Chapter 6

Implementation and Application

6.1 Introduction

The life cycle of a software application consists of multiple phases, analysis, design, programming, debugging and testing, maintenance and enhancement. One challenging task in software engineering is to develop a methodology which provides an optimal solution for the successful realization of software applications in all phases. The task becomes very difficult when large complex software applications are to be developed.

In the past thirty years, software development has experienced a great change in terms of the complexity and the implementation effort. To cope with the change, a new software methodology based on the object-oriented technique has emerged. The object-oriented approach attempts to address several critical issues of the software development, including the complexity decomposition, the code reusability and the software migration. The methodology has been successfully applied in various fields, such as database, expert systems, CASE tools and graphic user interface.

However, less effort has been made to adopt the object-oriented technique for solving scientific computational problems. One reason is that most scientific computational applications are algorithm driven, i.e. the software architecture is procedural based. In other words, procedures are the center of a program and the data are manipulated by procedures.

Contrary to the procedure centered approach, the object-oriented methodology is centered around objects/data. Objects actively drive the procedures and procedures are secondary and used only for describing the behavior of objects. For complex software applications, the object-oriented methodology has been shown to have a significant advantage in terms of the development cycle time, the program maintainability and the code reusability. As the scope of scientific computational problems becomes larger and the complexity of software tools to solve these problems increases, more developers are looking into the object-oriented technique as an alternative implementation.

The application of the object-oriented technique in the implementation of a 3D mesh generator is described in this chapter. Some experiences which we gained during the development process are presented. In addition to the description of the implementation of a mesh generator, the chapter presents a prototype implementation of a 3D device simulation system.

The chapter is organized as follows: Section 6.2 provides a general overview of the object-oriented technique. Section 6.3 discusses various implementation issues for the mesh generator, including a detailed description of various classes. Section 6.4 presents a prototype implementation of a 3D device simulation system and Section 6.5 gives a summary.

6.2 Overview of Object-Oriented Technique

6.2.1 Introduction

A key question to be answered by the software engineering is how to implement a complex software application. One well known technique is the divide and conquer technique [DIJK79]. Using the technique, a complex problem is decomposed into a set of simpler sub-problems and each sub-problem is further refined if necessary. The recursive process continues until each sub-problem is sufficiently simple to be solved. One example of the divide and conquer application is the octree based approach to generate 3D unstructured mesh, as discussed in the previous chapters. In this section, the divide and conquer technique is used to develop a systematic method for implementing large and complex software applications.

Two different ways have been proposed to decompose a complex software application. One is based on the algorithmic decomposition, where an algorithmic module is decomposed into a set of simpler algorithmic modules. Each sub-module is refined if the algorithm for the module can be further decomposed. The approach has been used extensively for solving scientific computational problems. For example, a numerical module to solve a system of partial differential equations can be decomposed into a number of sub-modules: the discretization module, the matrix assembly module, the matrix solver module, the algebraic solver module and the interpolation module. For direct solution methods, a further decomposition of the matrix solver module results the symbolic factorization module, the numerical factorization module, the forward solver module and the backward solver module. The decomposition is depicted in Figure 6.1.

The second way for decomposition is based on the object-oriented technique, where a complex software application is decomposed according to the abstraction of objects. A program is made of objects and each object has the state and the behavior. The execution of the program is the result of transactions of object states and interactions among objects through message passing. In the case where an object is considered to be too complex, it can be further decomposed into a set of simpler objects.

The advantage of taking the object-oriented approach is that the abstraction of objects enables us to concentrate more on the behavior of objects instead of the implementation. For most objects, it is much simpler to understand the behavior than its implementation. In

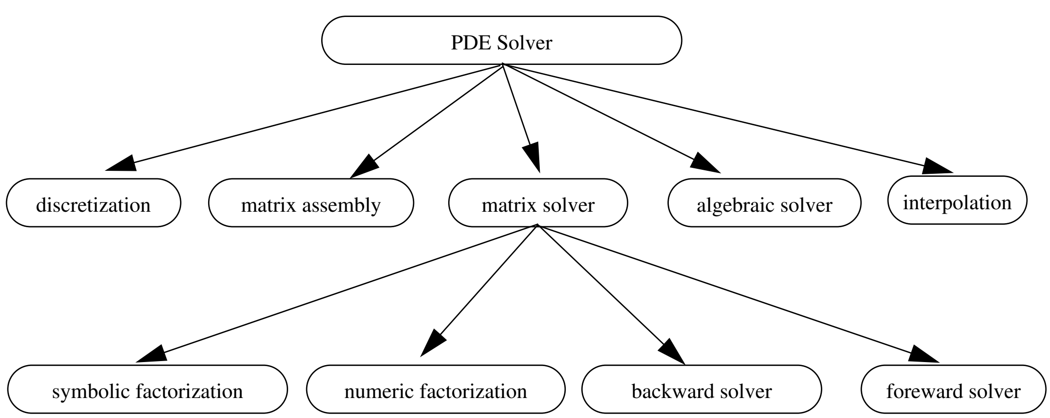


Figure 6.1 Algorithmic decomposition of a PDE solver.

other words, the realization of the complexity decomposition by the object-oriented technique is accomplished through the hiding of the implementation.

For example, in the implementation of a mesh generator, one important behavior belonging to an octant object is its tetrahedralization. As discussed in Chapter 4, different octants can have very different implementations, depending on whether it is an interior octant or a δ-zone octant. Using the object-oriented technique, only the behavior of the tetrahedralization is considered when we design the architecture of the program. The implementation of the tetrahedralization is delayed until the architecture issues are resolved.

One might ask between two approaches, by algorithm or by object, which is the right way? Traditionally, for most scientific computational applications, the first approach has been taken because the algorithm is the core of the solution to the problem. A successful implementation of an application largely depends on how good the implementation algorithm is. However, experiences have shown that the algorithmic driven approach is no longer adequate for providing software solutions for large and complex scientific problems. The success of the object-oriented approach in other fields has encouraged the use of the latter approach for solving scientific problems [YEAR93]. This is especially true when a scientific computational application becomes very complex and interfaces with many different software modules. The effort to handle software engineering issues, such as the user interface, the software migration and the code reusability, dominates the software development effort. As exemplified by our implementation of a mesh generator, the object-oriented approach can be a viable solution for the development of large complex scientific applications.

The further discussion requires a clear understanding of what is the object-oriented technique. As suggested by Booch, the object-oriented technique is defined to be[BOOC91]:

> *a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships [BOOC91].*

In the rest of this section, we discuss various aspects of the object-oriented implementation in more detail, including objects and classes, abstraction and encapsulation, class hierarchy and object inheritance.

To illustrate different concepts, various pieces of C++ code are used, where the following conventions are adopted.

1. The first letter of a *class* name is always capitalized. For example, *VertexObject* and *LevelControlObject* are all class names.
2. The first letter of an object name is always in lower-case. For example, *vertexA* and *theDevice* are names of the object instances.
3. An *int* is an integer type. A *float* is a floating point number type. A *boolean* is a data type taking only two values *TRUE* and *FALSE*.
4. A *floatVector3D* is a floating number vector of size 3.
5. A *List<int>* is a link list type where each element on the link list is of the integer type.

