OpenSG Symposium (2003)
D. Reiners (Editor)

A Framework for Dynamic Connectivity Meshes

J. Vorsatz and H.-P. Seidel

Max–Planck–Institut für Informatik,
Saarbrücken,Germany

Abstract

Implementing algorithms that are based on dynamic triangle meshes often requires updating internal data-structures as soon as the connectivity of the mesh changes. The design of a class hierarchy that is able to deal with such changes is particularly challenging if the system reaches a certain complexity.
The paper proposes a software design that enables the users to efficiently implement algorithms that can handle these dynamic changes while still maintaining a certain encapsulation of the single components.
Our design is based on a callback mechanism. A client can register at some `Info`*-object and gets informed whenever a change of the connectivity occurs. This way the client is able to keep internal data up-to-date. Our framework enables us to write small client classes that cover just a small dedicated aspect of necessary updates related to the changing connectivity. These small components can be combined to more complex modules and can often easily be reused. Moreover, we do not have to store related 'dynamic data' in one central place, e.g. the mesh, which could lead to a significant memory overhead if an application uses some modules just for a short time.*
We have used and tested this class design extensively for implementing 'Dynamic Connectivity Meshes and Applications [9]'. Additionally, as a feasibility study, we have implemented and integrated our concept in the OpenMesh[2]*-framework.*

1. Introduction

Triangle meshes are a well established standard to represent the outer skin of a 3D geometric object. Compared to *dynamic meshes*, algorithms that are based on *static meshes* are usually easier to implement from a design point of view. Modules that realize such an algorithm just need to store static data, with respect to the mesh, that does not change during the runtime of the application.

To give a simple example, the application might comprise one module that sets a flag whenever an edge of the mesh gets selected by a user. Since the connectivity of the mesh does not change the module can use an internal `vector-of-bools` that reflects the current status of each edge.

When it comes to meshes that change their connectivity during runtime, however, implementing algorithms that operate on them gets more involved. Data that is stored internally to some module has to be aware of these changes. This task is particularly challenging if the application reaches a certain complexity and one wants to implement components that can be reused. One way of solving such a problem is to store the data (edge-flags) outside of the module, e.g. directly along with the mesh-data-structure that obviously 'knows' when its connectivity changes.

One possible way to store data within the mesh are the so called *Meshtraits* or *Meshitems* that have been effectively used in several libraries [2, 5]. This is an excellent approach, if the data that is to be stored is an 'established property' that can be reused. This might for instance be a vertex-property such as the valence, a list of all adjacent triangles, texture-coordinates etc. The major advantage is the fact that multiple modules that work with the mesh have easy access to this data and the data is stored/updated in just one place. We found that in this case, the loss of data encapsulation is not a severe restriction (as long as it gets updated correctly). However, the documentation of a module that uses such a Meshitem should explicitly state that it needs a specific Meshitem and a compile-time check[1] should make sure that the appropriate Meshitems are present.

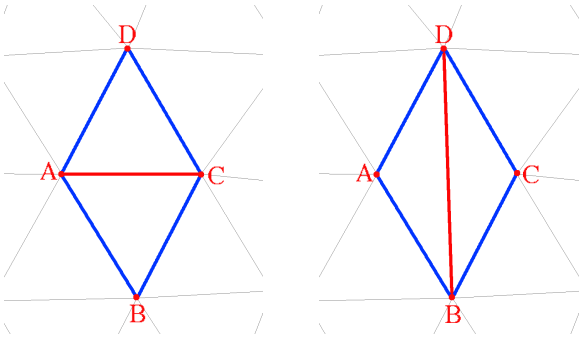

Figure 1: Illustration of an edge-flip, the example-operator in this paper. The common edge of the triangles $\triangle ABC$ and $\triangle ACD$ (left) 'flips' and forms the new triangles $\triangle ABD$ and $\triangle BCD$ (right).

On the other hand, if a module makes use of module specific data that is used only temporarily, inflating/polluting the Meshitems with this data is problematic. In particular for more complex applications one can easily lose control over all the different components, which clearly limits the maintainability. Even worse, the data consumes memory throughout the lifetime of the application even though it might get used for a short period only.

