An Abstract Model Of An Interactive Simulation System

Alexandru Telea

5th June 1998

Abstract

There exists a wide range of simulation systems which approach in different manners and solve only partially the problems of simulation specification, interrogation, and steering. As simulations grow more complex, one may need a formal, generic model in which the large variety of simulated problems can be easily described, and software systems able to implement the model. However, existing simulation software are often based on particular simulation representations, and hence are able to directly model and represent in software only specific problems, making thus the combination between various types of simulations form different areas of expertise a difficult task. The lack of control of many simulators' interactivity is another major problems the user experiences. We approach these problems by firstly devising a general abstract simulation model in which a very large class of problems can be represented. Next, we derive two formal simulator models from the simulation model and we use them to discover the causes of the interactivity problem frequently experienced by existing software. The formal framework we build lets us find easily the possible solutions to this problem and see the trade-offs they involve. We conclude by presenting some mechanisms which can implement these trade-offs in the context of the presented formal simulator models.

1 Introduction

With the advent of high performance computers, increasingly complex physically based processes, computer animations or virtual reality worlds can be represented, investigated and modified with the help of software tools we call simulation systems. As the complexity of the theoretical representation (the model) used to describe the simulated universe grows, the scientist needs more complex software that is able to cope with its modelling (description in terms of software concepts), interrogation (reading results or online visualization of produced data) and ultimately steering (changing the simulation parameters and monitoring the simulation evolution interactively).

Interactivity has become an important feature of simulation systems. Whether the simulation describes a physically based process or is a computer animation or virtual reality world model, the ability of the user to interact with the universe he simulates in terms of its modelling, interrogation and steering has become a critical requirement of the simulation tool. Interaction with a simulation comes in several flavours, as different kinds of simulation and visualization systems (or frameworks) have been conceived, approaching each of the modelling, interrogation and steering tasks on a different level.

The simplest simulation frameworks come as libraries dedicated to a limited range of operations, such as geometrical modelling [1], linear algebra [2] or visualization [3].

Object-oriented libraries such as Diffpack [5] or LAPACK++ [6] provide a more abstract application programming interface (API) by which the user can represent simulation concepts as *objects*.

Specification of complex simulations can be however only partially done by such libraries. Besides modelling the simulation's entities by objects, the programmer should represent the *relationships* between these entities. For example, a numerical simulation can have many parameters depending on each other in complex ways. Such dependencies impose *constraints* on the time evolution of the parameters they involve. Since it is complicated and error-prone for the user to 'steer' such a simulation by explicitly changing all its parameters and maintaining the constraints, a *constraint management* mechanism is provided to specify and automatically enforce constraint relationships. A good example of a simulation library offering constraint management is OpenInventor [4].

Adding *interactivity* to simulation systems takes yet another step in modelling reality. While some systems allow only the monitoring of time dependent data, interactive steering systems practically integrate numerical computations and visualization in one tool, which can monitor but also interactively steer a running simulation. Haber and McNabb [7] and Marshall et al. [8] give a good survey of interactive simulation systems.

Dataflow systems like AVS [9], Khoros, Iris Explorer or apE offer an interactive manner to simulation specification by means of a graphical user interface (GUI) which allows connecting various computational modules in a directed graph, called a flow network. While the simulation runs, data flows from its source through modules which perform various operations on it up to the modules which perform the effective visualization.

Although powerful, existent simulation frameworks are often designed for a specific problem domain, for which modelling concepts and tools are offered. The difficulty appears when a given framework has to be used to house a wide range of simulations, of which many might not directly map to the concepts provided by the system's designers. For example, dataflow systems are ideal for a user-driven investigation of scientific data, while animation systems perform their best when one needs to simulate the evolution in time of a universe, often described by means of a scripting language. It is however difficult to combine the two types of simulation in a single environment, since they are based on two different simulation engine policies (the dataflow systems work on a demand or event driven model, while many animation systems simply execute an imperative description of the temporal evolution of the modelled universe). Moreover, it often happens that simulation parameters (e.g. time) are treated in fundamentally different ways by different software systems, hence the conceptual difficulty which rises when one wants to map a simulation specification from one system to another.

We think the above problem can be solved in two steps : first, an abstract model of a generic simulation should be designed, offering all concepts necessary for describing virtually any simulation one could think of. Secondly, the concrete specification of a software system implementing the abstract model should be derived from the formal specification of the abstract model.

This paper presents such an approach, firstly describing a general abstract model of a simulation and then showing the structure of a simulation framework able to implement the abstract model, thus able to perform any simulation which can be reduced to the abstract model.

2 The State Model of a Simulation

Our goal being the design of a general purpose simulation system, a conceptual model for a simulation must be first developed. The model we propose is based on the state space description of a physical universe. We start by defining the notion of *state*. A state (or state vector) of a system is the set of all parameters that fully describe the system at a given time instant. These parameters are known as *state variables*.

Examples of such state parameters are all 'physical' parameters of a physical system (e.g. velocities, positions, pressures, voltages and any other physical quantity). Other state parameters which don't model directly physical quantities can be, for example, the viewing parameters of the visualization system used to monitor a simulation or the parameters of the input devices of a computer system(e.g. mouse and keyboard). Two states are said to be different when they differ by the value of at least one state parameter. Whenever a state parameter changes, a *state transitions* occurs and the system goes in a new state.

Since states can change, there is an inherent notion of time, so we can regard the state parameters as functions of time. A simulation whose state parameters are continuous functions of time will be called a continuous simulation (abbreviated CS). The evolution in time of a CS will be given by a sequence of states spaced infinitely close to each other in the time domain.

Time is a very important state parameter and will receive special attention in the following. Informally, the importance of the time state parameter comes from the fact that there are several 'views' on time in an interactive simulation system, like the 'physical' time (time of the simulated process), the 'user time' or 'real time' (time as being perceived by a user that monitors the simulation) and 'computational time' (time needed for the internal computational modules of the simulation).

The evolution in time of the CS's parameters is governed by *constitutive laws*, i.e. mathematical expressions which relate the values of the state parameters. A simulation specification (abbreviated SS) is hence the set of all the state parameters together with all the constitutive laws which interrelate them. Constitutive laws can be denoted functionally as, for example, $a = a(b, c, d)$, where we specify that the state parameter a depends on parameters b, c, d. The dependency can be as simple as an algebraic expression or as complex as a partial differential equation (PDE), as long as given the right hand parameters (b, c, d in the above example) one can compute the left hand parameter. Laws can involve more state parameters on the left hand side (i.e. a state subvector can depend on another state subvector, for example in the case of a linear system of algebraic equations, where the solutions depend on the coefficients and the right hand side vector).

