Timepark: A point-and-click modeling environment

Internship Report

Yannick WURM, idh@poulet.org

January 2004

Abstract

Here I present work on a point and click modeling environment code-named "Timepark". It is the result of an internship with Nicolas ZINOVIEFF, a self-employed software engineer. The work is divided up into two phases. During a first phase, we have developed a C++ framework for modeling, simulation and visualization of certain mathematical systems. The mathematical models the C++ framework is designed for are based on functions of time and their derivatives: time-dependent ordinary differential equations (ODEs). We tried to develop the framework in an open manner: cross-platform open-source tools were chosen whenever possible, and the source code for our framework will be distributed under an open-source license.

In the second phase, we have started development on a powerful yet accessible modeling, simulation and visualization environment, code-namend Timepark. It is built around our C++ framework. The intended applications for Timepark include use as a teaching or learning tool in schools and universities. In any context, it may one day be used to illustrate evolution of time series data in a visually stunning manner. As Timepark development continues, features and flexibility shall be added that could turn it into a modelers tool of choice for scientific use.

Both phases of our work are presented in this report.

A great thanks to:

Nicolas Zinovieff *my mentor during this internship, for guiding me and helping me out of some very sticky situations;*

Jean-Michel Fayard and Guillaume Beslon *from the Bioinformatics and Modeling department at Insa de Lyon, for making this project a recognized part of my studies;*

The open-source community *for being everything it is;*

Apple Computer *for creating an insanely great Operating System and development environment.*

Preamble: Internship host company

Nicolas Zinovieff is a self-employed software engineer. He obtained his masters-equivalent DEA (diplôme d'études approfondies) in September 2003, and has been making a living developing software since 1999. His major achievements include software development for solution maintenance and deployment at Apple Computer in Paris, France as well as web, security and network consulting and software development for RDM Hone (Philadelphia, PA, USA). More recently, Nicolas has started teaching an a course on embedded systems at Ecole Centrale d'Electronique in Paris. He is also currently developing DVD post-production tools for Amazing Studios, Paris.

I have known Nicolas Zinovieff since 1996, when he joined an Internet Relay Chat (IRC) discussion room named #poulet, that a number of Franco-phone Macintosh users spent time on (including myself). Many are successful independent software developers today.[1]

In May 2002, Nicolas and I spent 8 days at Apple's Worldwide Developer Conference in San Jose, California thanks to scholarships obtained as student-developers. We stayed in Palo Alto, and had a long bus ride to San Jose. One morning, Nicolas mentioned the idea of an easy-to-use visual modeling environment, and so Timepark was born.

Contents

1 Motivation

In schools, physical science is often taught using text-book examples and aided with practical experiments. Some experiments however, such as launching a satellite or observing an electron's movement, are not that simple to undertake. In the western world, most schools are now equipped with computers. Why not use computing to help students learn and ease the understanding of certain scientific concepts? What if teachers could quickly and simply create 3d computer-simulations and let students virtually experiment by changing parameters? What if a student working on his own could see the influence of initial parameters and choice of model on the behavior of a dynamic system through hands-on *in silico* experiments?

In a professional environment, static charts are frequently used as a means of conveying dynamic data. Although a chart may often suffice to illustrate the data, a graphical animation can increase attention and thus have much greater impact on the audience, particularly when projected onto a large screen.

Many computer-based modeling tools exist today. They usually fall into one of two categories: either they are versatile and have a very steep learning curve, or they are relatively straightforward but intended for a very limited number of specific models (such as "Dynamic" or "Satellites", used in French schools[2]). The formers' limitation constrains the user into certain usage cases, whereas the latter are inappropriate for young students or professionals who cannot dedicate the time necessary to master them.

Our aim with Timepark is to do for modeling what Apple Computer's products have done in many domains, including personal computing, digital music, and digital video production: to make modeling simple and accessible to everyone.

2 Concepts, Methods and Materials

Mathematical concepts: derivatives and integration

A defining characteristic of a dynamic system is that its variable's values change over time. A variable's variation (sign and speed) is quantified by its derivative: "the instantaneous change of one quantity relative to another" [Wor97].

For time-based systems, derivatives are often given relatively to time. Take for example a rocket flying in a straight line. The time-derivative of its position (flown distance) along the line is its speed, expressed instantaneously, for example in kilometers per hour or meters per second.
A variable's derivative function (relative to another variable) is the function giving all instantaneous derivative values over time. The value of $z_{position}$'s derivative at the time-point $T = t_a$ is the instantaneous change of position along the z-axis relatively to time: speed. Mathematically, $[d(z_{position})/dt]_{T=t_a} = z_{speed}(t)$.

The reciprocal concept is also used: a variable's values over time can be determined unambiguously if its initial value and the values of its time-derivative function are known. The process of obtaining a variable's values from its derivative and initial value is called integration. To take a very simple example, please consider that when our experiment begins, our rocket is at $z_{position} = 0$ $meters$. If we consider its speed to be constant at 100 $meters/second$, then after ten seconds our rocket will have reached $z_{position} = 1000$ $meters$ (this is illustrated in Figure 2 (a)).

