Alcom-FT Technical Report Series
ALCOMFT-TR-01-26

Smooth Animation of Algorithms in a Declarative Framework*

Camil Demetrescu† Irene Finocchi¶

Abstract

Thanks to the increasing sophistication of modern workstations and display devices, supporting smoothly changing images has nowadays become a valuable feature in software visualization systems, but it seems difficult to achieve in declarative frameworks, as they are basically data (and not event) driven. This paper focuses on the problem of specifying animations of algorithms in a declarative style, addressing both automation and customization issues. We propose a general functional approach to animation modeling where animated behaviors of graphical objects are "higher-order" attributes of the objects themselves expressed in the form of mathematical functions. These functions provide temporal and spatial information for creating smooth transitions between graphical scenes.

The paper also describes the practical embodiment of our mathematical framework in the logic-based software visualization system Leonardo. Differently from previous approaches to smooth algorithm animation, detection of visual events and in-betweening of scenes are completely automatic with our technique. Moreover, automation is supported without sacrificing customizability: default animated behaviors of graphical objects are automatically provided by the system, and practical solutions for specifying user-defined animation functions are supported as well. The quality of animations can be therefore incrementally improved by overriding the default behaviors as we show in some examples.

1 Introduction

Thanks to the increasing graphical possibilities offered by modern workstations and display devices, supporting *smoothly* changing images has nowadays become a valuable and almost mandatory aspect in a variety of applications; among them, in this paper we focus on algorithm animation systems. Algorithm animation is concerned with the use of graphics and animation techniques for portraying and monitoring the computational steps of algorithms and programs. In addition to the natural applicability in the educational field, algorithm animation tools revealed to be useful for debugging and for checking the correctness and the efficiency of algorithms on specific test sets: visualization systems that are able to detect visual events almost automatically can be used as high-level debuggers as unexpected or incorrect visual events may indicate that something is wrong with the execution of the program. We refer to [17, 18, 22] for comprehensive surveys on this topic and for the discussion of experiences with program visualization tools in education and teaching as well as in programming and software engineering.

It has been observed [19, 20] that *smooth animations*, as opposed to *jerky visualizations*, greatly help the viewer maintain context: smooth image transitions make program operations and data modifications more easily perceivable, can be used for capturing the viewer's attention and also for highlighting "semantic" properties of the code. Therefore, software visualization systems (briefly, SV systems) should provide practical facilities for creating and customizing smooth animations.

*This work was partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering".

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. E-mail: `demetres@dis.uniroma1.it`, URL: `http://www.dis.uniroma1.it/~demetres/`.

¶Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. E-mail: `finocchi@dsi.uniroma1.it`, URL: `http://www.dsi.uniroma1.it/~finocchi/`.

Relieving the visualization designer of the task of specifying animations is an appealing but not realistic strategy, as it greatly limits customization possibilities. On the other hand, specifying animations by means of a general-purpose programming language can be tedious and tricky, due to the lack of a specific vocabulary and of high-level abstractions for dealing with graphics. Even providing the programmers with powerful environments and low-level toolkits may have some drawbacks, since these tools usually have high learning curves and building animations with them is often difficult, time-consuming, and error-prone for non experienced designers. Hence, in order to obtain a compromise between flexibility, ease of usage, and customizability, a good choice seems to be supporting an animation language accurately designed to make the specification as simple as possible, to provide default animated behaviors for graphical objects, and to relieve the programmer of the task of specifying too many low-level details related to the presentation of the animation.

One of the most complete and rigorous study in this domain led to the statement of the *path-transition paradigm* [20], a general methodology for creating two-dimensional, real-time, color animations. The path-transition paradigm relies on the use of four abstract data types: location, image, path, and transition. Animations are realized by creating and modifying instances of these data types by means of suitable operations defined on them. The paradigm has an object-oriented style and so it is well-suited for use in *imperative* SV systems, i.e., systems where visualizations are produced by identifying *interesting events* and by annotating the source code with calls to visualization (and animation) routines. In particular, it has been used in an algorithm animation tool called TANGO (Transition-based ANimation GeneratiOn) [21]: the four abstract data types have been implemented as user-defined data types in the C programming language and the operations available for them as a package of calls provided to animation designers. Algorithms from disparate domains have been animated in TANGO, proving the feasibility of the approach to animation used by the system.

As opposed to imperative systems, *declarative* SV tools conceive the visualization as a mapping from program state into images [18] and are usually data (and not event) driven. These systems rely on the assumption that observing how program's variables change provides strong evidence about the actions performed by the program. This relieves animation designers of having a detailed knowledge of the source code, and allows them to focus on the meaning of data structures to realize a visualization. However, the lack of explicit user-defined visual event control makes data driven systems less suited for supporting smooth animations than imperative ones. A first solution has been implemented in the declarative tool Pavane [19], which strengthens the *simple mapping* from program state into images with a differential *animation mapping* and exploits the idea of recording one step of history. Namely, the animation designer defines separate animation rules for detecting appearance, change, or disappearance of each graphical object on the scene: in these rules, both the current and the *previous instance* of the object are accessed, so as to explicitly discover the differences between them. To realize the animation of a specific event, the time-dependent attributes of graphical objects involved in the event are interpolated explicitly between the old and the new values by the animation designer.

Functional approaches to animation have been considered [8, 9], too. Fran (Functional Reactive ANimation) [10] is a collection of data types and functions for specifying interactive multimedia applications. Fran carries on the idea of *temporal modeling*, extending the well-known concept of modeling across the time dimension: starting from static geometric models and images, an animation can be created thanks to the notion of *behavior*, that is a time-varying value of any type (e.g., number-valued, point-valued, even image-valued). Static values can be considered as degenerate behaviors, while the simplest non-static behavior is the time, which is a time-varying numeric value starting at 0 and increasing at the rate of one unit per second. The time behavior is used as the basic unit for creating more complex non-static behaviors, according to a compositional style: for instance, the size of a circle may vary according to the value returned by the number-valued behavior `sin(time)`, obtaining the effect of a cyclically growing and shrinking circle. At last, behaviors in Fran are reactive, i.e., they can be affected by external events such as mouse button presses, supporting in this way different kinds of user interaction.

Another interesting approach for adding animation to graphical objects has been adopted by the Amulet toolkit [15], a user interface development environment for C++. Among the various facilities offered for creating highly-interactive graphical user interfaces, Amulet provides built-in support for animation and other time-based effects [16]. In more detail, *animation constraints* written in C++ are attached to one or more slots of an object, where a slot contains properties of the object such as position or size. The constraints are automatically reevaluated whenever the value of the slot changes, and the value itself is smoothly modified over time. In this way, a good level of modularity and re-usability is reached.

This paper focuses on the problem of specifying animations of algorithms in a declarative style, addressing both automation and customization issues. We propose a general functional approach to animation modeling where animated behaviors of graphical objects are "higher-order" attributes of the objects themselves expressed in the form of mathematical functions. These functions provide temporal and spatial information for creating smooth transitions between graphical scenes. In particular, events of appearance, change, and disappearance of graphical objects are contemplated and the pace of animated transitions can be controlled. Based on these ideas, we describe an extension of the logic-based SV system Leonardo [3] in order to support smooth continuous animation. We also provide practical and flexible solutions for specifying user-defined animation functions and for associating them to graphical objects in the logic-based language ALPHA [6]. The interested reader can find up to date information about Leonardo over the Web at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

Compared to imperative systems, the amount of knowledge of the source code required from animation designers is consistently reduced with our technique: the focus of the visualization specification is on program variables rather than on control flow, thus avoiding to deal explicitly with interesting events. Compared to previous declarative systems, detection of visual events and in-betweening of animation scenes are completely automatic. The crux of our method for automatically detecting and animating scene changes is the use of logical names associated to graphical objects: this is an important novelty with respect to systems such as Pavane, which requires animation designers to compare explicitly successive visual states and arrange for the transition between them.

In Leonardo, automation does not limit customizability of animations: default animated behaviors of graphical objects are provided by the system, and user-defined animation functions can be easily specified. Visualization designers can therefore incrementally improve the quality of animations by overriding the default behaviors, as we illustrate in some algorithm animation examples. Last, but not least, treating animated behaviors as higher-order attributes of graphical objects, similarly to retinal and geometrical features, provides a uniform and well-structured framework for smooth animation.

The remainder of this paper is organized as follows. Section 2 presents our mathematical framework for modeling animations according to a functional style. Section 3 describes an extension of the logic-based architecture of Leonardo in order to support smooth animation: we detail how software modules cooperate and how changes of graphical objects can be automatically detected and animated. Section 4 focuses on the specification of smooth animation in a declarative style, i.e., on the use of declarations encoded in a logic-based language for associating animated behaviors to graphical objects and for controlling general attributes of the visualization system. Algorithm animation examples are presented in Section 5, where interesting issues and questions raised by our approach to animation are also discussed. Finally, the main features and advantages of our approach are summarized in Section 6.

2 A Functional Approach to Animation Modeling

In this section we introduce some notation for describing scenes of graphical objects and we provide a flexible mathematical foundation to animation modeling based on a general functional approach. As we will see in the next sections, this approach is well-suited for use in a declarative visualization setting. We start with elementary mathematical definitions of graphical objects and graphical scenes, and then we incrementally extend our framework to deal with smooth animations of scenes evolving over time.

