A Concept for Time-Dependent Processes

K. Polthier
FB 3 Mathematik, Technische Universität Berlin
Straße des 17. Juni 136, 10623 Berlin, Germany

M. Rumpf
Institut für Ang. Mathematik, Albert Ludwigs Universität
Hermann Herder Straße 10, 79104 Freiburg, Germany

Abstract

We develop a new concept to extend a static interactive visualization package to a time-dependent animation environment by reusing as many as possible of the existing static classes and methods. The discussion is based on an object-oriented mathematical programming environment and is applied to parameter-dependent structures, time-dependent adaptive geometries and flow computations but most of the ideas apply to other environments in scientific visualization too. We define new classes describing dynamic processes (including e.g. time-dependent adaptive geometries) and specify a protocol mechanism they must understand. This allows the definition of a class *TimeNode* supervising an arbitrary dynamic processes as a time-dependent node in a formerly static data hierarchy. We discuss mechanisms of such time-dependent hierarchies, and additionally the problem of algorithms on time-dependent geometries in a number of examples.

1 Introduction

Time-dependent processes occur in many different areas as for example in the description of natural phenomena and their computer graphical visualization, in computer animation packages, or in mathematics when analyzing parameter-dependent problems. The paper is centered around the question of how to describe such phenomena in a software package and what kind of algorithms operate on such data.

This paper describes a procedure to extend a static graphical programming environment to handle arbitrary time-dependent processes by easily upgrading existing classes and methods working on static data. Although this was our original goal, it turned out that the concept is useful in a more general sense. The restriction of having to start with a static environment and not to start with an animation system right from the beginning turned out to be extremely useful

when formulating the principal ideas. This might simply be true because even in animation packages a user usually is not able to work on an animated object during its animation but he stops the animation for modeling objects - meaning that he operates on a *static* time cut. Surely, this is a simplification and we will later discuss the problem of algorithms working on complete dynamic processes.

We start our paper with a discussion of the term *time* in chapter 2. For example, we generalize the meaning of time to be local to an object, which allows us to view the same dynamic process at two different speeds simultaneously. Surely, this is very useful when analyzing dynamic process. A major point is the definition of a new class *TimeNode* in chapter 3 describing a time-dependent subclass of the class *Node* in the static case. Using such nodes we extend a static hierarchy to a time-dependent hierarchy. A static node has a pointer to a static geometry. A instance of class *TimeNode* has (beside other instance variables) additionally a pointer to a dynamic process and reuses the pointer to the static geometry to store the current time cut of the dynamic process. If time does not change the hierarchy behaves as in the static case and all existing tools of the existing static environment may still be used.

Before discussing in chapter 5 time-control in a time-dependent hierarchy with multiple nodes of class *TimeNode*, we discuss in chapter 4 a number of example classes describing dynamic processes. For example, we extend the classical key-frame technique to allow discretization changes of animated triangulated geometries. Finally, we discuss in chapter 6 a number of mathematical algorithms working on dynamic processes. Such algorithms are not pure extensions of static methods, but may have much more functionality. Consider for instance clipping at a sphere where the algorithm is static but the sphere may have an animated diameter, or algorithms integrating over dynamic processes.

Summarizing, our major points are:

- Definition of a new class *TimeNode* as a subclass of the static class *Node* which points to a static geometry. The static geometry is now the static time cut of an attached dynamic process and updated via a simple protocol mechanism.
- Definition of a time-dependent hierarchy which extends a static hierarchy in a natural way based on instances of class *TimeNode* as hierarchy nodes.
- Definition of new classes describing dynamic processes, as for example the class *TimeStep* discussed in chapter 4 allowing time-dependent adaptive geometries.
- Discussion of algorithms for dynamic processes.

Our concept differs strongly from other concepts available in scientific visualization software [8].

The time-concept described in this paper has been implemented in the graphical programming environment GRAPE [2] developed at the Sonderforschungsbereich 256 for "Nonlinear Partial Differential Equations" at the University of

Bonn. GRAPE is an object-oriented, interactive visualization package with a library of classes and methods supporting besides others mainly solutions of problems from continuums mechanics [4] and differential geometry [1]. It is device independent and runs on a variety of workstations.

The first author wants to thank Charlie Gunn for fruitful comments on the paper.

Before we start with more detailed explanations let us clarify the term *time.*

2 What is *Time*?

This question may seem a little bit curious to some people, especially to scientists studying physical dynamic phenomena. Such problems carry in general an inherent time, just the physical time of nature. Also a video animator usually has this notion of *time* in his mind when designing a video. This is manifested for example in the unique time slider in many software animation packages.

