since more elements imply that we will need more time to generate the system of linear equations. Moreover, we would like to minimize the number of distinct element shapes used in the crystalline mesh, for reasons of simplicity and efficiency.

For example, we could perform the non-standard refinement as follows. First we refine the faces of the cube according to the appropriate standard patterns. Then we create one element for each of the faces appearing on the cube by connecting each of its vertices to the center of the cube. This process is easy to describe and to automate and will result in conformal elements. However, we found that it generates many more elements than necessary. Moreover, some of the transitional element shapes generated in this way have small angles and therefore may result in ill-conditioned systems of linear equations. By carefully choosing the way we perform the non-standard refinement we can generate meshes that are better conditioned and have fewer elements.

To illustrate the approach, we discuss in some detail one specific instance.

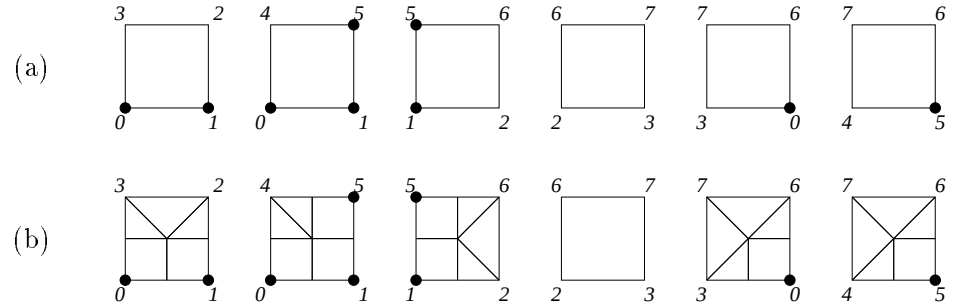


Figure 24: (a) All the faces of the non-conformal cube shown in Figure 23. (b) The faces of the cube marked with the appropriate standard patterns.

Specifically, we consider the case of a cube that shares three vertices, all placed on the same face, with higher-level cubes. This cube is shown in Figure 23(a), where vertices shared with higher-level cubes appear as solid circles. In the same figure, we show the standard numbering of the vertices of the cube. In Figure 23(b) the same cube is shown, where its three visible faces have been marked with the appropriate standard patterns as specified in Figure 20. In order to better explain how all the faces of the cube will be partitioned, we show all the faces of the cube in Figure 24(a), emphasizing the vertices that are shared with higher-level cubes. Figure 24(b) shows the appropriate standard pattern for each of the faces. Notice that if we were to use the procedure described above to refine this cube, we would generate 26 transitional elements. We will now show a much more efficient refinement.

The entire process is shown in Figure 25. We start by removing a cube, that is formed around vertex 5 of the original cube, which is shared with a higher-order element (2.). Another cube, formed next to vertex 1, is removed next (3.). We continue by removing two pyramids, whose square faces were formed when the first cube was removed (4. and 5.). We continue removing elements from the cube in the same fashion, always keeping in mind that the resulting elements should divide the faces of the original cube into the appropriate standard patterns, that shared faces and edges should be conformal, and that only cubes and the transitional elements shown in Figure 17 should be used (6.–15.). We conclude that this element can be refined by using only 15 elements (three of which are cubes).

Figure 26 shows the same refinement, in a more compact way. Figure 26.1 shows the cube, the vertices shared with higher level elements and the appropriate standard patterns for each of the three visible faces. Fig-