Without loss of generality, graphical objects can be formally represented as vectors in a multidimensional space of features U that we will refer to as the *visual universe*. Features of graphical objects naturally include geometrical and retinal attributes, but can be extended to represent other kinds of non-visual information, such as temporal or logical information. Informally, a *graphical scene* can be thought of as a subset $S \subseteq U$ of the visual universe U. For simplicity, in the following we consider graphical objects drawn from a basic 2D vocabulary including points, lines, and elementary shapes such as rectangles and ellipses. We assume that each object belongs to a fixed *category* (e.g., `line`) and has a *logical name* that uniquely identifies it in

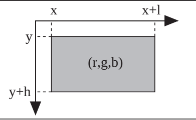
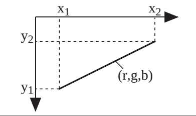
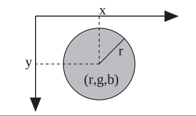
Category	Features	Reference Image
`rectangle`	$F_{rectangle}=\{((x,y),(l,h),(r,g,b))\} \subseteq \Re^2 \times \Re^2 \times [0,1]^3$	
`line`	$F_{line}=\{((x_1,y_1),(x_2,y_2),(r,g,b))\} \subseteq \Re^2 \times \Re^2 \times [0,1]^3$	
`circle`	$F_{circle}=\{((x,y),r,(r,g,b))\} \subseteq \Re^2 \times \Re \times [0,1]^3$	

Figure 1: Some basic graphical objects and their geometrical and retinal features.

a scene together with its category. More formally, we consider the following sets:

- $Categories = \{$**rectangle**, **line**, **circle**, $\ldots\}$ is the set of categories of graphical objects.
- $LogicalNames = \{0, 1, 2, \ldots\}$ is the set of logical names for graphical objects. Sometimes we will refer to these names as *identification numbers*.
- $GraphicalFeatures = \bigcup_{c \in Categories} F_c$ is the set of geometrical and retinal features (also called *attributes*) of graphical objects. Notice that each category c has its own set of graphical features F_c, that is, the graphical features of an object depend on its category. For instance, a possible geometrical feature for a circle is the value of its radius, but it does not make sense to define the same feature for a line. A few examples of basic graphical objects and some related features are shown in Figure 1.

According to the definitions above, a graphical object g can be formally represented by means of a triple (id, c, f), where id is the logical name of g, c represents its category, and f contains its geometrical and retinal features. The set of all such triples is the visual universe U:

$$U = LogicalNames \times Categories \times GraphicalFeatures$$

We can now define an *equivalence relation* $\sim$ over U as follows:

$$\forall\ g, g' \in U \text{ s.t. } g = (id, c, f) \text{ and } g' = (id', c', f'), \text{ we say that } \ g \sim g' \iff (id = id') \wedge (c = c')$$

This means that two graphical objects g and g' drawn from the visual universe U are equivalent if and only if they have the same logical name and the same category. Based on this relation, we formally define a graphical scene S to be a subset of the visual universe U such that $\forall\ g, g' \in S$: $g \not\sim g'$, i.e., we require objects in a scene to be logically different from each other according to the equivalence relation $\sim$. In this way, each object in a scene is completely and univocally identified by its category and its logical name. However, the same logical name can be associated to objects with different categories.

An *interpolating sequence* of two different scenes S_A and S_B is a sequence $\sigma_k = S_0 \ldots\ S_{k+1}$ of scenes such that $S_0 = S_A$ and $S_{k+1} = S_B$. We refer to S_A and S_B as *control scenes* for σ_k and to $S_1\ \ldots\ S_k$ as *inter-scenes*. The animation produced by rendering and sequentially displaying the scenes in σ_k is perceived by a human observer as *smooth* if the attributes of graphical objects are transformed along paths described by continuous functions and the frames generation rate is at least $10-15$ frames per second. Three situations may arise when passing from a scene S_A to a scene S_B (see Figure 2):

1. *Birth* of objects: graphical objects not existing in S_A appear in S_B.
2. *Changes* of objects: graphical objects change one or more of their features, such as dimension, color, or position, but stay the same according to $\sim$.

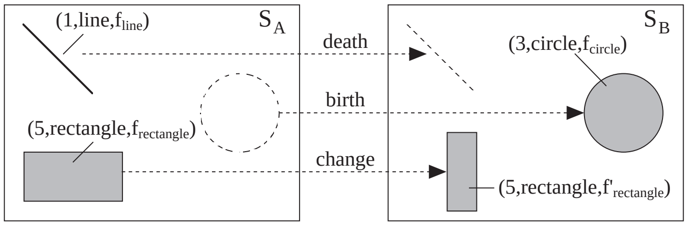


Figure 2: Possible differences between graphical scenes.

3. *Death* of objects: objects existing in S_A disappear from S_B.

Changes of objects can be naturally animated by interpolating the old and the new values of their features along continuous curves. On the other hand, the same interpolation technique cannot be applied in cases 1 and 3, since one end-point of the transformation is undefined. In order to solve this problem, we can think of the birth/death of a graphical object g as a particular change of its graphical attributes from/into the attributes of an *imaginary object* g_b/g_d. Imaginary objects are fictitious start and end points in the "life" of g and may be obtained from g itself by modifying some of its features. This choice makes it possible to customize the animation of the appearance/disappearance of a graphical object and to deal with such events as they were regular object changes.

Figure 3 shows six control scenes $S_0 \ldots S_5$ related to the history of a rectangle that appears, evolves over time, and eventually disappears. Figure 4 is obtained from Figure 3 by superimposing $S_0 \ldots S_5$ together with four dimmed inter-scenes for each pair (S_i, S_{i+1}). The smooth birth behavior of the rectangle g^{t_1} is obtained by giving g_b the same x-coordinate, length, and height as g^{t_1}, but opposite sign to the y-coordinate: in this way, g^{t_1} appears by falling down from the top edge of the viewport. Changes of both position and dimension are smoothly animated by linear interpolation. At last, g^{t_4} collapses into its central point, as the top left corner of g_d lies in the center of g^{t_4} and both its length and height are equal to zero.

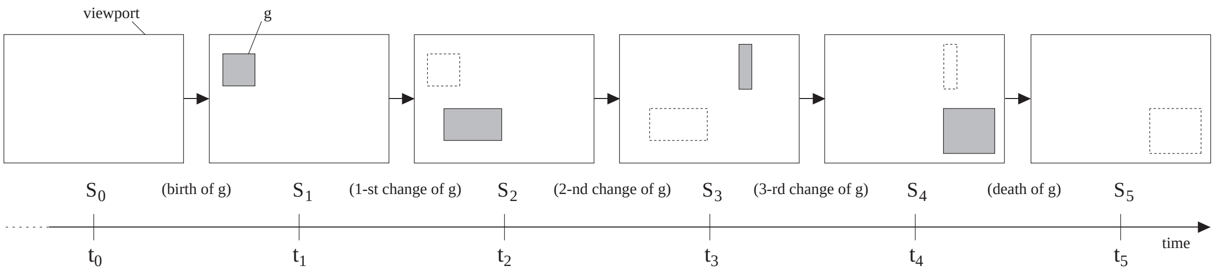


Figure 3: History of a rectangle: birth, evolution, and death

In the following we discuss in detail how to model animations of graphical objects according to a functional approach that consists of attaching to graphical objects additional information in the form of mathematical functions. These functions encode complete implicit information for creating interpolating sequences and can be considered as *higher-order* attributes of graphical objects. This approach is naturally well-suited for declarative specification and makes it possible to define highly customizable animations.

2.1 Synthesis Functions

In this section we deal with the appearance/disappearance of graphical objects in/from a scene. As already stated, such events can be reduced to regular object changes if any appearing/disappearing object g is transformed from/into a corresponding imaginary object g_b/g_d. To this aim, a straightforward way of generating imaginary objects consists of modifying some attributes of g such as its size or position: typically, an imaginary object will have null size or will fall outside the current viewport. This modification can be

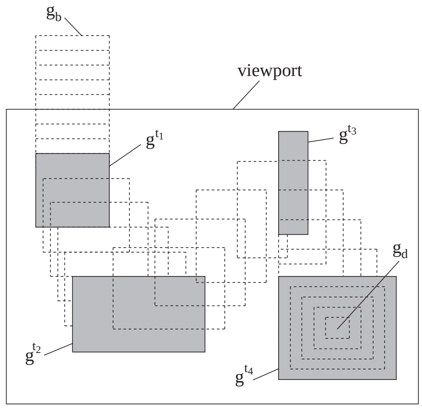


Figure 4: Smooth animation of the life of a rectangle.

represented through a *synthesis function* s that specifies which attributes have to be changed and how. The set of synthesis functions can be defined as follows:

$$Synth = \{s : \ GraphicalFeatures \rightarrow GraphicalFeatures : f \mapsto f'\}$$

As objects belonging to different categories may have different features, synthesis functions for transforming any input features f into new features f' must agree with the category of the involved objects. In order to deal with events of birth and death, we equip each graphical object in a scene with a pair of synthesis functions for generating imaginary objects. The extended definition of visual universe is given below:

$$U = LogicalNames \times Categories \times GraphicalFeatures \times Synth \times Synth$$

A graphical object $g \in U$ has now the form (id, c, f, s_b, s_d). The imaginary object g_b which g is transformed from when appearing into the graphical scene can be then obtained from g in the following way:

$$g_b = (id_b, c_b, f_b, -, -) \quad \text{where:} \begin{cases} id_b = id \\ c_b = c \\ f_b = s_b(f) \end{cases}$$

The imaginary object g_d which g is transformed into when disappearing from the graphical scene can be constructed similarly. Note that imaginary objects do not require any synthesis functions as they provide just a chance for other objects to smoothly appear or disappear. Moreover: $g \sim g_b$ and $g \sim g_d$, because they maintain the same logical name and the same category.

As an example, the synthesis functions for computing the imaginary objects g_b and g_d related to the birth and death of the evolving rectangle shown in Figure 4 can be defined as follows:

$$s_b : ((x, y), (l, h), (r, g, b)) \mapsto ((x, -h), (l, h), (r, g, b))$$
$$s_d : ((x, y), (l, h), (r, g, b)) \mapsto \left((x + \tfrac{l}{2}, y + \tfrac{h}{2}), (0, 0), (r, g, b)\right)$$

An advantage of the presented approach is that we do not need to encode explicitly the attributes of imaginary objects, but only to provide the know-how that makes it possible to synthesize them. This is very useful in a context where attributes of graphical objects may change over time, as in algorithm animation.

2.2 Interpolation Functions

In this section we focus on the problem of modeling interpolating sequences of graphical scenes. We first give some mathematical preliminaries on parametric curves in the plane and then we detail how $interpolation functions$ can be used to produce sequences of intermediate objects for smoothly animating changes of graphical attributes over time. Finally, we further extend the definition of visual universe by assuming that each graphical object in a scene is equipped with an interpolation function.