For this reason we have developed a framework that enables independent modules of an application to be alerted whenever the connectivity of the mesh changes. We ensure that the modules do not have to expose internal data to the outside, which facilitates their reusability. Our framework is based on a callback mechanism, all client classes that have to be aware of a changing connectivity supply a common interface (they derive from a common base-class).

In Section 2 we will build up the callback mechanism that informs a custom tailored client class whenever the connectivity of the mesh changes. Section 3 illustrates how we can make necessary information available to a client. In Section 4 we show the concept of informing *multiple* modules of a change in the connectivity while being completely independent of each other. To clarify our concept, we will describe an example application in Section 5 and point out some extensions of our framework in Section 6.

2. The Callback Mechanism

In order to keep our explanations of the class design and the examples as simple as possible, we will restrict ourselves to one single topological operation, the edge-flip (cf. Fig1). A fully-fledged library would of course comprise the complete set of operators that change the connectivity of the mesh, e.g. , edge-split, edge-collapse, face-split, etc. The additional operators can be integrated into our framework in a similar way as the edge-flip and are thus omitted here.

As we have illustrated in Section 1, we will store data that is sensitive to changes of the connectivity of the mesh along with the modules that are using this data, instead of storing them in Meshitems inside the mesh. This way however we are not able to update our data inside some private method of the mesh whenever `Mesh::flipedge()` gets called. Hence, we cannot call `Mesh::flipedge` directly and therefore outsource calls that change the connectivity of the mesh to another class. It makes sure that 'dynamic data' is always up-to-date and calls `Mesh::flipedge`.

So instead of calling

```
mesh->flipedge(edge_handle);
```

directly and thus risking that some modules remain clueless about the fact that the connectivity of the mesh has just changed and consequently that the data they store might be outdated, we wrap the call to `Mesh::flipedge` by two calls to methods which the users can custom tailor to their needs. Later we will show how a module can hook into these calls and is thus updates its own data whenever an edge-flip occurs.

For now we will show a very simple example of this concept and will later develop a more complex class that we are using in a real-world application.

The core piece of our framework is called `Dynamic`, the following listing illustrates the basic form of the callback mechanism.

```
struct Dynamic<Mesh> {

  void flipedge (EdgeHandle _edge_handle){
    if ( info->preFlip()){
      mesh. flipedge (_edge_handle);
      info->postFlip ();
    }
  }
 Mesh&     mesh;
 InfoBase* info;
};
```

Listing 1: `Dynamic::flipedge` shows the basic form of the callback mechanism a user can hook into.

in this example, `InfoBase` is implemented as follows:

```
class InfoBase{
 virtual bool preflip (){return true;}
 virtual void postFlip (){};
};
```

`InfoBase` is meant as a dummy base-class that does nothing but illustrate a certain interface, i.e. in its simplest form the call to `Dynamic::flipedge()` does nothing but flip an edge of the mesh just as a direct call to `Mesh::flipedge()` would have done. So what is the benefit of introducing this additional layer?

The users can derive their own `MyInfo`-class from `InfoBase` and replace `Dynamic::info` with it. This way additional functionality can be implemented and the appropriate `MyInfo::pre/postFlip`-method gets called whenever `Dynamic::flipedge` gets called.

As a simple example we implement `MyInfo` class as follows:

```
class MyInfo : public InfoBase
{
 bool preflip () { cout « "preFlip";return true;}
 void postFlip () { cout « "postFlip";}
};
```

Using an instance of `MyInfo` Listing 2 illustrates how to flip the edge with the EdgeHandle 0 while getting feedback about the `MyInfo::pre/postFlip`-calls.

```
int main()
{
 Dynamic dynamic;
 //read mesh & pass it to dynamic

 MyInfo myinfo;
 dynamic.info = &myinfo;
 dynamic.flipedge(EdgeHandle(0));

 //output of the program:
    preFlip—called postFlip—called
}
```

Listing 2: Getting feedback about the flip of edge 0 via MyInfo.

Please note that you can prevent an edge from flipping by returning *false* in your own `MyInfo::preFlip()`-method. This way you can (in addition to executing edge-flip specific code) influence the optimization process. We will come back to this point in Section 5.

Conceptually it would make sense to distinguish between the influence on the optimization process (cf. Section5) and the execution of edge-flip specific code. Consequently we should separate between, for example, a class `MyDataUpdateBase` which provides the `pre/postFlip`-interface and another `MyStrategyBase`-class where the users can implement different strategies to influence the optimization process. In practice however, we found it more convenient to have everything in one single `MyInfo`-object.

