Computer-Aided Design 31 (1999) 695–710

COMPUTER-AIDED DESIGN

www.elsevier.com/locate/cad

A programming approach for complex animations. Part I. Methodology

C. Bajaj[a], C. Baldazzi[b], S. Cutchin[a], A. Paoluzzi[b,*], V. Pascucci[a], M. Vicentino[b]

[a]*Department of Computer Sciences and TICAM, University of Texas, Austin, TX 78712, USA*
[b]*Dipartimento di Informatica e Automazione, Università di Roma Tre, Via Vasca Navale 79, Rome 00146, Italy*

Received 28 August 1998; received in revised form 15 June 1999; accepted 12 July 1999

Abstract

This paper gives a general methodology for symbolic programming of complex 3D scenes and animations. It also introduces a minimal set of animation primitives for the functional geometric language `PLaSM`. The symbolic approach to animation design and implementation given in this paper goes beyond the traditional two-steps approach to animation, where a modeler is coupled, more or less loosely, with a separate animation system. In fact both geometry and motion design are performed in a unified programming framework. The representation of the animation storyboard with discrete action networks is also introduced in this paper. Such a network description seems well suited for easy animation analysis, maintenance and updating. It is also used as the computational basis for scheduling and timing actions in complex scenes. © 1999 Elsevier Science Ltd. All rights reserved.

Keywords: Graphics; Animation; Design language; Collaborative design; Geometric programming; `PLaSM`

1. Introduction

This work introduces a high-level methodology oriented towards building, simulating and analyzing complex animated 3D scenes. Our first goal is the creation of compact and clear symbolic representations of complex animated environments. Our second goal is to allow for easy maintenance and update of such environments, so that different animation hypotheses can be analyzed and compared. The construction of complex animations using minimal man-hours and low-cost hardware also motivated the present approach.

The contributions of the present paper can be summarized as follows: (i) definition of a methodology for symbolic programming of complex 3D scenes; (ii) introduction of a minimal set of animation primitives for a geometric language; (iii) development of two experimental animation frameworks, both local and web-based. This approach goes beyond the traditional two-steps approach to animation, where a modeler is coupled, more or less loosely, with a separate animation system.

A programming approach to animation that provides simple but powerful animation capabilities as an extension of a geometric design language is discussed here. Hence both geometry design and animation can be performed in a unified framework where animations are programmed just like any other geometric information. In this way a designer or scientist may quickly define and simulate complex dynamic environments, where movements can be described by using well-established methods of mathematical physics and solid mechanics.

We show that using a functional language is serviceable both in quickly implementing complex animations and in testing new animation methods. For this purpose, look at the solution proposed for describing the storyboard of very complex animations and computing the relative time constraints between different animation segments. Notice that a full "choreographic control" is achieved by implementing a "fluidity constraint" of the whole animation, which *holds for any possible choice of expected durations of animation segments*.

For very complex animation projects, where several users cooperate to design an animated environment, we are developing a collaborative web-based interface where multiple client applications can interact with the same scene. Due to the compactness of the shared information, multiple client applications, possibly distributed in a wide area network, can interact with the same animated scene either synchronizing their views or working with independent focus on different aspects of interest or competence. This provides

* Corresponding author. Tel.: +39-06-5517-3214.
E-mail addresses: bajaj@cs.utexas.edu (C. Bajaj), baldazzi@dia.uniroma3.it (C. Baldazzi), cutchin@cs.utexas.edu (S. Cutchin), paoluzzi@dia.uniroma3.it (A. Paoluzzi), pascucci@cs.utexas.edu (V. Pascucci), vicenti@dia.uniroma3.it (M. Vicentino)

a shared workspace that makes easy the animation process and reduces its production time and cost.

We are experimenting the methodology in two settings. Firstly we have been using a simple Open Inventor-based animation server to allow local display of the animations on any system including low-cost PCs. Also, we are developing a web-based visualization framework, where the compactness of the language representation turns out to be useful when transmitting geometry and animation information over the network.

For this purpose we have extended the geometric language `PLaSM` [13], recently integrated into the collaborative and distributed 3D toolkit *Shastra* [3], with the ability to generate platform-independent animated scene descriptions capable of being read by 3D graphics libraries such as the Shastra animation module *Gati* [4], or *Open Inventor* [22]. The animation methodology introduced here is implemented as an extension of the `PLaSM` language, providing a hierarchical description of scenes and capable of being exported to generic animation engines, just by switching to an appropriate driver.

The proposed methodology allowed to implement a detailed reconstruction of a real catastrophic event which occurred in Italy three years ago. This large-scale example is described in the companion paper [14].

The present paper is organized as follows. In Section 2 some animation softwares are quickly described, in order to abstract their common features. In Section 3 a set of animation concepts is defined, and an animation design methodology based on discrete action networks is proposed. In Section 4 some background concepts from graphics, geometric programming and network programming are recalled. In Section 5 the design goals and implementation directions of the `PLaSM` animation extension are discussed. In Section 6 the static and run-time architecture of both a local Inventor-based animation server and of web-based Shastra's animation services are discussed. In Section 7 a complete implementation of a non-trivial animation example is given. In Section 8 the on-going extensions to the described animation environment are outlined.

2. Previous work

In this section we report on some main aspects of both commercial and academic animation software. In particular, we concentrate on the animation capabilities, even when the systems also offer advanced modeling tools. Commercial animation software usually allows to animate objects, lights, cameras, surface properties and shading parameters. The basic tools define animations on the basis of key-frames and motion paths. They also normally solve inverse kinematic problems. The user interface allows interactive definition and editing of parameters curves.

Power Animator [2] is a software from Alias/Wavefront. Interactive tools allow fine tuning of the interaction among actors and the background. It also allows to emulate physical phenomena like gravity, friction, turbulence or collisions. Advanced features include automatic generation of 3D solid and gaseous particles interacting realistically with the environment. Maya [1], also from Alias/Wavefront, allows for intuitive animation of synthetic *characters*. The character behaviors are defined by hierarchical controls. This includes hierarchical deformations, inverse kinematic, facial animation, *skinning* tools. It provides a wide number of special effects and includes the scripting language MEL. It also provides a C++ API that allows to modify a number of functionalities as needed by the user.

LightWave [12] from NewTek is platform independent. The system is mostly modeling and rendering oriented (e.g. real-time transformation of polygonal surfaces into NURBS). Advanced features are provided for dealing with evolving illumination and the definition of dynamic systems of particles. The animations can be defined through the scripting language `L-script`. Electric Image [6] is also platform independent. In Electric Image 3D models can be build directly from their skeletons. Shape transitions are performed through morphing techniques. Ad hoc features are available for sky and backgrounds.

Houdini [19] is a software from Effects Software. It is a procedural animation system providing a scripting language and a SDK for system customization. The procedural paradigm is implemented through a friendly user interface with a data-flow like structure. 3D Studio Max [11] from Autodesk is an object-oriented environment with a flexible animation control system, where every feature is editable and extensible. SoftImage [20] offers a high-level interface for sequencing *actions* (sets of animation data) along the time axis. This allows easy animation editing of segmented high-resolution characters. It provides advanced features for skinning and inverse kinematics of physically based characters.

The academic software Alice, by Randy Pausch's group [17] at CMU, is an interesting rapid prototyping interactive system for virtual reality. Alice is also based on a programming approach, which uses the object-oriented interpreted language Python [21]. When the Alice program is executing, the current state can either by updated by evaluating program code fragments, or by GUI tools. Alice's simulation and rendering are transparently decoupled. Cinderella [18] is a Java software for geometry definition and animation. It is oriented towards mathematical users experimenting with projective and descriptive geometry, but does not seem to be well suited for 3D modeling.

Published work on advanced animation systems is often focused on specific aspects of the problem. For example, Grzeszczuk et al. [9] explore the possibility to replace heavy physically based simulated animations with physically realistic sequences generated by trained neural networks. This approach seems very interesting but cannot be used within general-purpose animation systems. Gleicher [8] considers the problem of retargetting existing motions of animated characters to structurally equivalent ones. His approach minimizes the frequency magnitude of the difference between the original motion and the adapted one using a space–time constraint solver. In this way the visual artifacts of the adapted motion are strongly reduced. Such an approach could be used to reusing pieces of animation code in different settings.

3. Methodology

The animation methodology proposed in this paper is specified here. First some definitions are given, then the animation partitioning with network techniques is proposed, and the typical modeling and animation cycle is discussed.