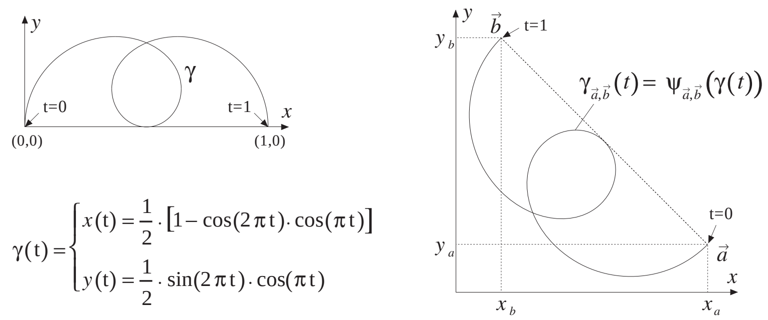


Figure 5: Transforming a canonical curve into a generic interpolating curve.

2.2.1 Curves in the Plane for Modeling Animation Paths

Let $\Gamma = \{\gamma : [0,1] \to \mathcal{R}^2\}$ be a family of continuous parametric curves in the plane. An *interpolating curve* of two points $\vec{a}$, $\vec{b} \in \mathcal{R}^2$ is a curve $\gamma_{\vec{a},\vec{b}} \in \Gamma$ such that $\gamma_{\vec{a},\vec{b}}(0) = \vec{a}$ and $\gamma_{\vec{a},\vec{b}}(1) = \vec{b}$. In particular, a *canonical interpolating curve* is a curve $\gamma \in \Gamma$ such that $\gamma(0) = (0,0)$ and $\gamma(1) = (1,0)$.

Let $\psi_{\vec{a},\vec{b}} : \mathcal{R}^2 \to \mathcal{R}^2 : (x,y) \mapsto (x',y')$ be a transformation pipeline that rotates, re-scales and translates points in the plane:

$$\psi_{\vec{a},\vec{b}}(x,y) = \begin{cases} x' = x\cdot\Delta_x - y\cdot\Delta_y + x_a \\ y' = x\cdot\Delta_y + y\cdot\Delta_x + y_a \end{cases} = (x,y)\begin{pmatrix} \Delta_x & \Delta_y \\ -\Delta_y & \Delta_x \end{pmatrix} + (x_a, y_a)$$

where: $\Delta_x = x_b - x_a$ and $\Delta_y = y_b - y_a$. Function ψ allows us to stretch any canonical interpolating curve γ as to become an interpolating curve $\gamma_{\vec{a},\vec{b}}$ for any pair of points $\vec{a}$ and $\vec{b}$:

Proposition 1 *Let $\gamma \in \Gamma$ be a canonical interpolating curve. For any pair of points $\vec{a}$, $\vec{b} \in \mathcal{R}^2$, $\gamma_{\vec{a},\vec{b}}(t) = \psi_{\vec{a},\vec{b}}(\gamma(t))$ is an interpolating curve of $\vec{a}$ and $\vec{b}$.*

Proof. $\gamma_{\vec{a},\vec{b}}$ is trivially a continuous parametric curve in the plane as it is obtained through a composition of continuous functions. Furthermore:

$$\gamma_{\vec{a},\vec{b}}(0) = \psi_{\vec{a},\vec{b}}(\gamma(0)) = \psi_{\vec{a},\vec{b}}(0,0) = \vec{a}$$
$$\gamma_{\vec{a},\vec{b}}(1) = \psi_{\vec{a},\vec{b}}(\gamma(1)) = \psi_{\vec{a},\vec{b}}(1,0) = \vec{b} \qquad \square$$

Figure 5 shows an example of this transformation operated on a Grandi's rose [14].

Pairs of interpolating curves having a common end point can be concatenated to form more complex ones. Let $\pi_n = \vec{p}_0, \gamma_0, \vec{p}_1, \gamma_1, \ldots, \vec{p}_{n-1}, \gamma_{n-1}, \vec{p}_n$ be an interleaved sequence of $n+1$ points $\vec{p}_0 \ldots \vec{p}_n$ and n canonical curves $\gamma_0 \ldots \gamma_{n-1}$. π_n can be used for defining a curve $\gamma_{\pi_n} \in \Gamma$ as follows:

$$\gamma_{\pi_n}(t) = \begin{cases} \psi_{\vec{p}_0,\vec{p}_1}(\gamma_0(n\cdot t)) & if\ 0 \le t \le \frac{1}{n} \\ \psi_{\vec{p}_1,\vec{p}_2}(\gamma_1(n\cdot t - 1)) & if\ \frac{1}{n} < t \le \frac{2}{n} \\ \ldots & \\ \psi_{\vec{p}_{n-1},\vec{p}_n}(\gamma_{n-1}(n\cdot t - n + 1)) & if\ \frac{n-1}{n} < t \le 1 \end{cases}$$

Note that γ_{π_n} is an interpolating curve for $\vec{p}_0$ and $\vec{p}_n$ and is forced to pass through all intermediate points $\vec{p}_1 \ldots \vec{p}_{n-1}$, as in the example shown in Figure 6.

In conclusion, basic curves can be described by means of their parametric equations, while more complex curves can be obtained as concatenation of basic ones joint in predefined points. We remark that other solutions may be provided, e.g., based on cubic splines or Bezier curves [11].

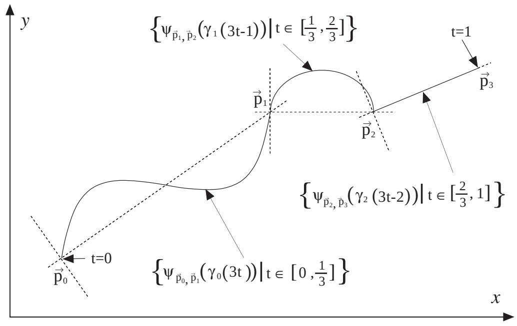


Figure 6: Complex curve obtained by concatenation of simpler ones.

2.2.2 Interpolation of Control Scenes

Using paths to control an animation is a well-accepted idea [1, 12]. Hence, in the following we base on the concept of interpolating curve in order to define formally interpolating sequences of control scenes. Let (S_A, S_B) be an ordered pair of control scenes in U issued during an animation process that generates S_A before S_B. Without loss of generality, we assume that the following property holds:

$$P(S_A, S_B) = [\forall\ g \in S_A,\ \exists g' \in S_B :\ g \sim g' \quad \wedge \quad \forall\ g' \in S_B,\ \exists g \in S_A :\ g' \sim g]$$

For brevity we say that S_A and S_B are *equivalent* if and only if they satisfy property P: indeed, if this is the case, S_A and S_B contain the same objects according to the equivalence relation $\sim$, even if these objects may have different graphical features. If $P(S_A, S_B)$ is false, then there is some object $a \in S_A$ such that no object $a' \in S_B$ is equivalent to a, or there is some object $b \in S_B$ such that no object $b' \in S_A$ is equivalent to b. These cases correspond to the disappearance of a from S_A and to the appearance of b in S_B, respectively. As we have already pointed out, the concept of imaginary object allows us to treat these events as they were simple changes of graphical attributes. We can then complete the scenes S_A and S_B with imaginary objects and define two new scenes $\widehat{S}_A$ and $\widehat{S}_B$ as follows:

$$\widehat{S}_A = S_A \cup \{g_b \mid g \in S_B \ \wedge\ \nexists g' \in S_A \text{ s.t. } g' \sim g\}$$
$$\widehat{S}_B = S_B \cup \{g_d \mid g \in S_A \ \wedge\ \nexists g' \in S_B \text{ s.t. } g' \sim g\}$$

g_d and g_b are the birth and death imaginary objects obtained from g as explained in Section 2.1. $\widehat{S}_A$ and $\widehat{S}_B$ always satisfy property $P(\widehat{S}_A, \widehat{S}_B)$, due to the fact that imaginary objects are always equivalent to the objects they are generated from.

Assuming to deal with equivalent scenes, we now define the set of *interpolation functions*:

$$Inter = \{p : GraphicalFeatures \times GraphicalFeatures \times [0,1] \rightarrow GraphicalFeatures : (f_1, f_2, t) \mapsto f\}$$

An interpolation function p takes as input two feature vectors f_1 and f_2 by the same category and a real parameter $t \in [0, 1]$. It uniformly produces a feature vector f by sampling at t interpolating curves for the attributes in f_1 and f_2.

As for synthesis functions, interpolation functions depend on the category of involved objects. If we assume that each graphical object in a scene is equipped with an interpolation function p we can further extend the visual universe U:

$$U = LogicalNames \times Categories \times GraphicalFeatures \times Synth \times Synth \times Inter$$

Hence, any graphical object $g \in U$ has the form: (id, c, f, s_b, s_d, p). Let $t_i = \frac{i}{k+1}$ be a sequence of $k+2$ *sampling points* such that $t_0 = 0$ and $t_{k+1} = 1$, where k is a constant and $0 \leq i \leq k+1$. An *interpolating sequence* $\sigma_k(S_A, S_B)$ of any two equivalent control scenes S_A and S_B is defined as follows:

$$\sigma_k(S_A, S_B) = (S_0, \ldots, S_{k+1}) \text{ where}$$

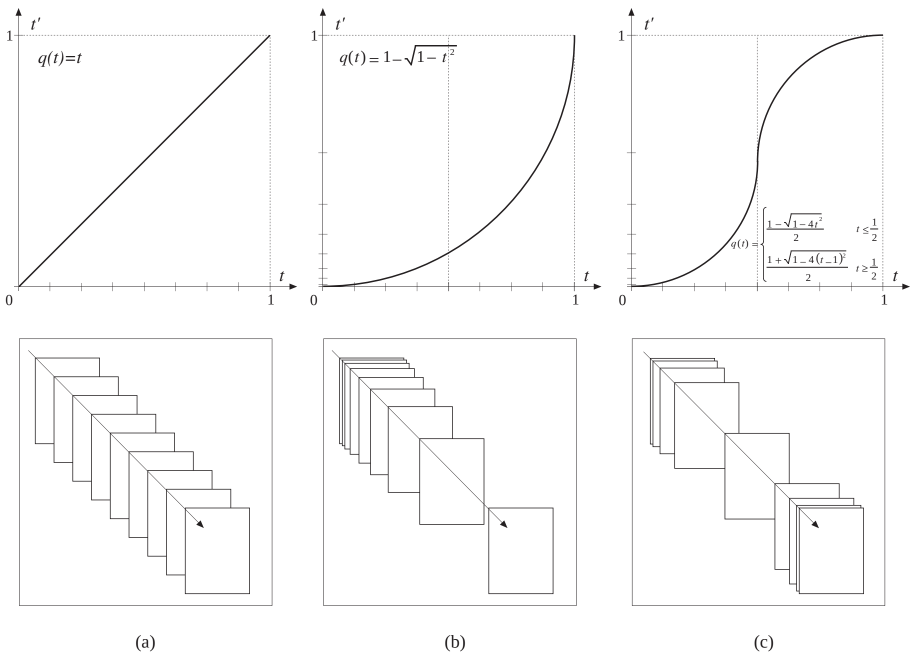


Figure 7: Examples of pace functions and their animation effect.