In the above example the term *time* is always used as the physical time, or at most to some extent modified, which nature gives us. We extend this characterization of time and define: *time* is an arbitrary but emphasized parameter of the object we study.

As the first consequence of the extended meaning of time let us assume our object depends on a set of parameters. We can emphasize one of those and consider the object as a dynamic process depending on this parameter, now called the time-parameter. During interaction we may choose a different parameter or the other parameters may for instance also depend on the time-parameter.

A second characteristic of our time-parameter is *locality.* In the extreme for example, every object belonging to a hierarchy may have an own time-parameter. This may seem unnecessary when creating a video which relies on a global time. But when analyzing dynamic processes it is important to see for example the same process running at different speeds or times simultaneously, or fix the time of one of the processes while keeping the others running.

Our definition of a time parameter extends the term *time* from the physical time to include any real parameters. This arbitrariness for instance allows in multi-parameter systems changing between parameters to analyze different aspects of the problem with the same tools. Surely, the new freedom in the definition of *time* shall not make things more complicated and possibly only introduce a new parameter. When studying flow problems one would still choose the physical time as *time*. But even here one might want to see two copies of the flow evolving simultaneously with different times, i.e. speeds.

Summarizing, we use *time* as follows:

- *time* is a word for an arbitrary emphasized parameter of a parameter-dependent object
- *time* is a parameter belonging locally to an animated object.

3 The Class *TimeNode* Supervising Dynamic Processes

In this chapter we discuss the concept of the class *TimeNode* to manage and supervise dynamic processes. Instances of class *TimeNode* will be nodes of the time-dependent hierarchy in chapter 5. Since our time-concept shall work together with existing software tools i.e. especially tools for handling static geometries, one of our guiding threads was to integrate the time-concept naturally with existing static tools. This includes continued use of our well-proved static classes for discretized geometries and tools operating on them as well as continued use of our data hierarchy.

An additional important advantage of building upon a static system was already mentioned in the introduction: a user working with a time-dependent system usually stops the animation when he tries to manipulate it. That means, he naturally works on a time-cut rather than on the animated system. Since the time-cut is a static object with respect to the current animation, static objects are still important.

Before we start extending a static hierarchy node to a *TimeNode* let us at first review our static object-oriented environment. All geometric objects are organized in a tree and each node may store a geometry object. One node of the tree is emphasized as the *current object* and methods are usually sent to this node. The node will then pass the message to his children and so on. In particular the principle applies to the method *display*, therefore only the subtree having *current object* as root node is usually visible. The user may choose any hierarchy node as *current object*.

The new class *TimeNode* is a subclass of the class *Node* with additional instance variables *dynamicProcess*, *localTime*, *currentTime*, and *syncFlag* as shown in a following pseudo code and in figure 1. The instance variable *dynamicProcess* points to an arbitrary dynamic process which we consider as a black box at first. The detailed description of such processes as for example an animated real variable, a deforming surface or a flow in a volume will be discussed in chapter 4. The major feature of such a black box will be to generate a static time cut on request via the *getObject*-protocol mechanism. In this chapter the instance variable *currentTime* shall be a global time. Later in chapter 5 we generalize its interpretation. *localTime* is used to store the value of *currentTime* each time a new time cut is generated. The instance variable *syncFlag* is a switch to prohibit updating of the time cut *staticObject*, we refer to chapter 5.

The instance variable *staticObject* inherited from the superclass *Node* will now be the current time cut of the attached dynamic process. Such a time cut is a static geometry in our former sense and can be handled by our existing system. The only difference to the *staticObject* of a static node is that here it is updated behind the scenes if *currentTime* changes. If *currentTime* is fixed (at the moment the global time) the *TimeNode* acts as a static object since all display methods as for example *display* or *clip* being sent to the *TimeNode* will be passed on to its instance variable *staticObject* by the same forwarding mechanism

as used by static nodes. On the other side, if the *currentTime* changes, an update mechanism assures that *staticObject* is updated before receiving any method.

The pointers to *dynamicProcess* and *staticObject* are of arbitrary type, therefore the *TimeNode* needs to know nothing about the class of the dynamic process or the returned time-cuts.