3.1. Definitions

In order to focus the main aspects of the animation problem several definitions are given. Some of them are quite standard in the animation field. Anyway, it seems useful to provide a precise meaning to concepts used in the remainder of this work:

Scene: A *scene* is defined here as a function of real parameters, with values in some suitable data type. Each feasible set of parameters defines a *configuration* of the scene. The time can be considered a special parameter and treated separately from the other ones.
Background: The scene part which is time-invariant is called *background*.
Foreground: The time-varying portion of an animated scene is called *foreground*. It can be subdivided into *static* and *dynamic* foreground. The former is visible only inside a time interval with a constant configuration. The latter is instead associated to variable configurations.
Behavior: The product of the time domain times the Configuration Space *CS* gives the *state-space* of the animation.[1] A continuous curve in the *state-space* of the animation is called a *behavior* of the scene.
Animation: An animation is a pair ⟨*scene, behavior*⟩.

3.2. Network approach

An animation description with discrete action networks is introduced here. Such a network description seems well suited for easy animation analysis, maintenance and updating. It is also used as the computational basis for scheduling and timing actions in complex scenes. The relevant concepts are given in the following:

Storyboard: A network representation of the animation foreground is called the *storyboard*. It is a hierarchical a-cyclic graph with only one node of in-degree zero (called *source* node) and only one node of out-degree zero (called *sink* node). The source node represents the animation *start*. The sink node represents the animation *end*.

[1] Remember that *CS* is the product space of the interval domains of scene DOFs.

From a practical viewpoint, the storyboard states a hierarchical partitioning of the animation behavior (and scene) into independent components. In other words, the storyboard partitions the animation into largely independent "segments" and specifies both time constraints between such components, and the projection curves of the animation behavior into lower-dimensional subspaces associated to segments:

Segment: A *segment* is an arc of the storyboard. It represents a foreground portion characterized by the fact that every interaction with the remainder animation is concentrated on the starting and ending nodes of the segment.

Each segment may be modeled *independently* from the others, by using a *local* coordinate frame for both space and time coordinates. The concepts of storyboard and segment are interchangeable, in the sense that each complex segment of the animation can be modeled by using a local storyboard, which can be decomposed into lower-level segments. *Elementary segments* are those without a local storyboard:

Event: An *event* is a storyboard node. The segments starting from it may begin only when *all* the segments ending in it have finished their execution.
Segment model: The *geometric model* of a segment is a description of both the assembly structure and the geometry of its elementary parts. In `PLaSM` every geometric model results from the evaluation of some polyhedrally typed expression.
Segment behavior: A *behavior* of the segment is the continuous curve restriction of the scene behavior to the segment's subset of degrees of freedom.
Actor: An *actor* (or *character*) is a connected chain of segments associated with a unique geometric model and with different behaviors. So, at any given time each actor has a unique fixed set of parameters, i.e. a unique configuration.

Notice that the behaviors of an actor are defined as different curves in the same subspace of state-space. In fact, since

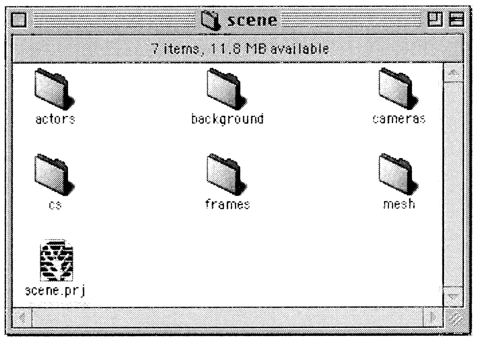

Fig. 1. Motion data set generated by an `ANIMATION` primitive. This representation can be played by a generic animation server.

the geometric model is invariant, the segment configuration space is also invariant. The different curves correspond to pairs of events where the considered actor interacts with other actors, according to the storyboard.

3.3. Modeling and animation cycle

The highest level `PLaSM` primitive, named `ANIMATION`, directly corresponds to the animation storyboard. The language interpreter generates at run-time a suitable set of *motion data* (see Fig. 1) to be interpreted and executed by the animation server.

Each animation segment is associated with a time length (duration), either introduced by the user for elementary segments, or automatically computed by using the segment storyboard and a suitable algorithm. In particular, the *critical path* algorithm is used for this purpose. This method is based on the computation of the longest-time paths of the storyboard network. According to standard PERT terminology each segment of a storyboard may be defined as either of type *earliest_start*, or *latest_end* or *critical* (both earliest start and latest end), to denote the specific time constraints.

Geometric models are implemented in `PLaSM` as Hierarchical Polyhedral Complexes (HPC) [15]. Such a model may be exported from `PLaSM` as either VRML or Inventor file. Container nodes referring to external VRML/Inventor files provide the geometric components for the run-time behavior of the animation.

The geometric model of a dynamic foreground segment must provide some degrees of freedom (DOFs), i.e. some unbounded parameters, often corresponding to parameters of affine transformations. Such a segment will be suitably exported using the `MOVE` primitive described in Section 5.2. The unbound parameters constitute one of main components of the motion files in the CS folder (see Fig. 1) associated by the `PLaSM` interpreter to the foreground segments.

The behavior curves are implemented in `PLaSM` by using parametric curves and splines. The starting and ending times are automatically computed using the dynamic programming formulas recalled in Section 4.5. A simple and often useful choice is to use Bézier curves to represent the behaviors. A behavior is completely specified by a sequence of points in state-space. In particular, any Bézier curve interpolates the first and last points and approximates the other ones. A sampled behavior, i.e. a finite sequence of state-space points, is exported to the external animation software, where a suitable engine will linearly interpolate between any pair of adjacent samples.

Animation cycle: To define a complex animation the following typical steps are required:

- Problem decomposition into animation segments, and definition of the storyboard as a graph.
- Modeling of geometry and behavior of the animation segments. Each segment will be modeled, animated and tested independently from each other.
- Non-linear editing of segments, describing in `PLaSM` their events and time relationships (look at example of Section 7). The segment coordination is obtained by using the critical path method.
- Simulation and parameter calibration of the animation as a whole.
- Feedback with possible storyboard editing, with the starting of a new cycle of modeling, editing, calibration and feedback, until a satisfying result is obtained.

4. Background

This section presents some background information for better understanding the material presented in the paper. First the `PLaSM` functional approach to programming is described, then the important graphics concepts of hierarchical assemblies and parametric curves are recalled. Finally some concepts of network programming, including critical path method, are briefly described.

4.1. `PLaSM` model of computation

The `PLaSM` language [13] is a geometry-oriented extension of a subset of the functional language `FL` [5]. Generally speaking, each `PLaSM` program is a *function*. When applied to some input *argument* each program produces some output *value*. Two programs are usually connected by using functional composition, so that the output of the first program is used as input to the second program. The composition of `PLaSM` functions behaves exactly as the standard composition of mathematical functions. For example the application of the compound mathematical function $f \circ g$ to the x argument

$$(f \circ g)(x) \equiv f(g(x))$$

means that the function g is first applied to x and that the function f is then applied to the value $g(x)$. The `PLaSM` denotation for the previous expressions would be

```
(f~g) : x ≡ f : (g : x)
```

where ~ stands for function *composition* and `g:x` stands for *application* of the function `g` to the argument `x`.

In `PLaSM`, a name can be assigned to a geometric model by defining it as a function without formal parameters. In such a case the *body* of the function definition will describe the computational process which generates the geometric value. The parameters that it depends on will normally be embedded in such a definition. For example we may have

```
DEF object = (Fun3~Fun2~Fun1):
parameters;
```

The computational process that produces the object value can be thought as the computational pipeline shown in Fig. 2.

In several `PLaSM` scripts the dependence of the model upon the parameters is implicit. In order to modify the generated object value it is necessary (a) to change the source code in either the body or the local environment of

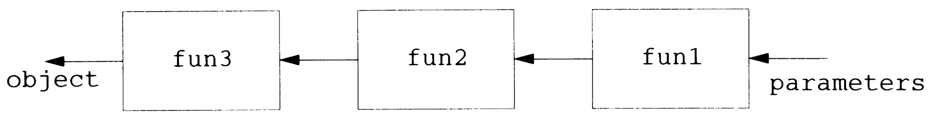

Fig. 2. Example of computational pipeline.

its generating function, (b) to compile the new definition and (c) to evaluate again the object identifier.

A *parametric* geometric model can be conversely defined, and easily combined with other such models, by using a generating function with *formal* parameters. Such a kind of function will be instantiated with different *actual* parameters, so obtaining different output values. It is interesting to note that such a generating function of a geometric model may accept parameters of any type, including other geometric objects.

