TECHNICAL RESEARCH REPORT

Generating 3D Models of MEMS Devices by Process Emulation

by S. Bellam, S.K. Gupta, A.K. Priyadarshi

TR 2002-57

Generating 3D Models of MEMS Devices by Process Emulation

S. Bellam, S.K. Gupta, A.K. Priyadarshi
Mechanical Engineering Department and Institute for Systems Research
University of Maryland
College Park, MD 20742

Abstract

MEMS designers often use numerical simulation for detecting errors in the mask layout. Numerical simulation involves generating 3D models of MEMS device from the mask layout and process description. The generated models can be meshed and simulated over different domains. This report describes an efficient algorithm that can generate 3D geometric models of MEMS devices. Specifically, the algorithm emulates the manufacturing of a single functional polysilicon layer MEMS devices using the MUMPS™ process.

1. Introduction

MicroElectro Mechanical Systems (MEMS) [Byrz94, Howe90, Tang97] are made up of extremely small mechanical elements, often integrated together with electronic circuitry. Micromachining, the enabling technology for MEMS device fabrication, is the technique for making structures and moving parts whose sizes are in the order of microns. These technologies are capable of making motors, pivots, linkages, and other mechanical devices of extremely small sizes. Micromechanical parts tend to be rugged, respond rapidly, use very little power, occupy very little space and have several other advantages over conventional macro machines. These devices have wide range of applications - inertial sensors, thermal sensors, optical switches etc.

One of the tools available to a MEMS designer is numerical simulation. Use of numerical simulation for detecting errors is computationally expensive but useful as it takes lesser time and is not as expensive as a trial and error design method. Numerical simulation involves developing the 3D models from the layout and process description. The model is then assigned material properties depending upon the process description. The model is then meshed either manually or by using an automatic mesh generator [Lakd99]. The finite/boundary element model is simulated using finite or boundary element simulation over different domains. Efficient and accurate emulation algorithms can rapidly generate 3D MEMS device models from the mask layouts. This will help designers in 3-dimensional visualization and design rule checking. Generating 3D models is also the first step in numerical simulation of MEMS devices. The generated models can be meshed and simulated over different domains. Hence efficient emulation algorithms are needed, as it would help speed up the design process. This report describes an efficient algorithm that can generate 3D geometric models of MEMS devices manufactured using the MUMPS™ process. This algorithm emulates the MUMPS™ manufacturing process described in [Koes94].

The reminder of this report is organized in the following manner. Section 2 describes some of the work done in MEMS process emulation. Section 3 describes the background required to understand this report. Section 4 describes the definitions required for this report. Section 5 describes the algorithms used to emulate the MEMS device fabrication. Section 6 provides the analysis of the algorithms described in Section 5. Section 7 describes the implemented system and Section 8 gives some of the results.

2. Related Work

Generating the three-dimensional solid model of the MEMS device involves the emulation or simulation of MEMS processes from the mask data available in the mask layout file and the corresponding process description. This geometric model is used for visual inspection and also for generating finite element meshes for subsequent simulation. The geometry-based algorithms used for process emulation are fast and usually result in a three-dimensional topography close to the fabricated reality. In quite a few cases where high aspect ratios are involved, the customary and desired etchants are highly anisotropic; the transformation from mask to shape is geometrically complex. In such cases there may be a requirement for generating the topography starting out with consistent physics-based process simulation algorithms. Process emulation is a geometry-based modeling approach. Non-idealities of the processes are ignored or simplified, which allows for the fast creation of the three-dimensional solid model using data from the design layout and the corresponding process characteristics.

Emmenegger *et. al.* developed a software called MemCel [Emme98]. It is a prototyping tool for the three-dimensional visualization and subsequent finite element simulation of MEMS devices. It generates a cell representation of the three-dimensional topology of the device. The cell representation correctly reflects the layer structure of the device in the direction of the normal of the wafer surface. It generates a tensor mesh for the device geometry, considering mesh generation for device simulations the ultimate goal. Mesh generation for finite element or boundary element stress analysis places stringent requirements on the fidelity of the resultant topography, as well as the accuracy of the built-in material stresses due to high temperature manufacturing processes. Here, only physics-based three-dimensional process simulation will provide sufficient accuracy. Meshes for solving the Maxell electromagnetic wave equation, e.g., by the finite difference time domain method, are somewhat less stringent.

DeVoe *et. al.* developed a software called 3DMX [DeVo98] which generates solid models for MEMS structures from the given process description and mask layout. It is capable of modeling a wide range of typical MEMS device features such as conformal and planar surfaces; vias, etch holes, etc. 3DMX is capable of handling devices with arbitrary aspect ratios, non-Manhattan geometries and supports both light and dark field layers. The 3DMX software algorithms are geometry based and cannot achieve the accuracy of physics-based algorithms. By allowing sufficiently detailed processing rules 3DMX can provide fast generation of solid models with sufficient accuracy to describe complex MEMS structures produced by a wide variety of fabrication techniques. The extracted solid models can be used for three-dimensional visualization, capacitance extraction and coupled system analysis. It emulates sidewall coverage but cannot support non-vertical sidewalls and rounded corners.

Osterberg and Senturia developed a script in the I-DEAS macro programming language called MemBuilder [Oste95]. The input is a 2D-mask layout design in CIF format and a user designed simple version of the process sequence using the MEMCAD 2.0 Process Editor. It supports surface micro-machined and wafer bonded processes. MemBuilder mimics the simplified process by a sequence of operations to create layers in the solid model. It supports conformal, planar, stacked, via deposit types. Each layer is built by a deposit followed by a corresponding etch. The constructed solid model is in the I-DEAS environment and can be used for visual checking and meshed for further analysis using software such ABAQUS, FASTCAP and CoSolve-EM.

3. Background

The algorithm described in this report has the ability to emulate the manufacturing of a single functional polysilicon layer MEMS device using the MUMPS™ process. This includes the deposition and etching of the nitride layer, Poly0, Oxide1 and Poly1 layers. Deposition processes for the Poly0 and Poly1 layer involve the use of "light mask fields" for patterning. A light mask field indicates that the material that is to be left behind after development is described in the mask layout. A dark mask field indicates the opposite. Oxide layer is patterned using dark mask layouts. Anchors, Dimples and Hole layouts are dark mask layouts and are used to pattern oxide and polysilicon layouts to open connections to the nitride layer for the Poly1 layer, and to put holes and dimples in the Poly1 layer.

The first step in creating 3D models of the device is the parsing of the CIF file. The CIF file contains a set of set of non-intersecting Manhattan rectangles. Each set of rectangles describes the layout of a mask. A separate list of rectangles is maintained to describe each mask layout. The largest and smallest co-ordinates of the rectangles are kept track of and are used to create a rectangle such that it overlaps every rectangle in the device layout. This rectangle is used to create the nitride layer. The nitride layer is a single cuboid that has co-ordinates so as to overlap all rectangles from the other layers. The height associated with the cuboid is as defined by the MUMPS™ process. The *Z-Map* is a list of cuboids that maintains the up-to-date status of the 3D model. At the end of the modeling process, this list will contain the list of cuboids that represent the MEMS device model. A brief description of the emulation process follows:

1. Cuboid corresponding to the nitride layer is created and placed in the list *Z-Map*. This corresponds to the nitride layer deposition.
2. The 2D region described by the Poly0 mask layout is extruded and placed on the Z-map. This corresponds to the Poly0 deposit and etch.
3. The 2D region described by the Anchor1 layout is subtracted from a rectangle whose dimensions equal the dimensions of the rectangle used to generate the "nitride" cuboid. The resultant 2D region is extruded and conformably stacked on top of the cuboids in list *Z-Map*. This corresponds to the Oxide1 deposit and subsequent etch using the Anchor1 mask layout.
4. The 2D region described by the dimple mask is extruded and placed in the model such that the top surface of the dimple is in contact with the top surface of the Oxide1 layer.
5. The 2D region described by the Hole1 layer is subtracted from the 2D region described by the Poly1 mask. The result is extruded and conformably stacked on the current device model. This corresponds to the deposit of the Poly1 layer and the etch using the Poly1 and Hole1 mask layouts.
6. The cuboids representing the Oxide1 layer in the list *Z-Map* are deleted. This corresponds to the oxide release to free the mechanical structure.

4. Definitions

The required definitions for understanding the remainder of this report are described below:

Rectangle: A geometric shape represented by four parameters X_{min}, Y_{min}, X_{max}, Y_{max}. Each of the four parameters represents a line and the area bounded by the four lines represents the rectangle. The line segments formed due to the intersection of the four lines form the edges of the rectangle. Using the point set notation, rectangle r_i can be defined as a set of points:

$r_i = \{(x,y) \mid X_{min} \quad x \quad X_{max}\ ,\ Y_{min} \quad y \quad Y_{max}\}$

The interior of rectangle r_i can be defined as:

$i(r_i) = \{(x,y) \mid X_{min} < x < X_{max}\ ,\ Y_{min} < y < Y_{max}\}$

The boundary of rectangle r_i can be defined as:

$b(r_i) = r_i - i(r_i)$

The boundary of Rectangle r_i is the union of the four edges E_l, E_t, E_r and E_b.