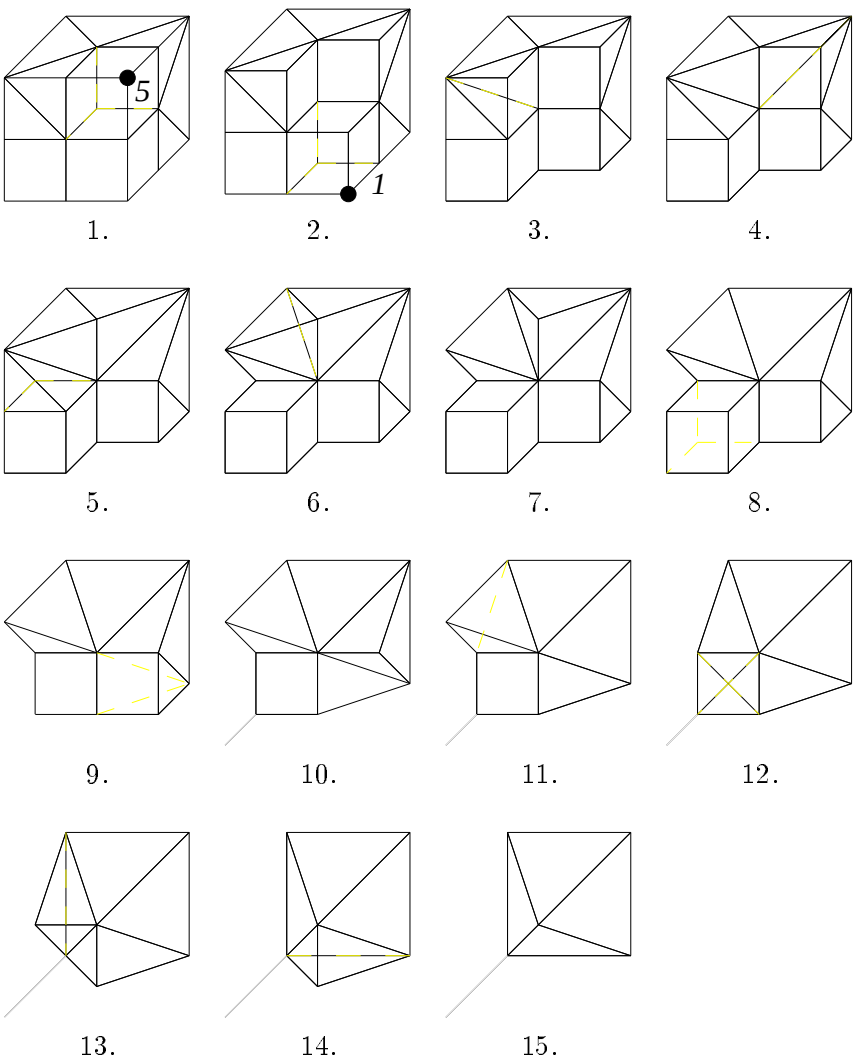


Figure 25: Non-standard refinement used for an element sharing three vertices, all on the same face, with higher-level cubes. Each figure can be obtained from the previous one by removing one element. Dashed lines show the "hidden" sides of the element that will be removed next. The refinement can be done using 15 elements.

ure 26.2 shows the refinement of part of the original cube. Each element has been marked with a number that denotes the last step in Figure 25 where the element is present. In the background we show the part of the cube that

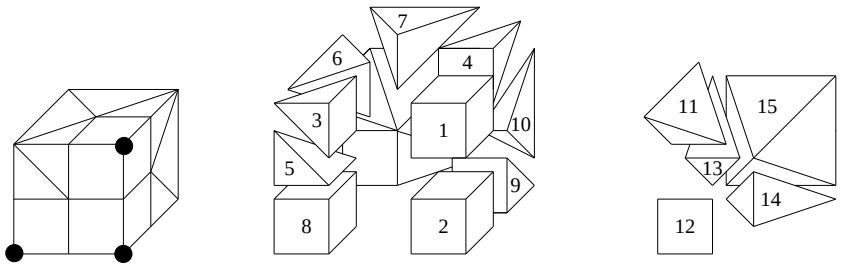


Figure 26: Another way of showing the same refinement. (1.) The faces of the cube are shown marked with the standard patterns. (2.) The refinement of part of the cube is shown. (3.) The refinement of the rest of the cube is shown. Elements are marked with a number corresponding to the step in Figure 25 where they appear for the last time.

remains after removing all the elements shown. It is identical to the one shown in Figure 25(11.). Finally, Figure 26.3 shows how the remaining part of the cube is refined, resulting in two pyramids and three tetrahedra.

In the appendix we show all 22 possible neighboring configurations as well as an appropriate refinement for each one of them, which was generated in the same way as the one described here.

6 Implementational Issues and Experimental Results

In order to test how crystalline meshes perform in practice, we have developed a prototype program (in collaboration with J. Castaños and building on previous work of J. Castaños and J. Savage [7, 21, 22], whose focus was the parallelization of refinements) that uses the finite element method to solve Poisson's equation with Dirichlet boundary conditions in both 2D and 3D. In order to be able to make comparisons, we implemented separate procedures that can also handle structured meshes as well as unstructured non-crystalline meshes. The prototype is written almost entirely in C++ (we use Blas and Lapack subroutines for solving the system of linear equations). We found that C++ gives us the flexibility needed, to handle different types of meshes and different types of elements, and allows for easy and efficient memory management. Every important data structure in the program is handled through smart pointers and is automatically deleted when it is no

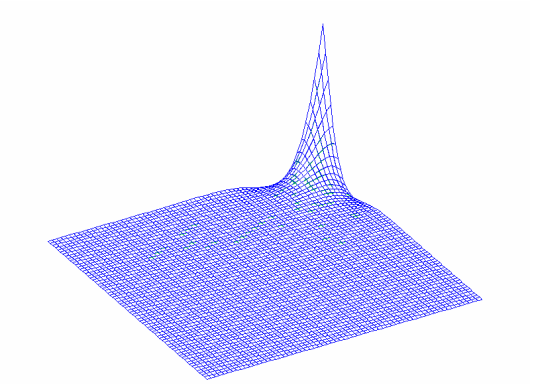


Figure 27: The solution to the problem used as a benchmark. The solution changes rapidly in the region next to one of the corners.

longer needed, freeing the allocated memory [11]. This allows us to handle very large meshes (up to 100,000 points) without encountering memory problems.