We say that a SS is *complete* if, given an initial state and a time instant t after the initial state, we can compute the values of all the state parameters at time t out of the constitutive laws (for that to happen, some laws should give the direct dependency of some state parameters p on time, and the remaining state parameters should depend on these p). . A SS is *deterministic* if for a given initial state and time instant t after the initial state, there is a unique value for the state at time t (the evolution of the system is totally determined by its initial state).

The problem of simulation can be now expressed as follows: given a deterministic and complete simulation specification, devise a system which computes all the states the simulation passes through after the initial moment. We shall call such a simulation system a *continuous simulation system* (abbreviated CSS).

3 Discrete Systems

The description of the CS is a perfectly general one so far. Any 'universe' which can be characterized by a state vector continuously varying in time can be described in the above terms. However it is impossible to build a computer simulation of a CS (i.e. a software CSS) since there is an infinity of states for a CS and it is impossible to compute and present all these states to the user. The solution is to *discretize* the continuous time, and obtain a *discrete simulation* (abbreviated DS) out of the CS. A discrete simulation's state vector can be seen as a sampling of the evolution of the CS's state vector in time Its state parameters will therefore be discrete sequences of values that are samplings of the corresponding continuous parameters of the CS. The (discrete) evolution in time of a DS will be given by the discrete sequence of state vectors corresponding to the discrete sequence of sampling time instants. An example of a DS is a sampling device which produces a discrete sequence of state parameters out of a continuous quantity. If the sampling of the continuous parameters is accurate enough, then we can say that the DS is equivalent to the CS, so instead of the (impossible) goal of simulating the CS, one should strive for a software system which can simulate the DS.

4 Discrete Simulation Systems

A discrete simulation system (DSS) is a software engine which simulates the evolution in time of a DS. A DSS will practically maintain a state vector, including the time parameter, and the constitutive laws, increment time in a discrete fashion, and use the constitutive laws to compute a sequence of state vectors. Since our simulation specification (SS) is complete and deterministic, it is possible to devise an automatic manner to compute the 'next' state vector (i.e. the state vector at time $t + deltat$, where $deltat$ is the discrete time increment taken by the DSS) out of the 'current' one (i.e. the one at time t).

The model of the DSS that we propose is a collection of software *modules* implementing the constitutive laws (i.e. being able to compute the left hand side of a law, given its right hand side state parameters). Time can be modelled by a time module that loops forever incrementing the simulated time parameter with a discrete amount $Deltat$ that corresponds to the next sampling instant.

The fundamental problem the designer of a DSS faces is how to efficiently automate the process of computation of states out of the current state and the constitutive laws. In many cases, these laws have complex forms, so their application is expensive in terms of computing time. We shall see in the following sections that there are several ways to implement a DSS, which trade accuracy for a faster system response time, or conversely. If the computations performed by the DSS are accurate enough, its evolution in time will generate a discrete sequence of state vectors which will closely parallel (or even equal) the evolution of the DS. Our goal is to design a DSS whose states should, firstly, follow as close as possible the states of the corresponding DS and therefore the one of the CS. The DSS should practically behave as if it directly sampled the continuous system's parameters. Secondly, the proposed DSS should have a short response time, i.e. should be interactive. We shall see how these partially divergent goals can be matched in the following.

5 Time, Sampling and Interactivity in a Discrete Simulation System

Time is an essential parameter of a simulation system, since all other state parameters are regarded as functions of time. There are two main interpretations of time in the context of a simulation system:

- *Physical time:* This is the time state parameter of the CS, DS or DSS. It is analogous to all other physical state parameters, like pressure, velocity or position. The DS produces its discrete time state parameter by sampling the continuous time parameter with a certain sampling step Δt. The same effect is obtained in the DSS by the time module described above, that keeps incrementing the time parameter with the same amount Δt. We can discretize time as accurately as we like in the DSS, by selecting an appropriate Δt.

- *Real time (user time):* This is time as perceived by a human user monitoring a DSS or as measured by the computer's real time clock. If we assume for simplicity that the time module increments the physical time with an amount Δt at each real time ΔT seconds, then there is a very simple mapping between physical time and real time, given by:

$$time_{\text{real}} = \frac{\Delta T}{\Delta t} time_{\text{physical}} \tag{5.1}$$

As previously outlined, state parameters in the DSS are computed by modules. Essentially speaking, such a module will repeatedly execute to evaluate its state parameter (either called by the simulation or looping continuously evaluating that parameter). If the module needs T_c real time seconds to evaluate its parameter, and we assume it continuously loops evaluating that parameter, the system will behave as if that parameter is sampled with a rate of:

$$rate = \frac{\Delta T}{T_{\text{c}}} \frac{1}{\Delta t} \tag{5.2}$$

samples per second of physical time. This essentially says that *the sampling rate of a state parameter in a DSS is bounded by the speed of the module that evaluates it.* We shall call T_c the *latency* of a module.

The notion of interactivity has a large range of meanings when associated with software systems in general and with simulation systems in particular. In the model we are presenting, we shall assume that the user interacts with the system by changing the values of some state parameters (e.g. via the system's input interface — mouse, keyboard, data acquisition tools) and by monitoring other parameters (via an output interface — displays, VR devices, etc). Interactivity can then be defined as the possibility for the user to *change* a certain parameter of the DSS and *observe* the effect that his change has had on the system, which actually means observing that some state parameters of the system have changed as an effect of his actions.

Interactivity can be measured by the amout of real time elapsed between the moment the user changed a state variable and the moment he can observe the changes of the state variables affected by the induced change. In other words, a *real time* interactive system should respond quickly (e.g. in subsecond time) to changes initiated by the user. Assuming that the user controls a state parameter read by a module and observes the state parameter evaluated by that module, the interactivity rate of that module can be defined as the inverse of its latency:

$$rate_{module} = \frac{1}{T_{\mathrm{c}}} \tag{5.3}$$

This gives the number of samples delivered by the module per real time second. In other words, the above equation tells us that the module responds with a delay of T_c real time seconds to a parameter change. It is important to note that the interactivity rate is a characteristic of the software system (DSS) itself and has nothing to do with the physical (simulation) time.