6.2.2 Object

One essential component for the object-oriented programming is the object. An *object* is

> *a tangible entity that exhibits some well-defined behavior and an object is completely characterized by its state, behavior and identity. [BOOC91]*

In C++, member variables (variables scope within a class of objects) are used to implement the object state and member functions are used to describe the object's behavior. The identity of an object is determined by the memory address of the object (also called the pointer to the object).

To illustrate the above definition, we use the vertex object as an example. Vertex objects are basic components for the description of a 3D geometry region. The state of a vertex consists of three real numbers *(x, y, z)*, i.e. the spatial coordinates of the vertex. The behavior of a vertex is described by two member functions, *move* and *moveTo*. When the *move* function is called, the location of the vertex is translated by a vector *(dx, dy, dz)*. When the *moveTo* function is called, the vertex object is move to a new location *(newX,* *newY, newZ)*, as illustrated in Figure 6.2. The following C++ code is used to implement vertex objects. The code contains all three components of the object definition, i.e. the state, the behavior and the identity. Two vertex objects *vertexA* and *vertexB* are declared.

```
CLASS VertexObject
{
    PUBLIC:
        float x, y, z; /* state of a vertex */
        move(float dx, float dy, float dz);
        moveTo(float newX, float newY, float newZ);
        ...
} vertexA, vertexB
```

Different objects may have the same state and behavior. But they are still different objects because of different identities. In the previous example, *vertexA* and *vertexB* may have same location coordinates *(0, 0, 0)*. In other words, they have the same state and the same behavior. But they are still two different objects.

In the above C++ code, we declare everything to be public, i.e. both variables and member functions are accessible by the outside. Such an implementation style is not a good practice for the object-oriented programming. Unlike the behavior of an object, the object state should be hidden from the outside world (an example of encapsulation). The state hiding is achieved through the declaration of PRIVATE, as illustrated by the following modified version.

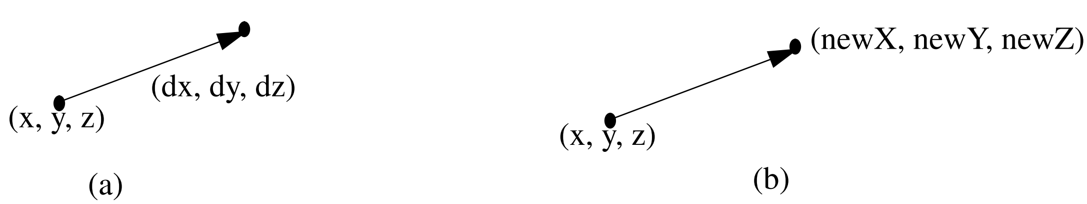


Figure 6.2 Illustration of different member functions for a vertex object: (a) *move* member function. (b) *moveTo* member function.

```
CLASS VertexObject
{
    PUBLIC:
        float x() { return _x; }
        float y() { return _y; }
        float z() { return _z; }
        move(float dx, float dy, float dz);
        moveTo(float newX, float newY, float newZ);
        ...
    PRIVATE:
        float _x, _y, _z; /* the state of a vertex object */
        ...
}
```

For the further discussion of the encapsulation, we need to introduce a concept called the client server model. In an object-oriented environment, a *client* is any object that uses the resources of another object (known as *server*). For example, when the *deviceObject*, an object to describe the mesh region, moves a vertex object by calling the member function *move*. Then we say that the *deviceObject* is the client object and the *vertexObject* is the server object.

The encapsulation of the state information from the outside clients has several advantages. First it hides the information of how the state is implemented. A change of implementation does not impact the outside clients, which is very important for supporting software migration as illustrated by the following example. The implementation of the vertex state can be realized by three float numbers. But it can also be done by three double precision real numbers. Due to the trade-off between the accuracy and memory usage, we may want to switch between these two implementations. If the state variables are hidden from the outside clients, no code change is required by any client object, as long as the interface of *VertexObject*, i.e. the PUBLIC portion of the declaration, remains the same. On the other hand, if state variables are in the PUBLIC portion of the declaration, i.e. *x, y, z* are used by some outside clients, then any change from the float to the double or vise versa requires the code change of the clients.

6.2.3 Class

The class concept is another essential component for the object-oriented programming. Unlike an object, a class is not a concrete entity existing in time and space. In other words, there is no memory allocation for a class. A class is only an abstraction which facilitates the grouping of a set of objects sharing a common structure and a common behavior. The abstraction through class declaration simplifies the way to describe the state and the behavior of objects.

For example, all vertex objects are of the same class *VertexObject*. Different vertex objects can have different states and different behavior. But the language to describe the state and the behavior is the same. For all vertex objects, the state consists of three float numbers, *x, y* and *z* and each of the objects has the behavior consisting of two member functions called *move* and *moveTo*.

In many cases, a class may contain a member function which takes different inputs. For example, the *VertexObject* class discussed before has a member function called *moveTo* which takes three float numbers *(newX, newY, newZ)* as its inputs. Sometimes, we would like to achieve the same behavior but by providing a position object as its input parameter, as illustrated by the following C++ code.

```
CLASS VertexObject
{
    PUBLIC:
        ...
        moveTo(float newX, float newY, float newZ);
        moveTo(floatVector3D newPosition);
        ...
}
```

This can be achieved by an object-oriented programming language such as C++ and the technique is called the polymorphism. The polymorphism is a very useful software technique which allows the same name to denote different implementations. Without polymorphism, a developer ends up writing code consisting of large case or switch statements. There are two types of polymorphisms. One is based on the different input parameters (also called the signature of a function) as discussed here. The other is based on the class hierarchy which is discussed in a later subsection.

6.2.4 Abstraction and Encapsulation

Abstraction and encapsulation are defined as follows:

> *"Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the difference"[DAHL72].*

> *"Encapsulation is the process of compartmentalizing the elements of an abstraction that constitutes its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation" [BOOC91].*

Abstraction and encapsulation are two key features to characterize an object-oriented implementation. Without abstraction, no class could be made because each object would be an unique entity. Without encapsulation, there would be no separation of the implementation from the interface.

To illustrate the concepts of abstraction and encapsulation, we again use the vertex object discussed previously. One major difference between a point and a vertex is that a point has a fixed location while a vertex can be moved. Since the *move* operation is often applied to vertex objects, we abstract this behavior out and attach it to all vertex objects, i.e. we declare the *move* as a member function of class *VertexObject*. Even though different vertices may be moved to different locations depending on where their origins are, the difference should not prevent us to abstract out the commonality of the *move* behavior, i.e. the vertex is translated by a vector *(dx, dy, dz)*.