Cuboid: A geometric shape represented by 6 parameters X_{min}, Y_{min}, Z_{min}, X_{max}, Y_{max}, and Z_{max}. Each parameter represents a half-space. The volume bound by these 6 half-spaces represents the cuboid. Using the point set notation, cuboid c_i can be defined as a set of points:

$c_i = \{(x,y,z) \mid X_{min} \quad x \quad X_{max}\ ,\ Y_{min} \quad y \quad Y_{max}\ ,\ Z_{min} \quad z \quad Z_{max}\}$

Line Segment: Here a line segment is a geometric entity that is represented by 4 parameters. The first parameter is a binary parameter and can be either *vertical* or *horizontal*. The second parameter defines line on which the line segment lies (i.e., for horizontal line we store the y value and for vertical lines we store the x value). The third and fourth parameters are the end points of the line segment (i.e., for the horizontal line we store the x-coordinates of the two end points and vice versa).

5. The Emulation Algorithm

This section describes the algorithms used to create the 3D models of MEMS devices.

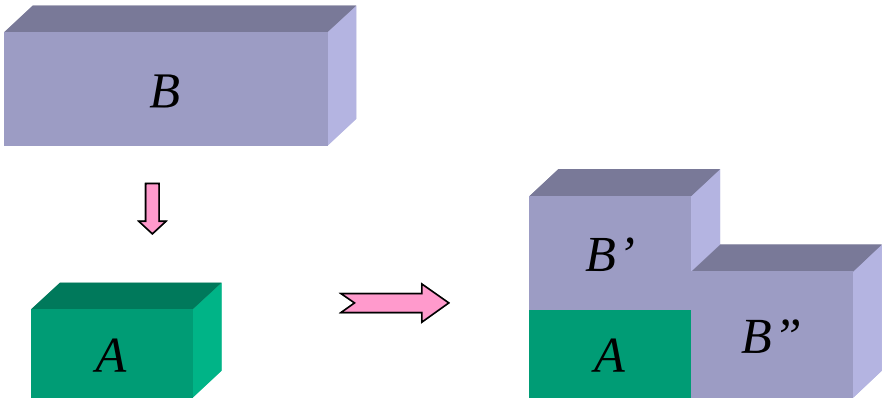
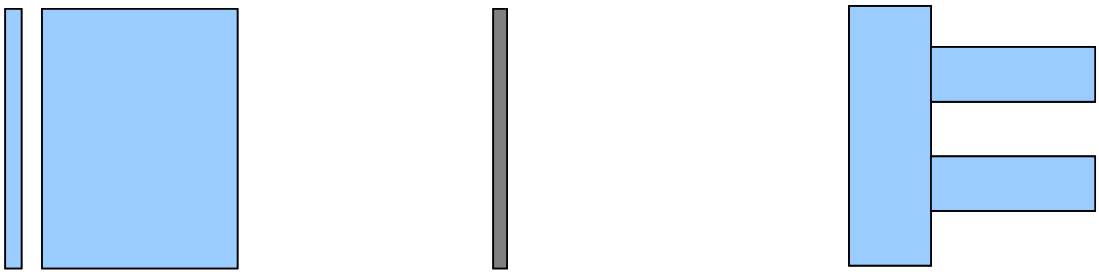


Figure 1. Conformal Stacking of cuboid *B* over *A*

5.1. The Main Algorithm

In the main program, the CIF file is parsed and all rectangles that are used to describe the layouts of various masks are read and initialized into separate lists. Then the effective layout for each layer is calculated by subtracting the layouts described by the dark mask layouts from the light mask layouts. Then these layers are stacked conformably after extrusion. Figure 1 illustrates the meaning of conformal stacking graphically. Cuboid *B* is stacked conformably on cuboid *A*. Cuboid *B* is split into two cuboids *B'* and *B"* to facilitate conformal stacking. An intuitive analogy for conformal stacking would be a snow pile on uneven terrain.

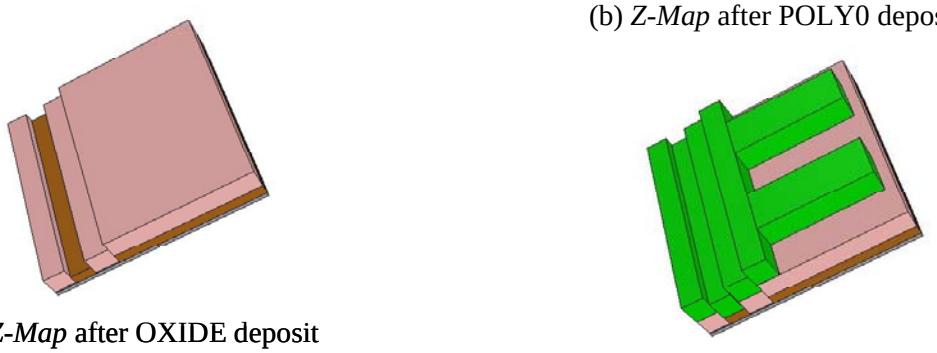


(a) POLY0 (b) ANCHOR1 (c) POLY1

Figure 2. Example Mask Layouts.

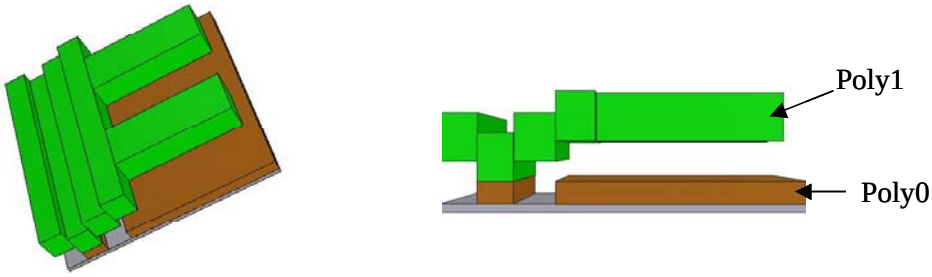
(a) *Z-Map* after NITRIDE deposit

(b) *Z-Map* after POLY0 deposit



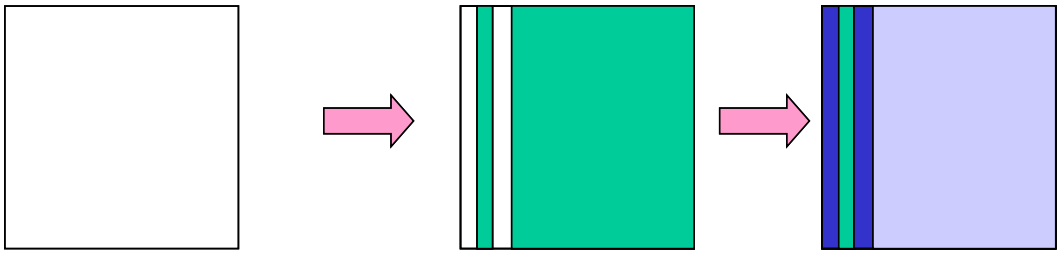
(c) *Z-Map* after OXIDE deposit and ANCHOR1 etch

(d) *Z-Map* after POLY1 deposit



(e) *Z-Map* after OXIDE release.The side view of the model described by *Z-Map* clearly shows the floating poly1 layer

Figure 3. Device Model as described by the cuboids in list *Z-Map*



(a) *Top- Surface* after NITRIDE deposit

(b) *Top- Surface* after POLY0 deposit. (Rectangles at different heights in different colors)

(c) *Top- Surface* after OXIDE deposit and ANCHOR1 etch.

Figure 4. 2D region described by the rectangles in list *Top-Surface*.