3. Passing Data to `MyInfo`

Up to this point we get informed via our own `pre/postFlip`-method whenever an edge-flip occurs. Of course, a very important fact we are interested in is where the flip actually took place. This is crucial for executing flip-specific code in `MyInfo`. It would be straight forward to pass the edge to the `pre/postFlip`-method as an argument. In practice however, we found that we often need more information about the flip that is going to take place or just took place. For this reason we pass a pointer to a whole `Data`-struct to `pre/postFlip`, which is of the form:

```
struct Data{
  EdgeHandle flip_edge_;
  ...
};
```

and is a member of `Dynamic`. Opposed to our toy-example, this struct also holds all the information that is needed for the other topological operations (cf.Section 2). This additional information can be exploited for instance if the user needs to know which was the last edge that collapsed prior to the current edge-split etc.

We have made `Data` a member of `Dynamic` instead of storing it directly in `InfoBase` for two reasons: First, for reasons of efficiency since we can use the members of `Data` to store the current edge directly while being in `Dynamic::flipedge()` and thus do not need any additional copy operation.

Second and more importantly, in the next expansion stage of our framework we will introduce the concept of multiple `InfoBase`-objects that work independently of each other. Each of them gets notified by a special instance of an `InfoBase`-object, however, we would like to avoid multiple instances of `Data`. We will see how our framework can benefit from this and point out some implementation issues in the next section.

4. Distributing to multiple Clients by an Observer

With the current implementation of `MyInfo` we would have to put all code, which has to be executed in order to respond correctly to an edge-flip, into `MyInfo`.

In practice, for a more complex application, a comprehensive, custom tailored `MyInfo` class can easily become a *Blob* [3], i.e. a single class with a large number of attributes and operations. Even worse, we would not have gained much compared to the 'embed-all-edge-flip-specific-code-in-the-mesh-class'-approach (cf. Section 1), meaning that if a module puts flip-specific code into `MyInfo` entails that this module cannot encapsulate and manage its own data anymore.

By using the observer/observable-pattern[4], the diagram in Figure 2 shows how we can get around the two problems that we have just mentioned. A client can register at the Observer, in our framework we call it `Hub`, and gets called whenever an edge-flip is performed. This way we can write small, independent client classes that are aware of changes of the mesh-connectivity. The whole concept works as follows.

We create a `Hub` that is derived from `InfoBase` and let `Dynamic::info` point to it - this way the Hub gets called by `Dynamic::flipedge()`. A client class derives from `InfoBase` and hence supports the `pre/post-Flip` interface. Now an object of this client class registers at the `Hub` by passing a reference to it. The `Hub` maintains a list of these client objects, the callees. Whenever `Hub::pre/postFlip` gets called, the `Hub` passes the call to all its callees. Additional information about that flip is available via `Data`. A pointer to this struct is passed to the `pre/postFlip` methods and can thus be exploited by the callees.

Again, using this approach we are able to hide client specific code and data-structures and do not have to expose it to some central instance. We found that the Hub also encourages users of our framework to write small and independent and thus reusable components.

In our current implementation the `Hub` holds a simple list of references to callees that get called one after the other. However, if the user needs fine grained control over the order of execution of the client classes, one can easily incorporate a more sophisticated calling strategy. This could either be calls by priority, but one can also think of a calling-tree similar to a scenegraph (the `Hub` serve as nodes, the clients are the leaves).

5. An Example Application

In this Section we will showcase a small application scenario that demonstrates how the framework can be put into practice. We just want to give an impression how the parts of our concept play together and show that the different modules form a closed entity that can be reused easily. Of course, many more applications can be realized similarly to our simple example and we hope that the pool of modules that uses our framework will grow rapidly.

Assume we are given a triangle mesh that contains vertices with high valences, i.e. many edges emanate from these vertices. A multitude of algorithms in geometric modeling prefer vertices with valence six (or at least close to six). The edge-flip is one operator that can reduce this valence-excess (cf. [6, 8] for a detailed description).