We start with a very coarse mesh that correctly approximates the problem's domain. Given a target error bound, the system of equations is assembled and solved, the regions of interest are found and, if the error is too large, refined. The process is repeated until the desired error bound is achieved. During the whole process the mesh is kept conformal. The entire refinement history of each of the elements is kept in the form of a tree. Elements in the current mesh are the leaves of the tree. They are also threaded to form the element list. When an element x is refined, the elements resulting from this refinement become the children of the data structure representing x in the refinement tree. Also, x is removed from the element list and its children are added in its place.

We compared the performance of crystalline meshes to that of both structured and unstructured meshes for problems with region of interest. In this paper we will use as a benchmark Problem (3.61) in [20] (more experimental results will be presented in a future paper). The domain of the problem, as well as the solution, are shown in Figure 27. Notice that the solution has the same value in almost the entire domain, but changes rapidly close to one of the corners. Since the problem exhibits regions of interest, we expect unstructured meshes to do better than structured meshes. We also expect adaptive refinement techniques to prove extremely useful.

We used the benchmark to compare the performance of four different

Target error L_∞- norm	Structured Mesh - Triangles				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	2048	1089	83	4	1
0.005	8192	4225	150	2	4
0.001	32768	16641	255	2	18
0.0005	32768	16641	255	1	10
0.0001	>524288	>263169	>374	>3	>180
Target error $L2$- norm	Structured Mesh - Triangles				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	32	25	7	1	0
0.001	512	289	43	3	0
0.0001	8192	4225	150	3	4
0.00001	32768	16641	255	2	16
0.000001	>524288	>263169	>374	>3	>180

Table 1: Results obtained when using structured meshes consisting of triangles. For each target error, we give the number of elements and vertices in the final mesh, the number of Conjugate Gradient iterations needed to solve the final system, the number of adaptive iterations and the time the entire adaptation process took (in seconds). For both error norms, the smallest error in the table was not achieved. The refinement process could not finish due to out-of-memory problems.

Target error L_∞- norm	Structured Mesh - Quadrilaterals				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	1024	1089	59	4	0
0.005	4096	4225	108	2	2
0.001	16384	16641	183	2	11
0.0005	16384	16641	183	1	8
0.0001	>262144	>263169	>235	>3	>30
Target error $L2$- norm	Structured Mesh - Quadrilaterals				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	256	289	31	3	0
0.001	1024	1089	59	2	0
0.0001	4096	4225	108	2	2
0.00001	16384	16641	183	2	11
0.000001	>262144	>263169	>235	>3	>30

Table 2: Results obtained when using structured meshes consisting of quadrilaterals to solve the problem. In this case, too, the smallest target error in the table could not be achieved due to memory problems.

types of meshes: structured meshes consisting of triangles or squares, unstructured meshes, consisting of triangles that have been adapted using the very popular Rivara refinement (presented in [18, 19]) and crystalline

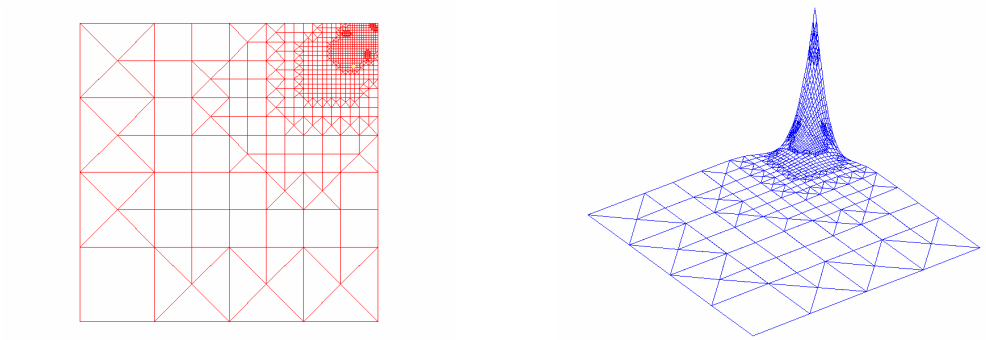
Target error L_∞- norm	Unstructured Mesh - Rivara refinement				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	174	101	28	7	0
0.005	244	137	33	3	0
0.001	2054	1060	90	11	1
0.0005	4554	2321	130	5	6
0.0001	>181444	>90872	>699	>15	>150
Target error $L2$- norm	Unstructured Mesh - Rivara refinement				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	32	25	7	1	0
0.001	218	124	30	5	0
0.0001	796	421	61	4	0
0.00001	2004	1039	92	3	1
0.000001	7594	3867	160	4	9

Table 3: Results obtained when using unstructured meshes consisting of triangles and the popular Rivara refinement method. The smallest error in the L_∞ norm could not be achieved.