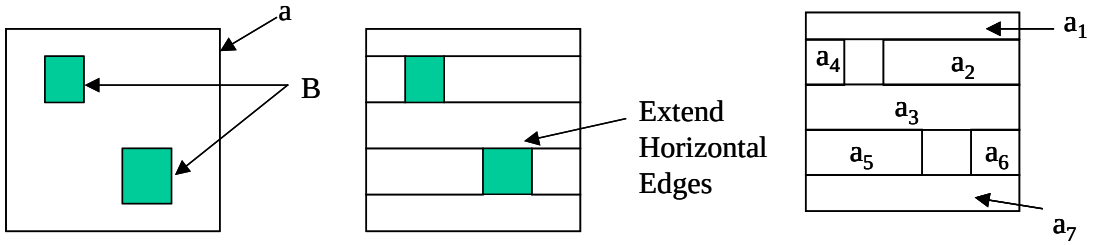
Top-Surface is a list of doubles, where each double is an ordered set with the first element being a rectangle and the second element being the z-coordinate of the plane containing the rectangle. *Z-Map* is a list of doubles, where each double is an ordered set with the first element being a cuboid and the second element indicates the type of the cuboid. Our emulation algorithm consists of the following steps:

1. Lists *Top-Surface*, *Z-Map,* and $L_{NITRIDE}$ are initialized as empty lists. Input CIF file is read and the lists L_{POLY0}, $L_{ANCHOR1}$, L_{DIMPLE}, L_{POLY1} and L_{HOLE1} are initialized. The rectangles described in the CIF file are copied into various lists depending upon the mask layout to which the rectangles belong. A rectangle *R* is initialized and the parameters are set such that *R* overlaps every rectangle described in the CIF file. *R* is inserted into the list $L_{NITRIDE}$. Figure 1 illustrates the three mask layouts Poly0, Anchor1 and Poly1.
2. The effective OXIDE1 2D region is calculated by calling the subtraction algorithm with $L_{NITRIDE}$ and $L_{ANCHOR1}$ as input arguments. The output is stored in L_{OXIDE1}.
3. The effective POLY1 2D region is calculated by calling the subtraction algorithm with L_{POLY1} and L_{HOLE1} as arguments. The output is stored in L_{POHO1}.
4. The rectangle in $L_{NITRIDE}$ is extruded to create a cuboid such that the Z_{max} of the cuboid is set as 0 and Z_{min} is set such that the difference between the two is equal to the height of the nitride layer. The cuboid is inserted into the list *Z-Map*. The associated type is set as "NITRIDE". Figure 3 (a) graphically illustrates the model described by the cuboid in *Z-Map* at the end of this step.
5. The rectangles in the list L_{POLY0} are extruded such that the height of the cuboids generated equal the height of the Poly0 layer. The parameter Z_{min} of the cuboids is set to 0. The cuboids generated are inserted into list *Z-Map*. The associated type is set as "Poly0". Figure 3 (b) graphically illustrates the model described by the cuboids in list *Z-Map* at the end of this step.
6. The subtraction algorithm is called with input arguments $L_{NITRIDE}$ and L_{POLY0}. The rectangles from the output list are associated with a z-coordinate of 0 and inserted into the list *Top-Surface.*
7. The rectangles from L_{POLY0} are appended to the list *Top-Surface* and the associated z-coordinate is set equal to the height of the Poly0 deposit. Figure 4 (a) graphically illustrates an example problem where the layout described by the list *Top-Surface* at this step is shown.
8. The Conformal-Stacking algorithm is called with the lists L_{OXIDE1}, *Z-Map*, *Top-Surface*, height equal to the Oxide1 deposit height and layer type "OXIDE1" as input arguments. Figure 3 (c) graphically illustrates the contents of list *Z-Map* at the end of this step.

9. The Place-Dimples algorithm is called with the lists L_{DIMPLE}, *Z-Map* and height equal to the depth of the dimple etch as input arguments.
10. Elements of *Top-Surface* are used to create a list of rectangles. The subtraction algorithm is called with this list and $L_{ANCHOR1}$ as input arguments. The rectangles in the output list are associated with a z-coordinate that is greater then the z-coordinate of the rectangles in *Top-surface* that contained them by the height of the oxide deposit. The output is copied to the list *Updated-Top-Surface*.
11. The intersection algorithm is called with lists *Top-Surface* and $L_{ANCHOR1}$ as input arguments. The output is appended to the *Updated-Top-Surface*. The rectangles in the output list are associated with a z-coordinate same as the z-coordinate of the rectangles in *Top-surface* that contained them. The elements of the list *Top-Surface* are deleted and the elements of the list *Updated-Top-Surface* are inserted into *Top-Surface*. Figure 4 (c) graphically illustrates the content of the list *Top-Surface* at the end of this step.
12. The Conformal-Map algorithm is called with the lists L_{POHO1}, *Top-Surface*, *Z-Map*, height equal to the height of the Poly1 deposit and layer type "POLY1" as input arguments. Figure 2 (c) is a graphical representation of an example POLY1 mask layout. Figure 3 (d) graphically represents the content of the list *Z-Map* at the end of this step.
13. The cuboids in list *Z-Map*, which are associated with the type "OXIDE1", are deleted. Figure 3 (e) graphically represents the content of the list *Z-Map*. The side view of the model clearly shows the floating polysilicon layer at the end of the release step.

5.2. The Subtraction Algorithm

Here we discuss an algorithm whose input arguments are two lists *A* and *B*. The content of these lists are non-intersecting Manhattan rectangles. The output is a list of non-intersecting Manhattan rectangles, which describes the equivalent 2D region when the 2D region described by list *B* is subtracted from the 2D region described by list *A*. Figure 5 illustrates the subtraction of rectangles of list *B* from rectangle *a*.



(a) For every rectangle a calculate a ∩ B

(b) Extend Horizontal Edges until they intersect the boundary or any vertical edge

(c) Split rectangle *a* using the horizontal edges into rectangles $a_1 - a_7$

Figure 5. Subtraction of list *B* from rectangle *a*

Subtract (List *A*, List *B*)
1. Initialize a list of rectangles *output* as an empty list of rectangles.
2. For each element a_i of list *A* do:
    a. Build a set of rectangles A_i of list *B* that overlap with a_i.
3. For each element a_i of list *A*, the following steps are carried out

a. Initialize a list of rectangles *H* as an empty list.
b. Initialize two lists of line segments *Vertical* and *Horizontal* as empty lists.
c. For each element *b* of set A_i the following steps are carried out.
    i. If a_i *b* is equal to a_i then a_i need not be added to the output, therefore continue with the next element in Step 3.
    ii. Find the line segments belonging to the boundary of *b* that are inside a_i. Add them to *Vertical* or *Horizontal* depending upon the orientation of the line segments. Create a new rectangle using the above line segments and append the rectangle to list *H*.
d. If lists *Vertical* and *Horizontal* are empty, add a_i to *output* and continue with the next element in Step 3.
e. Sort the lists *Vertical* and *Horizontal* using the line segment element's second parameter as primary key and third parameter as secondary key.
f. Build a Quadtree *Vertical-Tree* using the list *Vertical*.
g. For each element *h* of list *Horizontal* the following steps are carried out
    i. If left-end of line segment *h* is not on b(a_i), then it is extended until it intersects with a line segment from Quadtree *Vertical-tree* or b(a_i). The new edge is added to the list *Horizontal* immediately before element *h*.
    ii. If right-end of line segment *h* is not on b(a_i) then it is extended until it intersects with a line segment from list *Vertical* or b(a_i). The new edge is added to the list *Horizontal* immediately after element *h*.
h. Rectangle a_i is split using the edges in *Horizontal* and the new rectangles that are not present in *H* are added to *output.*

4. The list *output* is returned.

5.3. The Intersection Algorithm

This section describes an intersection algorithm. Input arguments for this algorithm are two lists of rectangles *A* and *B*. The content of these lists are non-intersecting Manhattan rectangles and the algorithm's output is a list of non-intersecting Manhattan rectangles that describes a layout which is the equivalent to the intersection of the 2D region described by list *B* and the 2D region described by list *A*. This algorithm computes the regular intersection. The algorithm is described below:

Intersect (List *A*, List *B*)
1. Initialize a list of rectangles *output* as an empty list of rectangles.
2. For each element a_i of list *A* do:
    a. Build a set of rectangles A_i from list *B* that overlap with rectangle a_i.
3. For each element a_i of list *A* the following steps are carried out
    a. For each element *b* of set A_i the following steps are carried out.
        i. Compute the intersection of i(a_i) & i(*b*). Add the result to *output*.
4. List *output* is returned.