The following pseudo code summarizes the class *TimeNode*. The type *instance* defines a variable of arbitrary type. The method *forward* is called if a node does not understand a received method, it allows a node to forward methods to *staticObject*, like e.g. *display* or *clip*. Hierarchy instance variables are hidden.

```
class TimeNode :  Node { /* TimeNode is subclass of Node */
    instance staticObject; /* inherited from superclass */
    instance dynamicProcess;
    float currentTime, localTime;
    int syncFlag;

    forward(Method method, Argument arg)
    { /* 'method' is arbitrary */
        if (localTime != currentTime) {
            /* generate a new time cut */
            staticObject = dynamicProcess:getObject(currentTime);
            localTime = currentTime;
        }
        staticObject:method(arg); /* invoke method of staticObject */
    }

    setTime(float time){/* invoked e.g. when changing a time slider */
        if (syncFlag == UPDATE_ON)
            currentTime = time;
    }

    setSyncFlag(int newFlag) {syncFlag == newFlag;}
}
```

An instance of class *TimeNode* may be used as a dynamic node in a static hierarchy in the same way as any static node. In chapter 5 we discuss the update mechanism in a hierarchy in more detail as well as the use of different current times.

The update mechanism of the time-cut *staticObject* is as simple as possible. It is realized by sending the method

getObject(currentTime)

to the dynamic process and getting returned a new time-cut. Sending *getObject* and updating the time-cut is done by the *TimeNode* itself, immediately when

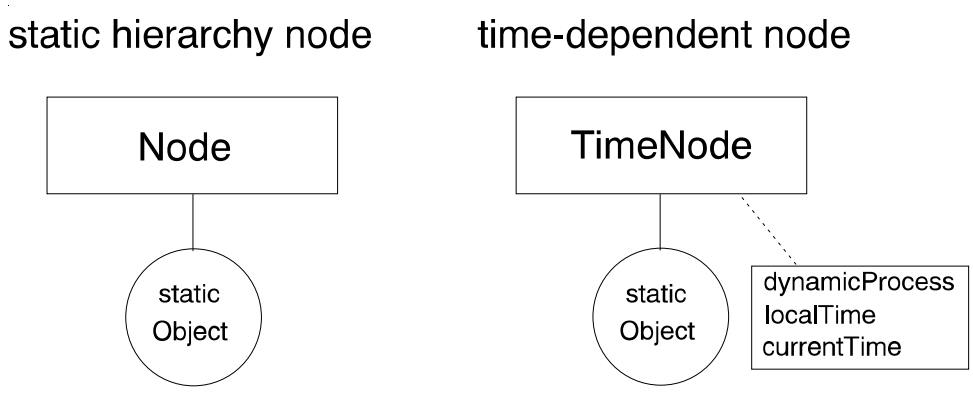

Figure 1: The class *TimeNode*

receiving a method and *currentTime* of the *TimeNode* is different from the time-cut's time.

This simple mechanism allows to consider dynamic processes as a black box. A dynamic process is not a class definition, we only require that on each dynamic process exists the method *getObject(time)* returning a time-cut. That is, the dynamic process observes the *getObject()*-protocol. We will see in the following examples how powerful this simple mechanism is and which wide class of dynamic processes can be handled with it.

It is obvious how to extend this protocol to include for example keyframing where the user modifies the *staticObject* at a specific time and forces the dynamic process to store it as a new key frame.

Remark The instance variable *localTime* is the time at which the actual time-cut has been generated. One might want to store that time inside the time-cut object, but this would require modifying each of the existing static geometry classes (for example putting a wrapper around the static classes), which we prefer to avoid.

4 Examples of Dynamic Processes

In the previous chapter we discussed the new class *TimeNode* supervising a dynamic process. The dynamic process was considered as a black box and every object understanding the protocol mechanism

getObject(time)

may serve as a dynamic process in a *TimeNode*. Let us now consider a few characteristic examples of dynamic processes in more detail. It will turn out

that their nature may be quite different.

4.1 The Class *TimeStep* Extending Keyframe Techniques

Let us discuss an example where the dynamic process of a *TimeNode* consists of a sequence of discrete geometries as in the keyframe technique and later extend it to allow changing of the discretization. As an example in mind we assume a triangulated surface in $\mathbf{R}^3$ deforming over the time. In our description we call such a dynamic process *TimeStep*. The purpose of this example is at first to demonstrate classical keyframe technique in our approach and at second extend it to include adaptive techniques of varying discretization in time direction.

The dynamic process *TimeStep* consists of a doubly-linked list and each node having also an instance variable *time* and a pointer to an arbitrary object. The value of *time* is obviously the time of the key-object, which is in our example the surface at this specific time. For the moment we assume all surfaces to have the same underlying discretization. Linear interpolation between two key surfaces is therefore done by simply interpolating corresponding vertices in $\mathbf{R}^3$. With this information we can describe the action of the method *getObject (time)* being sent to the dynamic process *TimeStep*: the method returns a key-object if *time* is identical to the corresponding key-time. Otherwise *time* is between two key-times and we apply the mentioned interpolation algorithm to generate a time-cut. Linear interpolation may be extended to higher order techniques, but would only complicate the current discussion.