On the other hand, the declaration of the member function *move* enables us to encapsulate the implementation of the *move*. A *move* can be implemented by updating the coordinates, i.e. replacing the *VertexObject* state *(x, y, z)* by *(x+dx, y+dy, z+dz)*. An alternative implementation is to record the translation vector *(dx, dy, dz)* as a separate vector whenever the *move* function is called and the original location *(x, y, z)* is not changed. The effect of the *move* is observed when the location access functions *x(), y()* and *z()* are called (an example of the lazy evaluation technique), as illustrated by the following C++ code.

```
CLASS VertexObject
{
    PUBLIC:
        float x() { return _x + _dx; }
        float y() { return _y + _dy; }
        float z() { return _z + _dz; }
        move(float dx, float dy, float dz) { _dx = dx; _dy = dy; _dz = dz; }
        moveTo(float newX, float newY, float newZ);
        ...
    PRIVATE:
        float _x, _y, _z; /* the initial state of a vertex object */
        float _dx, _dy, _dz; /* the translation vector;
        ...
}
```

6.2.5 Hierarchy and Inheritance

Class hierarchy is used to realize the "**is a**" relationship among objects. For example, a tetrahedron **is a** polyhedron. In other words, a tetrahedron class has the state and the behavior which a polyhedron class has. If the class A shares the structure and behavior of the class B, then the class A is called the derived class (or the subclass) and the class B is called the base class (or the superclass). We also say that the class A inherits from the class B. For example, the tetrahedron class is derived from the polyhedron class. There can be more than one derived classes. For example, a brick class is also derived from the polyhedron class. A C++ implementation of the polyhedron, the tetrahedron and the brick classes are illustrated as follows.

```
CLASS PolyhedronObject
{
    PUBLIC:
        ...
}


CLASS TetrahedronObject : PUBLIC PolyhedronObject
 /* inherits all Polyhedron structure and behavior */
{
    PUBLIC:
        Face face(int i); /* return the i-th face, where i = 1, 2, 3, 4
                             This member function is unique for Tetrahedron objects.
                          */
}
```

```
CLSSS BrickObject : PUBLIC PolyhedronObject
 /* inherits all Polyhedron structure and behavior */
{
    PUBLIC:
        /* The following 6 member functions are unique to Brick objects */
        float xMin();
        float xMax();
        float yMin();
        float yMax();
        float zMin();
        float zMax();
}
```

The advantage of using class hierarchy is that through the hierarchy a better abstraction is realized. Common properties and behavior can be abstracted. For example, the tetrahedron class and the brick class share many common properties and behavior. Both of them are 3D single connected geometry objects. They all contain a closed boundary which is made of some faces. To abstract these commonalities, a third class called polyhedron is defined, from which the tetrahedron and the brick classes are derived. The polyhedron class can abstract all common properties and behaviors.

In 6.2.3, we have discussed one type of polymorphism, i.e. an object can have the same member function which responds differently for different input parameters. Another type of polymorphism, discussed here, is that a derived class can inherits a member function from the base class but with different implementation. For example, a *PolyhedronObject* class may have a member function called *locationQuery* which checks if a location point is inside, on the boundary or outside of a polyhedron, as described by the following declaration.

```
CLASS PolyhedronObject
{
    PUBLIC:
        int locationQuery(Location P); /* =1 inside the polyhedron
                                          =0 on the boundary of the polyhedron
                                          =-1 outside the boundary of the polyhedron */
        ...
}
```

Since the brick class is derived from the polyhedron class, it also has the member function *locationQuery*. However, the implementation of *locationQuery* for the brick class can be very different from that for the polyhedron class. For a general polyhedron object, the algorithm to implement the member function *locationQuery* is much more complex than the algorithm for a brick class. In the latter class, we only need to check the x, y and z coordinates of the location point against the lower and upper bounds of the brick along the X, Y and Z axes. To have an efficient implementation for *locationQuery*, we would like to separate these two implementations. This can be achieved with the following declarations for the polyhedron and brick classes.

```
CLASS PolyhedronObject
{
    PUBLIC:
        virtual int locationQuery(Location P); /* =1 inside the polyhedron
                                               =0 on the boundary of the polyhedron
                                               =-1 outside the boundary of the polyhedron */
        ...
}

CLSSS BrickObject : PUBLIC PolyhedronObject
 /* inherits all Polyhedron structure and behavior */
{
    PUBLIC:
        virtual int locationQuery(Location P);
        /* The following 6 member functions are unique to Brick objects */
        float xMin();
        float xMax();
        float yMin();
        float yMax();
        float zMin();
        float zMax();
}
```

6.3 The Object-Oriented Implementation of a Mesh Generator

6.3.1 Introduction

In the previous section, a general overview of the object-oriented technique is provided. We introduced the object and class concepts, the abstraction and encapsulation techniques and the class hierarchy and inheritance. In this section, we demonstrate how the object-oriented technique is used for the implementation of a scientific computational application. The discussion is based on the C++ implementation of a 3D octree based mesh generator.

The mesh generator consists of three groups of objects, which are geometry objects, octree objects and user interface objects, as depicted in Figure 6.3. The geometry objects are used to describe the geometry of a semiconductor device for meshing and the geometry entities such as tetrahedrons, faces, edges, vertices and etc. The octree objects are for describing an octree and its construction process. The user interface objects are used for the mesh generator to interface with users, such as input/output file control and mesh generation process control. In the rest of this section, we discuss various objects in more detail.