Target error L_∞- norm	Unstructured Mesh - Crystalline				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	176	208	32	5	0
0.005	298	249	40	2	0
0.001	935	754	68	3	0
0.0005	3234	2530	127	5	5
0.0001	31183	27590	384	4	66
Target error $L2$- norm	Unstructured Mesh - Crystalline				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	112	97	23	3	0
0.001	192	161	31	2	0
0.0001	582	507	57	2	0
0.00001	1974	1795	107	2	1
0.000001	7169	6703	198	2	6

Table 4: Results obtained when using crystalline meshes and the refinement method suggested in this paper.

meshes. In all cases we have a target error and we compare the number of elements and vertices in the mesh that achieves it, the number of Conjugate Gradient iterations needed to solve the final system of linear equations, the number of adaptive iterations needed to obtain the final mesh and the time (in seconds) the whole adaptation process took. The results are presented in Tables 1–4.



(a) (b)

Figure 28: Crystalline mesh obtained by starting with a very coarse mesh and using adaptive refinement for the benchmark problem. (a) top view (b) the value of the solution is indicated by the hight of each point. Notice that at the region of interest the element density is much higher than in other regions of the domain

As expected, structured meshes are outperformed by both the unstructured and the crystalline meshes. Also notice that, consistently with the results presented in [1], quadrilateral meshes are performing better than triangular meshes. Both types of meshes could not achieve the smallest error in the table for both error norms. The number of elements required in order to obtain the desired accuracy is extremely high and the execution of the program had to be interrupted due to out-of-memory problems. This results are consistent with the results we obtained using several other problems as benchmarks.

We now turn our attention to comparing the triangular meshes obtained by using the Rivara refinement with the crystalline meshes. We see that when the required accuracy of the solution is low, the two methods give comparable results in terms of mesh size and solving time. However, when a high-accuracy solution is needed, crystalline meshes outperform meshes constructed using Rivara refinement. Furthermore, even in the cases where the size of the mesh is compatible, the time needed to achieve the error bound for crystalline meshes is smaller, and the adaptation step for crystalline meshes is simpler, easier to implement and faster, since the structure of the propagation guarantees that refinement paths have no cycles.

In Figure 28 we show one of the crystalline meshes that were generated by the adaptive refinement process. It is the mesh that achieves an error of

0.001 in the L_∞ norm and consists of 935 elements and 754 vertices. Notice how the element density is much higher at the corner of the domain where the value of the solution changes rapidly.

7 Conclusions

In this paper we have presented integer-coordinate crystalline meshes and have argued that they give us the benefits of both structured and unstructured meshes, by offering the flexibility of different element densities in different regions, and the ability to generate the system of equations quickly and accurately. We have also described how, given an crystalline mesh, we can refine regions of it to get a better approximate solution.

Some issues still remain. We are currently investigating the problem of generating the mesh, given a description of the domain (both in 2D and 3D). It seems that in the 3D case, and for problems where the boundary is irregular, we will need to use some tetrahedral elements that are not homothetic to the elements shown in Figure 17. However, we believe that by carefully choosing the size of the elements in the regions close to the boundary, we can get a very good approximation of it by using only a small number of distinct shapes.