The concept of *TimeStep* allows to include classical keyframe techniques in our time-concept in a natural way. As before, we only require the method *getObject(time)* to exist on *TimeStep*, i.e. the objects on which each *TimeStep* key points need to allow some kind of interpolation mechanism.

Let us discuss the keyframe type approach again, and this time assume a geometry with adaptive discretization in time. Classical keyframe techniques do not apply since interpolation now has to deal with different discretizations. Since such situations occur quite naturally in time-dependent adaptive techniques we have expanded the concept of *TimeSteps*: each key of a *TimeStep* list has not only one pointer to an object, but it has two pointers: a *preObject* and a *postObject*. If the discretization needs to change at a time we put at this time a key and store the same object twice: the object with the initial discretization is assigned to *preObject* of the key and the object with the modified discretization is assigned to *postObject*. It is essential that the geometry of the object does not change at this key, only its internal discretization. And it is also essential that the discretization of the *postObject* of a key is identical to the discretization of the *preObject* of the next key. Compare figure 2.

```
class TimeStep {
    float time;
    TimeStep preStep, postStep;
```

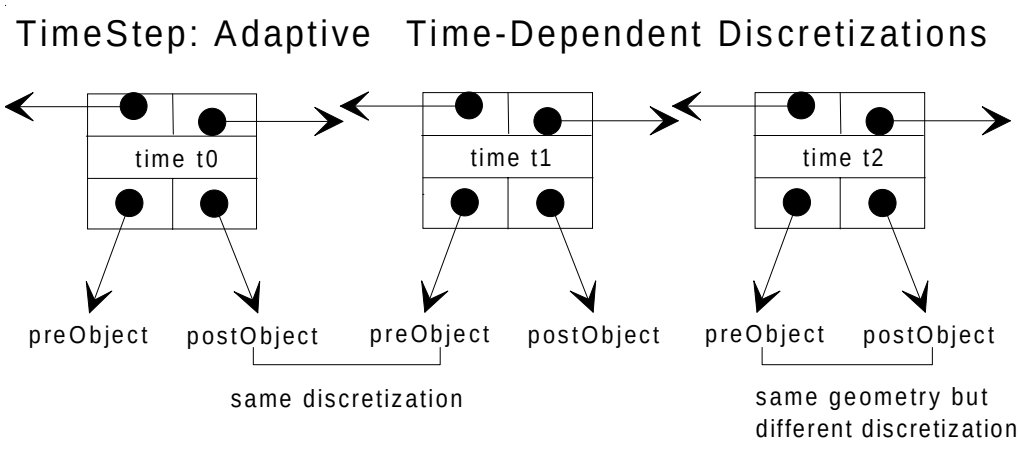

Figure 2: The Class *TimeStep*

```
    instance preObject, postObject;

    getObject(float time){
        step = self; /* first link of list */
        /* find two keys enclosing 'time' */
        while (step.postStep.time < time)
            step = step.postStep;
        /* interpolate between two keys (sloppy notation) */
        object = interpolate(step.postObject,
                step.postStep.preObject, time)
        return object;
    }
}
```

This technique allows us to change the discretization of an object during an animation. Every change of discretization happens exactly at a keyframe. Between two keyframes we interpolate by linear interpolation between the two identical discretizations of the *postObject* of a key-step and the *preObject* of the next step. Therefore interpolation between two successive keys works as in the non adaptive case. When during the animation we pass over a key-step where the discretization changes, we also have no problems. Since the geometry of the *preObject* and *postObject* at a key is identical, the user will not recognize the change of the internal discretization (assuming he is watching a shaded model). Compare figure 3.

In the case the discretization before and after a keyframe are identical, we need to store only one object at the key. Both instance variables *preObject* and