In our example we will use two client classes that register at a `Hub` (cf. Section 4). `ValenceStore` manages the valences of all mesh-vertices - it serves as an example for a module that holds its own data and updates it if the connectivity of the mesh changes.

```
class ValenceStore : public InfoBase{

  // assume the valences are stored in valenceMap_
  // and initialized by the constructor of this class.

  //update valences after an edge_flip
  bool postFlip(const Data& _data){

    EdgeHandle e = _data.flip_edge_;

    // Update their valences of the four adjacent
    // vertices v_i of e        //       v3
    valenceMap_[v[0]] -=1;      //       / | \
    valenceMap_[v[1]] +=1;      //    v0 | v2
    valenceMap_[v[2]] -=1;      //       \ | /
    valenceMap_[v[3]] +=1;      //        v1
  }

  std::map<VertexHandle,int> valenceMap_;
};
```

```
class BalanceStrategy : public InfoBase{

  // Pass a ValenceStore object to
  // this class in the constructor

  bool preFlip(const Data& _data){

    EdgeHandle e = _data.flip_edge_;

    // Now get the four adjacent vertices of e and their
    // valences val[0,...,3] from valenceStore_ ...

    //... and calculate the valence-excess ...
    current_excess = sqr(val[0]-6) + ... + sqr(val[3]-6);

    // New valences under the assumption that
    // a flip has taken place
    val[0] -= 1; val[1] += 1;
    val[2] -= 1; val[3] += 1;
    flip_excess = sqr(val[0]-6) + ... + sqr(val[3]-6);

    if( new_excess < current_excess ) return true;
    else return false;
  }
};
```

Note that we do not have to enumerate all adjacent vertices of a vertex to recalculate its valence, since we know how the valences of the four vertices are affected by an edge-flip.

The second class is an example for exerting influence on the execution of edge-flips. `BalanceStrategy` is a class that calculates the valence-excess of vertices incident to an

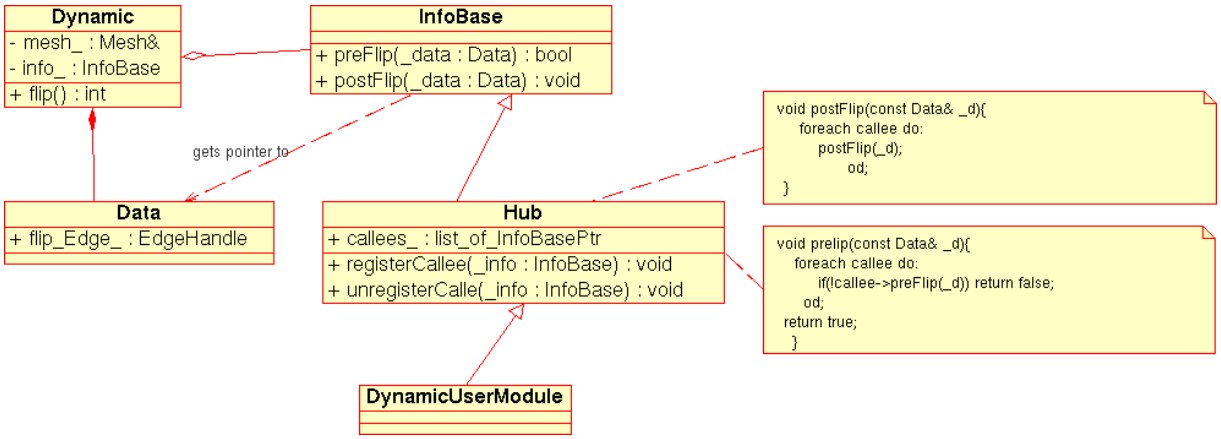

Figure 2: UML-diagram of our framework. A client class derives from `InfoBase` and registers at the `Hub`. The `Hub` gets passed to `Dynamic` and is thus 'aware' of a (scheduled) edge-flip. Additional information about that flip is stored in `Data` which is a member of `Dynamic`. A pointer to this struct is passed to the `pre/postFlip` methods and can thus be exploited by a client class.

edge and indicates via `preFlip` if an edge-flip would improve it.