5.4. The Conformal Stacking Algorithm

The inputs to the Conformal Stacking algorithm are the lists *Z-Map*, *Top-Surface*, *A*, Height *Z* and Layer Type *L*. List *A* is a list of non-intersecting Manhattan rectangles. *Z-Map* is a list of cuboids each associated with a layer type. *Top-Surface* is a list of rectangles each associated with a height. This algorithm updates the state of the 3D model by adding cuboids to *Z-Map*. The 2D region described by *A* is extruded and conformably stacked on the current 3D model. The current device structure is described by the list of cuboids in *Z-Map*. The newly created cuboids, which correspond to the new layer being stacked, are associated with the layer type *L* and appended to *Z-Map*. In this algorithm we create a new data structure *H* to store the neighborhood information for every rectangle *r* in *Top-Surface*. *H* is a list of quintuplets whose first element is rectangle *r*. Rectangles in *Top-Surface* that are neighboring rectangles to *r* are stored in the next four elements of *H*, organized in the following manner. The second element of *H* is a set of rectangles that are top neighbors of *r*. The third element of *H* is a set of rectangles that are the bottom neighbors of *r*. The fourth element of *H* is a set of rectangles that are the right neighbors of *r*. The fifth element of *H* is a set of rectangles that are the left neighbors of *r*. The conformal stacking algorithm is described below:

Conformal-Stack (List *Z-Map*, List *A*, List *Top-Surface*, Height *Z*, Layer Type *L*)

1. Initialize *H*.
2. Map the neighborhood of every rectangle in the 2D region described by the set of rectangles in list *Top-Surface* and store the map in *H*. This is done as described in Section 4.3.
3. For each rectangle *a* of list *A* the following steps are carried out:
   a. Search the rectangles in the list *Top-Surface* for a rectangle which overlaps with rectangle *a*.
   b. Split rectangle *a* recursively, such that split rectangles overlap at the maximum of one rectangle from the list *Top-Surface*. The split rectangles are extruded and the cuboids placed such that the bottom surface (Z_{min}) of the cuboid is at the same height as the rectangle it overlaps in list *Top-Surface*.
   c. The parameter Z_{max} of the cuboids formed in the above step is set equal to the sum of the parameter Z_{min} of the cuboid and height *Z*. The cuboids are associated with the layer type *L* and added to the end of the list *Z-Map*.
4. Return *Z-Map*.

5.5. The Dimple Placing Algorithm

Input arguments to this algorithm are the lists *Z-Map* and *Dimples*. *Z-Map* is a list of cuboids each associated with a layer type. *Dimples* is a list of non-intersecting Manhattan rectangles. The algorithm models the dimples and adds them to the list *Z-Map*. Steps in this algorithm are described below:

Place-Dimples (List *Z-Map*, List *Dimples*, Height *Z*)

1. For each element *a* in list *Dimples* do:
   a) For each element *z* associated with the layer type “OXIDE1” in *Z-Map* do:

      If the X-Y projection of *z* overlaps with rectangle *a*, extrude *a* and set the Z_{max} of the cuboid equal to the parameter Z_{max} of cuboid *z*. Set Z_{min} such that the difference between Z_{max} and Z_{min} is equal to height *Z*. Add the cuboid to list *Z-Map* and associate it with layer type “DIMPLE”.

6. Analysis of the Algorithm

In this section we discuss the worst-case complexity of the emulation algorithm and other algorithms described in this report.

6.1. Analysis of the Subtraction Algorithm

In this section we discuss the complexity of the Subtraction Algorithm. We will designate the size of the two lists, which serve as input to the algorithm as *N*, *M*.

1. Step 2 has a time complexity of O(*N* log *M* + *I*) where *I* is the number of intersections between the rectangles of list A and B [Ullm84, McCr80].
2. Step 3-a is an initialization step and has a constant time complexity.
3. Step 3-b is an initialization step and has a constant time complexity.
4. Step 3-c has a time complexity, which is equal to the size of set A_i. Let us assume this to be C_i for the i^{th} element in list *A*.
5. Step 3-d has a constant time complexity.
6. Step 3-e has a time complexity of $C_i \log C_i$ [Corm89]. The lists *Vertical* and *Horizontal* have a size proportional to the size of the A_i and hence have a size proportional to C_i.
7. Step 3-f has a time complexity of $C_i \log C_i$ [Corm89]. The size of list *Vertical* is proportional to C_i as stated above.
8. Step 3-g has a time complexity of *Ci* log *Ci*. This step is repeated for each element of list *Horizontal*, which has a size proportional to *Ci* and since each of the sub-steps have a time complexity of log C_i. This is the complexity of a search using the quadtree to find the nearest edge in the direction of extension. Hence this step has a total time complexity of $C_i \log C_i$ [Corm89].
9. Step 3-h splits the rectangle *a* using the edges in list *Horizontal*. The number of rectangles produced in this step is proportional to the number of edges in list *Horizontal* and hence the number of rectangles generated in this step is proportional to C_i. Since list *H* has to be searched for each of these rectangles generated and the complexity of a search is log C_i , the overall complexity of this step is *Ci* log C_i.

Steps 2-9 are repeated for each element in list *A*. Hence the total complexity for Step 3 of this algorithms is $\sum_{i=1}^{N} \left(C_i + 4C_i \log C_i \right)$

C_i corresponds to the number of rectangles produced when splitting the rectangles in *A* such that the resultant layout represents the subtracted layout. This is proportional to the number of intersections between the rectangles in *A* and *B*, which is designated as *I*. There are two possibilities:

1. The case where each rectangle in *B* intersects with a maximum of 4 rectangles in *A*; like when *B* represents holes that need to be subtracted from the Poly1 layout represented by *A*, *I* is proportional to *M*. In this case the overall complexity of this algorithm is O(*N* log *M* + *I* log *M*), which is equivalent to O((*N*+*M*) log *M*).
2. In a general case when the above stated assumption is not true C_i has a maximum value of *M*, which is based on the assumption that a rectangle from *B* intersects with every rectangle in *A*.

The number of rectangles produced in this case is proportional to *NM* and the overall time complexity is O(*NM* log *M*)

6.2. Analysis of the Intersection Algorithm

In this section we analyze the complexity of the algorithm "Intersect". We will designate the size of the two lists, which serve as input to the subroutine as *N* and *M*.

1. Step 2 has a time complexity of O(*N* log *M* + *I*) where *I* is the number of intersections between the rectangles of list A and B [Ullm84, McCr80]
2. Step 3 is carried out for each element of list *A*. The Step a has a time complexity which is equal to the size of the set A_i. Let us assume this to be C_i for the i^{th} element in list *A*. Hence Step 3-a has a time complexity proportional to C_i since Step i has a constant time complexity. Hence the overall complexity for Step 3 is given by: $\sum_{i=1}^{N} C_i$ which is proportional *I*, the total number of intersections. Therefore the complexity for this step is O(*I*).

Hence total complexity for this algorithm is O(*N* log *M* + *I*) where *I* represents the total number of intersections. Hence as described in the previous section, for the general case where every rectangle of list B intersects with every rectangle of *A*, the overall worst case time complexity is O(*N* log *M* + *NM*).

6.3. Analysis of the Conformal-Stacking Algorithm

In this section we discuss the complexity of the algorithm "Conformal-Stack". We will designate the size of the two lists *Top-Surface* and *A*, which serve as inputs to the algorithm as *M* and *N*.

1. Step 2 has a time complexity of *M* log *M*. This involves mapping the neighborhood of every rectangle as in list *Top-Surface* discussed in Section 4.3.
2. Step 3-a has a time complexity of *M* since it involves a search in a list of size *M*.
3. Step 3-b has a time complexity of C_i where C_i is the number of rectangles produced by splitting.
4. Step 3-c has a time complexity of C_i.

Steps 2-4 are repeated for every element of A hence the complexity of this algorithm is $\sum_{i=1}^{N} (M + C_i)$. C_i corresponds to the total number of rectangles produced during splitting. Since in a general case each rectangle in *B* can intersect with each rectangle in *A*, the total number of rectangles produced is proportional to *NM*. Hence the overall time complexity for the subroutine is O(*NM*).

6.4. Analysis of the Dimple Placing Algorithm

In this section we discuss the complexity of the algorithm "Place-Dimples". We designate the size of the two lists *Z-Map* and *A*, which serve as input to this algorithm as *M* and *N*. Here since

only the Oxide elements of the list *Z-Map* are considered, *M* represents the number of Oxide elements in the list.

1. Step a is carried out for every Oxide element of list *Z-Map* and hence has a complexity of O(*M*).

This is repeated for every element *A* and hence the routine has an overall complexity of O(*NM*)

6.5. Analysis of the Main Algorithm

In this section we discuss the overall complexity of the algorithm. Here we analyze the main algorithm from where the other algorithms are called. We denote the number of rectangles used to describe the layouts of masks POLY0, ANCHOR1, DIMPLE, POLY1 and HOLE1 as N_1, N_2, N_3, N_4 and N_5 respectively.