We can use the concept of interactivity rate to globally characterize a simulation system composed of several modules. For such a system where we monitor a parameter which depends on a user controlled parameter or on time via a set of n pipelined modules (i.e. for whose computation n laws have to be applied sequentially), the interactivity rate will be:

$$rate_{parameter} = \frac{1}{\sum_{j=1}^{n} T_{c_i}} \tag{5.4}$$

where T_{c_i} is the latency of the ith module. The above tells us a very important fact, namely that complex systems having large sets of constitutive laws tend to be slow, since these laws would often have to be applied sequentially (like in the extreme case of the pipeline presented above). A second important observation is that latencies of several state parameters in the system are different, depending on the latencies of the modules involved in their computation. For a computer of a given throughput, the only way to make some of its state variables 'behave' more interactively is to evaluate the other variables less often, i.e. to decrease their sampling rate on behalf of increasing the sampling rate of the former ones.

6 Sampling the State Vector

There are two distinct approaches a DSS can take when computing state vectors, related to the way in which the system correlates the sampling of the time state parameter with sampling the other state parameters. These approaches, which ultimately determine the system's interactivity, will be presented in the following together with their advantages and disadvantages.

6.1 Unique Sampling Rate Systems

Unique sampling rate systems feature a single sampling rate for all their state variables (Figure 2). The system's functioning is simple: the time parameter is incremented with the desired amount Δt by the time module and then *all* other state parameters are re-evaluated for the new time instant (all constitutive laws are applied). The re-evaluation of state parameters can proceed in parallel or sequentially, depending on the available computer, but the time incrementation is a synchonization point. After all parameters have been evaluated, a new state vector has been obtained. This could mean, for example, that a new 'picture' of the universe is rendered or a new data set is output. The system proceeds then to the next time increment and so on. This unique sampling rate policy is found in many systems that use an infinite loop to read input, compute the state parameters and output the results, usually looking like the following:

```
forever
{
   read user input
   increment time
   evaluate all state parameters
   output state vector              //this means sample the state
}
```

Figure 1: Unique sampling rate algorithm

The sampling instants do not necessarily have to be equally spaced on the time axis. The only constraint is that the system evaluates all state parameters at the same moments. The main advantages of this approach are that the computed samples are *consistent*, i.e. correct for the sampling moment and that it is very simple to implement it. Another advantage is that we can precisely control the sampling rate of all the parameters, by choosing the time increment, thus avoiding problems like undersampling.

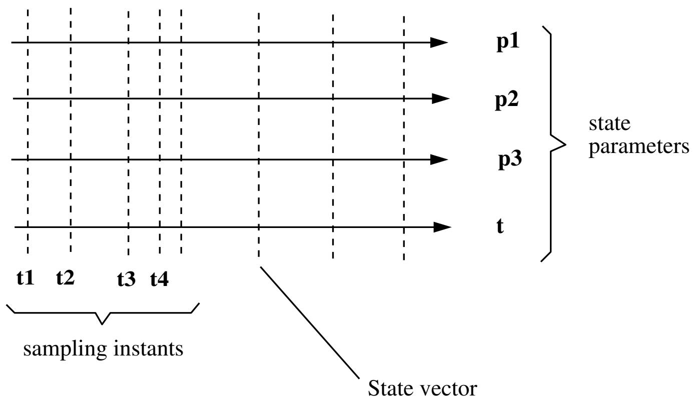

Figure 2: Unique sampling rate

The main disadvantage of the unique sampling rate is that the *latency* of the simulation system will be equal or larger than the greatest latency of all its parameters (the best case occurring when all parametes can be computed independently in parallel), due to the synchronization points imposed. In the case of a sequential system, the overall latency will be even greater, equal to the sum of the latencies of *all* modules called during a state computation step. If, for example, it takes 10 minutes to compute some state parameter, the system will not do any other parameter sampling during this time. In other words, the user will not be able to interact at all with such a system, not even for changing a viewpoint in a 3D camera, for example. Moreover, some parameters might be independent on the one whose computation slows down the whole system, and their reevaluation might be fast, so it would be desirable to evaluate (i.e. sample) them with a different (e.g. higher) rate. Unique sampling rate systems are also inapropriate for simulations in which we must evaluate (i.e. sample) state parameters at different time instants.

6.2 Multiple Sampling Rate Systems

We have seen that single sampling rate systems have a latency equal to the sum of the latencies of the composing modules. In order to decrease the latency of such a system, we have to perform some of the module computations in parallel. However, the overall system latency will still be higher than the largest latency of a state parameter p , due to the imposed synchronization points. To gain more interactivity, we have to give up the synchronization constraint, i.e. to allow modules to evaluate state parameters in an *asynchronous* manner.

Multiple sampling rate systems are essentially a collection of modules that run in parallel, each reading the state vector and evaluating (sampling) its output parameter (or parameters) at its own speed, given by equation (5.2). Such systems will behave like discrete systems (DS) that use different (multiple) sampling rates for their state variables (Figure 3):

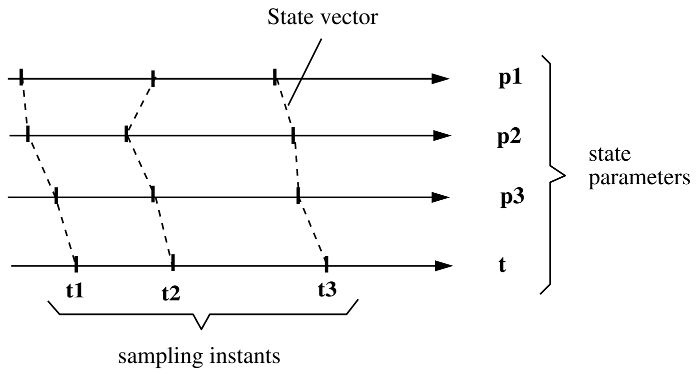

Figure 3: Multiple sampling rate

The main and fundamental advantage of multiple sampling rates resides in the interactivity it offers. Multiple sampling rate systems could, for example, sample the user-controlled state parameters (i.e. read the mouse motion to set a view point) and the output parameters (i.e. render a view) with a high rate limited only by the available hardware speed. In the same time the system can sample other parameters (e.g. some physical quantities computed by a slow PDE solver module) with a much lower rate. The system will ultimately allow the user to interact with the universe with a high speed without being delayed by the slow FEM computations. This philosophy can be extended to the sampling of all state parameters, not just the ones directly controlled by the user.

The fundamental problem raised by the multiple sampling rate model is consistency. In the above example the view point parameter and the PDE output parameters were independent, hence there was no need to correlate their sampling (evaluation) rates in any way (the user could change the viewpoint without to have to wait for the PDE to complete or conversely). In many cases however state parameters depend explicitly on each other. Having uncorellated sampling rates for these parameters might lead the system to an incorrect state (i.e. a state whose parameters are sensibly different from the corresponding CS or DS parameters). The next sections will describe the causes of these inconsistency problems and examine which of them can be avoided and how.