$$S_i = \{\ (id, c, p(f, f', t_i), -, -, -) \in U\ s.t.\ g = (id, c, f, -, -, p) \in S_A,\\ g' = (id', c', f', -, -, p') \in S_B, \text{ and } g \sim g'\}$$

$\sigma_k(S_A, S_B)$ is an ordered sequence of $k+2$ scenes such that $S_0 = S_A$ and $S_{k+1} = S_B$. The features vector of any object in any scene S_i is equal to $p(f, f', t_i)$, i.e., is generated from the start and end features f and f' by means of the interpolation function p of the starting object: this implies that birth imaginary objects must keep a copy of the interpolation function of the objects they are generated from. Finally, note that $p(f, f', t_0) = f$ and $p(f, f', t_{k+1}) = f'$ by definition of interpolating curve.

2.3 Pace Functions

In the previous section we defined interpolating sequences of scenes assuming that attributes of intermediate graphical objects are obtained by sampling interpolating curves at regular intervals through a succession of sampling points t_i. This means we have no direct control on the temporal modeling of animation: it is not possible, for example, to specify animation behaviors where some portions of an interpolating curve are traversed faster than others. However, slowing effects have been proved to be very useful for capturing the attention of the viewer, especially when applied at the beginning and at the end of a motion [13]. This effect is usually called slow-in/slow-out and is shown in Figure 7(c).

A *pace function* can be informally seen as “a way to specify the rate at which a curve is traversed” [23]. In this section we complete the definition of visual universe by assuming each object in a scene is assigned with a pace function q. As we will see, these functions can be composed with interpolation functions in order to control the temporal animated behavior of graphical objects. The set of pace functions is defined as follows:

$$Pace = \{q\ :\ [0,1] \to [0,1]\ :\ t \mapsto t' \text{ s.t. } (q(0) = 0) \wedge (q(1) = 1)\}$$

Even if we do not strictly require pace functions to be continuous functions, we remark that discontinuity points may compromise the smoothness of the animation. In the remainder of this paper, we assume that the visual universe is defined as follows:

$$U = LogicalNames \times Categories \times GraphicalFeatures \times Synth \times Synth \times Inter \times Pace$$

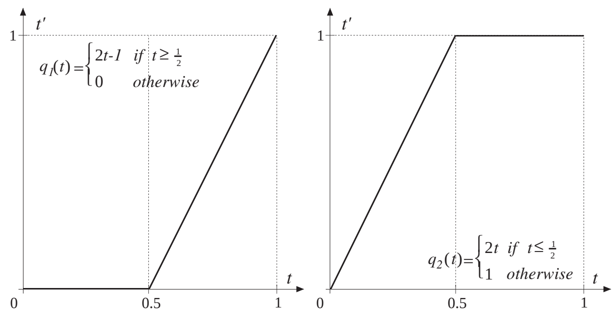


Figure 8: Pace functions for relative animation timing.

According to this extension, any graphical object g has the form: $(id, c, f, s_b, s_d, p, q)$. Thus, we can refine the definition of interpolating sequence $\sigma_k(S_A, S_B)$ by assuming:

$$S_i = \{\ (id, c, p(f, f', q(t_i)), -, -, -, -) \in U\ s.t.\ g = (id, c, f, -, -, p, q) \in S_A,$$
$$g' = (id', c', f', -, -, p', q') \in S_B, \text{ and } g \sim g'\}$$

The attributes of intermediate objects in S_i are obtained by composing the interpolation function p and the pace function q. This simple solution provides flexibility in the temporal modeling of animations, and makes it possible to customize even the relative animation timing of different objects in a scene. In a more complex scenario, however, graphical objects may have separate pace functions for different kinds of attributes, and more than one composition level may be allowed.

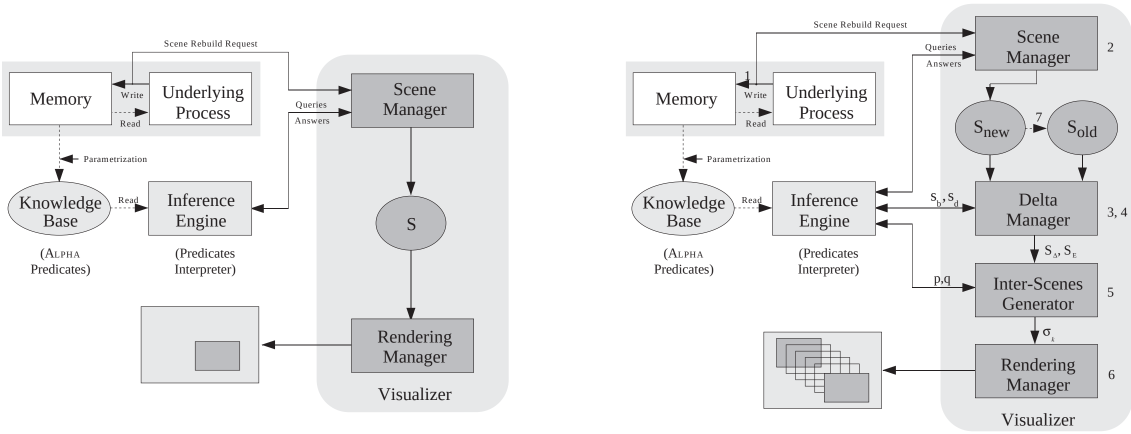
Figure 7 shows three pace functions and the animation effects they produce on a rectangle that moves along a straight path: the rectangle in (a) moves at a constant speed, while the ones in (b) and (c) act as they were subject to a force and progressively accelerate/decelerate along the way. The two pace functions q_1 and q_2 presented in Figure 8 can be used in order to obtain *sequential animations* of groups of objects in a scene. For instance, let us assume that q_1 and q_2 are assigned to different graphical objects g_1 and g_2, respectively. If both g_1 and g_2 change when passing from one control scene to the successive one, the whole transformation of g_1 is compressed in the first $\frac{k}{2}$ inter-scenes, while g_2 changes only during the successive $\frac{k}{2}$ ones, giving the effect g_1 is animated before g_2. Similarly, it is easy to design sets of pace functions with overlapping or even pace functions with oscillations.

3 Embodiment in a Working System

In this section we present an embodiment of the mathematical framework for smooth animation described in Section 2 in the software architecture of the logic-based SV system Leonardo [3]. Before discussing the extensions of the architecture, which have been implemented in Leonardo 3.5, we summarize the fundamental aspects of Leonardo's approach, referring the interested reader to [3, 6] for a detailed description.

In the logic-based approach, a visualization is specified by means of a set of logic assertions contained in a knowledge base. Such assertions describe a scene by declaring the existence of graphical objects and their attributes. As shown in Figure 9(a), they can depend upon the content of variables of the underlying program to be animated (we call this aspect *parametrization*). As we will detail in Section 4, assertions are encoded by means of definitions of *predicates* in the logic-based language ALPHA [6]: ALPHA predicates are Boolean functions with by name/by value parameters computed according to a backtracking-based Prolog-like computation model. Some of them, called *standard predicates*, have fixed predefined prototypes and are expected to return information about graphical objects in their parameters when computed on Visualizer's demand. Definitions of standard predicates are written by the animation designer to specify a visualization.

A module of the Visualizer, the Scene Manager, retrieves visual information from the knowledge base by querying an Inference Engine as shown in Figure 9(a). The interaction in Leonardo between modules of the Visualizer, Inference Engine, and running program can be summarized as follows:



(a) (b)

Figure 9: (a) Logic-based visualization system of Leonardo [3]; (b) Extension of the logic-based visualization system with smooth animation capabilities as implemented in Leonardo 3.5.

1. When the process being animated modifies the content of a variable referred to by an assertion in the knowledge base, a *scene rebuild* request is sent to the Scene Manager and the process is suspended.
2. The Scene Manager asks the Inference Engine for visual information that is retrieved from the knowledge base and builds a scene S that is displayed by the Rendering Manager.
3. The underlying process is resumed.

We remark that the temporal evolution of the underlying process' computation state may imply a continuous change of the information that the Visualizer is able to retrieve from the same knowledge base, ending up in different scenes being visualized over time. One of the main advantages of attaching an atemporal declarative description of graphical objects to an imperative specification of an underlying computation is that graphical objects never need to be explicitly created nor destroyed by the programmer: their existence is simply controlled by a dynamically reevaluated condition that may be true or false at any time. This guarantees a highly reliable programming style by providing strong separation between graphics and algorithms and makes it easier to portray the behavior of programs with respect to their data manipulations.

We now present an architectural extension of Leonardo that allowed us to support smooth scene transformations. The main novelties introduced in Leonardo 3.5 are:

- Assertions in the knowledge base describe an extended scene by declaring not only the existence of graphical objects and their geometrical and retinal features, but also their higher-order attributes, i.e., their synthesis, interpolation, and pace functions. The use of Alpha predicates for specifying animation functions is discussed in Section 4.
- The Visualizer is provided with the capability of generating automatically interpolating sequences of consecutive scenes. In particular, its internal structure is extended with two new modules, called *Delta Manager* and *Inter-Scenes Generator*, respectively.