1. Step 1 has a time complexity of $\sum_{i=1}^{5} N_i$ as the operation involves the parsing of the CIF file to read in the mask data.
2. Step 2 has a time complexity of O($N_3 \log N_3$). As the inputs to the subtract algorithm are such that all of the rectangles in *B* are wholly contained in the rectangle in *A*. The number of rectangles output is O(N_3).
3. Step 3 has a time complexity of O((N_4+N_5) log N_5). As the inputs to the subtract algorithm are such that all of the rectangles in *B* are wholly contained in the rectangles in *A*. The number of rectangles output is O(N_4+N_5).
4. Step 4 has a constant number of operations associated with it as only a constant number of operations are carried out.
5. Step 5 has a complexity of O(N_1) as the number of operation associated with this step is proportional to the size of the list POLY0.
6. Step 6 has a complexity of O($N_1 \log N_1$). As the inputs to the subtract algorithm are such that all of the rectangles in list *B* are wholly contained in the rectangle in *A*. The number of rectangles output is the order of N_1.
7. Step 7 has a time complexity of O(N_1). The number of operations associated with this step is proportional to the size of the list POLY0.The number of rectangles added to *Top-Surface* is equal to the size of list POLY0, which is N_1.
8. Step 8 has a time complexity of O(N_1N_2). As the Conformable-Stack algorithm is called with inputs, which are lists of sizes N_1 and N_2.
9. Step 9 has a time complexity of O($N_1N_2N_3$). As the dimple placing algorithm is called with inputs that are lists of sizes N_1N_2 and N_3.
10. Step 10 has a time complexity of O($N_1N_2\log N_2$). The subtraction algorithm is called with lists of sizes N_1 and N_2.
11. Step 11 has a time complexity of O($N_1N_2\log N_2$). The intersection algorithm is called with lists of sizes N_1 and N_2.
12. Step 12 has a time complexity of O(N_1N_2(N_4+N_5)). Conformal-Stack algorithm is called with input lists whose sizes are the order of N_1N_2 and N_4+N_5.
13. Step 13 has a time complexity of O(N_1N_2). The number of operations required to delete the cuboids of the oxide layer is proportional to the number of Oxide cuboids, which has an order of N_1N_2.

Hence the overall time complexity for this algorithm is O(N_1N_2(logN_2+N_3+N_4+N_5)).

7. Implementation

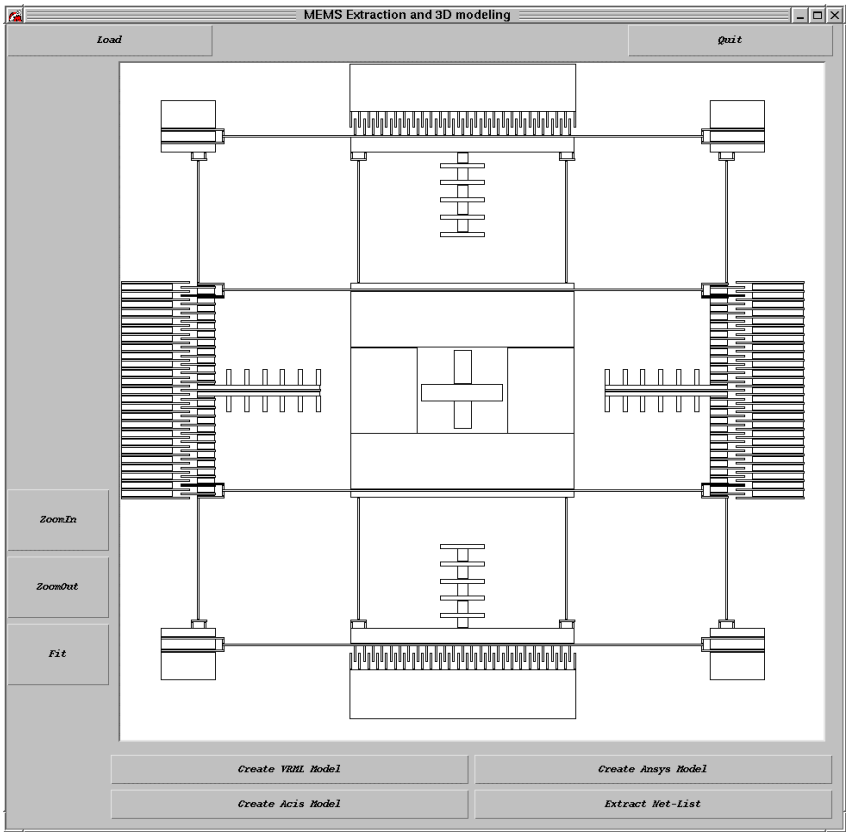


Figure 6. Screen Dump of the system GUI

The system has four modules: the system GUI, CIF file parser, 3D model generator and file exporter.

The system GUI is written in JAVA and uses its AWT package for rendering the 2D layouts. A CIF file is chosen from a file select menu. The POLY1 layout is displayed and the layout can be visually inspected and the GUI can be used to Zoom and Pan the layout image. The 3D model of the MEMS device is generated and written to a file in the format as desired by the user. Figure 6 shows the screen dump of the system GUI.

The parser reads the CIF file. We assume that the CIF file is generated for the MUMPS™ process. There are two parts to reading a data file: processing the characters and building the results into a data structure. The former is commonly known as syntax analysis and the latter semantic analysis [Trim87]. The syntax of the CIF file for the MUMPS™ process is pre-determined and hard coded in the implementation. Rectangles describing each mask layout are stored in different lists. The parser is implemented in C.

The modeling engine consists of the algorithms described in this report. The algorithms were implemented in C++. The 3D device model is represented by a list of cuboids.

The list of cuboids, which represents the 3D device model, can be written to a file in three formats. It can be written to a file, which can be directly read into the ANSYS finite-element analysis environment for subsequent meshing and analysis. It can also be written as VRML file.

A VRML file consists of a list of faces that represent the 3D model. Each face is represented by a set of points. Each point has a 3-dimesional co-ordinate value. Standard headers are added to the file from a text file stored on the secondary memory device. A VRML model can be viewed using standard Internet browsers with Cosmo Player. It can also generate an ACIS file containing a single body, which represents the 3D device model. The ACIS file is generated using the ACIS 6.0 geometric kernel. The ACIS model is compatible with leading CAD/CAE systems and can be used as starting input for further analysis in these CAD/CAE systems.

8. Results

Section 8.1 shows an example MEMS device model in different environments. Section 8.2 has a number of example MEMS device models constructed using the algorithms described in this report.

8.1. Environments for further processing of generated MEMS device models

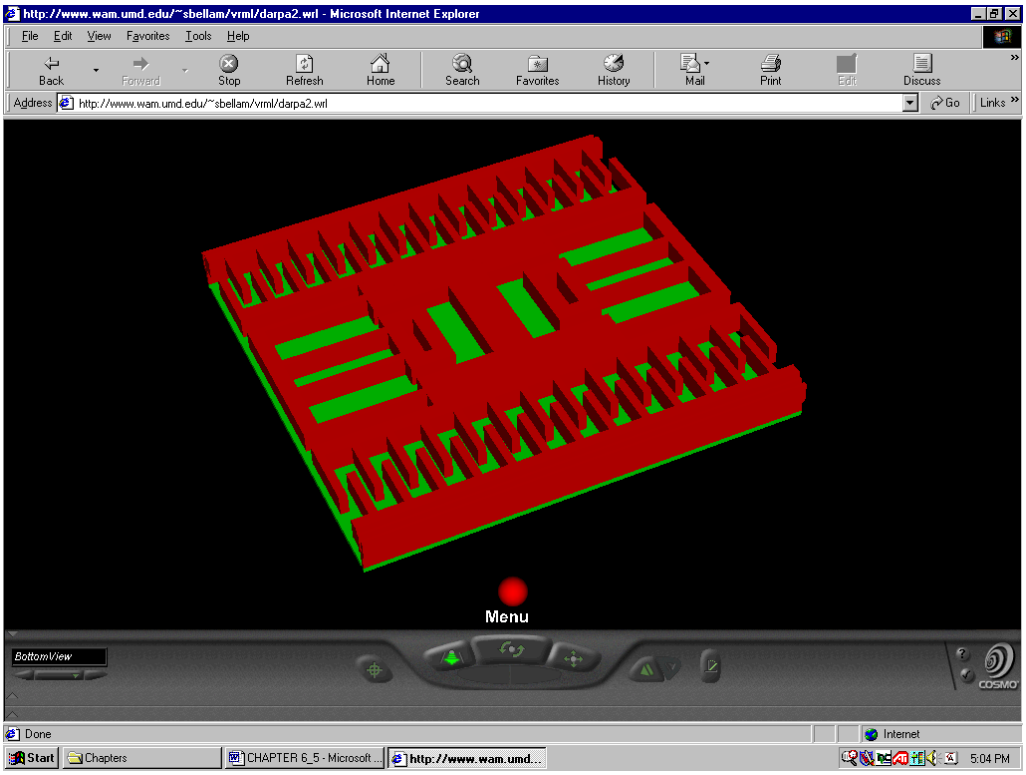


Figure 7 MEMS model visualized using VRML.

Figure 7 shows a MEMS device model saved in the VRML format, which can be visualized using a standard Internet browsers installed with Cosmo™ Player. Figure 8 is a screen shot of a generated MEMS device model imported into a CAD environment. The device model was saved in the ACIS file format. Figure 9 is a screen shot of generated MEMS device model in the ANSYS FEA software environment. The device model was written in file format that can be directly imported into the ANSYS environment.