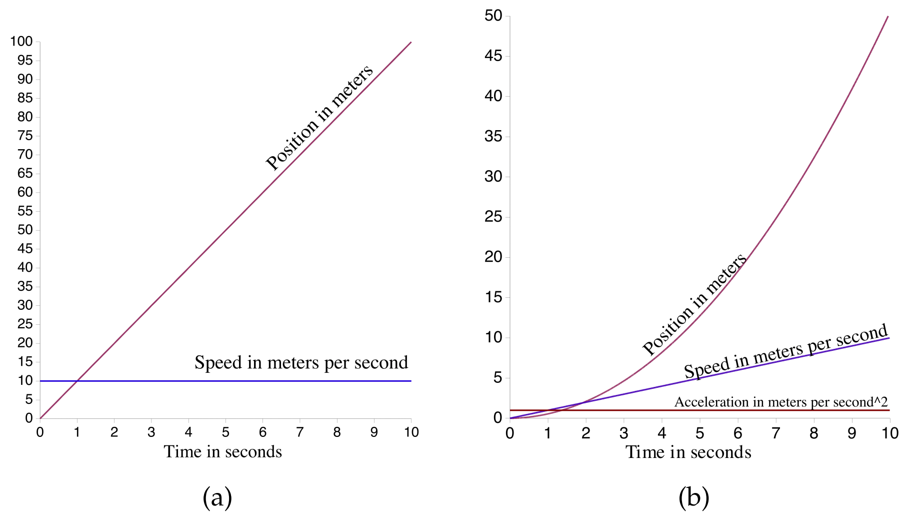


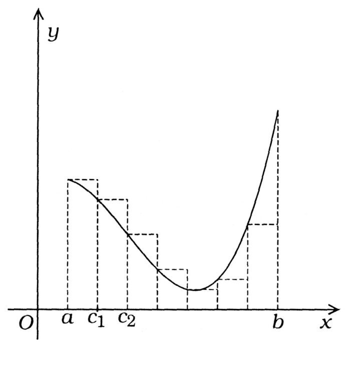
Figure 1: ***a:*** *Distance flown by a rocket flying at a constant speed of 100 meters per second.* ***b:*** *If speed is initially null. Constant acceleration leads to gradually increasing speed, which in turn leads to exponentially increasing flown distance*

The previous example of integration may seem very straightforward. This was because we considered speed to be constant. Actually, the function one wants to integrate is itself often a function of time. This is the case with a rocket at lift-off: its initial speed and position may be zero; its acceleration is not. Thus, if acceleration, the time-derivative of speed, is constant and positive, speed will increase over time. The more speed increases, the faster position will increase. This is illustrated in Figure 2 (b): z_{speed} and thus $z_{position}$ values rise rapidly if $z_{acceleration}$ is constant and positive. Once again, this type of relation becomes more complex if acceleration is not constant.

Geometrically, integrating a function over time means calculating the surface of the area between the curve representing the function's values and the horizontal abscissa line for which $T = 0$ (when the curve is below the abscissa line, the surface is taken negatively).

Computer-based integration of discrete values means approximation

Variation of values over time brings up a problem specific to computer-simulation: approximation. Time varies continuously, but representing time on the computer within a finite amount of memory requires time to be considered discretely. For a given problem a time interval such as $\Delta T = 0.1 seconds$ is chosen. For each time interval taken individually, the function to be integrated can be considered constant. Thus, a value of $z_{position}$ when 100 intervals have passed can be obtained by calculating values of z_{speed} and integrating to get $z_{position}$ for each new instant, one step at a time. This approximation, which consists in considering that during an entire time interval ΔT, the derivative is always equal to the value it takes at the beginning of that time interval (Figure 2). This is a first order approximation.



$$\begin{aligned} z_{speed,T} &= d(z_{position,T})/dt \\ &\simeq d(z_{speed,T})/dt \times \Delta T + z_{speed,T-1} \\ &\simeq z_{acceleration,T} \times \Delta T + z_{speed,T-1} \end{aligned}$$

(a) **(b)**

Figure 2: *The rectangle approximation method is one way of approximating integration when only discrete values are known*

Other less error-inducing integration approximation methods exist, such as the trapezoidal method (also first order), or the Runge-Kutta method. See more about this in the discussion section (on page 16).

Software engineering

Developing our modeling, simulation and visualization environment was divided up into two distinct tasks: the first was programming a robust cross-platform modeling, simulation and visualization framework. The second was using the framework as an engine around which a graphical user interface-based application was developed for end-users, Timepark.

Development methodology

After determining a global vision for the project, desired features were identified and realistic goals set. The early development process was exploratory and documentary; it helped in the decision-making process. Afterwards, a roughly iterative development scheme was used: short-term (weekly or bi-weekly) goals were set and obtained or revised, while trying to keep the code-base stable whenever possible and keeping longer-term goals in sight.

During development, a series of interface-independent tests (including unit-tests) were run systematically on the engine framework whenever its code was changed.

Development Tools

Code compilation was first managed using Apple Computer's Project Builder running on Mac OS 10.2, where compilation is jam[3]-based. In November 2003, we upgraded to Mac OS 10.3 and the X-Code development environment which use a native build system.

Code was mainly written in XEmacs 21.4 as well as in Apple's aforementioned integrated development environments. The graphical user interface was created using Apple Computer's Interface Builder. The compiler GCC 2.95.2 was used. Debugging was done with gdb, the GNU Project debugger[4], directly and indirectly using Apple's gdb-wrapper. Apple-supplied performance profiling tools were also used, namely Shark, OpenGL Profiler and MallocDebug.

Source code was managed using Concurrent Versions System (CVS[5]). Documentation was generated automatically from header files using doxygen[6]. Other documents were typeset using LaTeX2e [7].

Development platform

Code was mainly written on a iBook G3 (November 2002), and a 1.6GHz G5 Power Macintosh. CVS Server 1.11 ran on a NetBSD 1.6.1 Pentium-class machine.

Languages, libraries and Application Programming Interfaces (APIs)

The engine was programmed in C++, often using the GCC 2.95.2 C++ Standard Template Library (mainly container and input/output-related classes). OpenGL 1.2 was used for graphical visualization. Xerces 2.1 was used for XML-input validation (using XML Schema 1.0) and parsing. Flex++ 2.54 and Yacc 1.9 were used to generate code for lexical parsing of mathematical expressions.