Eventually we can assemble the components. A sketch of the main parts of the program that balances the valence-excess of a triangle mesh is shown in Listing 3.

```
int main()
{
 Dynamic dynamic;
 //read mesh & pass it to dynamic

 Hub hub;

 ValenceStore vStore(mesh);
 hub. registerCallee (&vStore);

 BalanceStrategy balance(vStore);
 hub. registerCallee (&balance);

 dynamic.info = &hub;

 //now iterate ...
 for( e = mesh.edges_begin(); e!= mesh.edges_end();++e)
   dynamic. flipedge(e);
}
```

Listing 3: Reducing the valence excess

6. Extensions

As capturing all details of our framework exceeds the scope of this paper, we have only outlined the core concepts. However, there are many ways to extend the concept we have shown so far and we want to highlight some of them.

In our implementation we have used an *iterator* for processing multiple edges at once. So, instead of passing edges to `Dynamic::flipedge()` one-by-one, we iterate over a whole set of edges. We will illustrate the advantage of this concept by means of a small example.

Let's stick to the valence-excess example of the previous section. Assume we have a module *A* that wants to prevent the four blue edges (cf. Figure 1) from flipping as soon as the red edge has flipped - the four edges remain locked as long as we have not processed every edge in the mesh. After one iteration over all edges, the status of these edges is set to 'free' again. If a module *B* of our application processes edges one-by-one, we need an indicator when it is done with processing. The problem is that both modules might be unaware of each other. We can solve this problem by leaving the control over the edge-flips to `Dynamic` and inform the user via the `InfoBase`-interface after all the edges provided by the iterator are processed. This way all clients that have registered at the `Hub` can respond in their specific way. Of course, the iterator is not limited to iterating over all edges of a mesh , but it can feed an arbitrary set of edges to `Dynamic::flipedge()`.

Listings 4 and 5 show the extension we have to make in order to realize the concept.

```
struct Dynamic<Mesh> {

  // canditate serves as iterator over a set of edge

  void flipedge (){
    for( candidate->init (); candidate->hasmore();
         candidate->next() ){
      if ( info->preFlip(const Data& _d)){
        mesh. flipedge ( canditate ->get());
        info->postFlip(const Data& _d);
      }
    }// end: for all candidates

    info->endFlip();
  }

  Mesh&     mesh;
  InfoBase* info ;
  Candiate* candidate ;
};
```

Listing 4: Incorporating the iterator-concept into Dynamic.

```
class InfoBase<Dynamic>{
 virtual bool preflip (const Dynamic::Data&){return true;}
 virtual void postFlip (const Dynamic::Data&){};
 virtual void endFlip (){}
};
```

Listing 5: Extended version of InfoBase.

Another venue for extending our framework is the design of small client classes that cover just one specific aspect of the changing connectivity. In this context we do not limit ourselves to the edge-flip, but think of the complete set of operators that change the connectivity of a mesh. The aspects can be as diverse as:

- a change of the adjacency list of a vertex.
- a notification that some triangles/edges/vertices have vanished.
- a change of normals in the vicinity of an operation.

For instance we could address the first item by designing a module that maintains a list of adjacent vertices for each vertex in a mesh. This modules registers at the `Hub` and updates its internal list with respect to a notification it gets via the callback mechanism - this can be done efficiently, since the module 'knows' how the adjacency lists have to be updated for e.g. an edge-flip. Now the module can grant a client access to these lists. The client does not need to worry if the lists are up-to-date or maintain its own list.

We can even go one step further and separate the data (the adjacency lists) from the information about the update (which list has changed in which way). Using this concept we can design a class *A* that registers at the `Hub` and just takes care of the update. Another class *B1*, which registers at *A*, can e.g. hold the adjacency lists. To carry on this thought, we also think of a class *B2* that just needs to be notified about a change of the adjacency-information, but does not hold a complete adjacency-list at all. Eventually, this concept will lead to a tree-like structure of callbacks that enable a client to register at those points that are vital for its algorithms.

7. Results

We have used the framework we have described in the last Sections to implement *FSR*, a program that comprises the algorithms propose in [8, 9]. One result we have achieved with *FSR* is shown on the first page. In *FSR* we register dozens of modules that inter-operate with each other, it showed that managing these modules without clearly separating between them is quite error-prone. We have also implemented a small example which is based on *OpenMesh*[2]. A tar-archive can be downloaded from our web-site [7]. Please note that the current version is just a feasibility study and is not mature enough to be used in a production environment.