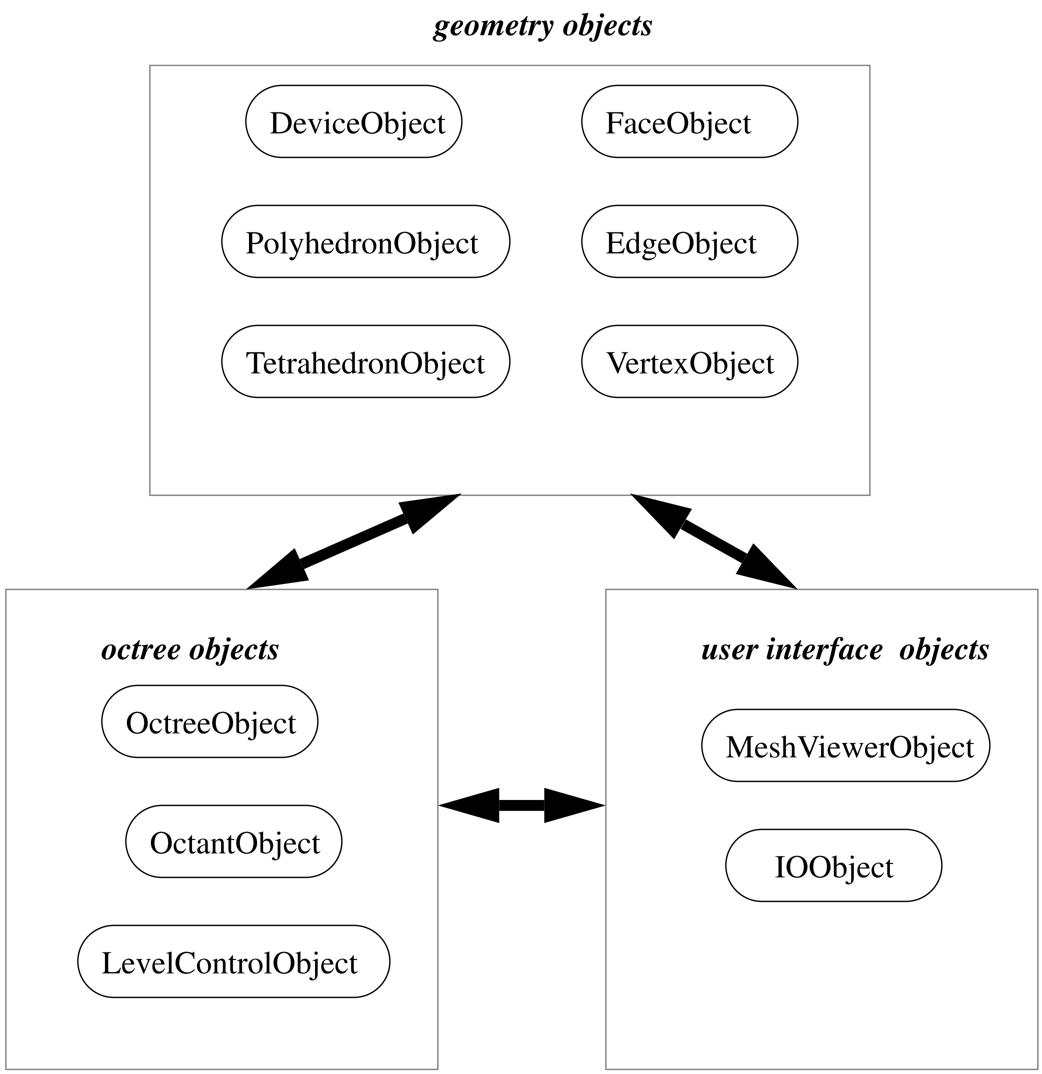


Figure 6.3 Object diagram for the mesh generator.

6.3.2 Geometry Objects

Objects in this group are DeviceObject, FaceObject, EdgeObject and VertexObject. They are used to describe the geometry structure of a mesh domain. These objects are created during the initialization process of the program.

DeviceObject is the entity denoting the semiconductor device to be meshed. For each execution, there is only one object in this class, i.e. the *deviceObject*. The d*eviceObject* contains the information about the mesh domain which consists of several geometry regions, i.e. the silicon region and the oxide region. One important behavior for the *DeviceObject* is the location query, i.e. to find if a given location is inside, on the boundary or outside of the device. Each of the geometry regions is represented by a polyhedron object with the material type. A *PolyhedronObject* contains a list of *FaceObjects*, which make up the boundary of the polyhedron. A description of the classes is given as follows:

```
CLASS DeviceObject
{
    PUBLIC:
        int locationQuery(Location P); /* =1: inside
                                          =0 : on the boundary of the device
                                          =-1: outside
                                        */
    PRIVATE:
        List<PolyhedronObject> pList;
}

CLASS PolyhedronObject
{
    PUBLIC:
        int materialType(); /* returns the material type */
    PRIVATE:
        List<FObject> fList;
}
```

FaceObject is used to describe the boundary face of a geometry region. The state of a *FaceObject* is represented by the plane which the face is on and by the edges which bound the face. The plane is determined by a point on the plane, called the *origin*, and the normal vector of the plane, called the *normal*. In our implementation, we assume that the normal always points outwards with respect to the polyhedron.

EdgeObject is used to describe a segment of the boundary for a face. It is determined by the line which the edge is on and by two bounding vertices. A line is described by a point on the line and the line vector. The boundary vertices of an *EdgeObject* are *VertexObjects*. Following are the C++ code for the declaration of the classes.

```
CLASS FaceObject
{
    PUBLIC:
        floatVector3D normal(); /*returns the normal vector of the plane */
        floatVector3D origin(); /* returns the origin of the plane */
        int edgeNum();          /*   returns the number of boundary edges for the face */
        ...
}

CLASS EdgeObject
{
    PUBLIC:
        floatVector3D lineVector();/* returns the line vector */
        floatVector3D lineOrigin();/* returns the origin of the line */
        VertexObject *start(); /* returns the starting vertex of the edge */
        VertexObject *end();  /* returns the ending vertex of the edge */
        ...
}

CLASS VertexObject
{
    PUBLIC:
        /* following 3 member functions return the coordinates for the vertex */
        float x();
        float y();
        float z();
        ...
}
```

One very useful geometry operation is the incidence checking operation between two geometry objects. For example, we have a *FaceObject* and a *VertexObject*. We want to check if the vertex is on the face or not. If not, we want to further determine if the vertex is on the positive or negative side of the face with respect to the face normal vector. There are three types of incidence operations: to check if two objects coincide, to find the intersections of two objects and to determine the orientation of one object with respect to the other. Examples of these three types of operations are illustrated in Figure 6.4 where (a) checks if a vertex is on a face, (b) finds the intersection point of an edge and a face and (c) determines which side (positive or negative) a vertex is on with respect to a face normal. In the following discussion, we discuss how polymorphism can be used to make the implementation of these operations easier. We use the operation of type (a) as an illustration. Similar discussion can be applied to operations of type (b) and (c).

Suppose we want to implement the operation which checks if two geometry objects are coincident, i.e. the operation of type (i). In a procedure based programming language such as C or PASCAL, these operations would be implemented by functions with different names, due to the different implementations. Using the object-oriented programming, different implementations can be abstracted by one common name called *checkIncidence*, an example of polymorphism. Depending on which geometry object the function *checkIncidence* belongs to and what is the input parameter, the same *checkIncidence* function can behavior differently. For example, the *checkIncidence* belonging to a *VertexObject* which takes a *FaceObject* as its input parameter checks if the vertex is on the face. On the other hand, the *checkIncidence* belongs to a *faceObject* which takes an *EdgeObject* as its input parameter checks if the face contains the edge.

To take advantage of the class hierarchy, we can further abstract the member function *checkIncidence* and make it a member function for general geometry objects. The following is a C++ implementation which has one base class called *GeometryObject* with the *faceObject, edgeObject* and *vertexObject* as its derived classes.

```
CLASS GeometryObject
{
    PUBLIC:
        virtual BOOLEAN checkingIncidence(vertexObject);
        virtual BOOLEAN checkingIncidence(edgeObject);
        virtual BOOLEAN checkingIncidence(faceObject);
}
```