A graphical user interface was created using Objective-C++ and Apple Computer's Cocoa API on Mac OS X.

3 First Result: A Preliminary C++ Framework for Modeling, Simulating and Visualizing

Modeling concept

The developed framework can be used for modeling certain time-dependent systems as described below: a system contains objects that have internal variables. These variable's values can be constant or vary over time. To have a variable's value change over time, it must be defined in one of two ways. The first is by directly applying a function of time. An example of the second is using speed to find out position: by using the variable's derivative. If the variable's initial value is given and the variable's derivative is defined as a function of time, the variable's values can be determined. To complicate things further, the variable's derivative can itself also be defined through it's derivative (the derivative of speed is acceleration).

The function defining a variables value can be unique, but it is also possible to have a choice made automatically between several, depending on the system's variable's values. To do this, instead of defining a single function, pairs of boolean control statements and functions must be defined. Two of these control statements may not be simultaneously true. During simulation, the single true control statement's corresponding function will be used.

We have chosen to name a function or set of control-statements and functions a relation. It is specific to one of an object's variables. Relations are grouped together as laws.

To eliminate the need to repeatedly define variables for different objects that may have things in common, an object class hierarchy was implemented. The basic class we use for graphical visualization is defined with the following variables: $x_{position}$, $y_{position}$, $z_{position}$, $x_{rotation}$, $y_{rotation}$, $z_{rotation}$ and $size$. For every object created, these variables are thus defined. Other classes can be defined (such as "Physical Object" or "Charged Object"), providing they contain at least one additional variable definition (such as $Mass$ or $ElectricalCharge$). A class can and must have one unique parent from which it inherits variables, but a parent can have several children (which inherit from its variables).

Mathematical possibilities

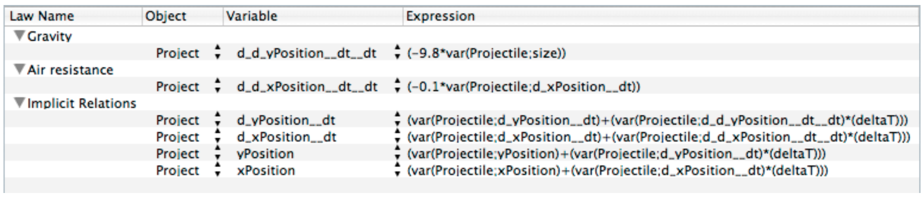
A variable's variation can be defined directly as a function of time, or indirectly, through the definition of its derivative (first or n-th order). Mathematical expressions made up of standard mathematical operators ($+$, $-$, $\times$, $\div$) and a number of constants and functions can be interpreted and evaluated (see the table in figure 3 for a short list).

Time representation

In the framework, time is represented as a decimal value of arbitrary unit. As mentioned on page 4, time is discretized, which means that it is cut up into indivisible steps. During one simulation run, all time steps have the same length.

Operators :	$+, -, \times, \div$
Precedence operators:	(,)
Trigonometrical functions :	cos, $acos$, sin, $asin$, tan, $atan$
Exponential operators :	$sqrt$ (square root), pwr (exponent)
Time-related:	T (time), ΔT (time interval)
Other:	constants, var (one of any object's variables)
Boolean operators:	$=, \leq, \geq, <, >, \neq$

Figure 3: *The developed framework can evaluate mathematical expressions composed of different elements, shown above the horizontal line in this table. A mathematical expression may be used as a function defining a variable's value. Two expressions, when combined with a boolean operator, make up a control statement which can be true or false.*



Law Name	Object	Variable	Expression
▼Gravity			
	Project	d_d_yPosition__dt__dt	(-9.8*var(Projectile;size))
▼Air resistance			
	Project	d_d_xPosition__dt__dt	(-0.1*var(Projectile;d_xPosition__dt))
▼Implicit Relations			
	Project	d_yPosition__dt	(var(Projectile;d_yPosition__dt)+(var(Projectile;d_d_yPosition__dt__dt)*(deltaT)))
	Project	d_xPosition__dt	(var(Projectile;d_xPosition__dt)+(var(Projectile;d_d_xPosition__dt__dt)*(deltaT)))
	Project	yPosition	(var(Projectile;yPosition)+(var(Projectile;d_yPosition__dt)*(deltaT)))
	Project	xPosition	(var(Projectile;xPosition)+(var(Projectile;d_xPosition__dt)*(deltaT)))

Figure 4: *A simple example of laws and single-function relations: An object named "Projectile" is submitted to gravity (downward acceleration), and air resistance (backwards acceleration). A series of "Implicit relations" were defined that take care of integrating the defined acceleration.*

Simulation algorithm

Simulation starts out with an incrementation of time's value by one time step. The time-dependent functions in the system's relations are evaluated; the corresponding variables obtain the new values. This is repeated many times.

Internally, every relation is stored within a law (see page 7). Also, derivatives are considered much like other variables. Taking this into account, the following pseudo-code gives the algorithm used for simulation:

```
/* Set up integration approximation */
1. For each law in the system,
      For each relation in the law,
        if the relation defines a variable's derivative's value.
           define the integrating relation(s) permitting to calculate
           the primitive's value through integration if necessary

/* Simulate */
2. For each law in the system,
      For each relation in the law,
        if the relation contains a unique time-dependent function,
           evaluate the function.
              The relation concerns one variable of one object.
              Set the variable's new value to the evaluation result.
        Otherwise, the relation contains pairs of boolean control-
        statements and functions.
              Evaluate the control-statements.
              Only one of them can be true.
      Evaluate the corresponding function.
              The relation concerns one variable of one object.
              Set the variable's new value to the evaluation result.
Increment the system's time by one time-step and repeat 2 until the
system's time counter has attained a certain value.
```

Main C++ classes

The main C++ classes used as a developer's building-blocks of a system created with our framework are:

TPWorld : This is the system "container". Once *TPObjects* and *TPLaws* have been added, *TPWorld's simulate* method can be called. Others include *verifyIntegrity, setTimeStep* and *setTimeFrame*.