The complete logic-based architecture for portraying the execution of an underlying program by means of smooth, continuous animation is shown in Figure 9(b). The cooperation between the different modules is now carried out as follows:

1. When the process being animated modifies the content of a variable referred to by an assertion in the knowledge base, a *scene rebuild* request is sent to the Scene Manager and the process itself is suspended.
2. The Scene Manager asks the Inference Engine for visual information that is retrieved from the knowledge base, and builds an extended scene S_{new}.

3. Provided S_{old} is the scene currently visualized on the screen (initially empty), the Delta Manager compares S_{new} with S_{old}, decides which objects are appearing, which are changing, and which are vanishing, and produces two lists of changed and unchanged objects (S_Δ and S_E, respectively). This is done by comparing the old and the new sets of objects' identification numbers according to their category. In case of death of an object g, the Delta Manager uses the synthesis function s_d to compute the attributes $f_d = s_d(f)$, used as they were new attributes for the vanished graphical object. Analogous considerations apply in case of birth of objects.

4. If $S_\Delta = \emptyset$ the control jumps to the last point.

5. The Inter-Scenes Generator, provided with both S_E and S_Δ, creates an interpolating sequence $\sigma_k = S_0 \ldots S_{k+1}$ of S_{old} and S_{new} using the information carried on by the interpolation functions p and the pace functions q. Namely, unchanged objects are directly copied from S_E to each intermediate scene. Moreover, each inter-scene S_i contains new objects whose features are generated by appropriately computing the interpolation function p composed with the pace function q.

6. $S_1 \ldots S_{k+1}$ are then rendered and sequentially displayed by the Rendering Manager.

7. S_{new} is copied onto S_{old}.

8. The underlying process is resumed.

We remark that the crux of our method for automatically detecting and animating scene changes is the use of logical names associated to graphical objects, and this is an important novelty with respect to previous declarative systems such as Pavane. In Section 4 we will show that synthesis, interpolation, and pace functions have their own logical names, too. This allows the animation designer to refer quickly to basic functions contained in the libraries of default animated behaviors that come with Leonardo, without sacrificing the possibility of specifying new customized behaviors, whenever required.

We finally point out that the manipulation of scenes S_{new} and S_{old} - and even their existence - is kept completely hidden from the animation designer. This is another important difference with respect to the approach followed by Pavane, which consists of providing the animation designer with a history-sensitive internal representation of the data to be visualized: *verbatim* from [5], "such an internal representation is almost *de rigueur* when animation (smooth visual motion) is involved, since the animator must be able to compare the two successive visual states and arrange for the transition between them". The animation designer in Pavane is therefore required to manage explicitly the previous instance of the visualization space to produce animations, while in our framework this step is completely automatic. As a consequence, our approach abstracts and simplifies the task of the animation designer, that has only to deal with animation functions as they were normal attributes of graphical objects.

4 Specifying Smooth Animation in Alpha

In Section 3 we introduced Alpha predicates as Boolean functions with by name/by value parameters. In general, although they are not targeted for functional specification since we designed them for declaring information about graphical objects, they can be used for calculating general-purpose mathematical functions.

In this section we present a new set of standard predicates that have been introduced in Leonardo 3.5 for specifying synthesis, interpolation, and pace functions and for associating them to graphical objects. We do not discuss scene modeling in Alpha, and we refer the interested reader to [6] for details: we only assume that a graphical scene already exists and each graphical object has a unique identification number which makes it possible to refer to it. We show the prototypes of new Alpha standard predicates related to geometrical attributes of rectangles. Similar predicates for retinal features and for other graphical categories are completely analogous. We also propose new Alpha predicates for specifying global attributes of the visualization system.

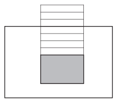
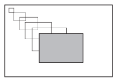
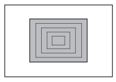
Definitions	Effect
`#define TOP_ -1` `#define LEFT_TOP_GROW_ -2` `#define GROW_ -3`	—
`RectangleSynth(TOP_,` `X,Y,L,H,` `Out Xs,Out Ys,Out Ls,Out Hs)` `Assign Xs=X Ys=-H Ls=L Hs=H;`	
`RectangleSynth(LEFT_TOP_GROW_,` `X,Y,L,H,` `Out Xs,Out Ys,Out Ls,Out Hs)` `Assign Xs,Ys,Ls,Hs=0;`	
`RectangleSynth(GROW_,` `X,Y,L,H,` `Out Xs,Out Ys,Out Ls,Out Hs)` `Assign Xs=X+L/2 Ys=Y+H/2 Ls,Hs=0;`	

Figure 10: A small library of definitions of RectangleSynth and their effect.

4.1 Specifying Synthesis Functions

Signatures of Alpha predicates for computing synthesis functions and for associating them to rectangles are given below:

```
RectangleSynth(SynthID,X,Y,L,H,Out Xs,Out Ys,Out Ls,Out Hs)
RectangleBirth(RectID,Out SynthID,V)
RectangleDeath(RectID,Out SynthID,V)
```

The Scene Manager invokes `RectangleBirth` and `RectangleDeath` in order to associate to each rectangle `RectID` in a view `V` a birth and death behavior, represented by a number `SynthID` identifying a synthesis function. If such predicates have not been defined, return an invalid `SynthID`, or fail (i.e., return false in their name) for some rectangle, the Scene Manager provides default values.

Any definition of the predicate `RectangleSynth` provided by the animation designer is expected to synthesize the geometrical attributes of an imaginary rectangle: it receives a number `SynthID` identifying a synthesis function and a list of geometrical attributes and returns a new list of attributes computed by modifying the input ones according to the value of `SynthID`. Definitions of `RectangleSynth` are easily reusable: it is possible to have several predefined `SynthIDs`, corresponding to libraries of different definitions of this predicate, in order to provide the most common behaviors. Hence, in most cases, the only task of the animation designer consists of defining `RectangleBirth` and `RectangleDeath`, without writing any new synthesis function. Note that the same `SynthID` can be specified for both appearance and disappearance events, producing "specular" effects. Figure 10 shows multiple definitions of `RectangleSynth` related to different `SynthIDs` and the animations obtained through them when a rectangle appears or disappears. Superimposed rectangles have been generated by linear interpolation.

4.2 Specifying Interpolation Functions

Signatures of Alpha predicates for defining curves in the plane are given below:

```
Curve(CurveID,float T,Out float X,Out float Y)
ConcatCurve(CurveID,Xa,Ya,Xb,Yb,Out Xi,Out Yi,Out SubCurveID)
```

These predicates are invoked by the Inter-Scenes Generator in order to produce interpolation scenes. Any instance of the predicate `Curve` defines a parametric canonical curve γ: `Curve` takes a number `CurveID` identifying the curve γ and a floating point value T$\in [0,1]$ and returns a pair (X,Y) such that (X,Y)=γ(T).

Similarly, any instance of `ConcatCurve` defines a sequence π_n that identifies an interpolating curve γ_{π_n}. This sequence π_n consists of a list of points (Xi,Yi) and curve identification numbers `SubCurveID`. The Inter-Scenes Generator computes it by calling `ConcatCurve` repeatedly as long as it is able to return new values. This powerful feature is a consequence of the backtracking-based computation model of Alpha

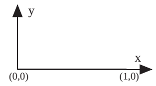
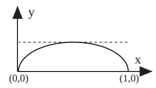
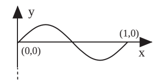
Definitions	Curves
`#define LINEAR_ -1` `#define ARC_ -2` `#define SIN_ -3`	—
`Curve(LINEAR_,` `float T,Out float X,Out float Y)` `Assign X=T Y=0;`	
`Curve(ARC_,` `float T,Out float X,Out float Y)` `Assign X=T Y=sqrt(1-sqr(2*T-1))/2;`	
`Curve(SIN_,` `float T,Out float X,Out float Y)` `Assign X=T Y=sin(2*pi*T)/(2*pi);`	

Figure 11: A small library of curve definitions.

predicates. In addition to the identification number `CurveID` of the curve itself, this predicate receives from the Inter-Scenes Generator the end points `(Xa,Ya)-(Xb,Yb)` of the sequence: these values may be useful for producing the list of `Xi`, `Yi`, and `SubCurveID`.

As for the synthesis functions described in the previous section, both `Curve` and `ConcatCurve` can be predefined in a library of predicates (see, e.g., Figure 11).

In addition to predicates for defining curves, we need predicates for assigning curves to position and dimension of rectangles. These predicates receive the logical name `RectID` of the rectangle and its enclosing view `V` and return the identification number `CurveID` of a curve:

```
RectanglePosPath(RectID,Out CurveID,V)
RectangleDimPath(RectID,Out CurveID,V)
```

If such predicates are not defined, generate an invalid `CurveID`, or fail for some rectangle, the Scene Manager should provide a default `CurveID`. Note that interpolation functions of graphical objects are specified by assigning separate interpolating curves for each kind of attribute. An example showing how these predicates work will be given in Section 5.

4.3 Specifying Pace Functions

We now give signatures of standard predicates for specifying pace functions and for assigning them to graphical objects. A pace predicate in Alpha takes as input a user-defined `PaceID` identifying a pace function q and a real value `inT`, and returns a real value `outT` such that `outT`$=q($`inT`$)$. The signature of the predicate for computing pace functions in Alpha is given below:

```
Pace(PaceID,float inT,Out float outT);
```

User-defined instances of predicate `Pace` are called by the Inter-Scenes Generator for computing interpolating sequences. The input value `inT` is assumed to be in the range $[0,1]$, and it is up to the animation designer to ensure that the same condition holds for `outT`: however, a robust Inter Scenes Manager should cope with programming errors by distorting any user-defined value to fit into the range $[0,1]$.

Below we propose signatures of Alpha standard predicates for assigning pace functions to graphical objects in a view. As for interpolation predicates, we focus on position and size attributes of rectangles:

```
RectanglePosPace(R,Out PaceID,V)
RectangleDimPace(R,Out PaceID,V)
```

Note that pace functions are assigned to graphical objects by specifying separate IDs for each kind of attribute. This is indeed more powerful than assigning the same pace function to all attributes, as we did in the mathematical framework presented in Section 2.3.