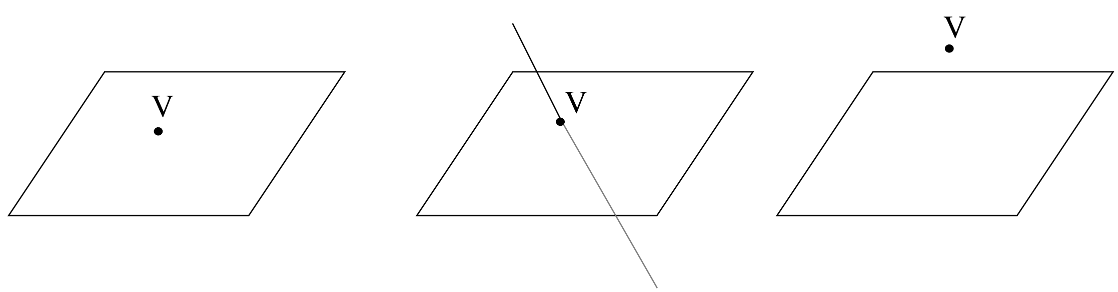


Figure 6.4 Examples of geometry operations: (a) Check if a vertex V is on a face. (b) Find the intersection vertex of an edge and a face. (c) Determine which side a vertex V is on.

```
CLASS FaceObject : public GeometryObject /* a derived class from GeometryObject */
{
    PUBLIC:
        realVector3D normal();    /*  returns the normal vector of the plane */
        realVector3D origin();    /*  returns the origin of the plane         */
        int edgeNum();            /*  returns the number of boundary edges for the face */

        /* the following member function checks if the face contains a vertex*/
        virtual BOOLEAN checkingIncidence(vertexObject);

        /* the following member function checks if the face contains an edge */
        virtual BOOLEAN checkingIncidence(edgeObject);
        ...
}
```

```
CLASS EdgeObject : public GeometryObject /* a derived class from GeometryObject */
{
    PUBLIC:
        floatVector3D v(); /* returns the line vector */
        floatVector3D o(); /* returns the origin of the line */
        VertexObject *start(); /* returns the starting vertex of the edge */
        VertexObject *end();  /* returns the ending vertex of the edge */

        /* the following member function checks if the edge contains the vertex */
        virtual BOOLEAN checkingIncidence(vertexObject);
        /* the following member function checks if an edge is on the face */
        virtual BOOLEAN checkingIncidence(faceObject);
        ...
}
```

```
CLASS VertexObject : public GeometryObject /* a derived class from GeometryObject */
{
    PUBLIC:
        /* following 3 member functions return the coordinates for the vertex */
        float x();
        float y();
        float z();

        /* the following member function checks if the vertex is on an edge */
        virtual BOOLEAN checkingIncidence(edgeObject);
        /* the following member function checks if the vertex is on the face */
        virtual BOOLEAN checkingIncidence(faceObject);
        ...
}
```

The above example demonstrates how the polymorphism and the class hierarchy can be used to abstract common behavior among different objects. Similar discussion can be applied to the operations of type (ii) and (iii).

6.3.3 Octree Objects

There are three types of object in this group, *OctreeObject, OctantObject* and *LevelControlObject*. There is one instance of the *OctreeObject* which denotes the entire octree. The *octreeObject* contains the root octant and has four basic member functions, *refineOctree, detectDeltaZone, tetrahedralizeInterior* and *tetrahedralizeDeltaZone*.

After the input is read and the *deviceObject* is constructed, the *octreeObject* contains the root octant. When the member function *refineOctree* is called (i.e. a message *refineOctree* is sent to the *octreeObject*), the *octreeObject* initiates the octree refinement process which recursively decomposes octants into sub-octants based on the level control function.

The member function *detectDeltaZone* generates the δ-zone after the balanced octree is created. Two tetrahedralization member functions are used to tetrahedralize leaf octants, which are *tetrahedralizeInterior* and *tetrahedralizeDeltaZone*. Detailed algorithms for these two member functions have been presented in Chapters 3 and 4. The following is a partial declaration of the *octreeObject* in C++.

```
CLASS OctreeObject
{
    PUBLIC:
        OctantObject *root();       /*  returns the root octant of the octree */
        refineOctree();             /*   the octree according to the level control function
                                         and generates a balanced octree.
                                    */
        detectDeltaZone();          /*  creates the delta-zone */
        tetrahedralizeInterior();   /*  tetrahedralizes the interior octants */
        tetrahedralizeDeltaZone();/*  tetrahedralizes the delta-zone octants */
        ...
}
```

The second type of objects is of class *OctantObject*, which is the core of the mesh generator. Each octant object has 6 faces, 12 edges and 8 vertices. It can be refined into 1 to 8 sub-octants if necessary. In addition, an object of class *OctantObject* provides many query functions about its properties such as to check whether an octant is a leaf. The following is a partial description of the *OctantObject* class.

```
CLASS OctantObject
{
    PUBLIC:
        /* member functions for the octant's tree structure */
        OctantObject *parent();             /*  returns the parent octant */
        OctantObject *subOctant(int i);     /*  returns the i-th sub-octant */

        /* member functions for the octant's structure */
        VertexObject *vertex(int i);    /*  returns the i-th vertex. */
        FaceObject *face(int i);        /*  returns the i-th face */
        EdgeObject *edge(int i);        /*  returns the i-th edge */

        /* member functions for the octant's properties */
        int isLeaf();                   /*  returns TRUE if the octant is a leaf octant */
        int level();                    /*  returns the level of the octant */
        ...
}
```

Another important behavior for the *OctantObject* is the tetrahedralization. We have shown that different types of octants have different tetrahedralization algorithms. The polymorphism is best suited for supporting such implementation variations. A generic tetrahedralization member functions is provided for the base class *OctantObject,* from which two different sub-classes *InteriorOctant* and *DeltaOctant* are derived. For the interior octant, we implement the tetrahedralization as discussed in Section 4.3. For the δ-zone octant, the implementation is based on the algorithm discussed in Section 4.4, as explained in the following C++ code.

```
CLASS OctantObject
{
    PUBLIC:
        ...                                 /* as discussed previously */
        virtual int tetrahedralization();   /* decompose the octant into tetrahedrons */
        ...
}

CLASS InteriorOctant : PUBLIC OctantObject
{
    PUBLIC:
        virtual int tetrahedralization();
}

CLASS DeltaOctant: PUBLIC OctantObject
{
    PUBLIC:
        virtual int tetrahedralization();
}
```

Another important class is the *LevelControlObject*, used for computing the level values. The base class contains a member function called *computeLevel*. As discussed previously, a user often wants to have a customized version of *computeLevel*. This can be easily implemented using the class hierarchy supported by the object-oriented technique. For example, we may want to use the doping profile for the initial grid and the post simulation error analysis for mesh refinement. We can accomplish this by creating two derived classes called *DopingObject* and *ErrorAnalysisObject*, as demonstrated by the following C++ code.

```
CLASS LevelControlObject
{
    PUBLIC:
        virtual int computeLevel(float x, float y, float z);
        ...
}
CLASS DopingObject : PUBLIC LevelControlObject
{
    PUBLIC:
        virtual int computeLevel(float x, float y, float z);
        ...
}
CLASS ErrorAnalysisObject : PUBLIC LevelCOntrolObject
{
    PUBLIC:
        virtual int computeLevel(float x, float y, float z);
        ...
}
```