4.2. Some FL & PLaSM syntax

FL is a pure functional language based on combinatorial logic. Primitive FL objects are characters, numbers and truth values. Primitive objects, functions, applications and sequences are expressions. An application expression exp1:exp2 applies the function resulting from the evaluation of exp1 on the argument resulting from the evaluation of exp2. Also the infix form is allowed for binary operators:

$+ : \langle 1, 3\rangle = 1 + 3 = 4$

Application associates to left, i.e. $f : g : h = (f : g) : h$. Also, application binds stronger than composition, i.e. $f : g\sim h = (f : g)\sim h$.

Construction functional cons allows the application of a sequence of functions to the same data. It can also be denoted by enclosing the functions between angle brackets:

$\mathrm{CONS}:\langle f_1, \dots, f_n\rangle : x = [f_1, \dots, f_n] : x = \langle f_1 : x, \dots, f_n : x\rangle$

Apply-to-all conversely allows the application of a function to a sequence of data:

$\mathrm{AA} : f : \langle x_1, \dots, x_n\rangle = \langle f : x_1, \dots, f : x_n\rangle$

The Identity function and the Constant functional have the standard meaning:

$\mathrm{ID} : x = x$
$K : x1 : x2 = (K : x1) : x2 = x1$

The Conditional function is defined as follows, where p, f and g must be functions:

$\mathrm{IF} : \langle p, f, g, \rangle : x = f : x \quad \text{if } p : x = \mathrm{TRUE}$
$\mathrm{IF} : \langle p, f, g\rangle : x = g : x \quad \text{if } p : x = \mathrm{FALSE}$

Other FL combining forms have a combinatorial behavior, and strongly contribute to the working of the language as a "combinatorial engine". In particular, the "Insert right" and "Insert left" functions allow for (recursively) applying a binary function to a sequence of arguments of any length:

$\mathrm{INSR}:f:\langle x_1, x_2, \dots, x_n\rangle = f:\langle x_1, \mathrm{INSR}:f:\langle x_2, \dots, x_n\rangle\rangle$
$\mathrm{INSL}:f:\langle x_1, \dots, x_{n-1}, x_n\rangle = f:\langle \mathrm{INSL}:f:\langle x_1, \dots, x_{n-1}\rangle, x_n\rangle$

The CAT (Catenate) function allows for appending sequences. Several other primitive functions for sequences are available, including LIST, FIRST, TAIL, LAST, and the selectors S1, S2, …, Sn of the first, second and the *n*th element of a sequence:

$\mathrm{CAT}:\langle\langle a, b, c\rangle, \langle d, e\rangle, \dots, \langle x, y, w, z\rangle\rangle = \langle a, b, c, d, e, \dots, x, y, w, z\rangle$
$\mathrm{LIST} : x = \langle x\rangle$
$\mathrm{S3} : \langle a, b, c, d, e, f, \rangle = c$

The two Distribute functions are applied to a pair composed by a sequence and by any expression and generate a sequence of pairs:

$\mathrm{DISTR} : \langle\langle a, b, c\rangle, x\rangle = \langle\langle a, x\rangle, \langle b, x\rangle, \langle c, x\rangle\rangle$
$\mathrm{DISTL} : \langle x, \langle a, b, c\rangle\rangle = \langle\langle x, a\rangle, \langle x, b\rangle, \langle x, c\rangle\rangle$

The Transpose function works exactly as a matrix composition, e.g.

$\mathrm{TRANS}:\langle\langle x_1, x_2\rangle, \langle y_1, y_2\rangle, \langle w_1, w_2\rangle\rangle = \langle\langle x_1, y_1, w_1\rangle, \langle x_2, y_2, w_2\rangle\rangle$

The language PLaSM can evaluate expressions whose value is a *polyhedral complex*. This is the only new primitive data type, with respect to FL. Clearly, it can produce higher-level functions in the FL style. Some important differences with FL exist. In particular no free nesting of scopes and environments is allowed in PLaSM, and no pattern matching is provided.

Two kinds of functions are recognized in PLaSM: *global* (or top-level) functions and *local* functions. Global functions may contain a definition of local functions between WHERE and END keywords. Local functions may not, and cannot contain a list of formal parameters. In order to make a local function *parametric* they must necessarily adopt the FL style of function definition, using combining forms and selector functions. Notice that the visibility of local functions is restricted to the scope of the global function where they are declared.

Top level functions may contain a list of formal parameters. They are always coupled to some predicate, which is used to test the type of the actual parameter:

```
DEF f(a :: type1) = body
DEF f(a1,...,an :: type2) = body
```

Notice that with more than one parameter, the application of the function to actual parameters requires the use of angle brackets:

```
f : x
f : ⟨x1,...,xn⟩
```

For a complete listing and meaning of pre-defined `PLaSM` operators, the interested reader should read paper [13]. Anyway, most of times the operator meanings are consistent with their names.

4.3. Hierarchical structures

Hierarchical structures are a basic graphics concept. A structure is inductively defined as an ordered set (sequence) of structures, affine transformations and elementary geometric objects. The semantics of the structure concept is very simple. Every affine transformation contained in a structure is (ordinarily) applied to all the subsequent objects that follow it in the sequence. This application returns a sequence of geometric objects all lying in the same coordinate space, when originally they were independently defined using local coordinate systems.

Structures are represented as oriented acyclic graphs. In particular, each structure is associated with a node, whereas the elements in its sequence are represented as the child nodes of the parent node. The structure

$$n = S, \qquad \text{with} \qquad S = \{n_1, n_2, \dots, n_k\},$$

is therefore implemented as an oriented graph with

$$\text{nodes } \{n\} \cup S \text{ and arcs } \{n\} \times S.$$

A *structure network*[2] or *hierarchical scene graph*[3] is simply the union of the graphs of the different structures. Such a graph *is not necessarily* a tree, since the same node may have more than one parent node at an upper level.

A traversal algorithm linearizes such a graph, suitably transforming all the geometric objects, each one defined in a local coordinate frame, into the coordinate frame of the root, often called *world coordinate system*. Such a traversal is often performed by some predefined *viewer* available in the chosen toolkit (e.g. in Open Inventor) and is executed at a suitable frequency to achieve fluid animation.

The animation effect is obtained when some of geometric parts of the scene change position, orientation or internal configuration. Such change is often implemented by editing the internal values of parameters of some affine transformation.

[2] ISO/PHIGS terminology.

[3] SGI/Inventor and VRML terminology.

4.4. Polynomial parametric curve

A polynomial parametric curve is a vector-valued function of a real variable u, obtained by the combination of some control points with the elements of a suitable polynomial basis in the u variable. In particular, a Bézier curve $\mathbf{c}$ of degree n is a polynomial combination of $n+1$ control points $\mathbf{q}_i \in E^d$:

$$\mathbf{c} : \Re \to E^d : \mathbf{c}(u) = \sum_{i=0}^{n} B_i^n(u)\mathbf{q}_i,$$

where the blending functions $B_i^n : \Re \to \Re$ are the Bernstein polynomials:

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}.$$

Notice that the image $\mathbf{c}(u)$ of the curve is a continuous set of points in the same space where the $\mathbf{q}_i$ control points are defined, which shortly describe the curve shape. Remember also that a Bézier curve interpolates the first and last control points, and approximates the intermediate control points.

4.5. Network programming

We use network programming techniques to edit the collection of animation segments. In particular, we use the dynamic programming algorithm of PERT (Program Evaluation and Review Technique) [10] to compute minimal and maximal times of events as well the completion time of the whole animation.

PERT, also known as *Critical Path Method*, is well known for managing, scheduling and controlling very complex projects and computing the optimum allocation of resources. Such projects may have tens or hundred of thousands of activities and events. This technique is the most popular variation of the network programming techniques, where a bundle of inter-dependent activities is represented as a directed acyclic graph.

Minimal (*maximal*) *spanning time*: t_k (T_k) of a node k is the minimal (maximal) time for completing the activities entering the node k. Notice that such activities can (must) be completed into this time, respectively.

Critical path method: The more important computation concerns minimal and maximal spanning time of nodes. The dynamic programming algorithm is very easy (see Fig. 3):

$$t_k = \max_{i \in \text{pred}(k)} \{t_i + T_{ik}\}, \qquad T_k = \min_{i \in \text{succ}(k)} \{T_i - T_{ki}\}$$

There are two subsequent steps in the computation, respectively, called forward and backward computation.