TPObject : A *TPObject* has a name and contains *TPVariable*s, identified by their name.

TPLaw : A *TPLaw* is a group of *TPRelation*s.

TPRelation : A *TPRelation* governs one of a *TPObject*'s *TPVariable*s. This happens through the definition of one or more mathematical *TPExpression*s that can be evaluated. If several *TPExpression*s are given, each must be paired with a boolean statement; only one of which should be true at a time. The *TPExpression* whose boolean statement is true is used.

TPExpression : To represent mathematical expressions, a trees of *TPExpression* subclasses are used.

Additional features

The developed framework has other features, including a simple system integrity verification. It currently checks whether the objects and variables referenced and the mathematical expressions used in relations are correctly defined.

The framework has support for writing and loading system definitions to and from files respecting the supplied XML Schema 1.0 structure.

After simulation, the variable's values over time are still stored in memory. The system's previous states can thus be obtained, and the system's variable's evolution can be written to a delimited text file.

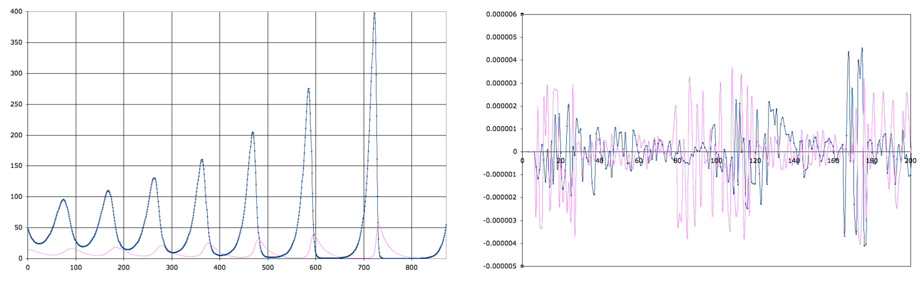
The system's evolution as obtained by simulation can also be displayed inside an OpenGL Context either real-time or post-processing. The system's objects are represented using models from standard files in the Alias Wavefront .obj format (or with generic shapes).

Mathematical validation

The mathematical simulation engine was validated using known mathematical models and simulation results from other sources. Dynamic system simulation data reproduced qualitatively includes Lotka-Volterra simulation points (as obtained from [Sha97]), shown in figure 3(a). The quantitative differences between data obtained in a Microsoft Excel Spreadsheet and using our mathematical framework are very small (see figure 3(b)): the relative error is less than 10^{-5}.

Distribution Policy

The C++ framework for modeling, simulation and visualization will be open-sourced under the Lesser Gnu Public License (LGPL).



(a) **(b)**

Figure 5: ***a:*** *Simulation results for a Lotka-Volterra Predator-Prey system defined by* $d(Predator)/dt = r \times Predator - a \times Predator \times Prey$ *and* $d(Prey)/dt = b \times Prey \times Predator - m \times Prey$, *with initially* $Predator = 15$, $Prey = 50$. ***b:*** *Error relative to the same simulation carried out in a Microsoft Excel spreadsheet [Sha97]*

Development pattern

The framework was developed in an object-oriented manner, using the Model-View-Controller paradigm. Sub-classing was used when deemed appropriate, so as to have a large amount of potential for extending the existing framework. This concerns for example the mathematical expressions, visualization methods and file output methods.

The API was used as the engine for the end-user application described in the next section.

4 Second Result: Timepark, an end-user's Modeling Environment

Using Apple Computer's Cocoa API as well as the previously described modeling, simulation and visualization framework, an end-user application was developed. It is currently at an early preview stage.

This graphical user interface wrapper for the previously described framework gives an end-user point-and-click access to many of its modeling, simulation and visualization features. Objects can be created, variable values modified and laws and relations defined to govern the variable's variations over time (these term's meanings were defined in "Modeling Concept" on page 7).

For simulation, the desired time-increment can be set and simulation is visualized in 3d in real time (frames are dropped if simulation is very slow).

The user can also visualize existing simulation data. Both playback of existing data and simulation can be paused and resumed. Time-dependent data can be exported, and the information defining a system can be saved to and opened from files.

Overview of the User interface

For a given system, one main window is used, shown in figure 4. It is meant to be maximized so that the viewing area is as big as possible. This window is divided up into several distinct parts. On the left side is a 3-d representation of the system and the objects it contains. The right side contains different user-interface elements to help create and inspect the system. Spanning the bottom of the window are controls to govern simulation and playback. The window and all main elements are re-sizeable. Some can be hidden.
Here are some details concerning the individual elements:

3-d Representation (Figure 4.a): Using the 3-d graphics standard OpenGL, the 3-d representation shows the objects in the current system in (x,y,z) coordinate space. The system is shown as viewed by a camera, which's height and position may be altered. Height is controlled by a slider, situated on the far left. Click-dragging within the 3-d representation area permits rotating the camera around the display's center. Display is usually centered around the system's coordinate origin, where three unitary vectors are shown, giving the x, y and z-axis directions.

Current Objects Inspector (Figure 4.b): This is where a list of the system's objects and their variable's precise values is displayed. It is structured hierarchically as a two-level outline-view. The top level items are the system's objects. More information about an object can be obtained by expanding the outline for that object. All of the object's variables and their values are then shown on a second level. The variable's values can be edited by the user as can an object's identifying name. When its name is selected, an object can also be removed from the system by clicking the *Remove* button.