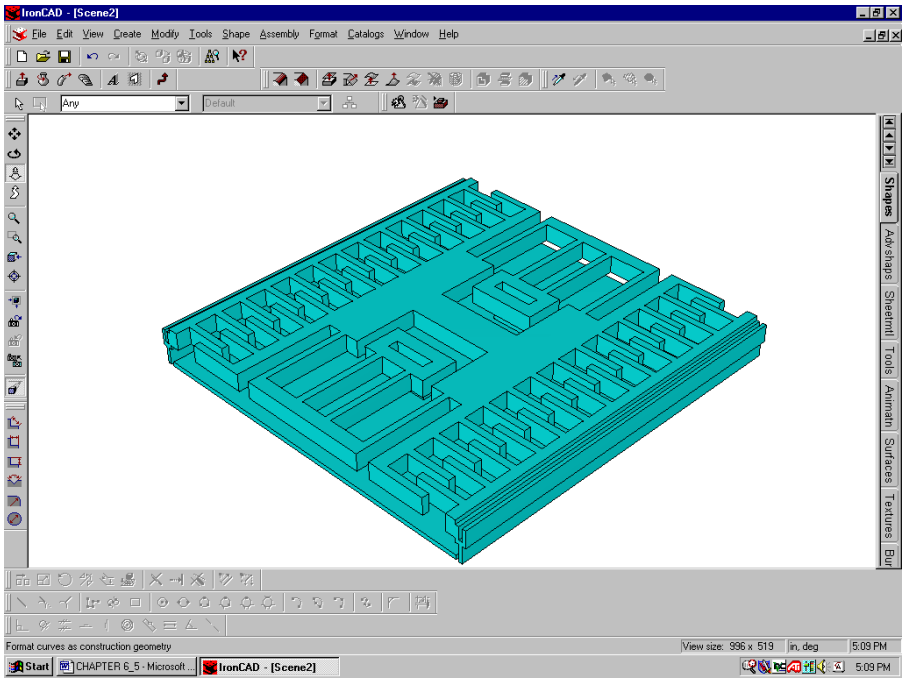


Figure 8. Generated MEMS device model in a CAD environment.

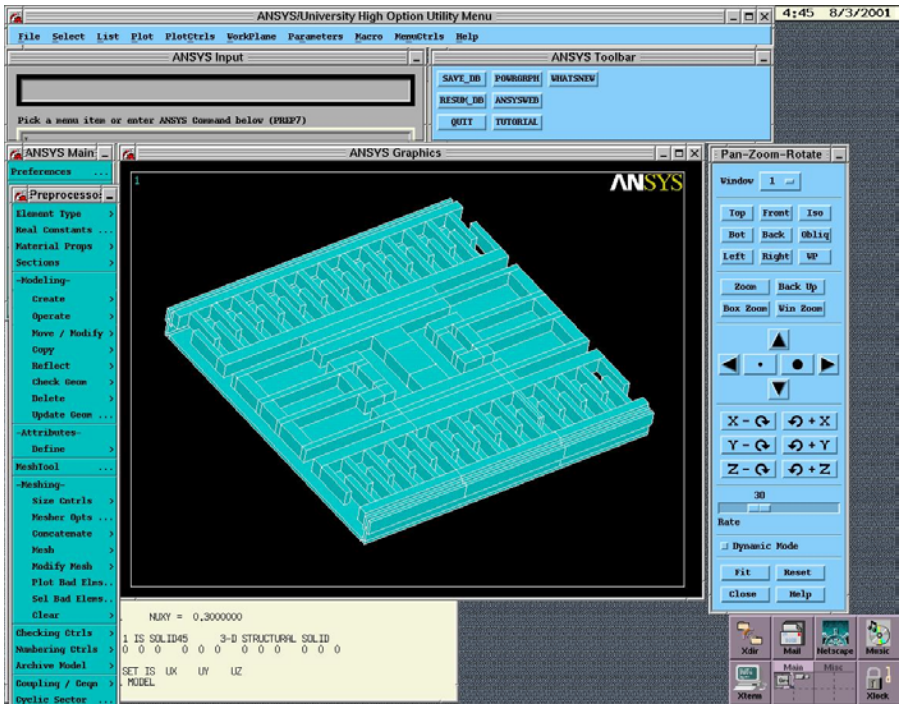


Figure 9. Generated MEMES device model in the ANSYS FEA software environment.

8.2. Examples

This section shows some example MEMS 3D models generated using algorithms described in this report. Figure 10 is the B-Rep model of a MEMS resonator. Figure 11 is a model of a MEMS resonator. Figure 12 is a model of a MEMS pressure sensor. Figure 13 is a model of a

MEMS Z-accelerometer. Figure 14 is a model of a MEMS gyroscope. Figures 15 and 16 are models of MEMS gyroscopes. Figure 17 shows a model of a MEMS accelerometer.