Forward computation of minimal times t_k. Set $t_0 = 0$. Then try to compute the minimal time t_e of ending node, i.e. the completion time of the whole project. The

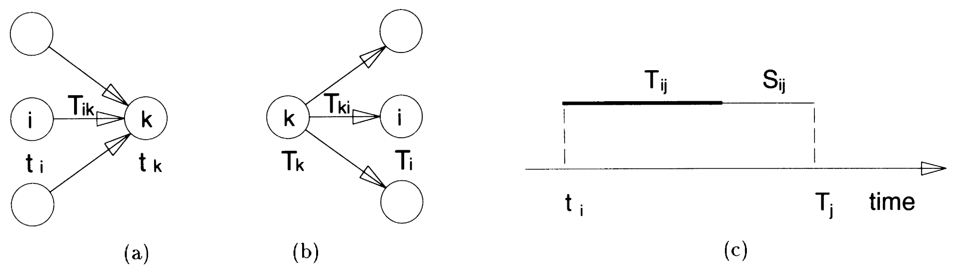

Fig. 3. (a) Forward computation of node k. (b) Backward computation of node k. (c) Slack S_{ij} shown in the earliest_start scheduling of the (i,j) segment.

recursive formula allows for computing the t_k of all nodes (see Fig. 3a).

Backward computation of maximal times T_k. Set $T_e = t_e$. Then try to compute the maximal time T_0 of starting node. The recursive formula allows for computing the T_k of all nodes (see Fig. 3b).

Activity slacks: The *slack* S_{ij} of the arc (i, j) is defined as the quantity of time which may elapse without a corresponding slack of the completion time of the project. The activity slack is given by

$$S_{ij} = (T_j - t_i) - T_{ij},$$

where T_{ij} is the expected duration of the activity (i, j) (see Fig. 3c). The *critical activities* have empty slacks: $S_{ij} = 0$.

5. Language extension

This section outlines the extensions to the PLaSM language implemented for defining and exporting motion data which may contain geometry, attributes and animations. Such exporting was targeted towards commonly available 3D toolkits. The reference model for such graphics extension of the PLaSM geometric language is the hierarchical scene graph. The described approach is currently performing only *off-line* geometric animations.

From the animation viewpoint the language extensions allow the modeling of the actors and their interactions on the scene, as well as the modeling of the independent behaviors of animation segments. The time assembly of independent animation segments is automatically guaranteed.

The previous PLaSM implementation allowed for geometric computations without considering non-geometric attributes such as colors, lights and textures. The current extension introduces such non-geometric features into the language, using as reference a hierarchical scene graph with non-geometric nodes. In particular, colors, lights and textures were added to the language [16].

A basic design decision was that of introducing only small modifications to the existing language interpreter. The language was mainly extended by modifying the predefined functional environment. A new internal data type and four new animation primitives were introduced into the language. Such minimal extension to the language has shown to be sufficient for implementing very complex animated scenes, as discussed in the companion paper [14].

5.1. Static animation with classic PLaSM

In PLaSM the animation of a polyhedral scene can be implicitly represented by explicitly giving the degrees of freedom as formal parameters of a function which generates the polyhedral values.

The desired behavior can also be modeled as a curve in configuration space. The movement can finally be emulated by: (a) sampling the curve; (b) applying the generating function of the shape on each sampled configuration (set of parameters); and finally by (c) aggregating all the resulting polyhedral complexes in a unique structure. The resulting covering of the work space will therefore represent the movement.

Example (Modeling a planar arm). Consider the very simple planar robotic arm with three degrees of freedom shown in Fig. 4a. There are three rotational degrees of freedom, respectively, named $\alpha 1$, $\alpha 2$ and $\alpha 3$

```
DEF rod = T:⟨1,2⟩:⟨ − 1, − 19⟩:
(CUBOID:⟨2,20⟩)
DEF DOF(alpha :: IsReal) = T:2:−18
~R:⟨1,2⟩:alpha;
DEF arm(a1,a2,a3 :: IsReal) = STRUCT:
⟨rod,DOF:a1,rod,DOF:a2,rod,DOF:a3,rod⟩
```

Example (Modeling a movement). The desired movement is modeled as a cubic Bézier curve defined by four points in configuration space $[-\pi, \pi]^3 \subset \Re^3$. Such a curve is shown in Fig. 4b. Notice that the Sampling function produces a sampling of the unit internal [0,1]. The implementation of the Bézier curve mapping is given in Appendix A.4

```
DEF CSpath = Bezier:⟨⟨0,0,0⟩,
⟨PI/2,0,0⟩,⟨PI/2,PI/2,0⟩,
⟨PI/2,PI/2,PI/2⟩⟩;
DEF Sampling(n :: IsIntPos) =
(AA:LIST~AA:/~DISTR):⟨0..n,n⟩;
```

Such a movement is emulated by applying the arm

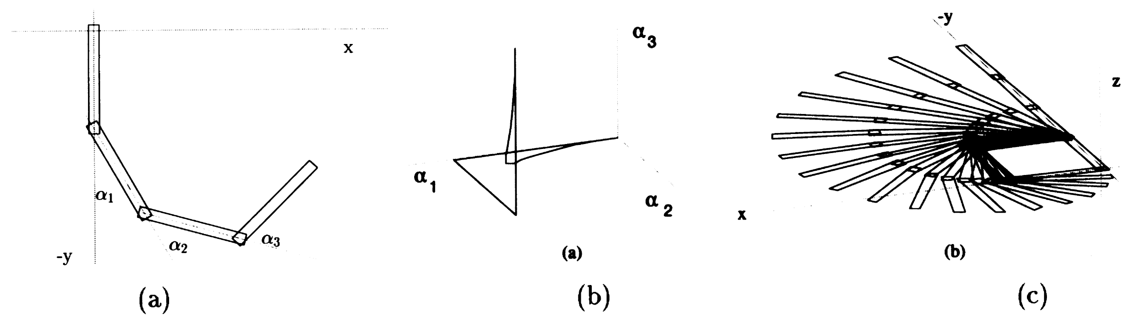

Fig. 4. (a) Planar arm configuration generated by `arm:⟨PI/6,PI/4,PI/3⟩`. (b) Cubic Bézier curve in configuration space. (c) Geometric object generated by `(STRUCT~AA:arm~AA:Cspath):(Sampling18)`.

function to a sequence of 18 curve samples and by accumulating the resulting polyhedral complexes in a structure:

```
(STRUCT~AA:arm~AA:CSpath):
(Sampling:18)
```

The geometric result obtained by evaluating the above `PLaSM` expressions is displayed in Fig. 4c.

5.2. Animation with extended `PLaSM`

For the purpose of defining and exporting animations, we have introduced four new primitives in `PLaSM`, respectively, denoted as `FRAME`, `XSTRUCT`, `MOVE`, `ANIMATION`. The representation of background objects does not require new primitives.

Background: The time-invariant objects in the scene are simply defined as standard polyhedral complexes, by using polyhedrally typed `PLaSM` expressions.

`FRAME` *primitive*: The `FRAME` primitive is used to represent a static portion of the foreground, i.e. a segment with constant configuration. Such a primitive is first applied to the static geometry, i.e. to a polyhedral complex. The result is then applied to an increasing pair of time values. The two time values define the existence interval of such geometry in the scene. Each animation is assumed to start at local time $t = 0$. The special time symbol -1 is used to denote the final time of the storyboard.

`XSTRUCT` *primitive*: The `XSTRUCT` primitive is used to define parameterized polyhedral complexes, which correspond to the new internal data type XHPC of extended `PLaSM`. The syntax and usage are equivalent to the `STRUCT` primitive, which is used to generate assemblies. Its semantics requires a lazy evaluation mechanism, where a set of parameters is left unbound for subsequent use.

`MOVE` *primitive*: The `MOVE` primitive is used to implement the behavior of a dynamic foreground segment. It requires three cascaded applications. First it is applied to a function, parameterized on the degrees of freedom of the segment and returning a value of XHPC type. The resulting function is then applied to the sampled trajectory in the segment configuration space. Finally, the function resulting from the latter application is applied to a conformal sequence of time steps.

`ANIMATION` *primitive*: The `ANIMATION` primitive is used (likewise a container) to define and export the set of motion data for a given background and foreground. Its input is a sequence of either polyhedral complexes, or `FRAME` expressions or `MOVE` expressions. As a side effect of its evaluation the motion files and directories displayed in Fig. 1 are generated. Such files will be used by the coupled animation server for the animation play-back.