Current Laws Inspector (Figure 3): The system's laws and relations are created here; it's hierarchy resembles that of the Current Objects Inspector. Since a relation is necessarily stored within a law, this outline-view's top level are laws. The second level are relations. The information defining laws and relations is displayed. Thus, a law's names and the relations it contains are displayed. For each relation, the object and variable it concerns is given, as is the mathematical expression which will be used to calculate new values.

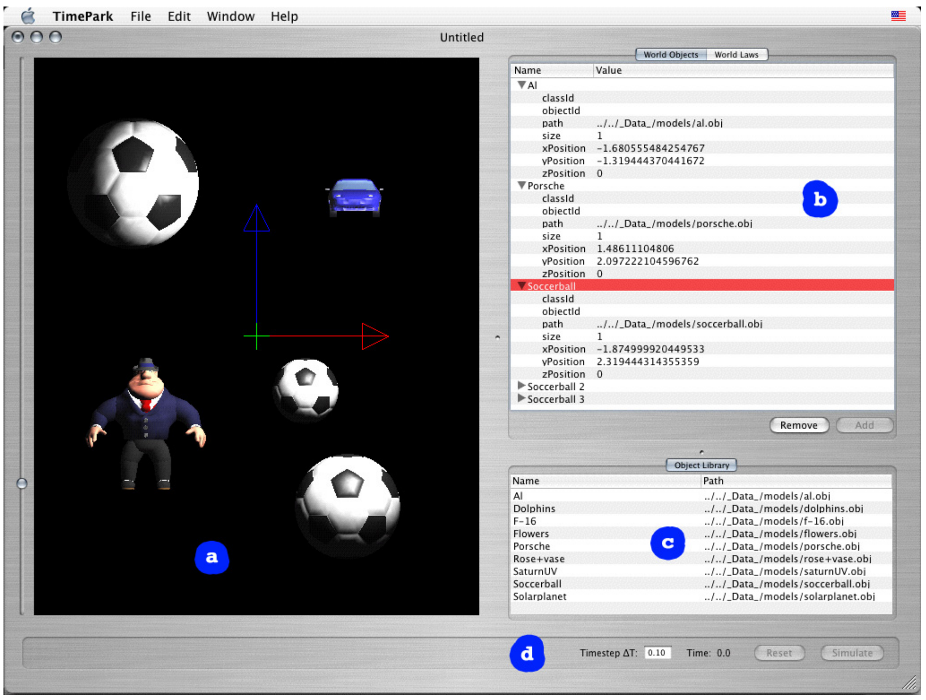
All of this information may be modified by the user. To change a relation's object, the user chooses one of a list of the system's objects (identified by name) from a pop-up-menu. Likewise, a pop-up-menu of an object's variables permits choosing between them.

Laws and relations can be added to and removed from the system by clicking the *Add* and *Remove* buttons.

Object Library (Figure 4.c): This is a table of all available 3-d object representations. An object instance is created in the system by the "drag and drop" metaphor: the user clicks on one of the table's rows and drags it into the 3-d representation on the left.

Simulation and Playback Controls (Figure 4.d) This area spanning the bottom of the screen displays the current value of the system's time. It also shows the time-interval which will be used during simulation. This value may not be changed while simulation is running.

Once the system has been defined, the user can click on the *Simulation* button. When simulation has started, the *Simulation* button no longer exists: it is replaced by a button titled *Pause*. When simulation is paused, this button is replaced once again. We will explain a little further down. Two additional controls also become available: a *Reset* button, and a horizontal slider. Clicking the *Reset* button puts the system back into it's initial state (simulation data is lost). Changing the horizontal slider's value lets the user scroll back to any available time point. A *Playback Data* button then appears where the *Pause* and *Simulation* buttons were. When clicked, the system's evolution is shown until reaching the instant at which simulation was stopped (playback may also be paused). Once that point is reached, simulation may be resumed from where it left off.



Name	Value
▼Al	
classId	
objectId	
path	../../_Data_/models/al.obj
size	1
xPosition	-1.680555484254767
yPosition	-1.319444370441672
zPosition	0
▼Porsche	
classId	
objectId	
path	../../_Data_/models/porsche.obj
size	1
xPosition	1.48611104806
yPosition	2.097222104596762
zPosition	0
▼Soccerball	
classId	
objectId	
path	../../_Data_/models/soccerball.obj
size	1
xPosition	-1.874999920449533
yPosition	2.319444314355359
zPosition	0
▶Soccerball 2	
▶Soccerball 3	

Name	Path
Al	../../_Data_/models/al.obj
Dolphins	../../_Data_/models/dolphins.obj
F-16	../../_Data_/models/f-16.obj
Flowers	../../_Data_/models/flowers.obj
Porsche	../../_Data_/models/porsche.obj
Rose+vase	../../_Data_/models/rose+vase.obj
SaturnUV	../../_Data_/models/saturnUV.obj
Soccerball	../../_Data_/models/soccerball.obj
Solarplanet	../../_Data_/models/solarplanet.obj

Timestep ΔT: 0.10 Time: 0.0

Figure 6: *Snapshot of a typical Timepark screen.* ***a:*** *3-d representation of the system's objects.* ***b:*** *summary of the system's Current Objects. Can be hidden to show Current Laws insted.* ***c:*** *Available objects are shown in this Object Library.* ***d:*** *Controls for launching simulation. When simulation is paused, a slider appears on the left. It permits scrolling back in time.*

Timepark in action

The user starts out with an empty system, represented by a completely black 3-d representation area (only the coordinate origin and directions are shown). He or she begins adding objects by dragging the desired ones from the Object Library on the lower right, and placing them into the 3-d representation at the desired $(x; y)$ coordinates. On the top right, either the system's Current Objects, or it's Current Laws are displayed. If the Current Objects view is displayed, it is updated as the user drops objects into the visualization area. Here, the user can change object's names and fine-tune their variable's initial values.