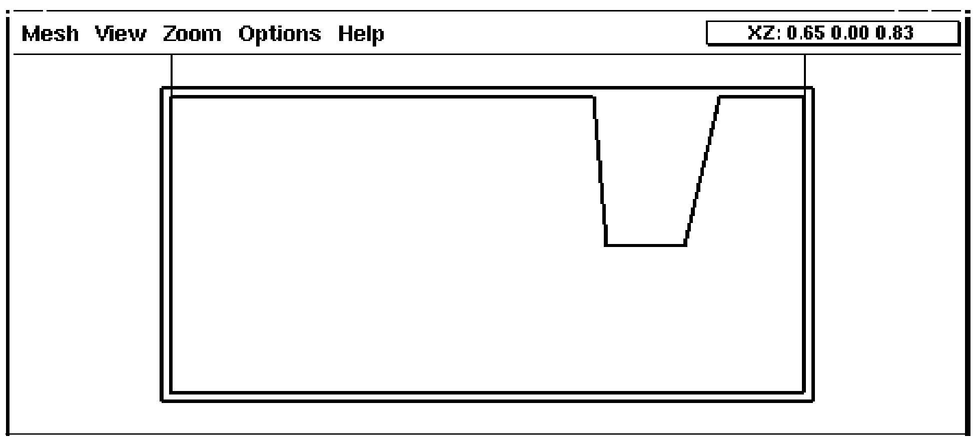
6.3.4 User Interface Objects

There are two types of objects used for handling the input/output and the user interface, *IOObject* and *MeshViewer*. The object of *IOObject* is used to access the SWR3D/ACIS data base (input) and write out the mesh in various formats (output). There are many different ways to read input and write output for a mesh generator. The encapsulation of different I/O activities into one object has a major advantage for future software maintenance and enhancement. Currently the input data is the SWR3D/ACIS database and the output is the AVS UCD database [AVS93].

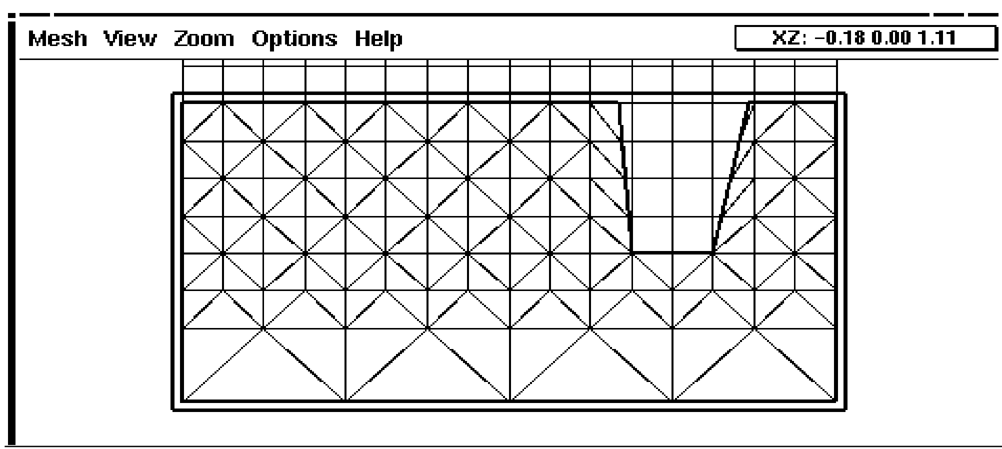
The object of class *MeshViewer* is the user interface manager for the mesh generator. It provides for the users a graphical environment to control the meshing process as well as the display of the mesh. Figure 6.5 displays (a) the geometry structure for meshing and

(b) the mesh. Such a graphical environment is very helpful for the users to achieve desired mesh and can greatly improve the tool's usability.

The implementation of the graphical user interface of the mesh generation is based on the object-oriented graphic widget system, *interview*[INTE92]. Since the objective of the graphical environment is not for visualization, the *MeshViewer* only displays the mesh in a 2D manner, i.e. at any time, only a 2D cross section of the mesh is provided. In the current implementation, the cross section is confined to be parallel to one of XY, XZ and YZ planes.



(a)



(b)

Figure 6.5 User interface for the mesh generator: (a) The geometry region for mesh. (b) Mesh generated.

6.4 A Prototype Implementation of a 3D Device Simulation System[1]

6.4.1 Introduction

In this section, we present a prototype implementation of a 3D device simulation system. The system is intended to demonstrate the framework concept and the individual tool capability. Our implementation of the 3D device simulation system is based on the SWR3D/ACIS framework. In addition to the framework capability including the mesh generation in both 2D and 3D, the system consists of a set of applications, which include tools from the commercial vendor such as CADENCE, tools developed by other university such as FASTCAP and tools developed internally such as a mixed mode simulator.

The system enables a user to describe a device by its mask layout using the layout editor developed by CADENCE [CADE92]. Using the mask layout and the process information, the 3D device generator creates the 3D structure of the device using various geometric operations provided by the SWR3D/ACIS[TSE93]. Based on the 3D structure, various simulators can be applied for device analysis and model generation. A 2D mixed mode simulator which couples SPICE and PISCES can be used for circuit simulation after the 2D cross section mesh is created [YU93]. The interconnect of the device can be analyzed by FASTCAP [NABO92]. Figure 6.6 depicts the system information flow for a 6 transistor SRAM cell. The rest of the section provides the discussion on each part of the system, which includes the mask layout generation, the 3D structure generation, the 2D cross section simulation and the 3D simulation.