Example (Modeling arm motion and geometry). In order to discuss the extensions to `PLaSM`, the robot arm example has been animated according to the behavior curve already presented. The extended `PLaSM` code for the description of both the planar arm and its motion is given in the following:

```
DEF background =
(EMBED:1~T:2: − 90~CUBOID):⟨100,100⟩;
DEF Rod = T:⟨1,2⟩:⟨ − 1, − 19⟩:
(CUBOID:⟨2,20,1⟩);
DEF dof(alpha :: IsReal) =
T:2: − 18~R:⟨1,2⟩:alpha;
DEF Arm(a1,a2,a3 :: IsReal) = XSTRUCT:
  ⟨Rod, dof:a1, Rod, dof:a2, Rod, dof:a3,
  Rod⟩;
DEF t1 = 2;
DEF t2 = t1 + 5;
DEF Sample(n :: IsIntPos) =
(AA:LIST~A:/~DISTR):⟨0..n,n⟩;
DEF Motion = Bezier:⟨⟨0,0,0⟩,⟨PI/
2,0,0⟩,⟨PI/2,PI/2,0⟩,⟨PI/2,PI/2,PI/2⟩⟩;
DEF Time = Bezier : ⟨⟨t1⟩,⟨t1 + 1⟩,⟨t2⟩⟩;
DEF CSpath = (AA : Motion~Sample) : 18;
DEF TimePath =
(CAT~AA:Time~Sample):18;
DEF FirstConfiguration =
(STRUCT~Arm):⟨0,0,0⟩;
DEF LastConfiguration =
(STRUCT~Arm):⟨PI/2,PI/2,PI/2⟩;
DEF Storyboard = ANIMATION : ⟨
```

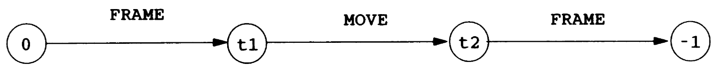

Fig. 5. The very simple storyboard of the plane arm animation example.

```
    background,
    FRAME:FirstConfiguration:⟨0,t1⟩,
    MOVE:Arm:CSpath:TimePath,
    FRAME:LastConfiguration:⟨t2, − 1⟩
  ⟩;
```

The previous example is a simple instance of our animation model, with only one animated actor. In this case the storyboard is a linear graph with three arcs, as shown in Fig. 5. The first and last segments are static, the second one is dynamic.

6. Test environments

The described animation methodology was experimented in two settings. A simple Open Inventor-based animation server has been implemented to allow local play-back of PLaSM-generated animations. A web-based visualization framework is also under development to address issues of user-collaboration over wide-area networks.

6.1. Open inventor animation server

The visualization server MOV (Motion with Open inVentor) was implemented on the IRIX and NT platforms, so allowing local play-back both on high-end workstations and desk-top PCs. MOV (see architecture in Fig. 6a) takes as input the motion data generated by PLaSM with information about: the geometry of actors; their configuration space path; the corresponding timeline data; the position, orientation and motion of animated cameras; the background objects, etc. The motion data set also includes a .prj project file, with a set of general directives to the player, like the number of frames per second, the starting and ending frames, the display resolution, the setting of static cameras, the background color, and so on. When the server is used interactively, the user can browse the animation frame by frame, backward and forward, with interactive camera control and selection, including all the standard capabilities of the Inventor's Examiner Viewer.

The motion data set is used to generate an Inventor scene graph (see Fig. 6b) and an internal database. The unbound parameters in the XHPC values are mapped symbolically by PLaSM into labels which are defined in the motion data and referred in the Inventor geometries. This mechanism allows to optimize the MOV play-back. Three execution modalities are available: real-time, step-wise and off-line. In real-time mode the server tries to synchronize the animation time to the clock time, eventually dropping some frames. In step-wise mode all the frames are displayed independently from the rendering time. In batch-mode a sequence of pictures (jpeg, rgb or bmp) is saved into secondary memory. This is useful for movie generation and editing.

6.2. Experimental web animation system

We have also built an experimental web animation system that uses a simple web-based visualization client (ShaPoly), two domain specific servers (Gati and PLaSM) and the Shastra collaboration toolkit to integrate the applications (see Fig. 7). The visualization client is a Java/VRML application downloadable from a remote web server. Gati is a dedicated animation server for generating complex animations from a high-level declarative language. The PLaSM server, as widely discussed in the previous sections, provides the modeling and animation functionalities. It

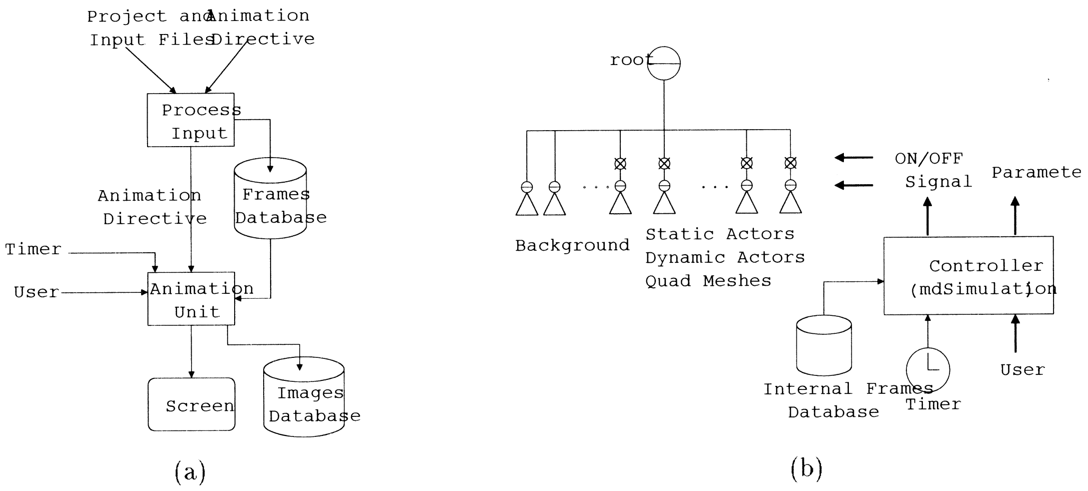

Fig. 6. (a) MOV server architecture. (b) Animated Inventor scene graph generated by MOV from PLaSM input.

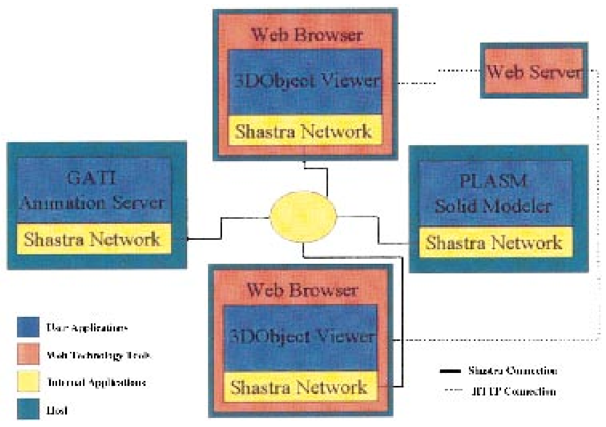

Fig. 7. Web animation architecture.

provides a powerful high-level language for describing complex geometric scenes and objects simply. The Shastra collaboration toolkit is a collection of libraries and applications that create a web-based collaborative networked environment.

6.2.1. Static architecture

ShaPoly is a Java/VRML collaborative object browser that can retrieve 3D objects, scenes, and animations from remote applications. It has a simple graphical user interface for displaying and manipulating three dimensional scenes and animations. Using the Shastra collaboration toolkit multiple ShaPoly applications can be used collaboratively.

The `PLaSM` interpreter accepts high-level descriptions of objects, scenes, and animations and in conjunction with the Gati animation server generates simplified descriptions of these scenes and animations. These simplified descriptions are then returned to the ShaPoly client for visualization.

Gati is a programmable animation server. It is capable of processing multiple remote requests for animations. Scene descriptions with complex animations are submitted to the server and simplified scenes and animations are returned as a result.

The Shastra collaboration toolkit is a collection of libraries and specialized applications that can be used to create powerful collaborative and distributed applications. The toolkit is both Java and C++ based, allowing both C++ and Java collaborative applications to be created and inter-operate. The toolkit provides communication, coordination, and collaboration mechanisms for applications.

6.2.2. Runtime architecture

Some users interact with the visualization client to send geometric descriptions to the `PLaSM` server that generates representations for scene and animation. This representation is passed to Gati which generates the low-level translation transmitted to the visualization client which displays the animation.

The animation is begun by the users pointing their web browser to a web server and downloading the Java ShaPoly visualization client. After the client is downloaded the user will be automatically connected to the Shastra collaboration environment. This allows them to access all the network resources active within the environment.