6.3 Requirements of a Discrete Simulation System

The abstract model of a simulation has been presented in terms of a simulation specification (SS) involving states, state parameters, state transitions and constitutive laws. The difference between the physical continuous system (CS), the discrete system (DS) and the discrete simulation system (DSS) have been outlined, introducing the concept of state sampling. Two simulation system types have been presented, the difference being made by the state sampling policy they use. The next step being the presentation of a model for a DSS, we first should specify which are the requirements such a DSS should comply with:

- Correctness: A DSS should produce a *correct* simulation, that is its states should match as close as possible the states of the DS we're simulating. The next sections present a development of the notion of correctness and show how (much) it can be satisfied.
- Interactivity: A DSS should be *interactive*, that is it should deliver the effect of a change in its state caused by the user within a reasonable amount of time. In terms of latencies, we'd like that the latency of a state parameter should not be influenced by the latency of other parameters, if their computation is algorithmically independent. (the user should not have to wait for slow modules if he desires to see effects that require only the operation of fast modules (e.g. changing visualization parameters for a time dependent data set which is delivered by some complex numerical computations module). We have shown that this requirement implies the use of different state parameter sampling rates.
- Open structure: A DSS should be *open*, i.e. it should be possible to easily extend the state vector with new state variables in order to describe a more complex universe. Modification of the functions that express the state transitions should also be easily done. These operations should be done with a minimal overhead, ideally while the system is running. This requirement imposes a modular system structure.
- Transparency: From user perspective, the system should allow an easy specification for simulations in terms of high level 'objects' like physical bodies, fields, laws, visualization tools, etc. Internally all these objects will be mapped on state parameters. From the system developer (programmer) perspective, the DSS should have a modular, layered structure with precise functionality attached to each layer. Changing the implementation of a system part (module) should not affect the rest. This requirement leads naturally to an object-oriented structure described in terms of *module interfaces* and *module implementations*.

The next sections present a proposed structure of a discrete simulation system that attempts to match the above general requirement list.

7 Structure of the Discrete Simulation System

This section describes a structural model for the multiple sampling rates DSS which was introduced in the previous sections. The problems induced by the multiple sampling rate policy will be discussed and some solutions will be proposed.

The central idea of the structure we propose is the *dependency*. We say that a state parameter b directly depends on a state parameter a if b appears in a constitutive law of the form $b = b(..., a, ...)$, i.e. whenever a changes b must change as well in order to maintain a law of the system. We denote graphically this dependency by two nodes containing the state parameters and an arc indicating the sense of the dependency:

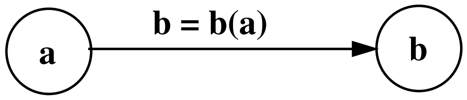

Figure 4: Dependency between state parameters: b depends on a, or b is a function of a

In this manner, the whole DSS can be seen as a complex dependency graph between the state parameters. Examination of this graph allows one to see which is the sequence in which the state parameters should be evaluated (i.e. the sequence in which the modules should execute to enforce the constitutive laws) when a change of one of them appears in the system (note that such a graph can be cyclic. The interpretation of dependency cycles will be treated later). When a state parameter changes, we can see that the change 'propagates' through the graph and all nodes encountered during this traversal re-evaluate (re-sample) their state parameters.

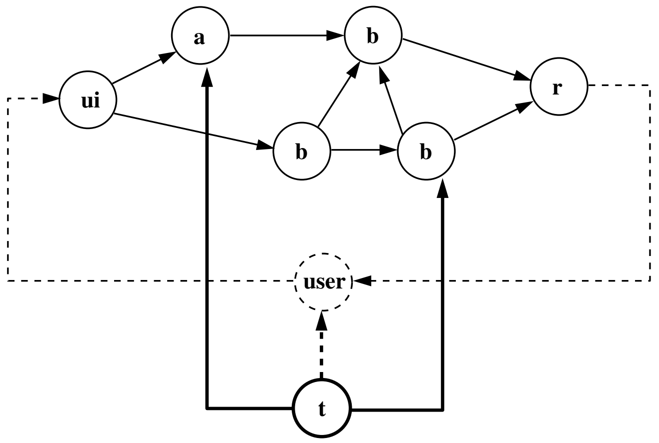

Figure 5: Dependency graph

Figure 5 depicts a dependency graph. Note the user interface state parameter *ui* that models all system's parameters directly controlled by the human user and the output (renderer) module *r* that models all system's parameters directly observable by the user. An interesting feature of the above model is that it can be generalized to include the human user as a node in the dependency graph. Indeed, the user is a node having the input connected to the rendering node (he takes decisions that depend on what he sees on the screen) and output connected to the user interface node (his decisions are passed to the mouse and keyboard) and has a (very) complex transfer function. The time node *t* has a particular status, being the only node with no input, since in the universe model we considered the time is the single state parameter which evolves on its own. Note also that the user node has an implicit dependency on time, drawn as a dotted line.

The dependency concept is fundamentally connected to the idea of *causality*, understood as follows: if b depends on a then we say that a is a *cause* of (or causes) b's changes. We see the change of b as an *effect* of a's change. In other words, the dependency graph expresses the set of constitutive laws which, each represented as a dependency arc in the graph. Dependencies are connected to causality since a state parameter change will *cause* changes of other state parameters which *depend* on it.

We say that a is on the *causal path* of b if and only if a change of a will cause (imply) a change of b. This means that b depends (directly or not) on a:

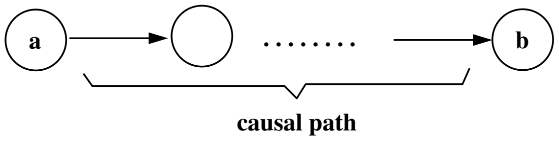

Figure 6: Causal path between a and b

A causal path between two state variables is therefore a sequence of other state variables that will be changed when the leftmost variable on the path changes. There may be several causal paths between two variables, and we'll see that this is the cause of the 'inconsistency' problems mentioned previously. In practice there must be only one possible causal path between two state variables.

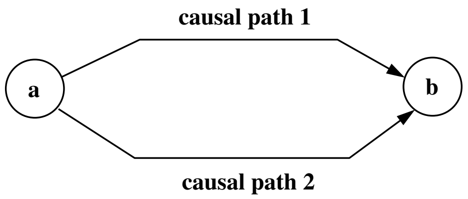

Figure 7: Nondeterministic behaviour and multiple causal paths

If several causal paths exist between two nodes a and b (figure 7) and the node a changes, this change will propagate independently on the two different paths to b. The result will be *nondeterministic* in the sense that b will finally keep the value assigned by the change that arrives the last. This will depend on the speeds of propagation of the changes on the two paths. In any case such a behaviour is not tolerated if the values of b will depend on the order of arrival of changes.