The proposed solution for dealing with pace functions is very robust, but it may be extremely CPU-time consuming: in fact, for each varying object, which has assigned a given pace ID, the inference engine has to search and compute the corresponding predicate matching that ID. Thus, if x different pace predicates matching x different IDs are currently defined, and there are n changing objects involved in a transition, the total time spent just in searching a pace predicate while building an interpolating sequence σ_k is $O(x \cdot n \cdot k)$. It turns out that, for large values of x, n, and k, the animation may result too slow. A more efficient approach could be that of having regular C functions instead of ALPHA predicates for computing pace functions, and then of assigning graphical objects no more with logical identification numbers, but with pointers to those C functions. This solution is straightforward as there is no need of searching any matching predicate, but it is error-prone, due to the possibility of assigning wrong pointers to graphical objects that might cause the system to crash. A solution could consist of having some ID numbers (e.g. from 0 to 10) allocated for the most commonly used pace functions, and then of keeping inside the system a table of pointers to those built-in functions. The system should be then responsible of deciding, according to the ID of the pace function, if searching and computing an ALPHA predicate that implements a user-defined function, or directly invoking a built-in C function by indexing the internal table of pointers through the ID. A more sophisticated addition to the foregoing solutions could be that of caching at run-time previously computed values of pace functions in order to avoid recomputing from scratch the same values for each request. This can be efficiently done through hash tables. Similar considerations apply also to synthesis and interpolation functions.

4.4 Specifying Attributes of the Visualization System

So far, we modeled interpolating sequences of control scenes by assuming some global parameters, such as the number k of inter-scenes per sequence, are given *a priori*. As a matter of fact, this assumption was not restrictive for developing our mathematical framework.

In general, however, it is interesting to think of a graphic system as a special object having its own attributes just as we think of a graphical object as having a color and a size. Below we propose a list of general attributes for the visualization system described in Section 3, and we give signatures of ALPHA standard predicates for declaring their value:

- *Scene rebuild flag*: a Boolean value that describes the status of the Scene Manager. If true, the underlying process gives the control to the Scene Manager every time a variable referred to by an ALPHA predicate has been modified, and the Scene Manager rebuilds the current graphical scene after consulting the Inference Engine. If false, the underlying process is never interrupted and no change of the currently visualized scene ever occurs, which means that the graphic system is frozen. We assume that the scene rebuild flag is true if the predicate `SceneRebuildOn` is declared, false otherwise. Controlling the Scene Manager status is extremely useful for reducing the number of control scenes, and invalidating `SceneRebuildOn` might be strictly required as long as there are variables looked by predicates that hold temporary or uninitialized data.

- *Screen update flag*: a Boolean value that describes the status of the Rendering Manager. If false, the Rendering Manager is frozen and no frame is ever displayed, even if the Scene Manager is active. We assume the screen update flag is true if the predicate `ScreenUpdateOn` is declared, false otherwise. Unlike `SceneRebuildOn`, `ScreenUpdateOn` can only be used for reducing the number of control scenes.

- *Smooth animation flag*: a Boolean value that describes the status related to the smooth animation capabilities of the visualization system. If false, the Rendering Manager displays only the last scene of any interpolating sequence which corresponds to the new scene and displays no inter-scene. In this case, only the control scenes are displayed and the result might be a jerky animation. If true, all the inter-scenes are displayed over time. We assume the smooth animation flag is true if the standard predicate `SmoothAnimationOn` is declared, false otherwise.

- *Number of inter-scenes*: number k of inter-scenes generated and displayed by the visualization system during each transition. The number of inter-scenes can be declared through the standard predicate `InterFrames(Out K)`.

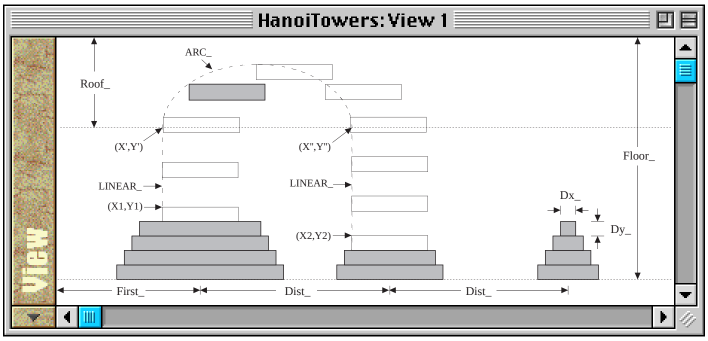


Figure 12: The towers of Hanoi example.

- *Inter-frame delay*: minimum amount of time the Rendering Manager has to wait before displaying any new scene in a sequence. The inter-frame delay can be declared through the standard predicate `InterFrameDelay(Out D)`. Generally, in order to achieve the best fine-tuning of animation timing, the delay measure unit should depend upon the target machine.

5 Algorithm Animation Examples

In this section we present some smooth animation examples that have been realized with Leonardo 3.5. First, we show how to animate a C program for solving the towers of Hanoi problem. We follow an incremental construction of the animation, starting from a simple jerky visualization of control scenes and then providing graphical objects with animated behaviors, both default and customized ones. As a second example, we focus on animating sorting algorithms and we discuss some interesting issues and questions raised by our declarative animation technique. We conclude with an example related to Graham's algorithm for computing the convex hull of a set of points in the plane.

5.1 Animation of the Towers of Hanoi Challenge

The towers of Hanoi challenge can be stated as follows: consider three pegs and a number of discs of different sizes. Initially, all the discs are stacked on a single peg, with decreasing sizes from bottom to top. The problem consists of transferring all the discs, one by one, to a target peg, paying attention not to put larger discs over smaller ones.

Though the algorithm for solving the towers of Hanoi challenge is very simple, we do not show the C functions that implement it, since our data-driven approach allows us to create an animation by considering only the data structures, regardless of how the program uses them. This is the main advantage with respect to event-driven systems. Actually, the only two data structures we are interested in are:

```
int Towers[3][MAXDISCS];
int TowersHeight[3];
```

`Towers` contains numbers representing the sizes of the discs stacked on each peg. According to the C language, pegs are implicitly numbered 0, 1, and 2. The height of each peg, i.e., the number of discs stacked on it at any instant during the execution of the algorithm, is maintained in the array `TowersHeight`. For instance, if `TowersHeight[0]`≥ 4, the first peg contains at least 4 discs; therefore, the value `Towers[0][3]` is well-defined and represents the size of the fourth disc stacked on that peg, counting from bottom to top.

In order to create a first visualization, let us assume that the following numerical constants have been defined: `Dx_`, `Dy_`, `First_`, `Dist_`, `Floor_`, and `Roof_`. These constants are used to describe the geometric layout of the rectangles used for representing the discs as shown in Figure 12. The following two Alpha predicates are enough to obtain a basic visualization:

```
View(Out 1);
```

```
Rectangle(Out ID,Out X,Out Y,Out L,Out H,1)
   For T: InRange(T,0,2)
   For D: InRange(D,0,TowersHeight[T]-1)
   Assign ID=Towers[T][D]
          X=First_+Dist_*T+Dx_*ID/2
          Y=Floor_-Dy_*(D+1)
          L=Dx_*ID  H=Dy_;
```

Predicate `View` declares a visualization window by giving it an identification number. The system computes this predicate at run-time, gets the identification number 1, and opens a new window. Note that keyword `Out` allows values to be passed back to the predicate caller as in any traditional "by reference" parameter passing.

Predicate `Rectangle` enumerates for each tower as many rectangles as the number of discs stacked on it and provides them with identification numbers (`ID`) and geometrical attributes. Such attributes are the `X` and `Y` coordinates of the left-top corner, as well as the length `L` and the height `H` of a rectangle in pixels. Observe that the last parameter of predicate `Rectangle` is equal to 1, meaning that all rectangles are contained in the previously declared window with identification number 1.

It is worthwhile to note the use of the `For` statement in the body of predicate `Rectangle`. This statement invokes the built-in ALPHA predicate `InRange(A,B,Out N)`, which enumerates all integer values in the range [`A`,`B`] and returns them, one by one, in its parameter N. Hence, in our example, the statement `For T: InRange(T,0,2)` means "for any tower T": during successive repeated invocations of predicate `Rectangle`, `T` is progressively assigned with values 0, 1, and 2. Analogously, the statement `For D: InRange(D,0,TowersHeight[T]-1)` means "for any disc D stacked on peg T"; this statement progressively returns the position, that is the height, on peg T of any disc stacked on it, from bottom to top. Such enumeration statements are possible in ALPHA because of its backtracking-based computation model [6]: thus, successive repeated calls to `Rectangle` made by the visualization system return different values of `ID`, `X`, `Y`, `L`, and `H`, that is, different rectangles. In particular, each pair `(T,D)` identifies a rectangle having `ID` equal to the size `Towers[T][D]` of the corresponding disc: this `ID` is unique since no two discs have the same size. The geometrical attributes of a rectangle only depend on the peg it is stacked on and on its position on that peg.

A basic smooth animation can now be easily obtained by turning on the smooth animation capabilities of the Visualizer through the declaration of the predicate `SmoothAnimationOn`. If no other piece of information is given, default synthesis (e.g., `LEFT_TOP_GROW_`) and interpolation (e.g., `LINEAR_`) function IDs are associated to all rectangles.

To customize the animation, we first specify how the rectangles representing discs have to appear/disappear in/from the graphical scene. Namely, we want them to automatically drop down from the top of the viewport when appearing and to shrink up when disappearing. The following predicates accomplish this task:

```
RectangleBirth(_,Out TOP_,1);
RectangleDeath(_,Out GROW_,1);
```

These predicates assign the same behavior to all rectangles in view 1 regardless of their logical names: this is achieved by accepting any `ID` as first argument through the wild-card identifier "`_`".