A user uses the ShaPoly visualization client to create a scene graph with objects that can be imported by the `PLaSM` interpreter. A simple graphical user interface is provided for creating scenes of user defined polyhedral objects. After creating a scene a user can then attach a `PLaSM` script to the scene specifying the motion to be applied to objects within the scene. Alternatively the user may sketch out space curves for objects within the scene to follow and graphically attach them to the scene objects (the interface will generate the corresponding `PLaSM` code).

Once defined a scene and animation can be submitted to a remote `PLaSM` server. The user selects a `PLaSM` server from a list of available servers provided by the Shastra kernel resource manager. If no `PLaSM` server is executing the user may request that one by starting and the Kernel will start a remote `PLaSM` server.

After the remote `PLaSM` server has been selected the user defined scene and animation is converted to a `PLaSM` geometric language description and submitted to the server using the Shastra communication layer. The `PLaSM` server then processes the code and generates the motion data described in the previous sections of this paper. Such data are submitted to a Gati server specified within the geometric language code. The scene and motion data are transferred to the Gati server using the Shastra communication layer.

Finally Gati generates a sequence of atomic transformations (events) that the viewer can execute directly. The resulting scene and animations are then streamed to the original visualization client where they can be played and replayed by the user.

7. Example

In this section a complete example of scene modeling and animation is discussed. The example closely resembles the famous *Luxo Lamp* animation by Michael Kass and Andrew Witkin [7]. In our case two similar lamps are moving together by describing a quite complex path in their configuration spaces.

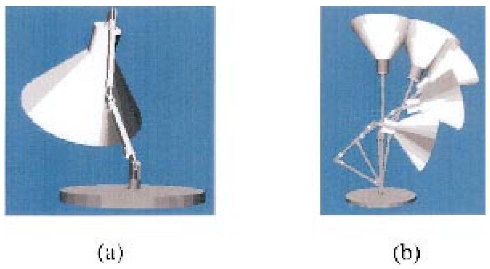

Fig. 8. (a) The lamp model in a given configuration. (b) Some key-frames along a segment animation.

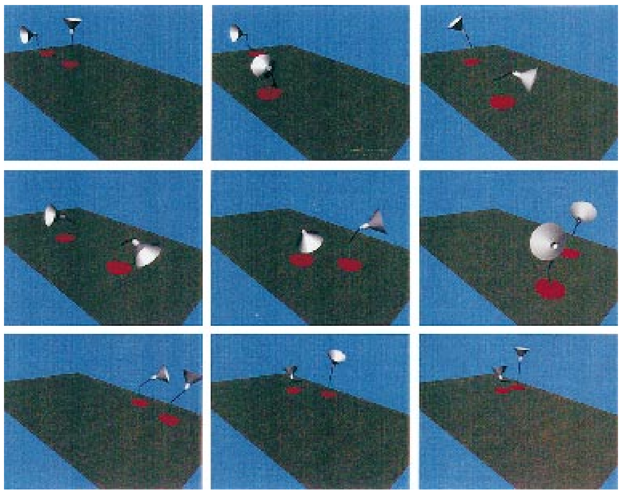
Fig. 9. Some key-frames of the animated example.

In Section 7.1 the geometric models of the lamp components are generated, and the lamp assembly is defined in local coordinates (see Fig. 8). In Section 7.2 the storyboard of the animation is given, where the movements of the two actors are both specified and coordinated (see Fig. 9). In the appendices a quite small set of related PLaSM code is given, together with the specification of the CS paths of the various animation segments.

Notice that the given example code is a complete working implementation, which can be directly executed under a PLaSM interpreter, obtaining a set of files to be executed under the control of either a Gati or MOV interface.

7.1. Geometry modeling

Design parameters: In order to generate different lamp instances, the lamp code has been parameterized with respect to the length and side of rods and to the radius and height of the basis.

```
DEF rodHeight = 20;
DEF basisRadius = 20;
DEF rodSide = SQRT:2;
DEF basisHeight = 2;
```

Geometric model of parts: First the lamp rod, named JointedRod, is generated. It is an assembly, along the *z* coordinate, of a maleJoint, a rod, and a femaleJoint.

```
DEF halfHinge2D = circle:PI:1:12;
DEF hinge2D = STRUCT:
⟨halfHinge2D,T:⟨1,2⟩:⟨ − 1, − 3⟩,Q:2*Q:3;
DEF hinge = hinge2D*Q : 0.5;
DEF DoubleHinge = STRUCT:
⟨hinge,T:3:1.2,hinge⟩;
DEF hbasis = circle:(2*PI):1.2:24*Q:2;
DEF femaleJoint = STRUCT :
  ⟨T:3: − 5:hbasis,T:2:0.85,R:⟨2,3⟩:
  (PI/2):DoubleHinge⟩;
DEF maleJoint = STRUCT:
⟨R:⟨2,3⟩:PI,
  T:3: − 5:hbasis,T:2:0.25,R:⟨2,3⟩:
  (PI/2):Hinge⟩;
DEF rod = T:
⟨1,2⟩:⟨rodSide/−2,rodSide/−2⟩:
  (CUBOID : ⟨rodSide, rodSide, rodHeight⟩);
DEF JointRod = maleJoint
TOP rod TOP femaleJoint;
```

In the previous code notice that infix binary operators (like TOP) are left-associative. Then the lamp basis and head are defined, where the conic part as well the cylinder part are both generated by using the function TrunCone, whose definition is given in the appendix. The head function depends on an implicit integer parameter, which specifies the grain of the polyhedral approximation of surfaces.

```
DEF basis = (circle:(2*PI):
basisRadius:32*Q:basisHeight)
  TOP femaleJoint;
DEF head = STRUCT~[K:maleJoint,
K:(T:3:5),embed:1~circle:(2*PI):4,
  TrunCone:⟨4,4,8⟩,K:(T:3:8),
  TrunCone:⟨4,20,20⟩];
```

Luxo lamp assembly: The lamp as a parametric hierarchical geometric model is defined as follows. Notice that the offset of joints from their center of rotation is 5, and that a JointedRod contains two such joints. This explains the term 10 in the *z* translation parameter (rodHeight + 10).

```
DEF Luxo(a1, a2, a3 :: IsReal) = XSTRUCT : ⟨
  basis,
  T:3:(basisHeight + 5),R:⟨1,3⟩:a1,
  JointedRod,
  T:3:(rodHeight + 10),R:⟨1,3⟩:a2,
  JointedRod,
  T:3:(rodHeight + 10),R:⟨1,3⟩:a3,
  head:32
⟩;
```

Notice also that the Luxo function has signature $\Re^3 \rightarrow$ *XHPC*. So, the generated values are polyhedral complexes, extended with non-geometric entities like colors, and depending on three degrees of freedom. When used with angle values in degrees, a conversion function to gradients must be applied to input data.

```
Luxo:(convert:⟨ − 90,90,90⟩);
```

7.2. Motion modeling and coordination

Mobile lamp: A 3D object sliding on the ground has three additional degrees of freedom, corresponding to one rotation and two translations

```
DEF mobileLuxo
(a1,a2,a3,a4,a5,a6 :: IsReal) =
```

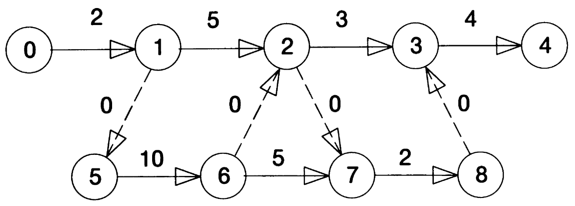

Fig. 10. Representation of the storyboard as an abstract graph.

```
XSRUCT:⟨T:1:a1,T:2:a2,R:⟨1,2⟩:a3,
Luxo:⟨a4,a5,a6⟩⟩;
```

Luxo's and friend's paths: As already stated, the animation storyboard is given as a network model, with animation segments associated to the arcs and (coordination) events associated to the nodes. The expected durations of single animation segments are first given. The dummy segments, drawn as hatched in Figs. 10 and 11, are used only as constraints and have duration zero. The PLaSM representation `luxo_pert` of the graph in Fig. 10 is the sequence of the graph's arcs. Each arc is a triple ⟨`s`,`e`,`t`⟩ where `s` is the starting node, `e` the ending node and `t` the duration time of the arc.

```
DEF luxo_pert = ⟨⟨0,1,2⟩,⟨1,2,5⟩,
⟨2,3,3⟩,⟨3,4,4⟩,⟨1,5,0⟩,⟨6,2,0⟩,⟨2,7,0⟩,
⟨8,3,0⟩,⟨5,6,10⟩,⟨6,7,5⟩,⟨7,8,2⟩⟩;
```

Coordination events: The minimal and maximal spanning times of storyboard events are computed from the storyboard network shown in Figs. 10 and 11. The minimal times t_i are computed as `tmin:0,…,tmin:8`. The maximal times T_j are computed as `tmax:0,…,tmax:8`. Appendix A.3 reports of a possible simple implementation of the functions `tmin` and `tmax`. *Example of curve in segment CS*: One of the configuration space curves of the storyboard segments is given in the following:

```
DEF CSpath_0_1 = AA:((Bezier~AA:XCAT):⟨
  ⟨100,0,convert:⟨0,0,0,0⟩⟩,
  ⟨150,0,convert:⟨30,30,0,−10⟩⟩,
  ⟨200,50,convert:⟨−150,−20,90,0⟩⟩,
  ⟨200,100,convert:⟨90 + 180,
   − 60,105,60⟩⟩
⟩);
```

It is generated as a Bézier curve of degree 3 in a configuration space of dimension 6. The PLaSM function generating Bézier curves of any degree in any *d*-dimensional space is given in Appendix A.4. The remaining CS curves are also given in the appendix.