In order to study this further we have to distinguish between two kinds of dependencies:

7.1 AND Dependencies

We say that two dependencies are *AND dependencies* if the state parameter of the node using them will depend on both of them *independently*. This means that a parameter c depending by two AND dependencies on two variables a and b is given by a constitutive law where *both* a and b are independent variables:

Figure 8 says that $c = f(a, b)$, i.e. for any combination of values of a and b a value for c can be computed (we assume for simplicity that f can be evaluated for any values for its parameters a and b). Furthermore, a system that has two causal paths starting from a node m and ending up in the same AND node c is deterministic (i.e. safe) (see Figure 9). The system will arrive in the same state c when m is changed, regardless of the relative propagation speeds of the changes on the two causal paths. The intermediate

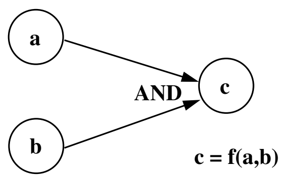

Figure 8: AND Dependency: c depends on a AND b

states the system passes through might be different though but the final state is guaranteed to be unique.

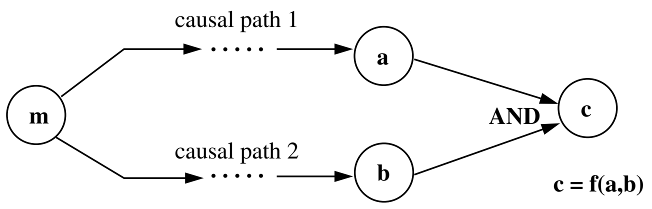

Figure 9: AND dependency and multiple causal paths

Indeed, there are two possible situations for the AND node c: a changes before b or b changes before a. If we take the first case and denote by a_i and a_c the initial and changed value of a, the system will respect the following invariants (Figure 10):

$\{ a = a_i ; b = b_i ; c = f(a_i , b_i) \}$	**Initial state**
$a \quad a_c$	**a changes**
$\{ a = a_c ; b = b_i ; c = f(a_c , b_i) \}$	**Intermediary state**
$b \quad b_c$	**b changes**
$\{ a = a_c ; b = b_c ; c = f(a_c , b_c) \}$	**Final state**

Figure 10: AND node states and invariants

If we consider the second case (b changes before a), we shall have the same final state (i.e. $c = f(a_c, b_c)$) but the intermediary state will be different (i.e. $c = f(a_i, b_c$ in the intermediary state).

The meaning of the above is that the variables a and b can freely and independently change and the changes will be *superimposed* to yield a change of c. It is safe to have multiple causal paths ending into an AND node. The order in which the changes of the AND node inputs occur is irrelevant if we ignore the intermediate states. AND nodes will appear in all cases where some state parameters changes are superimposed to yield another parameter change (the previous example can be thought as having two forces a and b that act on an object. The object's behaviour will be determined by the resultant force $c = a + b$).

A question which arises is whether it is reasonable to call an AND node 'safe' if it passes through a set of intermediary states which are determined by the relative speeds of the several causal paths ending in that node. We answer this question positively, since the parameters the AND node depends on are allowed to *freely* change, therefore *any* combination of their values is allowed (this is the very definition we have given to an AND node). In conclusion, any system composed of AND nodes will reach a well determined state when its input parameters are changed, regardless of the relative propagation speeds of changes in different parts of the dependency graph. Since the dependency graph is finite, the system will reach the final 'equilibrium' state after a finite amount of time (in practice this time can be very short if we have a highly interactive simulation with short latencies).

7.2 OR Dependencies

We say that two dependencies are *OR dependencies* if the variable of the node using them will depend *separately* on both of them. This means that a variable c depending by two OR dependencies on two variables a and b will have two different expressions on them (Figure 11).

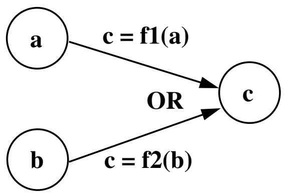

Figure 11: OR Dependency

This type of dependency can give nondeterministic behaviour. When we have two causal paths starting from the same node and ending in an OR node, the final value of its variable will depend on the *order* the changes will arrive to the OR node. Informally we can view an OR node as assigning to its state variable a value depending only on one of its inputs (on one *OR* on the other one, hence the dependency's name). The value of the OR node's variable will therefore depend on the *last* changed input.

For the same example used for the AND nodes (Figure 9), if a changes before b then obviously $c = f(a_c)$ and if b changes before a then $c = f(b_c)$. The system will not behave in a deterministic manner since, in most cases, $f(b_c) \neq f(a_c)$. An OR node will behave as though it tried to force the *same* state parameter to have two *different* values at the same time, which is impossible. An example of an OR node is a system where two users try to place the same object at two different positions in the same time. The object's position is the c parameter and the two users are the a and b nodes. The object will appear to 'jump' from one position to the other and back as the users try to force it to assume the two different positions. The object's position is therefore nondeterministic (we cannot say more about it than that it 'exists in two different positions at the same time'...)

A system including OR nodes is however not necessarily non-deterministic or useless or unsafe. We can have situations which are elegantly modelled by OR dependencies (i.e. cases when a state parameter appears on the left hand side of more than one constitutive law, but when some higher order semantics of the system can ensure that those laws will never be effected in the same time). Moreover, OR nodes can be converted into AND nodes by applying some superposition between the two mutually exclusive constraints on the node's state parameter (which means combining the two 'conflicting' laws in a single one, hence having the trouble-making state parameter appear once on the left hand sides of the system's laws).

7.3 Cyclic Dependencies

The dependency graph might have cycles. This section shows how these cycles can be understood and how the related problems can be solved.

Cycles are strongly connected with the notion of causality. In its general sense, a causal event-based system has the property that *an event will never be allowed to cause itself.* By event we understand here a state parameter change. In other words, a causal path in the dependency graph will not be allowed to have self intersections (loops), i.e. to pass twice through a given node. Note that this does *not* mean that the graph can not have cycles but it means that a causal path can not have cycles. There is a very simple algorithm one can use to enforce this rule when propagating a change form a node a to a node b:

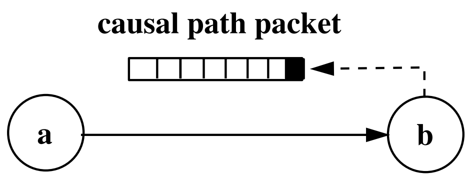

Figure 12: Cyclic dependency algorithm

Conceptually, we use a 'causal path packet' that travels together with the change signals through the dependency graph. This packet basically contains identifiers of all nodes having been encountered on the current causal path. When the node a must pass a change signal to node b, it examines its causal path packet and signals b only if b is not in the causal path packet. If so, it puts b at the end of the packet (which functions as a queue) and passes the packet and change signal to b, else nothing is done. This simple algorithm ensures that a change signal will not loop twice through the same node(s).