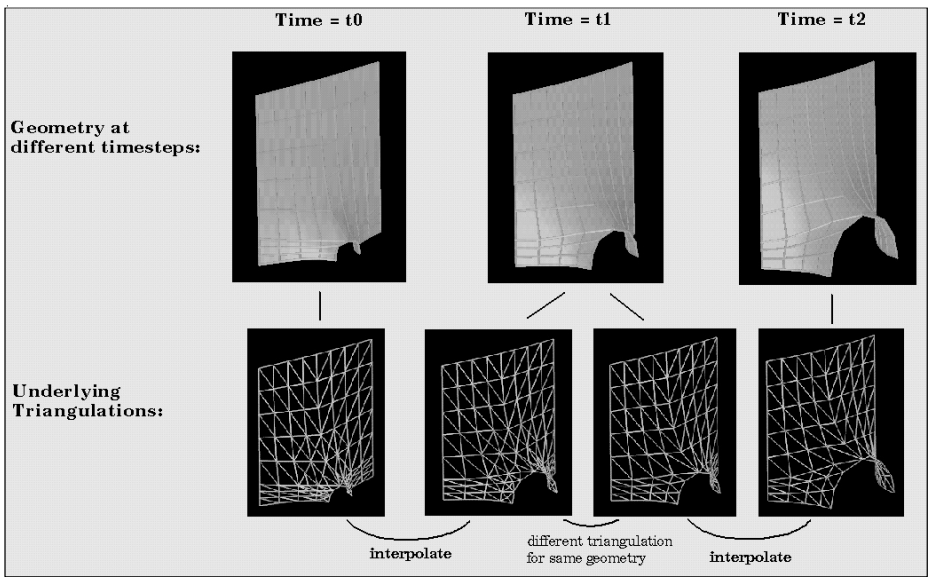

Figure 3: Interpolation Mechanism Between Adaptive Triangulations

postObject may point to the same object.

4.2 Time-dependent Variables and Functions

Very simple time-dependent geometric objects are for example animated points. But we will discuss an even simpler example: a float variable. As mentioned before, the dynamic process can be arbitrary, so we can animate continuous variables as for example a real number. Such a variable may be in the static case a constant with a fixed value, or it may have different values. In the latter case the software environment may support sliders to control and modify the variable during run-time. We will attach to the variable a slider and call this variable/slider pair the static situation in the following, despite the variability of the variable's value.

Let us create an instance of class *TimeNode* to handle the situation. We need a dynamic process controlling our animated variable: a time-dependent variable is simply a real-valued function depending on the time. As a dynamic process we therefore take a function curve (e.g. a spline) depending on the time. The required method *getObject(time)* sent to the function curve simply returns the function value at the requested argument *time*. The *TimeNode* has the function curve as dynamic process and the variable is the time-cut stored in the instance variable *staticObject*. When time changes, the variable/slider is updated by evaluating the function curve, i.e. sending of *getObject(time)*.

The user e.g. may still use only the variable, i.e. *staticObject,* in his program code and has nothing to do with the update mechanism. He uses the variable as an ordinary variable available in his programming language. Except that there is an intelligent control mechanism working in the background to allow control of the variable over the time. The function curve may for instance be a spline curve and interactively modifiable by the user. When initially defining such a time-dependent variable, the underlying function curve might be set to a variable's initial value on default.

```
/* static case */
float variable;

/* dynamic case */
TimeNode *timeNode;
timeNode.staticObject = variable;
timeNode.dynamicProcess = spline;

/* class Spline must understand the method 'getObject' */
spline:getObject(float time){
    /* call evaluation routine */
    return evaluateSplineAtTime(time);
}
```

The above description sounds as if the slider is superfluous in the time-dependent case, since the function curve directly updates the variable. But assume, time is fixed for a short moment. Then we might still want to vary the variable at this specific fixed time-cut to found an optimal value. Having found such we might edit the function curve and incorporate the value permanently, otherwise changing the time of the system would reset the variable and the slider via the update mechanism according to the function curve. Therefore, the function curve controls the time-dependency of the variable and the slider may still be used to control the variable temporary at an arbitrary time-cut.

The example of a time-dependent variable may be easily extended to higher dimensions. Moving points in $\mathbf{R}^3$ may be controlled by three function curves, one for each coordinate. Or one may even enlarge the domain of the function curve to allow parameterized, time-dependent curves or surfaces:

$$f : T \times \Omega \rightarrow \mathbf{R}^3,$$

where T is the time interval and $\Omega \subset \mathbf{R}$ for curves or $\Omega \subset \mathbf{R}^2$ for surfaces. Obviously $\Omega = \Omega(t)$ may also depend on the time.

4.3 Further Examples

Dynamic processes may be quite different. We have already seen this in the two proceeding examples. The action of the method *getObject(time)* was once the

evaluation of one or more functions at the requested time, and in the other example *TimeStep* an interpolation mechanism between two discretized geometries. As mentioned there, the interpolation mechanism may become quite complex when geometries carry additional information as for instance finite-element descriptions or general functions as shown in figure 3. Another possibility of a dynamic process would be a request to a data base. But the dynamic process may even be a hierarchy of dynamic processes, and sending *getObject(time)* would return a time-cut of the whole hierarchy.

5 Time-Dependent Hierarchies

Our concept of *TimeNode*s suggests an obvious definition of a time-dependent data hierarchy:

- a *time-dependent hierarchy* is a static hierarchy where *TimeNodes* are allowed as nodes in the tree.

Using this definition a time-dependent hierarchy is perfectly conforming with standard static hierarchies. We only allow an additional class of objects, namely *TimeNodes*, to occur as hierarchy nodes.