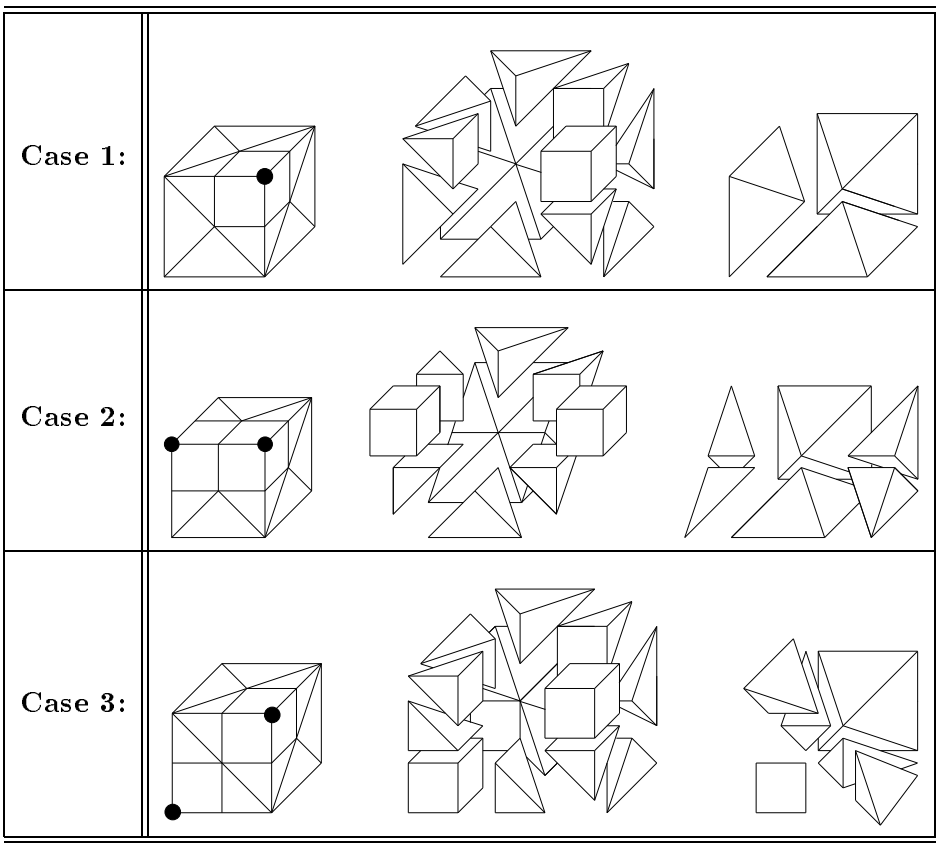
Another issue worth investigating is the parallel mesh generation and refinement. We are particularly interested in parallel algorithms for the Coarse Grained Multicomputer model, which seems to be close to the parallel computers available in the market today. Both problems seem to have enough locality and therefore efficient parallel algorithms seem likely to exist. The refinement problem seems to be the most difficult because of the propagation step and the load imbalance that every partial refinement step is likely to generate.

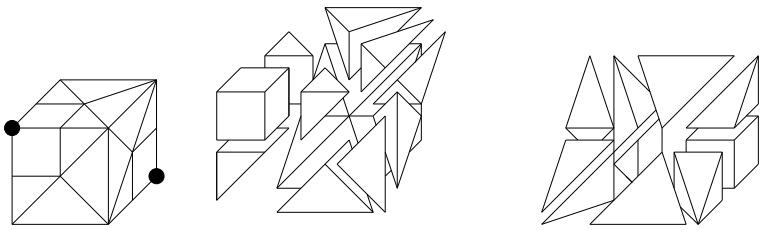
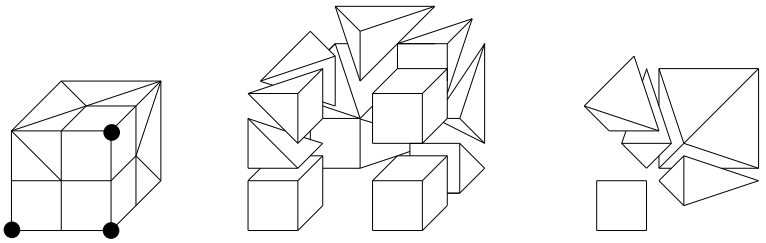
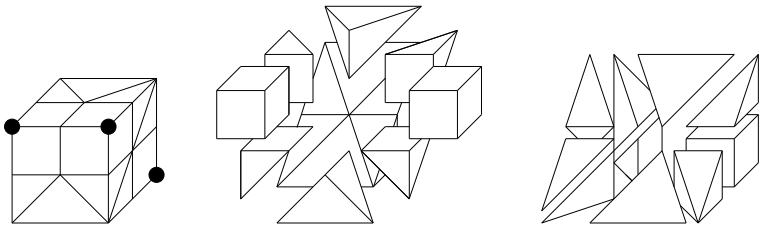
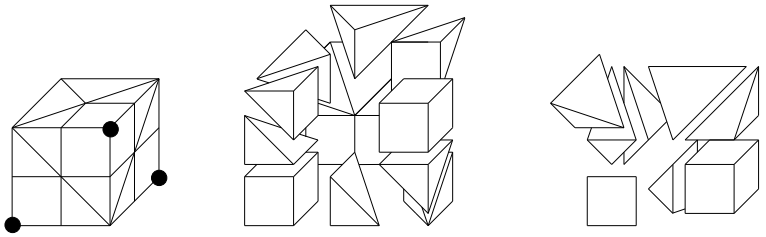
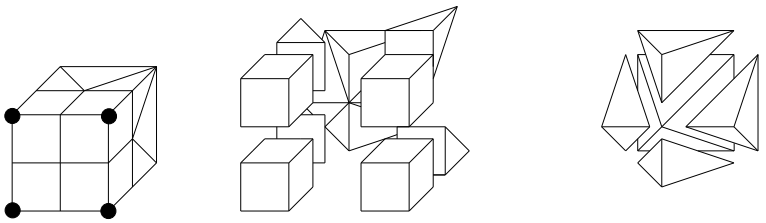
We currently have a working prototype, implemented in C++, that uses the finite element method to solve Poisson's equation with Dirichlet boundary conditions both in two and three dimensions. It has been programmed so that it can handle crystalline and non-crystalline unstructured meshes, and compute error bounds which are used to selectively refine or coarsen regions of the mesh. We used it to obtain comparative results. Our plan for the near future is to develop a mesh generator for crystalline meshes that cover complex domains, both in 2 and 3 dimensions.