Even if the path that rectangles traverse when moved from a peg to another one is a fundamental point in the animation of the towers of Hanoi, nothing so far has been said about it. Thus, we can further improve the animation by adding to the program the following predicate:

```
RectanglePosPath(_,Out ARC_,1);
```

In this way, when a rectangle changes its position (i.e., a disc is moved to another peg), it moves automatically along the arc curve shown in Figure 11. However, a more natural and pleasant effect could be obtained by first lifting it, then moving it along that curve, and finally by dropping it down. This can be done by means of the composite curve defined below:

```
#define MyCurveID_   1
```

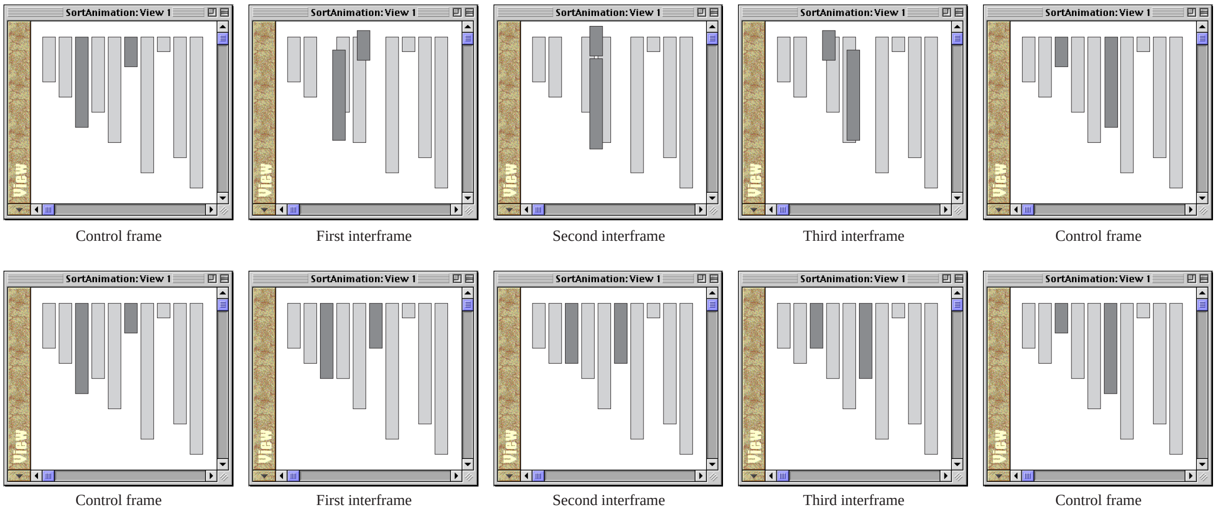


Figure 13: Storyboards concerning two different styles of animation of a swap operation.

```
ConcatCurve(MyCurveID_,X1,Y1,X2,Y2,Out X,Out Y,Out CurveID)
   Assign X=X1 Y=Y1    CurveID=LINEAR_  Moreover
   Assign X=X1 Y=Roof_ CurveID=ARC_     Moreover
   Assign X=X2 Y=Roof_ CurveID=LINEAR_;

RectanglePosPath(_,Out MyCurveID_,1);
```

According to the previous definitions, a moving rectangle follows the path identified by `MyCurveID_` and specified by `ConcatCurve`. `ConcatCurve` is automatically computed by the system for each rectangle that moves from `(X1,Y1)` to `(X2,Y2)`, and enumerates a sequence π_3 that defines a composite interpolating curve for these end-points. The final animation effect is shown in Figure 12.

5.2 Animation of Sorting Algorithms

Sorting algorithms are among the first algorithms ever animated since the birth of modern computer graphics [2]. Due to the simplicity of data they operate on, many animation styles have been proposed in the literature. The most common one is based on the sticks visualization, where elements drawn from a totally ordered set are typically represented as rectangles with heights proportional to their value. According to this visualization style, an ordered array of items looks like a sequence of growing sticks. Every known sorting algorithm applied to an unordered array of items progressively manipulates it, possibly using some auxiliary temporary memory, until it gets ordered. In general, input data can be thought of as composed of two logical entities: the array - that is an ordered collection of slots -, and the items - that are values placed into the slots -. Every slot contains an item and every item has a slot containing it. Thus, sorting algorithms look for a mapping that assigns slots having increasing indices with items having increasing values, and this process can be always reduced to a sequence of swap operations.

A natural question arises: how can we animate a swap operation? In other words: Do we animate the *slots* that change their items? Or: Do we animate the *items* that change their slots? In the literature, the second option is usually preferred, and we always see a pair of sticks that smoothly jump from one slot to the other in opposite directions. Instead, if we focus on the two slots involved in a swap, we could have their changing values animated as two sticks being smoothly expanded and shrunk, respectively. In the sequel we show how both these points of view can be naturally supported in our declarative framework.

Let us consider the following C definitions:

```
int v[]={3,5,2,9,6,4,1,8,0,7};
int n=10;
```

`v` is an array of 10 slots initialized with arbitrary integer values and `n` contains the number of slots of `v` in use. As usual in C, the slots of `v` are numbered from 0 to 9: in our example, slot 1 contains item 5 and item 4 is contained within slot 5. Note that no two slots hold the same value, that is, items are distinct.

We remark that, as in the example presented in Section 5.1, we do not need to show the C code for sorting the input data, and virtually *any* sorting function working on `v` could be animated with the ALPHA declarations that we are going to present. Indeed, a basic graphical representation of `v` can be obtained by adding to the C code the following two declarations:

```
View(Out 1);
Rectangle(Out ID,Out X,Out Y,Out L,Out H,1)
    For N:InRange(N,0,n-1)
    Assign X=20*N Y=0 L=18 H=20*v[N] ID=N;
```

We invoke the predefined predicate `InRange` to enumerate as many rectangles as the number of used slots and we properly set the graphical attributes to obtain an elementary stick visualization.

The assignment `ID=N` lets identification numbers of rectangles be equal to the slot indices, that is, *sticks logically represent slots*. In this case, IDs of rectangles are always unique since no two slots have the same index. Whenever a swap operation is performed by the sorting function, the Scene Manager builds the new scene and the Delta Manager finds out that two rectangles, corresponding to the involved slots, have changed their *size*. Therefore, if the smooth animation capabilities of the system are turned on, we can observe a rectangle being smoothly expanded and another one shrinked.

If we replace the assignment `ID=N` with `ID=v[N]` in the above definition of the predicate `Rectangle`, we obtain exactly the same control scenes, but inter-scenes are very different. Indeed, the assignment `ID=v[N]` lets identification numbers of rectangles be equal to the slot values, that is, *sticks logically represent items*. Whenever a swap operation is performed, the Delta Manages finds out that two rectangles, corresponding to the items being swapped, have changed their *positions* when passing from the old scene to the new one. Of course, the interpolation effect will be that of showing two moving rectangles, instead of two shrinking and growing ones. Graphical storyboards for both animation styles are illustrated in Figure 13.

Observe that while in the first case slot indices are unique regardless of the input data, in the second case two slot values may coincide, letting IDs of rectangles be not unique within the same scene. This is the reason why we required items in the array `v` to be distinct from each other.

As a matter of fact, an interesting drawback of the data driven approach is that comparing two successive scenes according to the logical names of graphical objects not always makes it possible to unambiguously infer what the program really did on its data structures, unless we are able to ensure uniqueness of logical names: in our example, if we want to animate items (and not slots,) and items are not distinct, it is impossible for the Delta Manager to unambiguously map sticks of the old scene onto sticks of the new one. It is, however, always practicable to extend the data structures of the underlying program with additional information in order to remove ambiguities. The following solution is not elegant, as it requires a "visualization-worried" C code, but it works correctly even when values to be sorted are not unique:

```
struct {
    int id,key;
} v[]={{0,5},{1,3},{2,5},{3,3},{4,5},{5,3},{6,5},{7,3},{8,5},{9,3}};
int n=10;
```

With this definition, items to be sorted are now pairs (`id`,`key`) where `id` fields hold unique values, and `key` fields contain the original array values. Observe that sorting keys are not unique in `v`. Of course, we require comparison and swap operations of sorting functions to be accordingly modified. If we assume IDs of rectangles are now given through the assignment `ID=v[N].id` and their heights as `H=20*v[N].key`, the Delta Manager is able to detect changes of positions of rectangles unambiguously, and the smooth animation with moving sticks is achieved even when values to be sorted are not distinct.

5.3 Animation of Graham's Convex Hull Algorithm

To conclude this section, we provide a further C + ALPHA example related to the animation of the well-known Graham's algorithm for finding the convex hull of a set of points [4]. As in the previous examples,

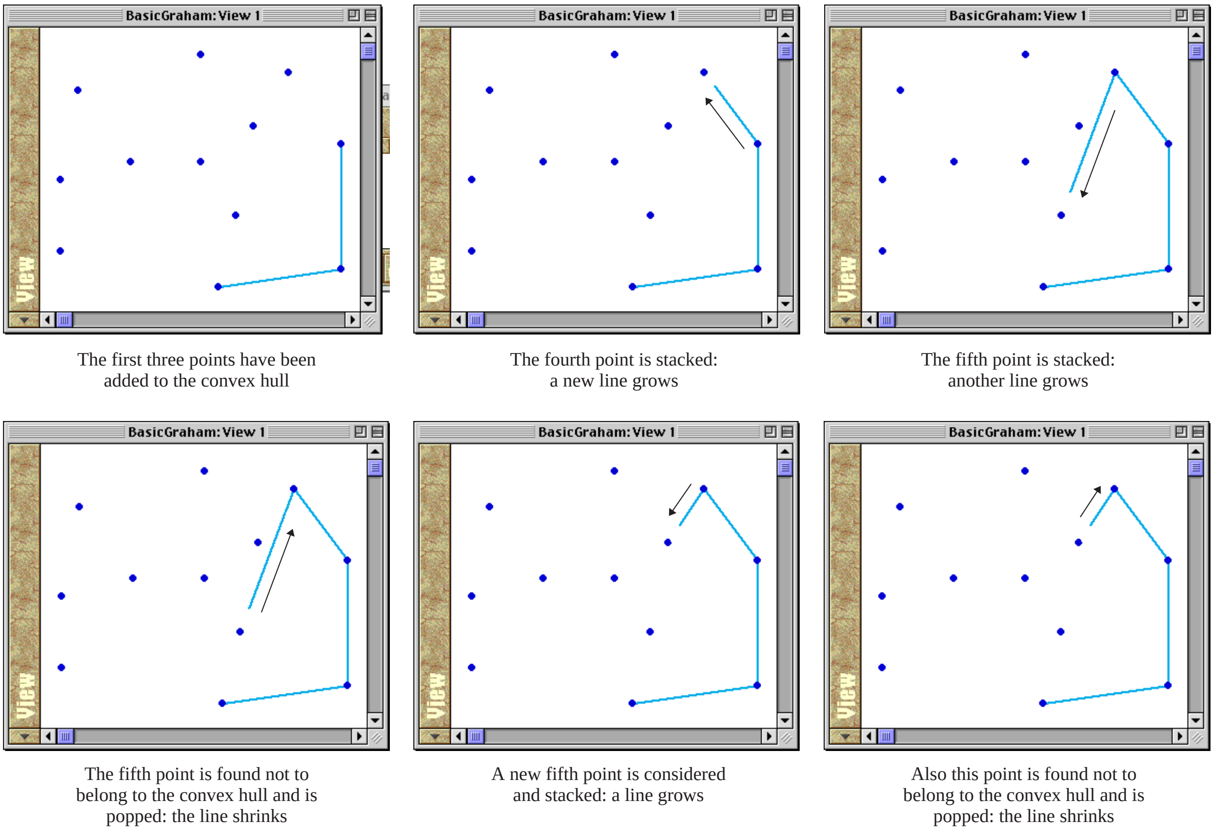


Figure 14: Animation storyboard concerning Graham's convex hull algorithm.