The next step is creating laws. Accessing the Current Laws view hides the Current Objects view. A first untitled law and empty relation are created by clicking on the *Add* button. They are each on one row. The law may be renamed. For the relation, an object is first chosen from a pop-up-menu list of the systems's objects. In the next pop-up menu, one of the chosen object's variables may be selected. Then a mathematical expression is typed into the last of the relation's fields. It will give the referenced variable it's value. This process can be repeated as desired, with other objects or variables.

The user can freely make changes to the values set previously. Objects, laws and relations can also be removed by selecting them and clicking a *Remove* button. The time increment (the duration one simulation step represents) can also be changed. It affects simulation precision and speed.

Once the user is ready to launch simulation, Simulate may be clicked. Timepark internally verified that the system the user has defined is ok. If there is a problem, it is pointed out to the user. Otherwise, simulation begins. Time starts increasing (shown at the screen's bottom right), and the system's objects sizes and positions may begin to change. Besides seeing real-time qualitative changes in the system's 3-d visualization, the Current Objects table shows the variable's quantitative values.

If something interesting has just happened, the user can pause the simulation run, scroll back, and play back what was calculated. Simulation can then be resumed.

If the user is unhappy with simulation results, they can be reset by clicking the *Reset* button. Simulation can be started again using the same conditions, after making minor modifications by changing initial values or time increment, or after major changes involving addition or removal of objects, laws and relations.

Once the user is satisfied with the simulation run, the File menu has an "Export Time-series Data" item which permits data exporting to a tab-delimited text-file. It can be read into any other program such as Microsoft Excel for creating graphs.

The system's state can be saved to file at any time to be loaded back later (on the same computer or not).

When a problem is detected, the user is alerted with detailed information as to what went wrong. When an error is unexpected (and probably due to an internal error), a single click is sufficient to send us a bug-report.

Availability

Timepark's distribution policy is not yet determined. A public preview release will be available shortly (by the end of March, 2004).

5 Discussion

The aim of my work was not scientific discovery but to develop a tool. Thus the work and choices presented here are primarily of technical nature. They are not based on a purely scientific evaluation of known or reported statements which could could be referred to in peer-reviewed articles in prestigious journals. Rather, the choices were made from a pragmatic software engineer's viewpoint, meant to solve a problem that we have identified: creating a modeling environment that would be simple to use.
At first, I wanted this discussion to be structured in two main parts: one discussion for each result. However, the discussed issues are often transversal, so a different structure was chosen. I begin by speaking about the limitations linked to the modeling concept we have chosen. After giving reasons for dividing the work into two main parts, I first focus on issues we had with the framework and in particular mention ways its precision could be improved. Subsequently, I will turn to Timepark and discuss implementation choices then features and shortcomings. Finally, I give ideas for promoting Timepark's use in the real world.

Modeling concept

The chosen concept permits object-oriented modeling. These objects have variables. Their variation is governed directly or indirectly through time-dependent functions or integration of time-dependent functions. Control-statements can be used to choose between different functions.

From a mathematical viewpoint, the concept we have chosen permits modeling of a large number of dynamic systems based on ordinary differential equations (ODEs). Thus many known physical and biological models can be implemented. However, the concept is inappropriate for a very large range of models (statistical, neural-network based, matrix-based just to name a few), but it was not our aim to make a universal modeling architechture.
Our aim was to create an architecture sufficient for modeling most systems seen in science classes at high-school or early university level.
The significance of our modeling concept comes from the fact that the used variables are necessarily linked to objects: instead of being just abstract placeholders, the user is required to give them meaning (through names). This is reinforced when a graphical visualization is used, such as the 3d-visualization possibility we have supplied.

Additional flexibility will be provided to Timepark users by giving them the opportunity of importing data: the time series import feature should permit the user to link external data to object variables. It could then be used for visualization on its own, or as a basis for simulation and visualization when combined with our ODE-based math.

The use of control-statements is possible for users of our framework although it was not yet implemented in Timepark as of this writing. We believe that this opens the door to many modeling possibilities. One of these is collision detection and management. Another could be a system in which interaction between two objects is due to different forces whose intensity depend on the distance separating the objects. Different cases can then be made, some of which could also be used to make processor load lighter. We envision using control-statements to trigger more complex actions, such as object creation and destruction. For example: if two hydrogen atoms meet under certain conditions, they could fuse and be replaced by a H_2 molecule. Control-statments could also be used for creating simple multi-agent systems.

An important element missing from the implemented tools is stochasticity. By definition, a model is an approximation of reality. Thus it is inherently incorrect. This may be due to the inherent randomness of the world, imperfect or incomplete knowledge of things that should be known, or errors made while establishing the model. These uncertainties can sometimes be at least partially compensated by introducing random noise into the model. One can consider that it plays the role of existing parameters missing from the model (see for example [BBL$^+$01]. This is the case for Brownian movement: an explanation for some of an atom's basic movements are unknown, so they are considered random. Whatever the reason, stochastic models are often better at capturing real-world behavior than deterministic ones [SYSY02].
Adding standard stochasticity generators is on our framework's to-do list.

Cross-platform development and distribution

Work described here was structured in two parts: developing a cross-platform framework and developing a platform-specific user interface. We had several reasons for doing this, the main being our desire to choose the tools best-adapted for each problem. Creating a cross-platform user interface would have limited our flexibility in choosing a user-interface toolkit and thus in creating an interface. On top of that, different platforms have different user-interface metaphors their users are accustomed to. Users often feel uncomfortable using graphical interfaces that were developed with a different platform in mind[Jeg].