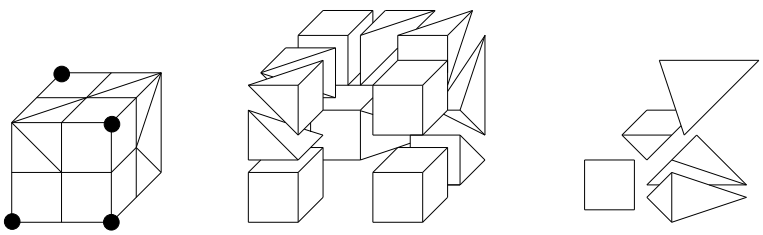
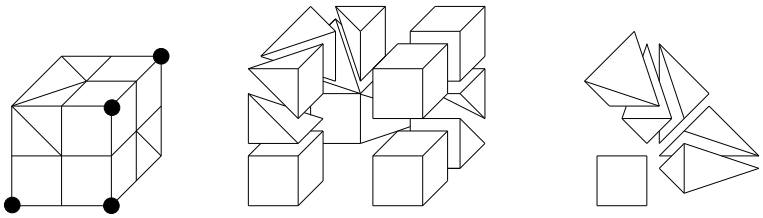
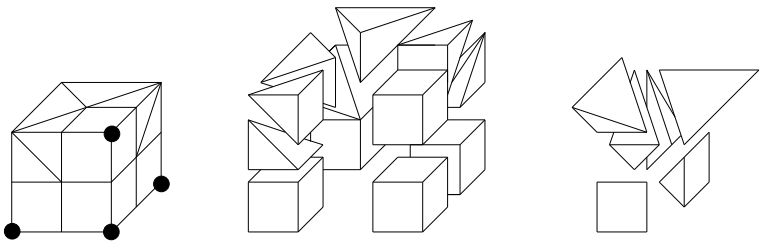
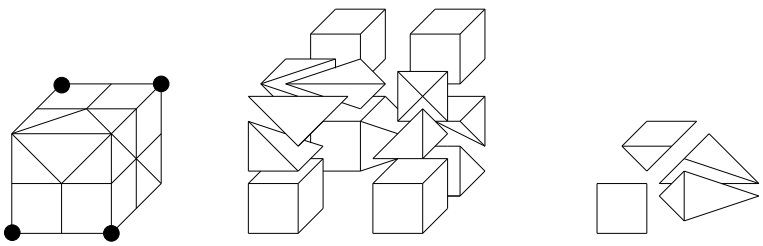
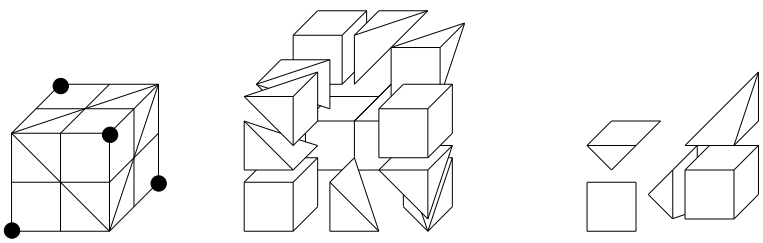
Appendix