Choreographic control: The timing of the (*i,j*) segment of the storyboard may be computed by using either minimal or maximal spanning times for the starting and ending events of the segment. If the $(t_i, t_j + T_{ij})$ pair is used, then the segment is executed “earliest”. If the $(T_j - T_{ij}, T_j)$ pair is used instead, then the segment is executed “latest”.

More interesting, when the pair (t_i, T_j) is used, the segment timing is automatically adapted to the finish and start times of the incident segments. If such a choice is done for all the “critical” segments, i.e. for all segments on the critical path, their animation will be executed as completely fluid, and no actors must stop and wait for restart in such storyboard segments.

This choice cannot by directly assumed for non-critical segments, since if (*h,i*), (*i,k*) are two incident non-critical segments, it would be $T_i \neq t_i$, so that the timing given by (t_h, T_i) and (t_i, T_k) would produce a time overlap for their execution.

A good solution is that of assuming for all events the timing given by their average spanning time, i.e. to use the timing (t_j^m, t_j^m) for each segment (*i,j*), with

$$t_i^m = \frac{t_i + T_i}{2} \qquad \text{for all } i \tag{1}$$

This clearly implies $t_i^m = t_i = T_i$ for critical events, and prevents from time overlaps for non-critical events.

It is very important to notice that the “fluidity constraint” of the whole animation induced by Formula (1) holds for any possible choice of expected durations of animation segments.

In PLaSM the average spanning times t_i^m, with $i = 0,\ldots,8$, can be computed as `tm:0,…,tm:8` using the function `tm` below

```
DEF tm(node :: IsInt) =
(tmax:node + tmin + node)/2;
```

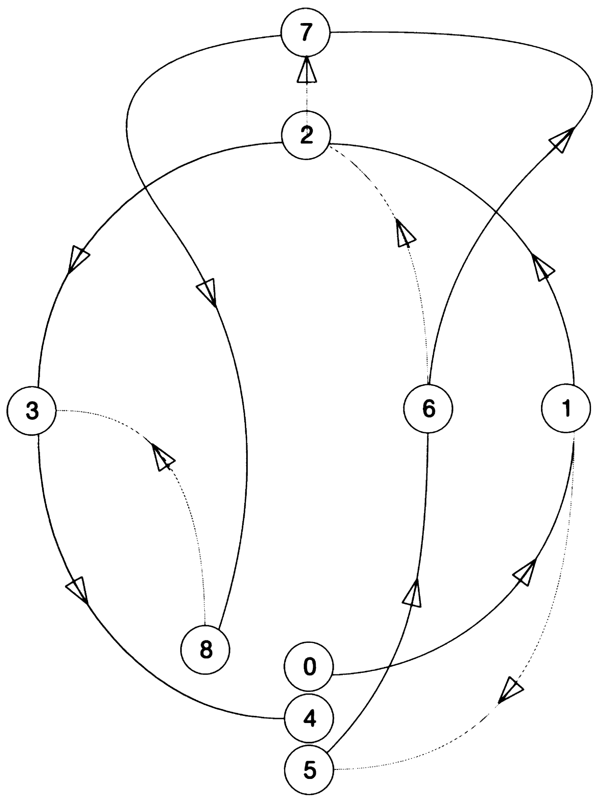

Fig. 11. Projection in E^2 of the storyboard embedded in configuration space.

Scene animation: The whole animated scene can be finally generated by just evaluating the following ANIMATION expression, which contains a MOVE expression for each storyboard segment. A number of 10 behavior samples is generated here for each segment

```
DEF steps = Sampling : 10;
DEF step = Sampling : 1;
DEF SceneAnimation = ANIMATION : ⟨
  FloorPlane,
  MOVE:mobileLuxo:(CSpath_0_1:steps):
  (time:⟨tm:0,tm:1⟩:steps),
  MOVE:mobileLuxo:(CSpath_1_2:steps):
  (time:⟨tm:1,tm:2⟩:steps),
  MOVE:mobileLuxo:(CSpath_2_3:steps):
  (time:⟨tm:2,tm:3⟩:steps),
  MOVE:mobileLuxo:(CSpath_3_4:steps):
  (time:⟨tm:3,tm:4⟩:steps),
  FRAME:((MkStruct~mobileLuxo):
  ((S1~CSpath_5_6):step)):⟨0,tm:5⟩,
  MOVE:mobileLuxo:(CSpath_5_6:steps):
  (time:⟨tm:5,tm:6⟩:steps),
  MOVE:mobileLuxo:(CSpath_6_7:steps):
  (time:⟨tm:6,tm:7⟩:steps),
  MOVE:mobileLuxo:(CSpath_7_8:steps):
    (time:⟨tm:7,tm:7 + 0.6*
    (tm:8 − tm:7),tm:7 + 0.8*(tm8 − tm:7),
    tm:8⟩:steps),
  FRAME:((MkStruct~mobileLuxo):
  ((S2~CSpath_7_8):step)):⟨tm:8, − 1⟩
⟩;
SceneAnimation;
```

Segment sub-timing: The approach shown in this paper allows any sub-timing of segments, according to the degree of the Bézier map $[0,1] \longrightarrow [0,\infty]$ used to generate the time samples passed to the MOVE primitive. This gives the animator the maximal freedom in accelerating/decelerating the animation speed within any segment, also *maintaining the constraint of fluidity of the full animation.*

As an example of non-linear sub-timing, look at segment (7,8) of the above example, where the time samples are generated by the partial map

```
time:⟨tm:7,tm:7 + 0.6*(tm:8 − tm:7),
  tm:7 + 0.8*(tm:8 − tm:7),tm:8⟩:steps
```

where tm:7, t:m8, respectively, stand for t_7^m, t_8^m. Since the Behavior function, given in Appendix A.2, is applied to a sequence of four values, the time sampling is done with a cubic real-valued Bézier map. According to the four control values, which are not equally spaced in the interval $[t_7^m, t_8^m]$, the samples are accumulated towards the interval end, so