Framework implementation

While developing the framework, pain was taken to open it to a wide range of possible architectures. This is visible at several levels. First of all, we chose development libraries and tools which are known to work in many environments and are actively used in large computing projects. This is the case for Xerces, OpenGL, libstdc++, Flex++/Yacc as well as the compiler (GCC).
On a second level, use of a published XML-Schema can permit anyone to create and edit the system definition files our framework uses natively. The possibility of exporting data obtained through simulation permits post-processing in many other applications.
Finally, our framework is open because we distribute it under the Lesser Gnu Public License (LGPL). This means that anybody may access its source code, and use it as they wish, providing the changes made are contributed back to the community by redistributing the modified framework under the same LGPL license. Thus, anyone can modify or verify parts of our code. Obviously, contributions and feedback are most welcome!

Compiler issues

Despite the fact that the tools we chose to use are relatively well established, we encountered issues with some of them. Using the mathematical expression parser generated by the pair of code generators Flex++/Yacc, requires including an older header-file, FlexLexer.h. This file includes others from an older C include hierarchy. The resulting namespace conflicts prevented compilation with GCC 3 or newer. Using GCC 2.95.2 required us to use an older version of the Xerces XML-library (2.1 instead of 2.3), but then all issues were resolved. Being able to use a more recent version of GCC (3 or later) would have been nice, since many interesting changes were made, one of the more visible being optimization for faster code execution. From a developer's point of view, many other improvements in newer GCC compilers could have eased the development process by accelerating compilation, linking and debugging.

Room for improvement

Additional features, such as further expanding the modeling concept by adding support for stochasticity (described above) still need to be implemented. Other shortcomings exist, such as:

- there is no support for derivation of one variable relatively to another (any variable including time);
- direct importing of time-series data is currently not possible;
- there is no support for importing 3-d object files that are not in the Wavefront .OBJ format.

Simulation speed could be increased by optimizing the algorithm, and by optimizing for certain hardware features (multiple processors, specific processor instructions...). Depending on the realm in which the framework is used, quantitative precision may or may not be an issue.

Improving integration: The integral-approximation method currently used for simulation is known as the rectangle-approximation. It is well known that this approximation is rough when compared to other simple methods such as the Trapezoidal Integration Method or the Simpson Method [GJ96]. Iterative integration schemes such as the Runge-Kutta method can give much higher precision (RK4 has an error which is significantly smaller than $(integrationinterval)^4$, [Wik03]).
Implementing the trapezoidal integration method would be relatively straightforward and significantly reduce error.
Integration precision can also be improved by the user on case-by-case by making the time interval, ΔT, smaller: Fitting the form of a curve with rectangles is easier when using a larger number of rectangles.

Improving number representation: Another issue is that of representing numbers on computers. As with any enumeration system, we are limited: We cannot represent all possible numbers, there are just too many of them [Sto96]. On computers, numbers are represented using bits, binary numbers that are either 1 or 0. Given any fixed number of binary numbers, most calculations with real numbers will produce quantities that cannot be exactly represented using that many binary numbers. A surprisingly simple example is the real number 0.1: In standard floating-point base 2 representation using 32 bits, it cannot be represented exactly, but it is approximated with $1.10011001100110011001101 \times 2^-4$ [Sun91]. Likewise, due to roundoff errors, the associative laws of algebra do not necessarily hold true for floating-point numbers: The expression $(x+y)+z$ has a totally different answer than $x+(y+z)$

when $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ (it is 1 in the former case, 0 in the latter).

A partial remedy can be found. Different high-precision math libraries are available, such as MAPM [Rin01] or apfloat. Using one of these or developing our own custom fixed-point mathematics implementation for representing real numbers would permit us to strongly reduce the error (given that computer memory is finite, the problem cannot be completely eliminated). However, one must keep in mind that higher precision would have the undesired effect of slowing down every calculation. For example, a standard addition of two numbers requires one step. Doing the same addition on the same computer with twofold improved precision usually requires four steps (this is the case when adding two numbers stored over 64 bits each on a 32 bit processor).

I believe that precision would only be an issue for certain uses. Where the visual aspect prevails, qualitative accuracy is sufficient and should be attained in most cases as long as the numbers involved are not near the limits of the floating-point representation used. For scientific use, finer control is definitely needed; a high-precision math library should be used. Precision could then be set on a case by case basis as desired, keeping in mind that higher precision slows things down. Rigorous scientific validation would then be required; the mathematical validation presented on page 9 having little scientific value.

End-user Application

We wanted to give our end-user application the best user-experience possible: our aim is to make modeling intuitive. We want the user to see the computer not a hurdle that must be overcome, but as a helpful and fun-to-use tool. Although APIs for developing cross-platform user-interfaces are plentiful (such as wxWindows, QT, Tcl/TK and Glut derivatives...), we believe the best user-experience can be obtained using Apple's Cocoa framework. This mature object-oriented framework was born more than fifteen years ago [Sin04] and makes providing an elegant user-interface relatively fast and straightforward. Most notably, the relatively strict Apple Human Interface Guidlines are easily respected; so the user feels at home right away.

Features

As stated before, the end-user application is still in development. Before several additional key features are implemented, it will not be fit for day to day use. What follows is a list of missing features, and, in certain cases, possible but un-intuitive workarounds that require the user to adapt to the computer:

- The interface for entering mathematical expressions is not very user-friendly (accessing certain functions (such as $\sqrt{x}$ or x^y and values of object's variables) currently requires knowledge of specific syntax. We plan on making this easier by letting the user visually select functions (through a palette or pop-up menus). Additionally, we would like variable references to be highlighted when displayed inside mathematical expressions.
- The class hierarchy is not implemented (See page 7). Only "physical" variables are defined (3 positional, 3 rotational and one controlling size). This makes representing mass, electrical charge, or population count difficult. As a workaround, one of the other variables can be used instead; the user is thus currently limited to seven variables.
- Control structures are not available: only one function may be defined per variable, and it is always used. This limits usability for complex models. For example, some pharmaco-kinetic models of medicine's location

in a patient's body require administration of multiple doses of medicine. Until control-structures are implemented, they can still be administrated "manually", by pausing simulation, increasing the quantity of medicine present in the first compartment and resuming simulation.