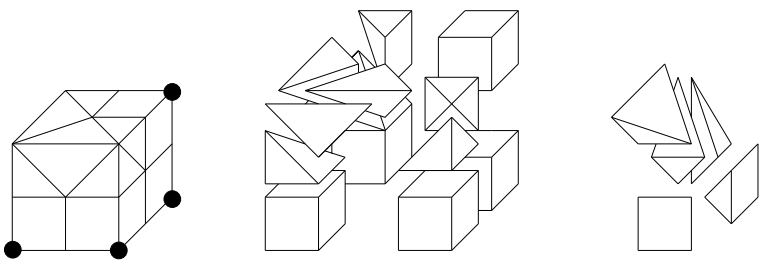
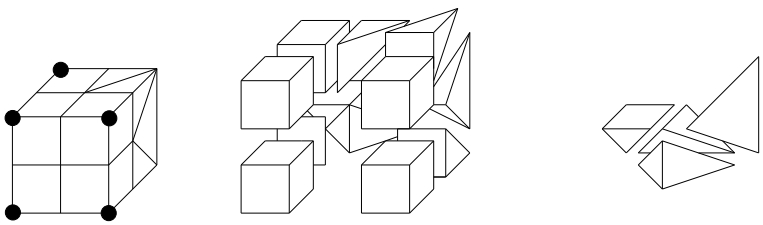
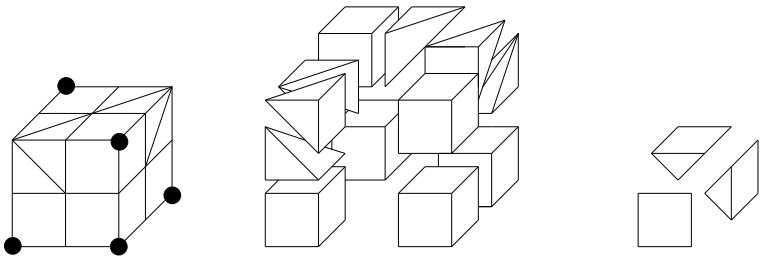
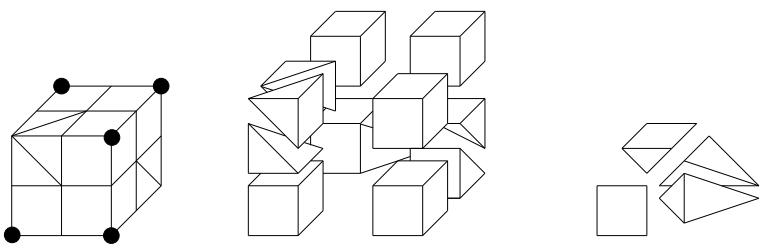
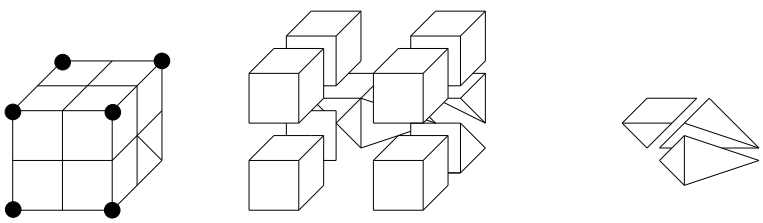
A Non-standard refinement of non-conformal cubes

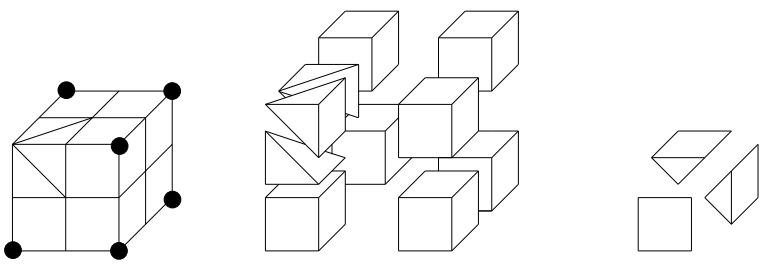
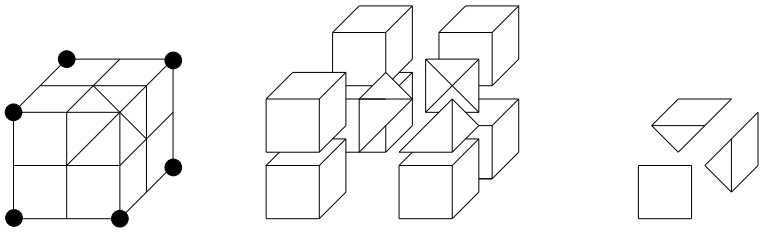
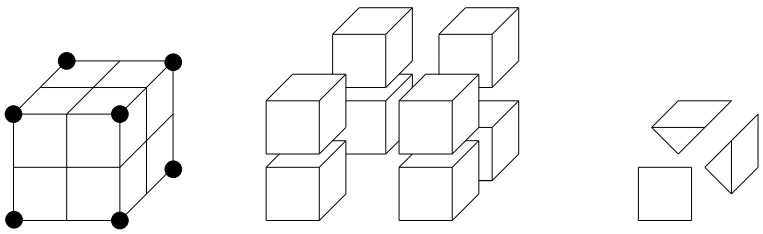
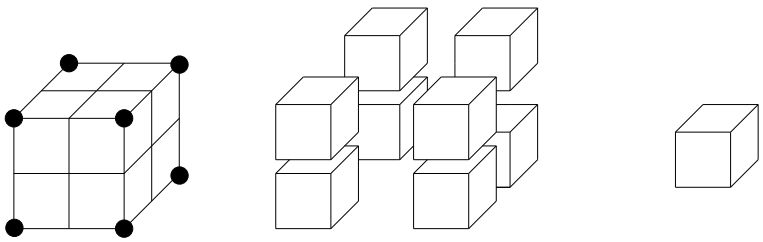
In Section 5 we saw that during the refinement of 3D crystalline meshes we need to identify cubes that share one or more vertices with a higher level cube and refine them (using non-standard refinement) in order to obtain a legal mesh from a conformable mesh. We also saw that there are 22 distinct neighboring configurations and that if each element is refined so that its faces are cut in a way consistent with Figure 20, the resulting mesh will be conformable. Now we will present the refinement used in all 22 configurations. Notice that in all cases only cubes and the shapes shown in Figure 17 are used.