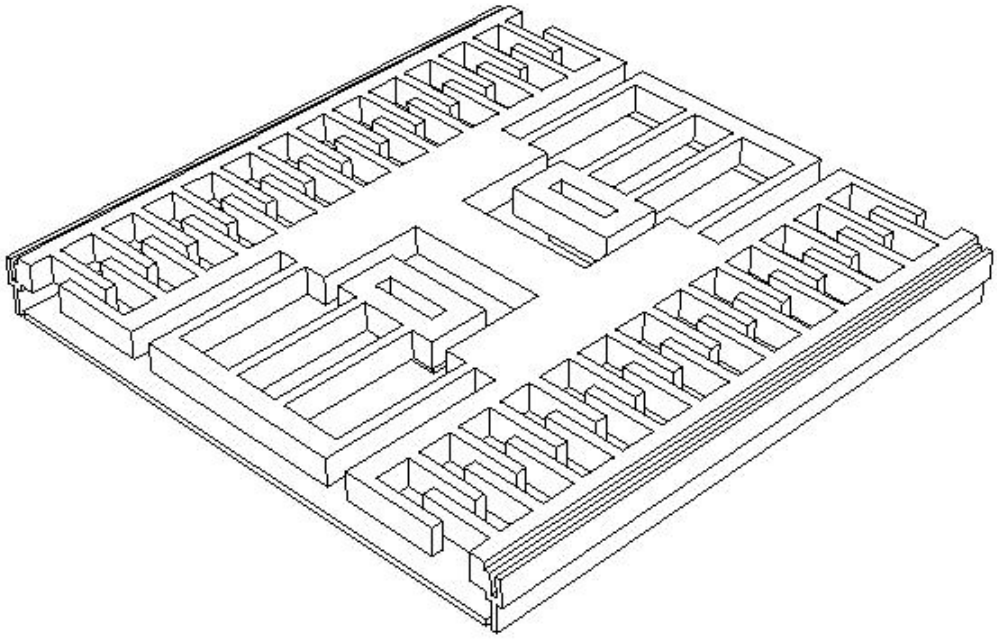


Figure 10. A MEMS resonator model

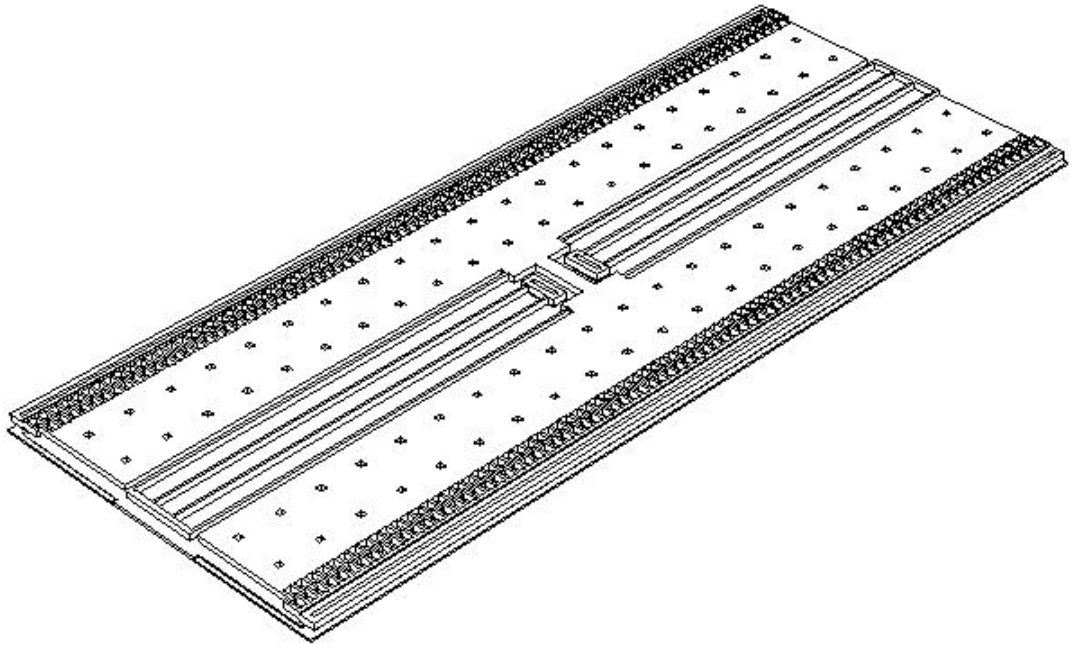


Figure 11. A MEMS resonator

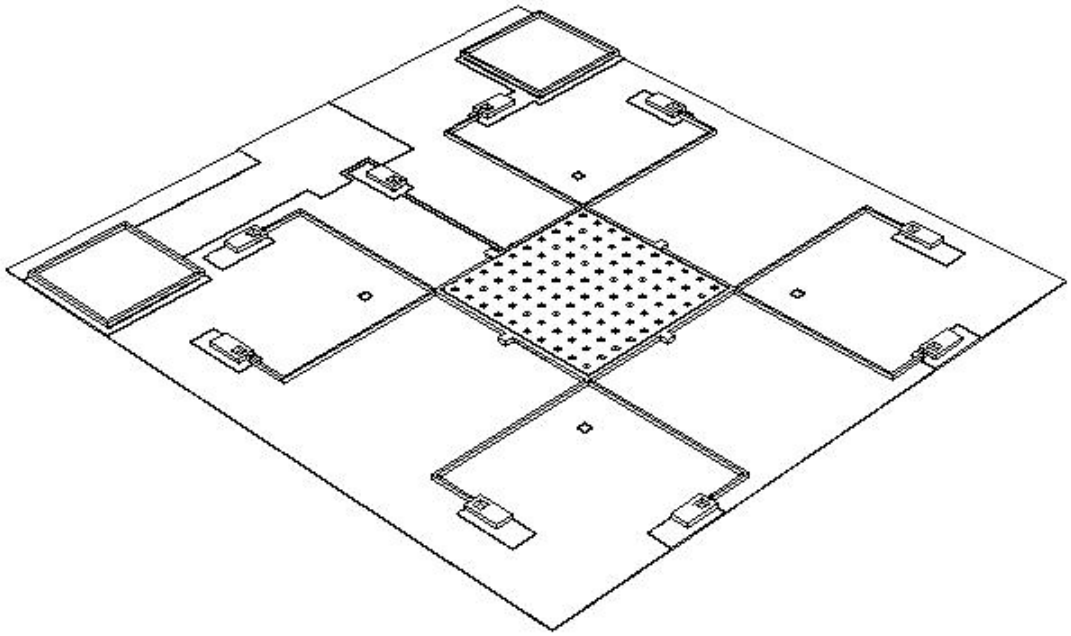


Figure 12. A MEMS pressure sensor

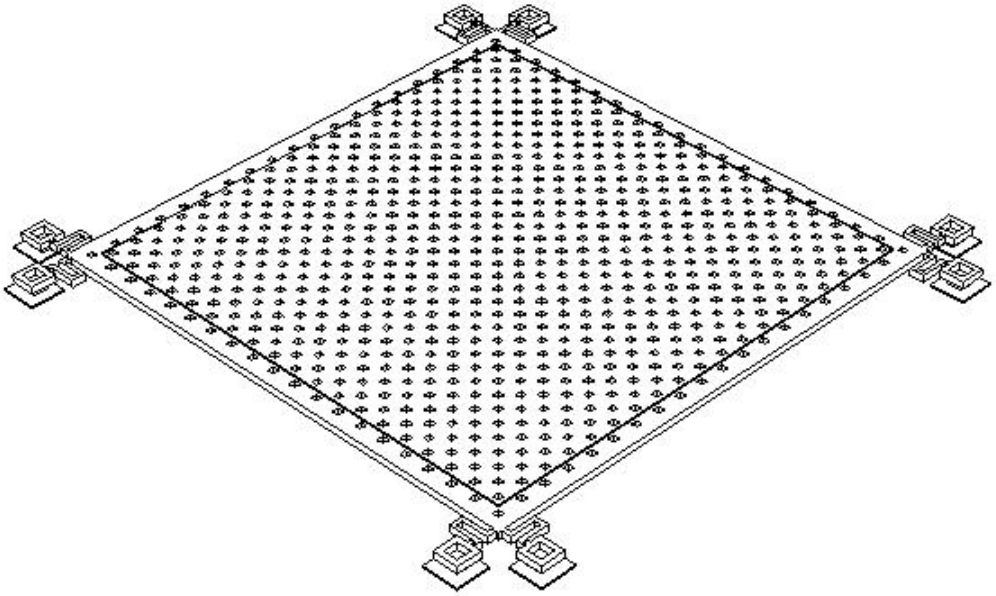


Figure 13. A MEMS Z-accelerometer

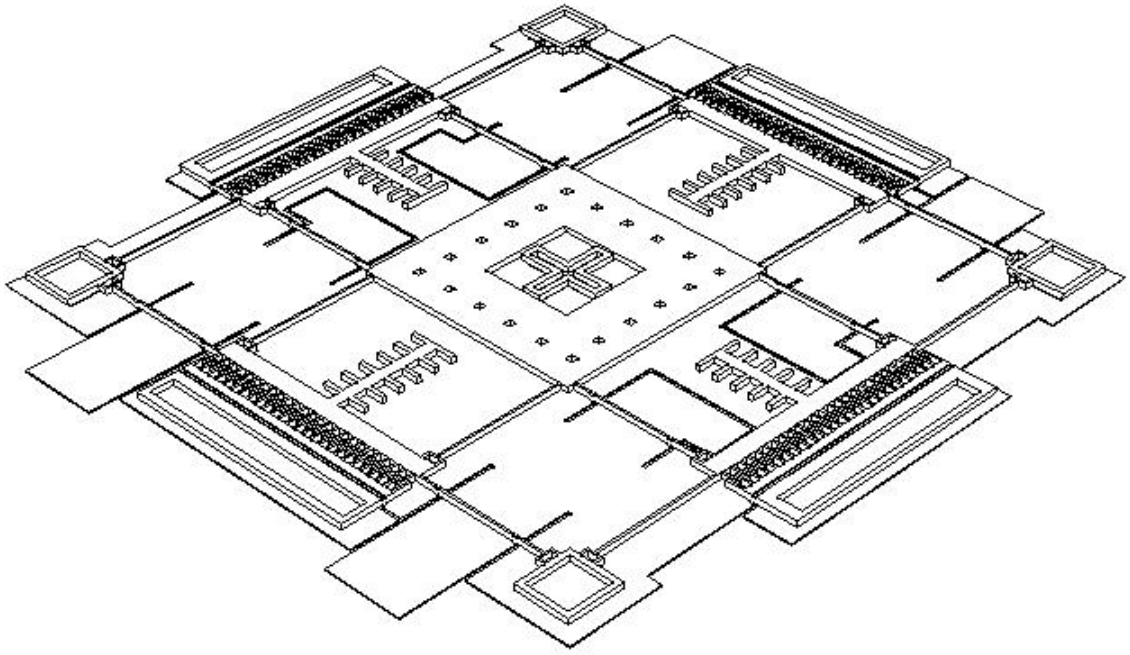


Figure 14. A MEMS gyroscope

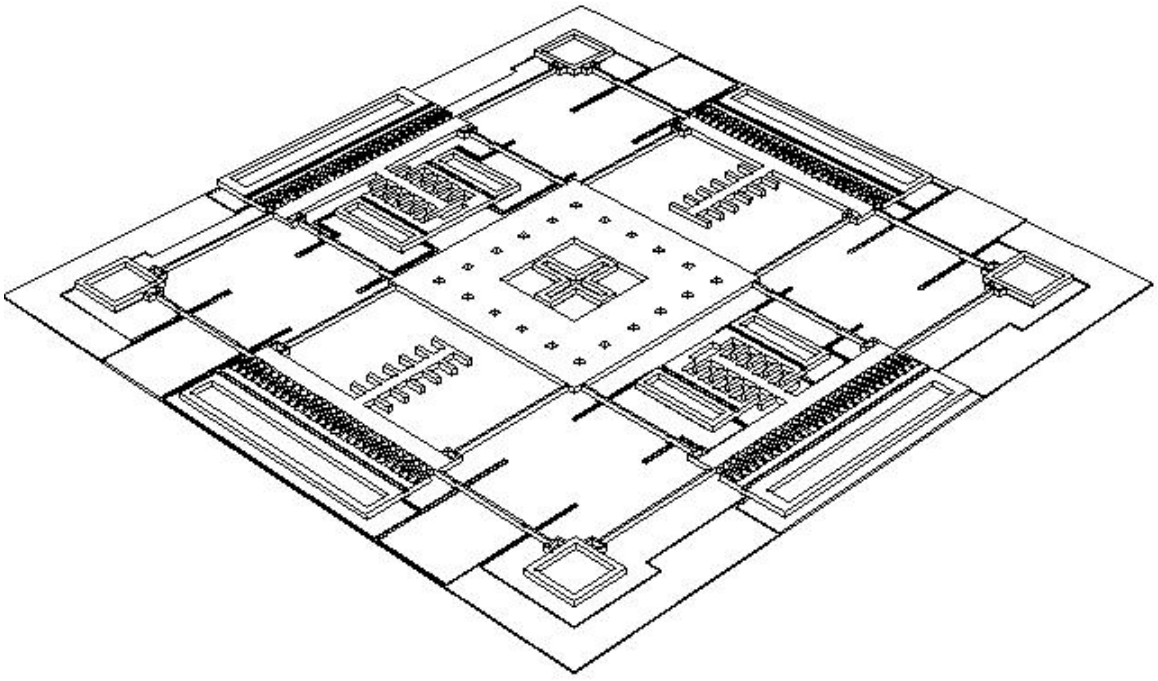


Figure 15. A MEMS gyroscope

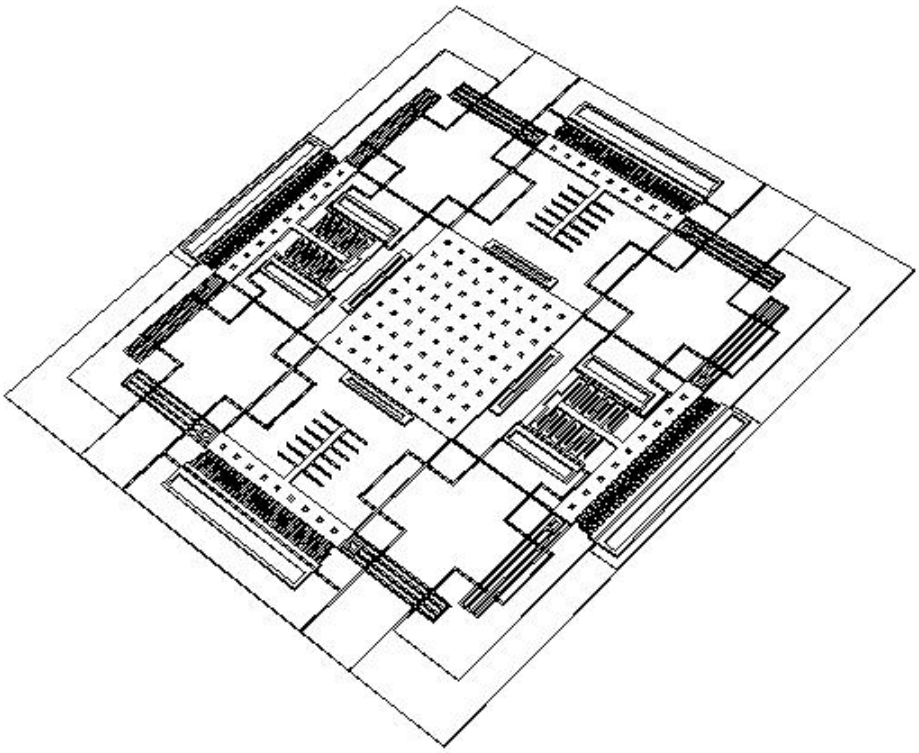


Figure 16. A MEMS gyroscope

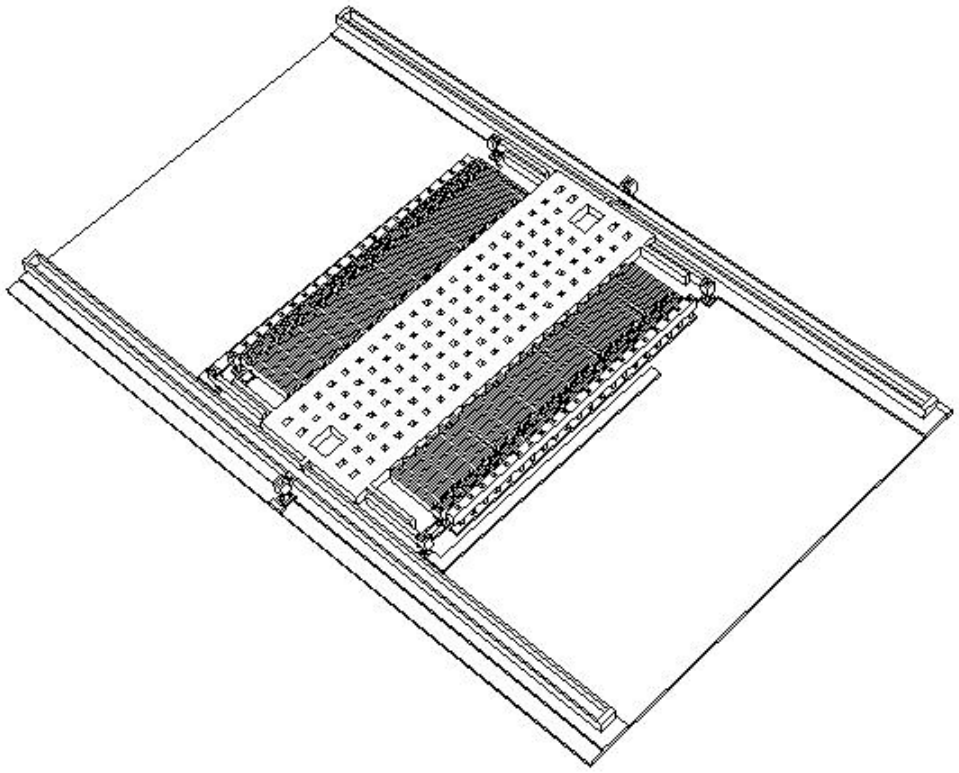


Figure 17. A MEMS Accelerometer

Acknowledgement. This research has been supported by a subcontract on NSF grant CCR9901171. Opinions expressed in this paper are those of authors and do not necessarily reflect opinion of the sponsors.

9. References

[Bryz94] J. Bryzek, K. Petersen and W. McCulley, "Micromachines on the march," *IEEE Spectrum*, May 1994, pp. 20-31.

[Corm90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, Introduction to Algorithms. New York, McGraw-Hill, 1990.

[Devo98] D.L. Devoe, S.B. Green, J.M. Jump, "Automated Solid Model Extraction for MEMS Visualization", Proc. Int. Conf. On Modeling and Simulation of Microsystems, Sensors and Actuators, pp. 292-297, 1998.

[Emme98]Markus Emmenegger, Krovink J. G. , Baltes Henry, "MemCel An Inexpensive and Efficient Tool for 3D MEMS Prototyping." *Micro-Electro-Mechanical Systems(MEMS) ASME 1998* DSC-Vol. 66, p. 559-563

[Howe90] R.T. Howe, *et. al.*, "Silicon Micromechanics," *IEEE Spectrum*, July 1990, pp. 29-35.