7.4 Self Dependencies

Self dependencies are a particular case of cyclic dependencies. Let us examine the meaning of a self dependency of the form $u = f(u)$.

The first meaning we can give to the above expression thinks of the same u on both sides of the expression. In this case, we have *an equation* rather than a functional dependency. If we want the expression be valid for a continuous (infinite) set of u's, this can be satisfied only if f is the identity function. This means that *the only self dependency of a variable on itself is the trivial one*, which is never explicitly present, nor used.

The second meaning takes different u's on the two sides of the expression. Namely we can think of the two values of u as being taken at different time instants. The expression then becomes:

$$u_{\text{new}} = f(u_{\text{old}}) \tag{7.5}$$

But this actually means:

$$u = u(t) \tag{7.6}$$

i.e. u depends explicitly on time. The potential problem with cyclic dependencies is now solved, since time is a state parameter which depends on no other state parameter but itself, therefore we can not have a cycle closing through the t node. The algorithmic explanation of the above is that a node for a variable that depends on itself will practically keep an internal copy of the old value(s) u_{old} and use them and the value of the time variable to compute the new value of u.

The conclusion of the above is that the so-called self dependencies $u = f(u)$ can actually be only time dependencies $u = u(t)$, therefore do not pose cycle problems since the time variable can not be involved in cyclic dependencies. Self dependencies are actually time dependencies.

8 The AND/OR Dependencies And Existing Simulation Systems

It is interresting to see how existing simulation systems fit in the formal description involving the AND and OR dependencies we presented. We shall examine two kinds of systems which cover a very large number of the available simulation software. The first kind is composed of *dataflow* systems, which were shortly introduced in Section 1. They include basically all systems in which the simulation is modelled as a set of operators connected by data channels. The second kind includes the systems described in Duclos and Grave (1993), where a (visualization) environment is presented as a set of operators acting on a shared data space (SDS), which the end user can apply on data represented as objects in the SDS. All operators interact implicitly by acting on the same, shared data objects. The interactivity of such a system is high, the user being able to apply any operator on any data objects at any time, as long as the operator and operands' types are compatible.

We can see that both dataflow and SDS systems are particular cases of the model we presented. Indeed, both the SDS operators and the dataflow 'nodes' are instances or our constitutive laws, more exactly of AND dependencies (in the SDS case the operators are practically equivalent with our constitutive laws). On the other hand, a dataflow system has usually a fixed topology, which corresponds to a dependency graph involving only AND dependencies, while a SDS system can apply several operators to modify a given data object, which corresponds to a dependency graph involving OR dependencies. (see Figure 13, which can be seen as the possible application of two operators e_1 and e_2 to yield e, while each operator is equivalent to a constitutive law).

9 Undersampling and Inconsistent States

When we have introduced the multiple sampling rate model, we have seen that potential problems can appear due to the uncorrelation of the sampling rates of the different state parameters. Two basic types of problems can appear:

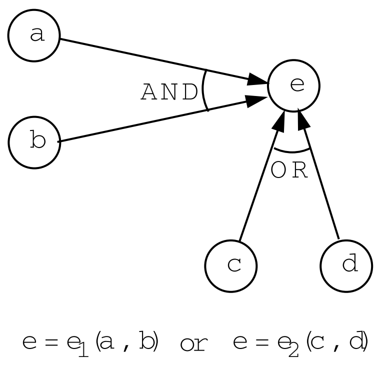

Figure 13: Combination of AND/OR dependencies. The above example reads as "*e* is function of *a* AND *b* OR of *c* AND *d*"

9.1 Undersampling Problems

An *undersampling problem* appears when a state parameter can not be computed (sampled) with a speed exceeding the change rate of the other parameters it depends on. A simple example is a user that drags a complex object with the mouse (Figure 14):

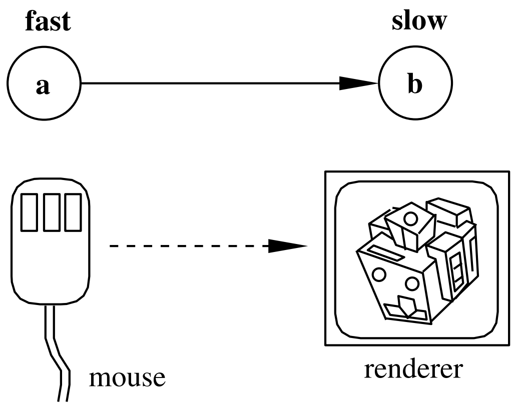

Figure 14: Slow module (renderer) undersamples fast changes of a parameter (mouse)

The renderer will 'undersample the user' if the motion events the user causes come faster that the renderer can draw the picture of the object. Other undersamplings appear for time dependent parameters which take longer to evaluate than the real time elapsed between two consecutive increments of the time parameter.

Undersampling is not a problem that always has to be avoided. Variations of some parameters can sometimes simply be 'lost' (not sampled) and the system will still behave correctly (e.g. some mouse motion events will be lost but the user still drags the object to the final desired place).

There are however cases when the system's behaviour critically depends on a correct sampling of a parameter so we must devise a means to avoid its undersampling. Good examples of such systems are accurate time integrators that can not afford to undersample time.

After having analyzed this problem it appears to us that the only safe and general way to avoid undersampling is *back synchronization*. By this we understand that a module computing a parameter *b* that critically depends on the sampling of some parameter *a* will force the module *a* to stop computing new values until *b* has finished and is ready to compute (sample *a*) again:

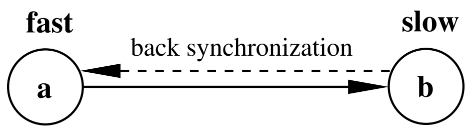

Figure 15: Slow module back synchronizes with fast module

Indeed, when a module *b* is slower than a module *a* it depends on, we can either make *a* as fast as *b* or *b* as slow as *a* in order to have the same speed for the two modules. The first option is not always possible. The second option is the presented back-synchronization.