How does one work with such a hierarchy? As mentioned in chapter 3 in the static case the user chooses an arbitrary hierarchy node as the current node, i.e. as the current root node, to work on this branch of the hierarchy. If this is now a *TimeNode* its *currentTime* will be the global time for the currently chosen branch. A global time slider will be attached to the current node and may be used to modify its *currentTime*. Modifying the slider will send an update method to the current node, which in turn will send further updates to its children updating the whole branch of the hierarchy, i.e. the *currentTime* off all children are set to the *currentTime* of the current root node.

The handling of methods being sent to a hierarchy node is identical to the static case: a node receiving a method will apply it to its *staticObject* and then send it to its children. If a node happens to be of class *TimeNode*, it will at first update its *staticObject* if necessary and then apply the method to the *staticObject* (methods working on dynamic processes will be discussed in the next chapter). Therefore, the only different action compared to static hierarchies is the additional update performed by *TimeNodes*, regardless of the type of method being send to it.

5.1 Using *currentTime* of TimeNodes

The major use of the instance variable *localTime* is to store the time at which the current time-cut has been taken. As an additional use the *currentTime* of a node may be set to a fixed specific value to display the dynamic process always at the same time while other nodes in the hierarchy are changing in time. To accomplish this, choose the desired node as current object, set its *currentTime*

using the time-slider, set its *syncFlag* to *UPDATE_OFF* and change back to your previous current object.

There exist a number of methods to control *currentTime* of each hierarchy node. For instance *setTime(time)* will set the *currentTime* instance variable of all nodes to the value *time*. Such an action might be forbidden on some nodes if they shall always show a fixed time-cut: this is done by switching the *syncFlag* of such a node, therefore not allowing to change his *currentTime* variable.

5.2 Animation Control Panel

An easy implementation of an animation control panel with play, playback etc. functionality works as follows: using the control panel buttons and its time-slider the user might interactively change the slider's time or let the slider automatically change by pressing a *play* button. This releases the sending of multiple method pairs *setTime(time)* and *display* to the current active hierarchy node. As a result an animated video will be shown on the screen.

5.3 Viewing Different Time-Cuts Simultaneously

An additional feature is viewing different time-cuts of the same process simultaneously: just create a number of *TimeNodes* under the same parent and instancing the same dynamic process. Then modify the instance variables *currentTime* of each *TimeNode* to the requested different times. Notice that the dynamic process itself exists only once. This is not only useful for large data sets but also allows modifications of the dynamic process to take immediate effect on all different time-cuts. The same procedure might also be chosen if for instance one would like to always look at the same dynamic process but using different display methods (e.g. clips, iso-lines, grid-model,...). In such a case we leave the different *TimeNodes* synchronized but change their local display-method to the requested tasks. Compare figure 4 where different time-cuts are shown simultaneously and additionally the trace of some points are drawn.

6 Algorithms on Dynamic Processes

In this paragraph we discuss some algorithms on dynamic processes. The main reason of this paragraph is not to explain the algorithms in detail, but to discuss their working principles in connection with the idea of *TimeNode*. Therefore we take a few example algorithms and see how they are applied to dynamic processes, how they work on *TimeNode* and behave in a hierarchical tree.

The algorithms can be divided into several groups:

- working on the entire dynamic process, e.g. reflecting a time-dependent geometry at a time-dependent symmetry plane
- integrative algorithms on dynamic processes, e.g. computing particle traces of a time dependent vector field on a surface or in a volume