Case 4:	
Case 5:	
Case 6:	
Case 7:	
Case 8:	

Case 9:	
Case 10:	
Case 11:	
Case 12:	
Case 13:	

Case 14:	
Case 15:	
Case 16:	
Case 17:	
Case 18:	

Case 19:	
Case 20:	
Case 21:	
Case 22:	

References

[1] S. E. Benzley, E. Perry, B. Merkley, K. Clark, and G. Sjaardema. A comparison of all hexagonal and all tetrahedral finite element meshes for elastic and elastic-plastic analysis. *4th International Meshing Roundtable*, pages 179–192, 1995.

[2] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1980.

[3] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *Journal of Computer and System Sciences*, 48(3):384–409, June 1994.

[4] M. W. Bern, D. Eppstein, and S. H. Teng. Parallel construction of quadtrees and quality triangulations. In *Proc. 3rd Worksh. Algorithms and Data Structures*, number 709 in Lecture Notes in Computer Science, pages 188–199. Springer-Verlag, August 1993.

[5] D. S. Burnett. *Finite elements analysis: from concepts to applications.* Addison-Wesley, 1987.

[6] G. F. Carey. *Computational grids.* Taylor & Francis, 1989.

[7] J. G. Castaños and J. E. Savage. The dynamic adaptation of parallel mesh-based computation. Technical Report CS-96-31, Department of Computer Science, Brown University, October 1996. Sun, 13 Jul 1997 18:30:15 GMT.

[8] V. Chatzi. Finding basis functions for pyramidal finite elements. *3rd CGC Workshop on Computational Geometry*, 1998.

[9] W. J. Coirier and K. G. Powel. An accuracy assessment of Cartesian-mesh approaches for the Euler equations. *Journal of Computational Physics*, 117:121–131, 1995.

[10] B. Delaunay. Sur la sphère vide. *Bull. Acad. Science USSR VII: Class. Sci. Mat. Nat.*, pages 793–800, 1934.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns.* Addison-Wesley, 1994.

[12] P. L. George. *Automatic mesh generation: Application to finite element methods.* John Wiley & Sons, 1991.

[13] M. A. Harrison. *Combinatorial problems in Boolean algebras and applications to the theory of switching.* PhD thesis, University of Mitchigan, 1963.

[14] C. Johnson. *Numerical solutons of partial differential equations by the finite element method.* Cambridge University Press, 1995.

[15] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 8th Annual Symposium on Computational Geometry (SCG '92)*, pages 212–221, Berlin, FRG, June 1992. ACM Press.

[16] J. T. Oden, L. Demkowicz, T. Strouboulis, and P. Devloo. *Adaptive methods for problems in solid and fluid mechanics.*, chapter 14. John Wiley & Sons, 1986.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[18] M. Rivara. Selective refinement/derefinement algorithms for sequences of nested triangulations. *Int. J. Numer. Meth. Eng.*, 28:2889–2906, 1989.

[19] M. Rivara and C. Levin. A 3-d refinement algorithm suitable for adaptive and multigrid techniques. *Com. in applied num. methods*, 8:281–290, 1992.

[20] U. Ruede. *Mathematical and computational techniques for multilevel adaptive methods*, volume 13. SIAM, 1993.

[21] J. E. Savage and J. Castanõs. Object migration for parallel dynamic load balancing. In *Proceedings of the 8th symposium on parallel and distributed processing*, New Orleans, October 1996.

[22] J. E. Savage and J. Castanõs. The dynamic adaptation of parallel mesh-based computation. In *Procs. of the eighth SIAM conf. on parallel processing for scientific computation*, March 1997.

[23] R. Schneiders, R. Schindler, and F. Weiler. Octree-based generation of hexahedral element meshes. In *Proceedings 5th International Meshing Roundtable*, 1996.

[24] H. R. Schwarz and J. R. Whiteman. *Finite element methods.* Academic press, 1988.

[25] M. S. Shephard and K. G. Marcel. Automatic three-dimensional mesh generation by the finite octree technique. *Int. J. Numer. Meth. Eng.*, 32:709–749, 1991.

[26] M. A. Yerry and M. S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *Int. J. Numer. Meth. Eng.*, 20:1965–1990, 1984.

[27] O. C. Zienkiewicz and R. L. Taylor. *The finite element method.* McGraw-Hill, 1994.