Back synchronization actually makes the two modules *a* and *b* behave like a whole, as if they were a single module. Back synchronization will of course not eliminate the undersampling problem but move it one module 'upstream' on the causal path (to module *a* in our case). However there will be a certain point in the dependency graph where undersampling will not be a problem any more. All modules might ultimately back-synchronize with the time module (which does not depend on any other parameter) and the system will become a unique sampling rate system. In this view, a unique sampling rate system is a degenerate multiple sampling rate system with omnipresent back synchronization. (see section 1).

9.2 Inconsistent States

An *inconsistent state* is a state of the DSS which does not correspond to a state of the DS or of the CS we're trying to simulate. As described in section 3, the main goal of a DSS is to produce states that are as close as possible to the states of the 'physical' CS, ideally being exact samples of the CS's states. Since the multiple sampling rate DSS varies its parameters with finite steps and in an asynchronous manner, the above goal can be only partially fulfilled, therefore its states are sometimes quite different from the CS's states. When these differences are unacceptably large, the DSS might evolve on a totally different path in the state space than the CS. We say that it starts having inconsistent states:

It is impossible to guarantee a total removal of inconsistent states for all possible DSSs since a discrete system has no information of what it happens between the sampling instants.

Multiple sampling rate systems are more prone to inconsistent states, since their modules work with independent speeds, corresponding to independent sampling rates of the state parameters. Practically this means that parameters that correspond to the same state (the same physical time instant) will be computed and delivered at different real time instants. This can be seen in figure 3: the state vector is not a straight line (as it was for the unique sampling rate systems, see figure 2) but a skewed line showing that parameters are computed and delivered at different time instants.

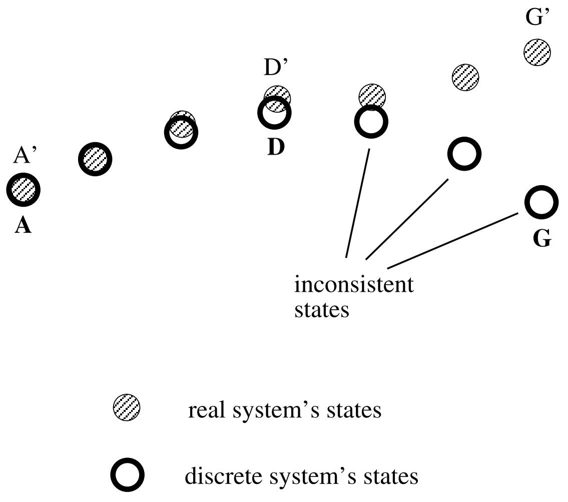

Figure 16: System evolution in state space: real system evolves on path A'..G' while discrete system follows path A..G. Discrete system starts being inconsistent from state D

Figure 17 is an example of system prone to inconsistent states. Two modules *a*,*b* with largely different speeds depend on a module *u*. There is a final module *c* depending on both *a* and *b*. A practical example is a system where *u* is the user interface (e.g. mouse), *a* is the position of some simple object controlled by mouse and *b* is the position of a very complex object, also controlled by mouse. *c* is the rendering of both objects.

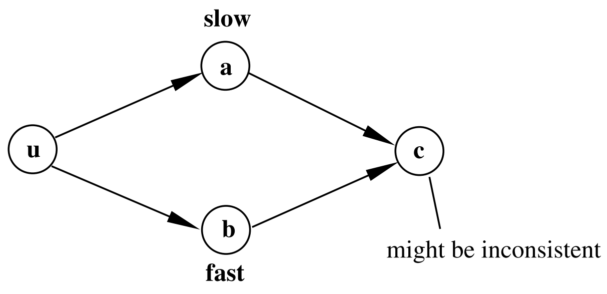

Figure 17: Example of system prone to inconsistent states

Let us take the following scenario: *u* starts changing and *a*,*b* start changing as well since they depend on *u*. Since *a* is fast, it will be able to 'follow' the changes of *u*, i.e. to accurately sample the variation of the *u* state parameter. Since *a* is slow, it will not be able to follow *a*'s changes, i.e. the module *a* will 'miss' most of *u*'s changes (i.e. it will undersample *u*). The rendering module *c* (assumed to be fast) will keep showing the correct position of *b* but an incorrect position of *a*. In other words, the object *a* will appear 'behind' its real position, besides having a jerky motion.

The above example illustrates two problems: firstly, the jerky motion of *a* as compared to a 'smooth' motion of *b*. This problem is caused by the slow speed of module *a* and can not be solved unless we accelerate *a*'s software or the hardware, therefore this is not of an issue when talking about inconsistent states. The second problem concerns the 'lagging behind' of the *a* object. This is a clear example of inconsistent state of the discrete system: although both *a* and *b* should be drawn in the same position (they're controlled by the same *u* parameter), our discrete system shows them in different places.

Remark that inconsistent states can happen only if the dependency graph is a graph and not a tree, i.e. if it exhibits 'diamond'-like structures as the one in Figure 17, or, ore formally, several causal paths between two nodes (hence the different speeds).

There are cases when a simulation can not tolerate such inconsistent states (e.g. a numerical simulation which is very sensitive to some parameters). We need a way to ensure that the states of our system will be as consistent as possible.

Several solutions to this problem will be outlined in the following.

9.2.1 Avoiding Inconsistent States by Back Synchronization

Back synchronization can be used to avoid inconsistent states. Taking the example shown in figure 17, we need to ensure that the values of *b* and *c* will always correspond to the same state. This means that the modules *b* and *c* must 'work as a whole' and not independently (we must ensure that the sampling rates of *b* and *c* will be identical).

In the general case, there is no connection between the speed (and sampling rates) of *b* and *c*. In order to ensure that they will always be equal, we need to add two back synchronizations and obtain the system in figure 18:

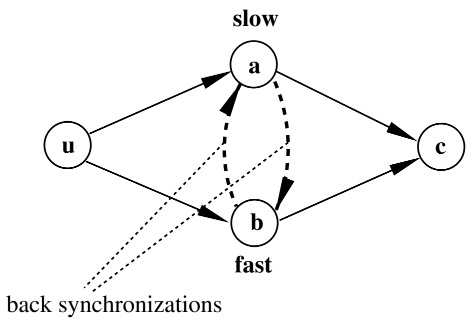

Figure 18: Avoiding inconsistent states by back synchronization

The two back synchronizations will actually force *a* and *b* work at the same speed, i.e. they will practically behave like an atomic 'macro' module that computes two state parameters. This ensures that the computed *b* and *a* will always refer to the same state, i.e. they will be fully consistent. In other words, the subsystem (a, b) is a unique sampling rate system (see 6.1).

The advantage of using back synchronization is that it provides a safe way to ensure consistency of a set of parameters. The drawback is that it forces a group of modules down to the speed of the slowest module in the group. Ultimately we can back-synchronize all modules with the time module and obtain a consistent unique sampling rate system.