1. This work is also contributed by William Wong, Eric Tse, Melissa Kok, Ken Wang, Anyu Lee and Francis Rotella.

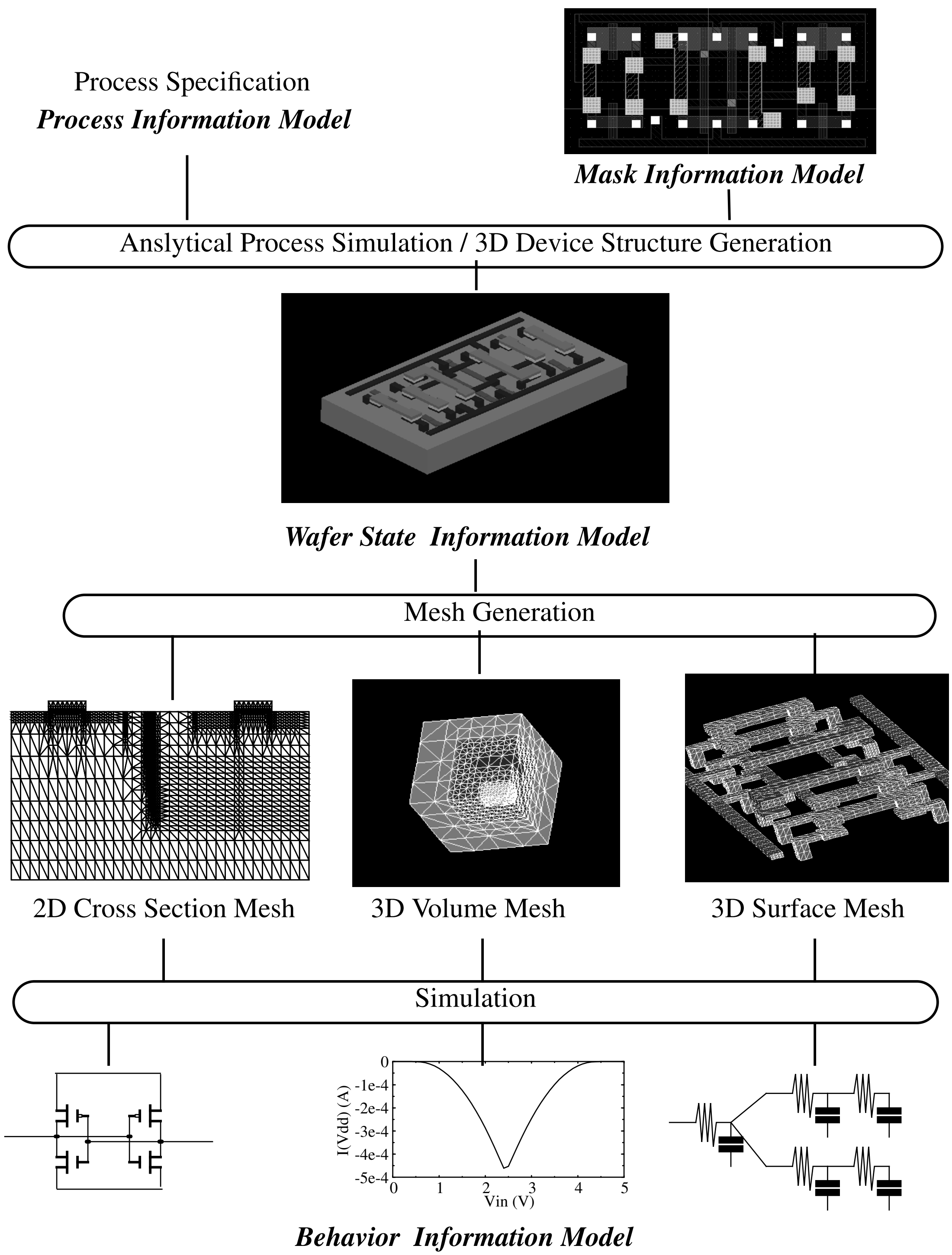


Figure 6.6 3D Device simulation information flow.

6.4.2 Mask Layout Generation

The mask layout editor developed by CADENCE [CADE91] is used as the device design entry tool for generating the mask layout of a device, as illustrated in Figure 6.7. Non-Manhattan polygonal geometry regions are supported for defining masks on multiple layers. After the mask layout is created, the mask information is saved to an object-oriented information data base. The mask layout information data base is implemented on top of a heterogeneous distributed database system [EITH93]. The objects provided by the mask layout information model include the mask set and the polygonal mask regions.

6.4.3 3D Structure Generation

The 3D structure generator incorporates the semiconductor process information and the mask layout information to create the 3D device structure. The generator consists of different process modules which perform various process steps such as etch, decomposition, implantation and etc. The current implementation of these modules uses 3D geometry operations provided by the SWR3D/ACIS. For example, a uniform growth step can be modeled by a sweeping operation, as illustrated in Figure 6.8.

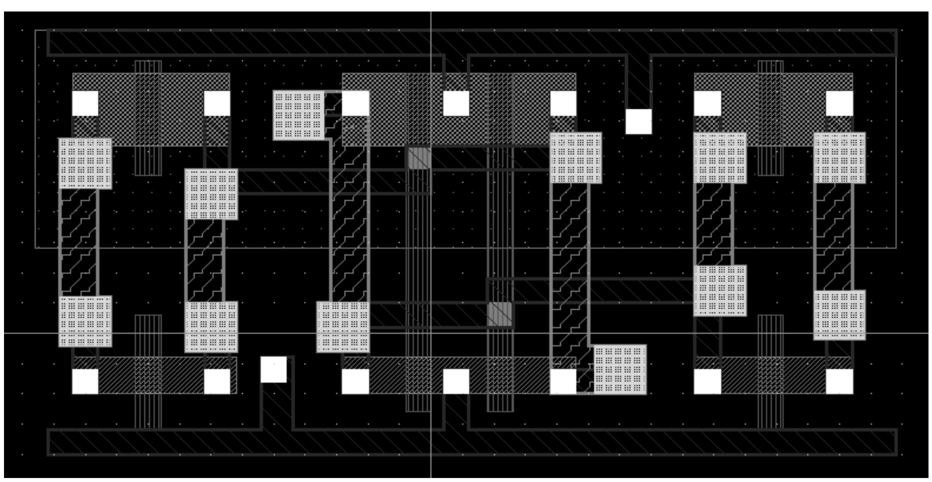


Figure 6.7 Mask layout of a 6 transistor SRAM cell.

The 3D device structure is modelled using objects provided by the SWR3D/ACIS. In addition to the geometry, physical properties attached to various geometry regions are implemented using the attribution scheme provided by the SWR3D/ACIS. It should be noted that the usage of attribution scheme to describe the physical properties has its limitation due to its lack of mesh structure. A field server is required in order to fully model a semiconductor device. An example of a 3D device structure is illustrated in Figure 6.9.

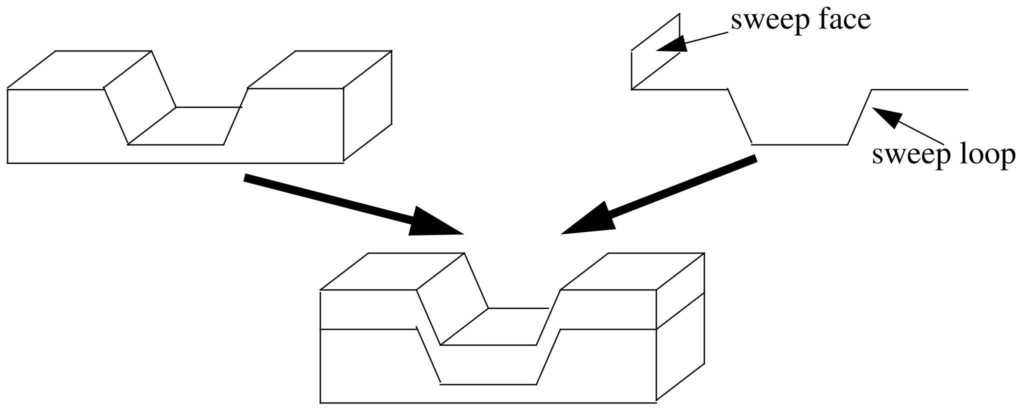


Figure 6.8 An uniform growth process step implemented as a sweeping operation in 3D modeling.

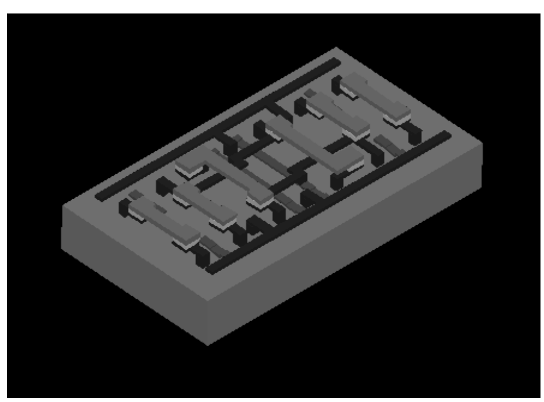


Figure 6.9 The 3D structure of the SRAM cell.