- The initial time frame cannot be set (simulation necessarily starts at $T = 0$). This can be overcome once again by modifying values while the simulation run is temporarily paused.
- Using Timepark for visually stunning presentations would require full-screen visualization and visualization export facilities. The currently displayed area could be recorded using third-party continuous screen-capture utilities such as Ambrosia Software's Snapz Pro[8].

Real world use

The primary goal of Timepark is to be used in schools. Use for educational purposes could be promoted by including ready-made experiment templates. A given "experiment template" would feature a defined system, ready for simulation (much like the Wizard concept used in Microsoft Office). The user could be guided to vary certain parameters so as to understand a specific concept. Educational tools like this exist, but they are usually specific to one problem or a small set of problems, requiring the user to adapt to a different user-interface for each experiment. With Timepark, the same tool could be used for different simulated experiments in physics, chemistry as well as biology. The power of setting up the experiment is put into the hands of the teacher, letting him or her adapt the experiment to his or her course and not the other way around.
This could be implemented at the framework's level in an object-oriented cross-platform scripting language such as Ruby[9].

Let me illustrate this with a concrete example: an experiment template for launching a satellite into orbit around earth. Upon opening this template, a presentation window could appear, with text and eventually graphics describing the experiment. A second window could list parameters that might influence the satellite's trajectory (acceleration, how long to accelerate...), and prompt the user to enter values for these parameters. Simulation could then be launched and shown right away, without the user having to use the standard Timepark interface.

Another feature many could find interesting is displaying an object's trajectory. Thus at the end of simulation, one would end up with a dashed-line, each point representing a position the object has held from its initial to its final position. Additional control should obviously be given, such as the time interval between two displayed points, the time range for which points should be shown and the option of activating and disabling trajectory display after simulation.
This would be particularly useful in an educational institution where children could run their simulation on the computer and subsequently have a print-out they can keep and that could be used for pencil-and-paper exercises (such as verifying that $\sum \vec{F} = m \times \vec{a}$.

An innovative approach

Despite the aforementioned shortcomings in the current implementation, we believe the idea behind Timepark is promising. Use of user-interface metaphors such as drag-and-drop object creation and pop-up-menus to reduce typing bring about a novel modeling experience. Also, what exists in the system is

immediately visible and quantifiable on-screen, something that can require effort in text-based tools.
A solid user interface linked to Timepark's expected versatility may make the difference between Timepark and other modeling tools, that are either complex or straightforward and limited to a specific model.

6 Closing Remark

For the near future, development will be focused on our graphical modeling environment, Timepark. Source code for the framework will be available shortly at ⟨`http://yannick.poulet.org/timepark/`⟩. We eagerly await the feedback we will get upon releasing Timepark to the general public. The amount, quality and diversity of such feedback is unpredictable. Nevertheless, I am sure that it will provide us with novel ideas and solutions.

Notes

[1]#poulet developers include ⟨`http://www.inferiis.com/`⟩, ⟨`http://www.macanalysis.com/`⟩, ⟨`http://www.pommsoft.com/`⟩, ⟨`http://www.rhapsodyk.net/`⟩

[2]Some software used in physics classes in France: ⟨`http://perso.wanadoo.fr/saintgregoire/scphys/logiciel.htm`⟩

[3]⟨`http://www.perforce.com/jam/jam.html`⟩

[4]⟨`http://sources.redhat.com/gdb/`⟩

[5]⟨`http://www.cvshome.org/`⟩

[6]⟨`http://www.doxygen.net/`⟩

[7]⟨`http://www.latex-project.org/`⟩

[8]⟨`http://www.ambrosiasw.com/utilities/snapzprox/`⟩

[9]⟨`http://www.ruby-lang.org/en/`⟩

References

[BBL$^+$01] F. Beck, B. Blasius, U. Lüttge, R. Neff, and U. Rascher. Stochastic noise interferes coherently with a model biological clock and produces specfic dynamic behaviour. *The Royal Society Proceedings: Biological Science*, 268(1473):1307–1313, June 2001.

[GJ96] D. Guinin and B. Joppin. *Précis de Mathématiques: Analyse-Géométrie. Prépas MPSI 1ère année*, chapter Intégration sur un ségment, pages 245–250. Bréal, 1996.

[Jeg] Mark Steven Jeghers. Cross-platform toolkits: Potential problems in gui development.

[Rin01] Micheal C. Ring. Mapm, a portable arbitrary precision math library in c. *C/C++ Users Journal*, November 2001.

[Sha97] Alexei Sharov. Quantitative population ecology: Lotka-volterra model on-line lecture, 1997.

[Sin04] Amit Singh. What is mac os x?, 2004.

[Sto96] John David Stone. Fundamentals of computer science: Ieee floating-point representations of real numbers, 1996.

[Sun91] Sun Microsystems. What every scientist should know about floating-point arithmetic. *Computing Surveys*, March 1991.

[SYSY02] Srivastava, You, Summers, and Yin. Stochastic vs. deterministic modeling of intracellular viral kinetics. *Journal of Theoretical Biology*, 2002.

[Wik03] Wikipedia. Runge-kutta methods, 2003.

[Wor97] WordNet. *WordNet 1.6*. Princeton University, 1997.