9.2.2 Avoiding Inconsistent States by Barrier Modules

Back synchronization is a safe method for avoiding inconsistent states but forces a group of modules to have the same speed. This will slow down the fast modules, which is not always desirable. There are cases when we'd like to give up a certain degree of consistency for the possibility of having independent sampling rates. In such cases, we talk about *more or less consistent states* and not about fully consistent states. A fully consistent state is composed of a set of state parameters that all refer to the same time instant (are sampled at the same time instant). Extending this definition, a most consistent state is composed of a set of state parameters which have a small time difference between their sampling instants. A measure of consistency of a set of state parameters can be the sum of the absolute time differences between their sampling instants or other statistical norm (see figure 19).

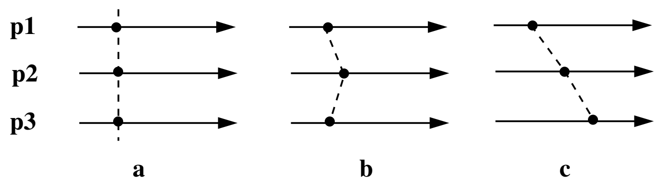

Figure 19: State examples: state *a* is fully consistent. State *b* is less consistent than *a* and more consistent than *c*

Taking a closer look to the example in figure 17, we see that what we actually want is that the *c* module 'sees' a consistent subvector (a, b) (we want to see a correct rendered view in which the objects *a* and *b* have the same position). This does not imply that the modules *a* and *b* must necessarily have the same speed and sampling rate.

An alternative solution to back synchronization is to introduce a module *B* between a, b and c. The *B* module (that will be called a 'barrier') will monitor the flow of *a* and *b* values received from modules *a* and *b* and 'extract' the most consistent pairs:

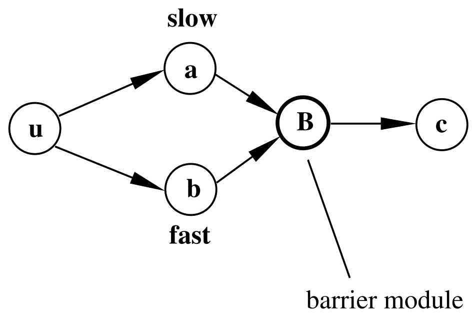

Figure 20: Barrier module for increasing consistency

There are several ways to implement a barrier module. One possibility is to have it store local copies of all its input parameters (*a* and *b* in our example). The barrier will keep updating its local copies immediately when some of the input parameters have changed and output them as a whole whenever all its inputs have been changed. At this moment, all inputs are marked 'unchanged' and the barrier waits for them to change. Another possibility is to connect time stamps to any modification of a state parameter. The time stamps have to be set by the time module which keeps the system's global time. The barrier module will then 'filter' its input parameters and output the most consistent sets, computed on the basis of the inputs' time stamps. This second solution has the disadvantage that the whole system must be modified in order to accommodate the time stamps management.

In conclusion, barriers are a simple means for obtaining a high consistency and still having modules with independent speeds. When we need to have fully consistent state parameters, we have to use back synchronization to enforce identical sampling rates.

10 Conclusions

Starting from the observation that many existing simulation systems have inherent limitations that restrict them to certain problems, a general simulation model has been introduced. Out of the general model, a discrete simulation (DS) model has been inferred by means of the sampling theory principles. The presented DS is able to describe any time-dependent process executable by a computer program. Next, we have presented a model for a discrete simulation system (DSS), i.e. a software system able to implement a DS. The conflicting requirements of corectness and interactivity have been treated in the formal framework of the DSS, showing how one can control both of them on a fine-grained, per-parameter basis. The use of the presented DS and DSS models is to establish a more formal framework in which the causes of the problems met in simulation systems can be easier represented and understood, and general solutions could be easier found. Ultimately, the DSS model presented here can be implemented in a software architecture, thus obtaining a very powerful, generic simulation system, able to directly model complex time behaviour and to offer a comprehensive control of its interactivity. The presented models can be also used to understand the limits of existing simulation and interactive visualization systems and why they fail to offer certain features or to cope with certain simulation classes.

References

[1] J. Barry, *GEOMPACK - A Software Package for the Generation of Meshes using Geometric Algorithms*, Adv. Eng. Software **13**, pp. 325–331.

[2] S. Carney, M. A. Heroux, G. Li, and K. Wu, *A Revised Proposal for a Sparse BLAS Toolkit*, Army High Performance Computing Research Center Technical Report 94-034, June 1994.

[3] J. Neider, T. Davis, M. Woo, *The OpenGL Programming Guide*, Addison-Wesley, 1993.

[4] J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.

[5] A. M. Bruaset, H. P. Langtangen, *A Comprehensive Set of Tools for Solving Partial Differential Equations: Diffpack*, Numerical Methods and Software Tools in Industrial Mathematics, (M. Daehlen and A.-Tveito, eds.), 1996.

[6] J. J. Dongarra, R. Pozo, D. Walker, *LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra*, Proceedings of Supercomputing '93, IEEE Press, 1993, 162–171.

[7] R. B. Haber, D. McNabb, *Visualization idioms: a conceptual method for visualization systems*, In *Scientific Visualization: Advances and Challenges*, Academic Press, 1994.

[8] R. Marshall, J. Kempf, S. Dyer, and C. C. Yen, *Visualization methods and simulation steering for a 3D turbulence model of Lake Erie*, Computer Graphics **24**, 1990.

[9] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, *The Application Visualization System: A Computational Environment for Scientific Visualization.*, IEEE Computer Graphics and Applications, July 1989, 30–42.

[10] W. Schroeder, K. Martin, B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1990

[11] C. Gunn, A. Ortmann, U. Pinkall, K. Polthier, U. Schwarz, *Oorange: A Virtual Laboratory for Experimental Mathematics*, Sonderforschungsbereich 288, Technical University Berlin. URL http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html

[12] B. N. Freeman-Benson, A. Borning, *Integrating Constraints with an Object-Oriented Language*, Proceedings ECOOP'92 – European Conference on Object-Oriented Programming, (O. Lehrmann Madsen, ed.), Utrecht, 1992.

[13] D. H. H. Ingalls, *A Simple Technique for Handling Multiple Polymorphism*, In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pp. 347–349, November 1986.

[14] R. Pozo, K. A. Remington, A. Lumsdaine, *SparseLib++: A Sparse Matrix Class Library, Reference Guide*, World Wide Web document **http://math.nist.gov/iml++/**, April 1996.