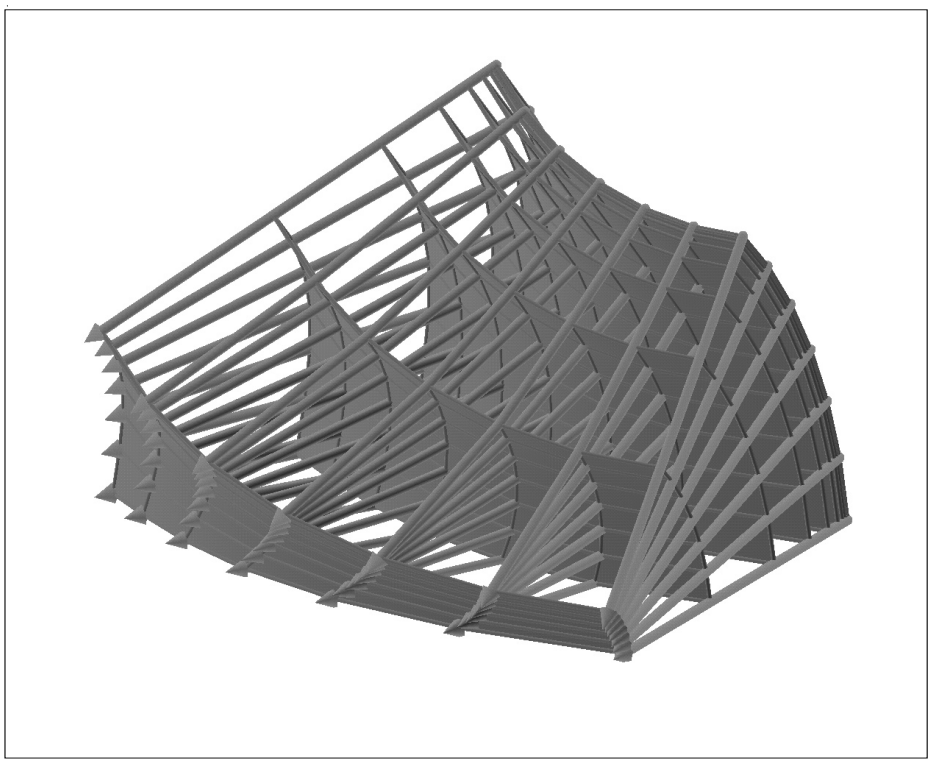

Figure 4: Simultaneous View of Several Time-Cuts

- algorithms generating dynamic processes, e.g. computing the evolution of a surface under some force including adaptive refining in time direction
- extraction of time-cuts by interpolation techniques(see discussion in the previous chapter)

6.1 Dynamic Processes are Single New Objects

The first class of algorithms is best suited to underline an important difference between keyframe techniques and our concept of *TimeNode* and dynamic processes. In our approach we consider a *TimeNode* as a unity, a single new object to which we can apply geometric and dynamic operations to obtain other *TimeNodes*. Also the dynamic process appears as a new object which it is best indicated by its black box behavior. This is in strong contrast to an ordinary collection of keyframes. To make this claim more concrete consider the following example.

Let us see how this point of view conceptually simplifies operations working on these objects.

Assume a time-dependent surface, which can be extended to a larger surface by reflection at a part of their boundary lying on a symmetry plane. The plane is therefore determined by the boundary arc. In the keyframe technique one needs to take an arbitrary key-surface, specify the symmetry plane and apply the reflection to all other keys. This operation will extend each key-surface to a larger surface and one ends up with a modified keyframe sequence. This operation is very technical and determined by the internal representation of the data.

In the time-concept we apply now the same operation: the user might see an arbitrary time-cut on the screen and decide to execute a reflection operation. For the user the actual time-cut represents the whole dynamic object, so if he operates on the time-cut, then internally the *TimeNode* itself should control and automatically invoke the necessary actions on the dynamic process. Instead of working with the internal representation of the dynamic process, which might be very complicated, we only act on an arbitrary time-cut as a representative.

This provides us not only with a very natural interface (operate on the data you see on the screen and not on technical details, [7]) but also gives us a necessary degree of abstraction in the passing of information from an action on the time-cut to the dynamic process: in the above example of reflecting a time-dependent surface one has to specify in an abstract way the symmetry plane on the time-cut such that it has a meaning for all objects of the dynamic process. Picking just three points on the time-cut and passing their positions is not unique since the symmetry plane might vary during the dynamic process. It is also not sufficient to pass to the dynamic process the indices of the picked points since the dynamic process may be adaptively refined during time. A solution would be to pass a unique identifier for a boundary component.

So, the user will pick a boundary component of an arbitrary time-cut which is then passed to the dynamic process together with the reflect message. The dynamic process has now the necessary information how to reflect itself, for example to reflect all members of a time-step. At the end we have duplicated the whole *TimeNode*. At no time the user has to be aware of the underlying dynamic process in background. He only operates on the currently visible time-cut of a *TimeNode*.

The further difference to an ordinary keyframe approach is also, that we generate an additional new *TimeNode* in contrast to extending each of the keyframes to a bigger one and preserving just one sequence. At first sight the difference may look not too big, but at the end the different point of view leads to many simplifications.

Our point of view as described in the above example has also big limitations. Surely many operations can not be decided on a single time-cut as a representative of the whole *TimeNode*. For example, if the picked symmetry plane vanishes somewhere in the sequence. This are problems for future research.

6.2 Integrative Algorithms

The type of algorithms in this and the following section are not new, but they demonstrate the working principles of our new objects. Integrative algorithms on *TimeNodes* are for example the integration of a time-dependent vector-field on a surface or in a volume. As an input we take for example a finite element description of a vector field in a volume as objects in a *TimeStep*, which varies over the time. This is an ordinary differential equation [5][3]. The user may want to insert test particles in such a system and study their traces. In the time-concept this is realized by picking an initial point or choosing an initial test surface on an arbitrary time-cut. This initial condition is then passed to the dynamic process which performs the necessary computations and returns the trace of the test set. In this case the trace is also a dynamic process and will be inserted somewhere in the data hierarchy as a *TimeNode*. Its time-cuts are just a test set at a position corresponding to the current time.