6.4.4 2D Cross Section Simulation

The 2D cross section simulation consists of a set of applications including a 2D cross section generator, a 2D mesh generator and a 2D device simulator, as illustrated in Figure 6.10. The 2D cross section generator reads the plane cross section information and

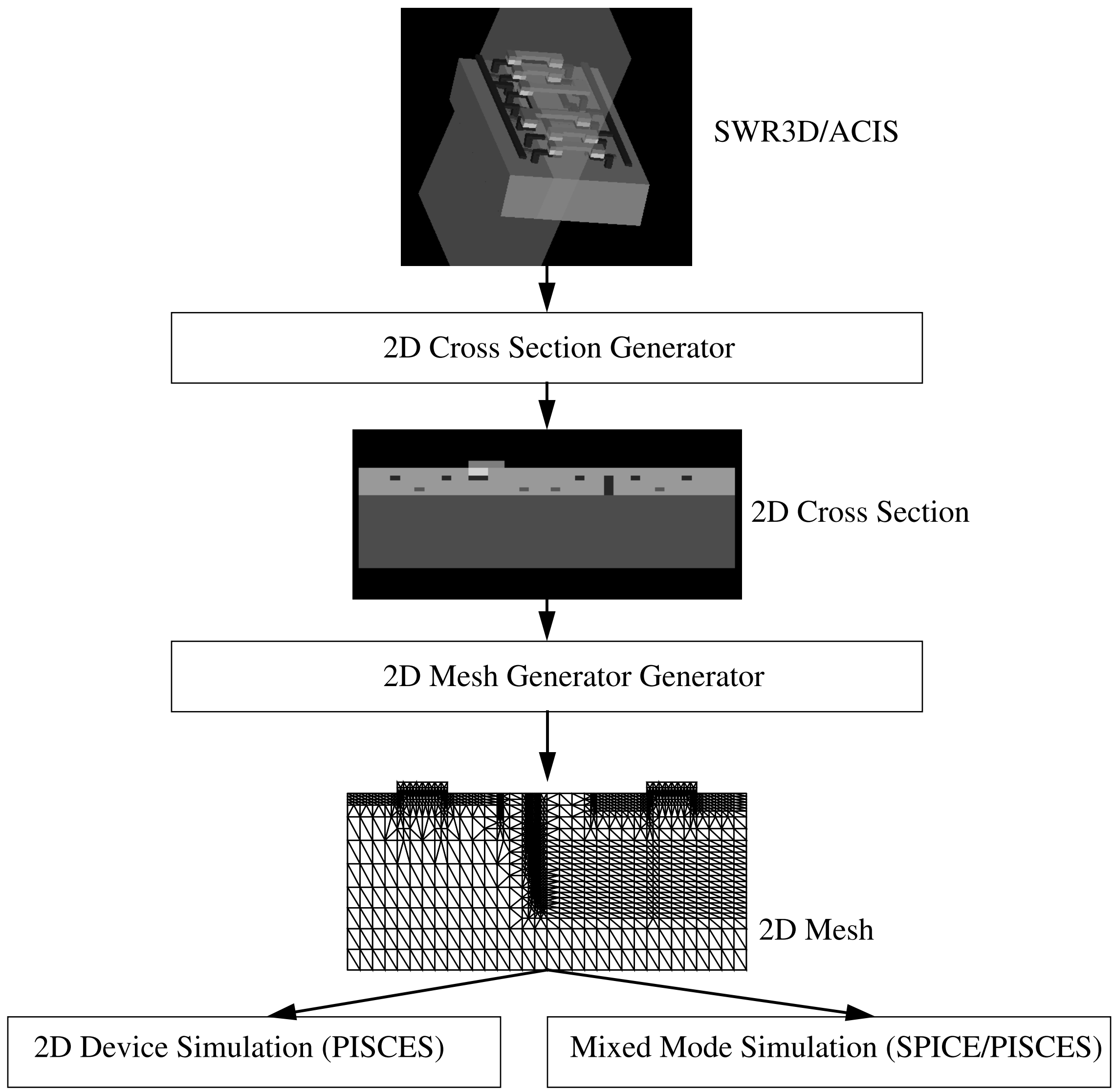


Figure 6.10 A 2D Cross Section Simulation.

calls the function *sectioning* supported by SWR3D/ACIS. After the 2D cross section is formed, the 2D quadtree based mesh generator is applied to generate the unstructured mesh which is then used by PISCES, a 2D device simulator, to perform 2D device simulation. Recently, we have incorporated a mixed-mode simulator into the system which consists of PISCES for the device level simulation and SPICE for the circuit level simulation.

6.4.5 3D Simulation

In addition to the 2D cross section simulation discussed previously, we can apply 3D device simulation directly. The octree based mesh generator discussed previously is used for generating unstructured mesh for 3D device simulation.[1]

Another important application based on the 3D device structure is the interconnect analysis. In today's VLSI circuit, the interconnect effect becomes very significant in terms of the chip performance and reliability. Traditionally, interconnect analysis is based on 2D layout extraction, which is not sufficient in many cases where a 3D simulation is required. Based on the 3D device information model, we can apply the interconnect simulation. A 3D capacitance extractor, FASTCAP [NABO92], has been incorporated into the system. The surface mesh required by FASTCAP is provided by the system, as illustrated in Figure 6.11.

1. An effort to enhance STRIDE [WU90], a 3D device simulator, to support unstructured mesh is currently underway.

6.5 Summary

In this chapter, we first provide an overview of the object-oriented technique in software engineering. We discuss the object and class concepts, the abstraction and the encapsulation, the class hierarchy and the inheritance. Then we discuss various implementation issues regarding to the octree-based mesh generator. We illustrate how the object-oriented technique can be used to develop a scientific software application. Lastly, we presented a prototype implementation of a 3D device simulation environment.

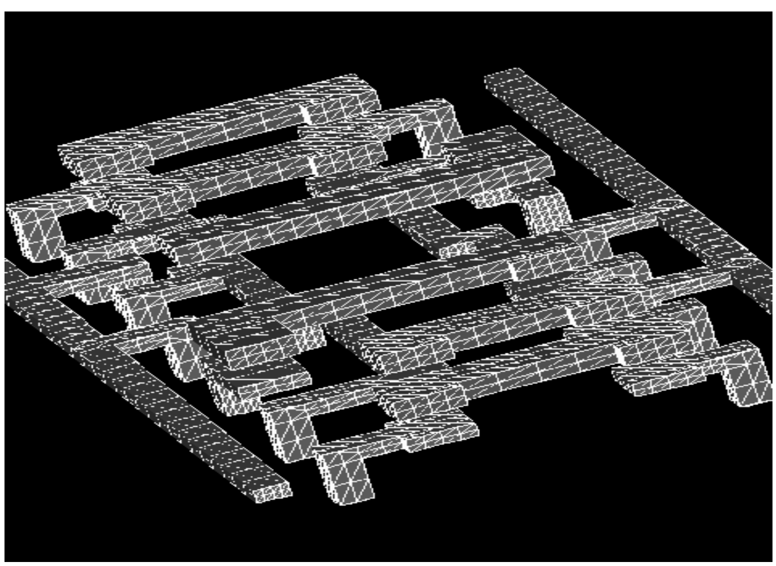


Figure 6.11 The 3D surface mesh for the SRAM interconnect