Certainly, our framework comes along with some overhead compared to directly calling the member functions (edge-flip/edge-split, etc.) of the mesh. As a worst case scenario we have tested our implementation for *OpenMesh*[2]. We have passed an empty `InfoBase`-object to `Dynamic` in order to disable the callback-mechanism and have executed one single edge-flip via our iterator-interface. This setup is 2.5x slower that the direct call to `mesh.flip()`. However, after changing our *FSR*-program to the proposed concept, we not only found that it was easier to incorporate new algorithms, but we were also able to discarded many calls to redundant update routines and eventually make *FSR* significantly faster.

8. Conclusion

We have proposed a framework for efficiently handling and working with dynamic connectivity meshes from a design point of view. In particular we have shown how encapsulated modules that depend on the changing connectivity of a mesh can keep internal data-structures up-to-date. Our callback-mechanism facilitates implementing new algorithms which base on dynamic meshes and shows how to add this new functionality to complex applications that make use of our framework.

Of course, there are many venues for further extensions and improvements. We expect, that a rich pool of small and reusable clients, which cover one specific aspect of the changing connectivity, will significantly speed up the development of algorithms that depend on dynamic meshes.

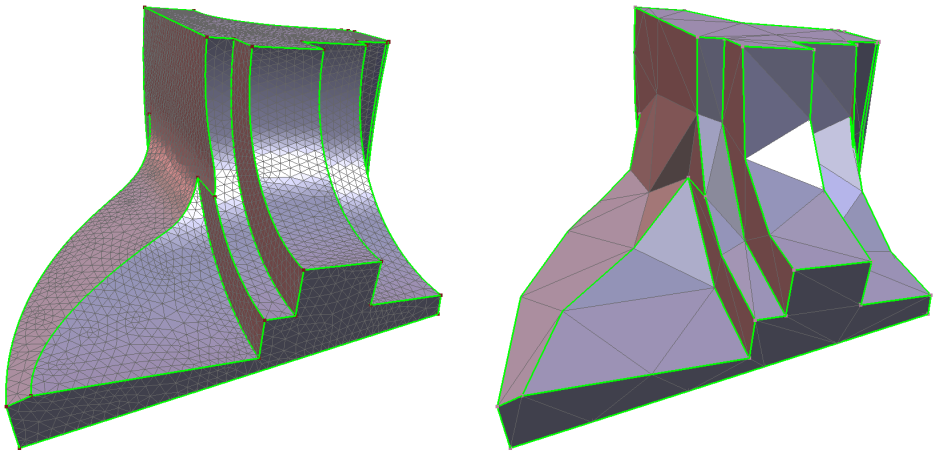

Figure 3: A result achieved with our design-framework. The original fan-disk (left) gets coarsened by successively changing the mesh-connectivity with simple topological operators [9]. In one module of our application, we use our framework to automatically update the selected edges (green) whenever a topological change occurs. Another module takes care of equally distributing vertices across the surface. Both modules work independently of each other and there is no need to expose internal data-structures to the outside. Thus, the modules can easily be reused in another context.

References

1. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001. 1
2. M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. Openmesh - a generic and efficient polygon mesh data structure. In *OpenSGPlus Symposium*, 2002. 1, 6
3. William J. et al. Brown. *Antipatterns*. John Wiley and Sons, 2000. 3
4. Gamma, Helm, Johnson, and Vlissides. *Design Patterns*. Addison-Wesley, 1995. 4
5. L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *CGTA: Computational Geometry: Theory and Applications*, 13, 1999. 1
6. L. Kobbelt, T. Bareuther, and H.-P. Seidel. Multiresolution shape deformations for meshes with dynamic vertex connectivity. In *Computer Graphics Forum (Eurographics 2000)*, volume 19, pages 249–260. 4
7. J. Vorsatz. homepage. pages www.mpi–sb.mpg.de/ vorsatz. 6
8. J. Vorsatz, Ch. Rössl, L. Kobbelt, and H.-P. Seidel. Feature sensitive remeshing. In *Computer Graphics Forum (Eurographics 2001)*, pages 393–401. 4, 6
9. J. Vorsatz, Ch. Rössl, and H.-P. Seidel. Dynamic remeshing and applications. In *Solid Modeling and Applications*, to appear 2003. 1, 6, 7