Integrative algorithms take a *TimeNode* and generate a new one usually of different type. In the example we took a time-dependent vector field and generated an animated particle. See figure 5 for integration of vector fields on static geometries.

6.3 Generating Dynamic Processes

During the previous discussion we have already seen a few algorithms generating new *TimeNodes* and new dynamic processes. A further important example is to start with a static object, and apply a sequence of operations, each generating a new static object. Arranging this sequence as objects in a time-step creates a dynamic process which can be used to expand the original static object to a dynamic process.

More precise, take a surface in R^3 as a static object and for example let it flow in normal direction with a certain speed. This is a very common procedure in mathematics to obtain solutions of differential equations [6]: constant speed leads to solutions of the eikonal equation, speed equal to the mean curvature leads to minimal surfaces, one could study wave propagation of let the surface be driven by arbitrary energy gradients. Collecting the surface at different stages of the evolution in a time-step, the user is free to interactively refine the surface discretization in evolution direction, according to areas with high or low curvature for example or criteria derived from the behavior of the underlying differential equation. Compare figure 5.

This way the user is able to store the evolution of the surface in a *TimeNode* with a *TimeStep* attached. The interpolation techniques inherent in the object *TimeStep* allow the surface not only to change discretization. It also includes change of the topological structure, e.g. splitting a long cylindrical bone into two spheres by letting the small waist shrink.

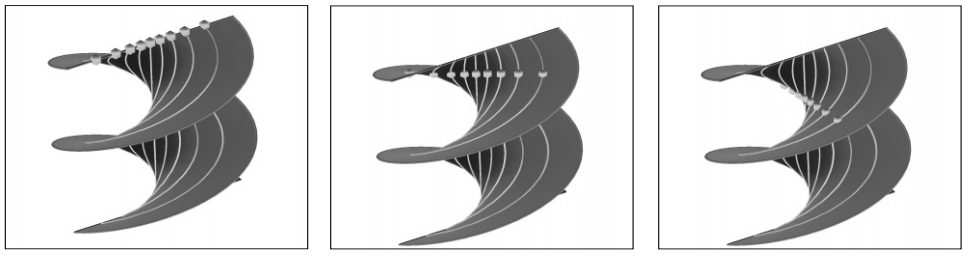

Figure 5: Flow along Vectorfield on Surface

7 Conclusion

Summarizing, we discussed a new concept to extend a graphical environment for static mathematical problems to an animation system. We introduced the new class *TimeNode* as an extension of a static hierarchy node without a need to change the existing classes and methods, and we discussed descriptions of dynamic processes and dynamic algorithms.

We have assumed in our description that the geometry data is organized in a tree. A more general discussion should be made allowing dynamic data to be organized in different ways although this does not seem to be a principal problem. The concept using time-cuts as representatives of a whole dynamic process and interact only with a single time-cut as in the reflection example surely has its limitations as already remarked in chapter 6.1. This is a large area with open problems for future research since many static algorithms wait for being made dynamic. For example consider time-dependent clipping, boolean operations or picking. Also our concept of *TimeStep* allowing keyframe techniques with adaptive triangulation should be extended, for example to higher order degree of interpolation - both in time direction and in the approximation of the static geometry steps.

References

[1] A. Arnez, B. Oberknapp, K. Polthier, M. Steffens, C. Teitzel, Time-dependent Curves and Surfaces in Differential Geometry, in preparation

[2] GRAPE Manuals, Vol. 4.1, SFB256 University of Bonn, Aug. 94

[3] M. Geiben, M. Rumpf, Moving and Tracing in Time-dependent Vector Fields on Adaptive Meshes, Report No. 12, SFB256 Bonn

[4] M. Geiben, M. Rumpf, Visualization of Finite Elements and Tools for Numerical Analysis, Eurographics Workshop on Scientific Visualization, Delft 1991

[5] B. Oberknapp, K. Polthier, Vector Fields on Surfaces, in preparation

[6] U. Pinkall, K. Polthier, Computing Minimal Surfaces and Their Conjugates, J. Experim. Math., Vol. 2 (1), 1993

[7] K. Polthier, M. Rumpf, WYSIWYO in Differential Geometry, Report No. 10, SFB256 Bonn

[8] C. Upson, et. al., The Application Visualization System: A Computational Environment for Scientific Visualization, IEEE Computer Graphics and Appl. July 1989