we just mention the variables whose content is interesting for visualization purposes:

```
int n;
struct { int x,y; } p[MAX_POINTS];
```

`MAX_POINTS` is a pre-defined constant that specifies the maximum number of points that the program is allowed to deal with: the actual number of input points is kept by the variable `n` and their coordinates in the plane are stored in the array of integer pairs `p`. The algorithm also uses a stack `hull` which stores indices of the points selected for insertion in the convex hull, and a variable `top` which is the index of the top of the stack:

```
int hull[MAX_POINTS], top;
```

A simple basic visualization of the data structures described above consists of displaying the set of input points `p` as dots and the stack `hull` as a chain of line segments connecting the first `top`+1 points selected for insertion in the convex hull:

```
View(Out 1);
Point(Out P,Out X,Out Y,1)
    For P: InRange(P,0,n-1)
    Assign X=p[P].x  Y=p[P].y;

PointColor(P,Out Blue,1);
PointShape(P,Out Round,1);
PointSize(P,Out 7,1);
```

```
Line(Out L,Out X1,Out Y1,Out X2,Out Y2,1)
    For L: InRange(L,1,top)
    Assign X1=p[hull[L-1]].x  Y1=p[hull[L-1]].y
           X2=p[hull[L]].x    Y2=p[hull[L]].y;

LineThickness(L,Out 2,1);
LineColor(L,Out Cyan,1);
```

ALPHA standard predicates `Point` and `Line` are used for declaring dots and line segments, respectively, in view 1. Notice that each dot and each line segment has a unique identification number. Also observe that line segments, whose end-points are denoted by (`X1`,`Y1`) and (`X2`,`Y2`), connect consecutive points `p[hull[L-1]]` and `p[hull[L]]` in the stack. Some attributes of graphical objects are also customized through suitable standard predicates and pre-defined constants: for example, `PointColor` is used for declaring that the color of any dot in view 1 is blue. Default values are anyway provided by the system for any missing piece of information related to object attributes. Previous declarations are enough for producing a basic jerky visualization where each manipulation performed by the algorithm on the stack is automatically portrayed by the system through line segments that suddenly appear and disappear as the stack grows or shrinks. A more pleasant animation effect that makes it easier for the observer to maintain the context can be obtained by simply adding to the code the following three declarations:

```
SmoothAnimationOn;
LineBirth(L,Out StartGrow,1);
LineDeath(L,Out StartGrow,1);
```

We animate stack manipulations as line segments that smoothly grow (push) or shrink (pop) from one end-point: this effect is obtained by assigning lines with a simple library synthesis function identified by the constant `StartGrow`. Figure 14 shows a storyboard concerning the animation of Graham's algorithm in Leonardo as specified through the declarations above. As the print medium does not convey effectively animation effects, we superimpose extra lines and arrows to the snapshots so as to provide a feeling of motion.

6 Conclusions

We have considered the problem of realizing smooth animations of algorithms in a declarative style: this seems difficult to achieve in data-driven frameworks, because of the lack of explicit user-defined visual event control. Our goal was to combine the typical advantages of declarative systems with the capability of modeling smooth transitions between scenes even without explicit control of visual events. In particular, we aimed at keeping the burden and the knowledge of the source code required from animation designers as low as possible: automatically detecting visual events and focusing on program's data structures rather than on its control flow are indeed two key concepts of our approach.

Our solution exploits the idea of modeling animated behaviors of graphical objects by means of "higher-order" attributes of the objects themselves expressed in the form of mathematical functions. These functions encode complete implicit information for creating smoothly animated transitions between graphical scenes: they specify how each graphical object appears/disappears in/from the scene, describe curves which varying attributes of graphical objects must be interpolated along by means of standard in-betweening techniques, and make it possible to control the pace of animated transitions. It is interesting to observe that in-betweening yields visualized scenes which are semantically inconsistent with the underlying state of the computation of the program being animated. Nevertheless, intermediate scenes are extremely useful for capturing viewer's attention in the extreme cases when two consecutive scenes are very similar or very different from each other. The first case may happen when the involved scenes are complex and just a few objects change their attributes.

We have embodied the mathematical framework in the version 3.5 of the logic-based SV system Leonardo [3] and we have devised practical and flexible solutions for specifying user-defined animation functions and for associating them to graphical objects in the logic-based language ALPHA [6]. The interested reader can find up to date information about Leonardo at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

The approach that we have addressed guarantees strong separation both between graphics and algorithms and between visualization and animation code: actually, the algorithm can be implemented without worrying about the visualization and the visualization can be specified without worrying about the animated behavior of objects. Moreover, if ALPHA predicates are left undefined, default behaviors are anyway provided by Leonardo. This implies that an incremental design of the animation can be approached, as shown in the examples given in Section 5. Last, but not least, in case of simultaneous changes of graphical objects the composition of their smooth transitions is obtained at no cost.

References

[1] Baecker, R.M.: *Picture-driven Animation*, Spring Joint Computer Conference, volume 34, pp. 273–288, AFIPS Press, 1969.

[2] Baecker, R.M.: *Sorting Out Sorting*, Morgan Kaufmann, Los Altos, CA. Narrated color videotape, 30 minutes, presented at ACM SIGGRAPH '81 and excerpted in ACM SIGGRAPH Video Review #7, 1983.

[3] Crescenzi, P., Demetrescu, C., Finocchi, I., Petreschi, R.: *Reversible Execution and Visualization of Programs with LEONARDO*, Journal of Visual Languages and Computing, 11(2), 2000, pp. 125–150. Leonardo is available at the URL: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[5] Cox, K.C., Roman, G.C.: *Experiences with the Pavane Program Visualization Environment*, Technical Report WUCS-92-40, Washington University, Department of Computer Science, School of Engineering and Applied Science, 1992.

[6] Demetrescu, C., Finocchi, I.: *A General-Purpose Logic-Based Visualization Framework*, Proceedings International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media, 1999, pp. 55–62.

[7] Demetrescu, C., Finocchi, I.: *A Technique for Generating Graphical Abstractions of Program Data Structures*, Proceedings of the 3rd International Conference on Visual Information Systems, LNCS 1614, 1999, pp. 785–792.

[8] Elliott, C.: *A Brief Introduction to ActiveVRML*, Technical Report MSR-TR-96-05, Microsoft Research.

[9] Elliott, C.: *An Embedded Modeling Language Approach to Interactive 3D Multimedia Animation*, IEEE Transactions on Software Engineering, to appear.

[10] Elliott, C., Hudak, P.: *Functional Reactive Animation*, Proceedings ACM SIGPLAN International Conference on Functional Programming, 1997, pp. 263–273.

[11] Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990.

[12] Hayward, S.: *Computers for Animation*, Focal Press, 1984.

[13] Lassiter, J.: *Principles of Traditional Animation Applied to 3D Concurrent Animation*, Proceedings SIGGRAPH'87, 1987, pp. 35–44.

[14] Lockwood, E.H.: *A Book of Curves*, Cambridge University Press, 1961.

[15] Myers, B.A., McDaniel, R., Miller, R.C., Ferrency, A., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A., Doane, P.: *The Amulet Environment: New Models for Effective User Interface Software Development*, IEEE Transactions on Software Engineering, 23(6), 1997, pp. 347–365.

[16] Myers, B.A., Miller, R.C., McDaniel, R., Ferrency, A.: *Easily Adding Animations to Interfaces Using Constraints*, Proceedings of the ACM Symposium on User Interface Software and Technology, 1996, pp. 119–128.

[17] Price, B.A., Baecker, R.M., Small, I.S.: *A Principled Taxonomy of Software Visualization*, Journal of Visual Languages and Computing, 4, 1994, pp. 211–266.

[18] Roman, G.C., Cox, K.C.: *A Taxonomy of Program Visualization Systems*, Computer, 26, 1993, pp. 11–24.

[19] Roman, G.C., Cox, K.C., Wilcox, C.D., Plum, J.Y.: *Pavane: A System for Declarative Visualization of Concurrent Computations*, Journal of Visual Languages and Computing, 3, 1992, pp. 161–193.

[20] Stasko, J.T.: *The Path-transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces*, Journal of Visual Languages and Computing, 1, 1990, pp. 213–236.

[21] Stasko, J.T.: *TANGO: A Framework and System for Algorithm Animation*, Computer, 23, 1990, pp. 27–39.

[22] Stasko, J.T., Domingue, J., Brown, M.H., Price, B.A.: *Software Visualization: Programming as a Multimedia Experience*, The MIT Press, 1998.

[23] Stasko, J.T., McCrickard, D.S.: *Real Clock Time Animation Support for Developing Software Visualizations*, Technical Report GIT-GVU-95-21, Georgia Institute of Technology.