[Koes94] D.A. Koester, R. Mahadevan, Busbee Hardy and K.W. Markus, *Multi-User MEMS Processes (MUMPs) Design Handbook*, available from Cronos Integrated Microsystems, 3026 Cornwallis Road, Research Triangle Park, NC 27709, rev. 6, 2001, 39 pages. http://www.memsrus.com/cronos/mumps.pdf

[Lakd99] H. Lakdawala, B. Baidya, T. Mukherjee and G.K. Fedder, "Intelligent Automatic Meshing of Multi-layer CMOS Micromachined Structures for Finite Element Analysis," *Proc. of MSM '99,* San Juan, Puerto Rico, pp. 297-300, April 19-21, 1999.

[McCo84] S.P. McCormick, "EXCL: A Circuit Extractor for Integrated Circuit Designs", Proceedings of the 21st DAC, June 1984, pp. 616-23.

[Oste95] P. M. Osterberg and S. D. Senturia, ""MEM-BUILDER":An Automated 3D Solid Model Construction Program for Microelectromechan-ical Structures," *Technical Digest of the 8th Int. Conf. on Solid-State Sensors and Actuators (Transducers '95*), Stockholm Sweden, v.2, pp. 21-24, June 1995.

[Tang90] W.C. Tang, T.-C.H. Nguyen, M.W. Judy and R. T. Howe, "Electrostatic-comb Drive of Lateral Poly-silicon Resonators," *Transducers '8*9, Vol. 2, pp. 328-331, June 1990.

[Tang97] W.C. Tang, "Overview of Microelectromechanical Systems and Design Processes," *34th DAC Pro-ceedings*,1997, pp. 670-3.

[Trim87] S.M. Trimberger, An Introduction to CAD for VLSI, Kluwer Academic Publishers, 1987.

[Ullm87] J.D. Ullman, Computational Aspects of VLSI, Computer Science Press, 1987.

[Wagn85] T.J. Wagner, "Hierarchical Layout Verification," *IEEE Design and Test,* February 1985, pp. 31-37.