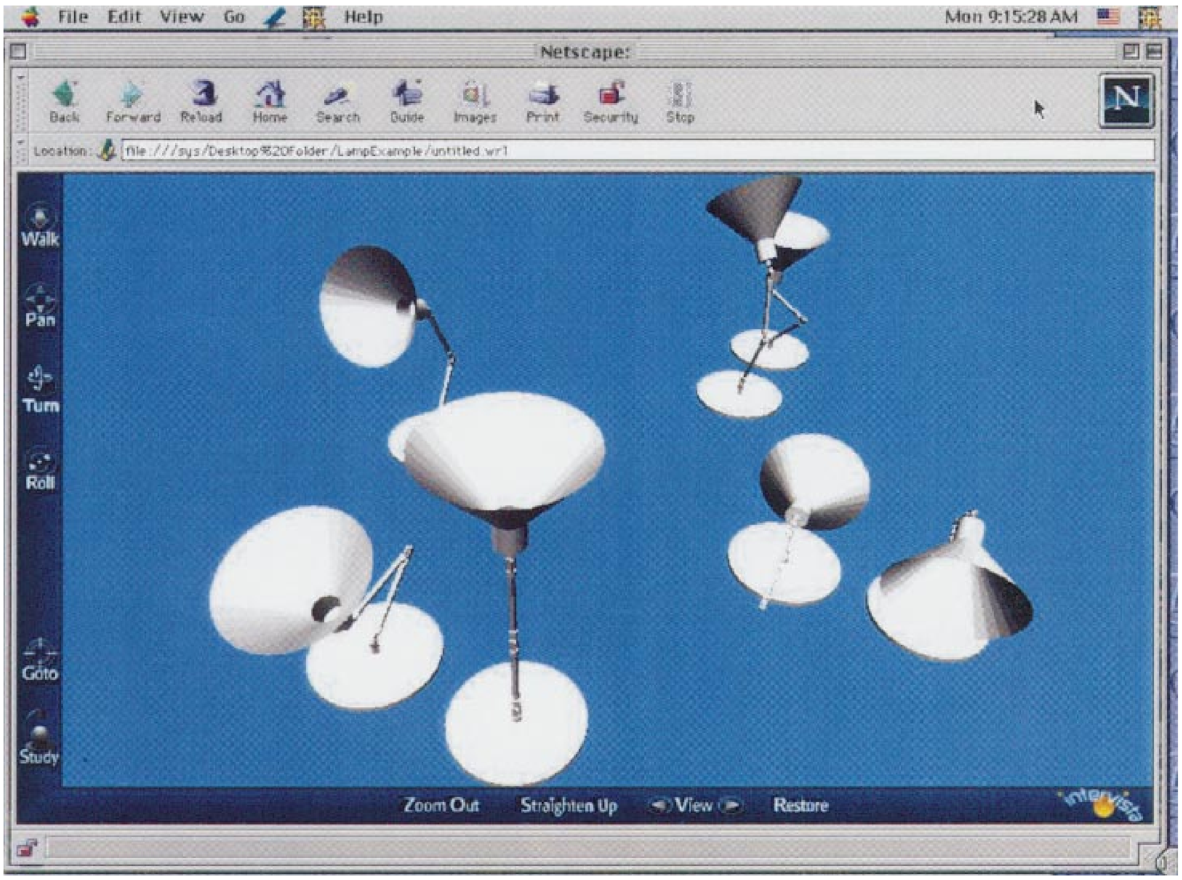

Fig. 12. Animation preview by super-imposition of segment key-frames.

suitably decelerating the first part and accelerating the second part of the (7,8) segment animation (see Fig. 12).

8. Conclusions and future directions

The present paper introduces a new approach for the symbolic definition of complex animated geometric scenes. The full integration between the design of both the shape of the elements in the scene and their dynamic behaviors is obtained by implementing both modeling and animation within the PLaSM language. For this purpose the language has been extended with a minimal set of new primitives shown to be sufficient to design complex animated scenes.

The flexibility of the presented methodology derives from the representation of the animation storyboard as a discrete action network that allows first to define/edit each animation segment independently and then to automatically integrate all the segments in the global animation. Moreover, any animation segment can be in turn decomposed into a local sub-network of lower-level animation segments in a hierarchical fashion.

The effectiveness of the present approach has been tested also in practice. A substantial test-bed was for example the large-scale reconstruction of a catastrophic event described in the companion paper [15]. In this case all the typical challenges that arise in dealing with real-life data were successfully met.

Further development of this research is planned along two main directions. First the PLaSM animation kernel needs further extensions to remove the current limitation to off-line definition of animated scenes. We plan to introduce the concept of online animation of reactive environments with the concurrent definition of alternative behaviors. Second we plan to improve the user-level interaction implementing a new visual interface for automatic generation of PLaSM functions. In this way we aim to relieve most of the time the user from the need to write directly PLaSM definitions.

Acknowledgements

The work of C. Baldazzi, A. Paoluzzi and M. Vicentino was partially supported by CERTIA Research Center project on visualization of simulation.

Appendix A

The PLaSM packages used in the examples are given here. Together with the code formerly given they constitute a full implementation of the examples, including Bézier curves of any degree and critical path method algorithm.

A.1. Geometry toolbox

The smallest set of geometry functions to implement the Luxo geometry is given here. Remember that PLaSM do not have a significant set of pre-defined shapes, but allows for quite simple implementation of needed shape generator functions

```
DEF Q = QUOTE~IF : ⟨IsSeq, ID, LIST⟩;
DEF ButLast = REVERSE~TAIL~REVERSE;
DEF circle(a :: IsReal)(r :: IsReal)
(n :: IsInt) =
  (S:⟨1,2⟩:⟨r,r⟩~JOIN):(MAP:
  ([cos,sin]~s1):((Q~#:n):(a/n)));
DEF TrunCone(r1,r2,h :: IsReal)
(n :: IsInt) =
  MAP:[x*cos~s2,x*sin~s2,z]:
  (Q:1*(Q~#:n):(2*PI/n))
WHERE
  x = K:r1 + s1*(K:r2 − K:r1), y = K:0,
  z = s1*k:h
END;
```

A.2. Motion toolbox

The small toolbox of utility functions to sample the [0,1] interval, to convert from degrees to radiants, etc. is given here. The recursive MkStruct function allows to transform a XHPC value with bounded parameters to a HPC value. It is used to generate a polyhedral actor value for a specified value of degrees of freedom

```
DEF Sampling(n :: IsIntPos) =
(AA:LIST~AA:/~DISTR):⟨0..n,n⟩;
DEF convert(seq :: IsSeqOf:IsReal) =
(AA:*~DISTL):⟨PI/180,seq⟩;
DEF XCAT = CAT~AA : (IF : ⟨IsSeq, ID, LIST⟩);
DEF time(tseq :: IsSeqOf:IsReal) =
(CAT~AA:(Bezier:(AA:LIST:tseq)));
DEF behavior(Constraints :: IsSeq)
(CSpath :: IsFun)(nSamples :: IsInt) =
  (aa:AL~trans~[time:Constraints,
  CSpath]):(Sampling:nSamples)
DEF MkStruct = IF:⟨OR~[IsFun,IsPol],
ID,STRUCT~AA:MkStruct > ;
```

A.3. Utility functions to query the graph luxo_pert

A full PLaSM implementation of the critical path method is given here. A more efficient solution is out the scope of the present paper. The functions rmin and rmax, respectively, return the minimum and maximum element of a sequence of reals; the inarcs and outarcs functions return the sequences of either the entering or the leaving arcs from a given node, where the arcs are represented as triples (see Section 7.2), and a graph is represented as a sequence of such triples

```
DEF greater(a,b :: IsReal) =
IF:⟨GT:a~s2,s2,s1⟩:⟨a,b⟩;
DEF lesser(a,b :: IsReal) =
IF:⟨LT~s2,s2,s1⟩:⟨a,b⟩;
DEF rmax(seq :: IsSeqOf:IsReal) =
INSR:greater:seq;
DEF rmin(seq :: IsSeqOf:IsReal) =
INSR:lesser:seq;
DEF inarc(node7 :: IsInt)(arc :: IsSeq) =
IF:⟨EQ~[K:node,S2],[[s1,S3]],K:⟨⟩⟩:arc;
DEF outarc(node :: IsInt)(arc :: IsSeq) =
IF:⟨EQ~[K:node,S1],[[s2,S3]],K:⟨⟩⟩:arc;
DEF inarcs(node :: IsInt)(graph :: IsSeq) =
(CAT~AA:(inarc:node)):graph;
DEF outarcs(node :: IsInt)
(graph :: IsSeq) =
(CAT~AA:(outarc:node)):graph;
DEF tmin(node :: IsInt) = rmax : (tpredecs)
WHERE
  predecs = inarcs : node : luxo_pert,
  tpredecs = IF:⟨~[LEN,K:0],K:⟨0⟩,AA:
  (+~[tmin~S1,S2])⟩:predecs
END;
DEFtmax(node :: IsInt) = rmin : (tsucces)
WHERE
  succes = outarcs : node : luxo_pert,
  tsucces = IF:⟨EQ~[LEN,K:0],
  K:⟨tmin:node⟩,AA:(−~[tmax~S1,S2])⟩:
  succes
END;
```

A.4. Bézier curves of any degree

Bézier curves of any degree *d* embedded in any dimensional space are given here. They are implemented as maps from a 1D polyhedral complex to the target *n*D space. The mapping is generated as a linear combination of the *d* + 1 control points with the Bernstein/Bézier polynomial basis of degree *d*

```
DEF Fact(n :: IsInt) = * : (CAT : ⟨⟨1⟩, 2..n⟩);
DEF BinCoeff(n,i :: IsInt) = Fact:n/
(Fact:i*Fact:(n − i));
DEF Bernstein(n :: IsInt)(i :: IsInt) =
  *~[K:(BinCoeff:⟨n,i⟩),**~[ID,K:i],
  **~[−~[K:1,ID],K:(n − i)]]~s1;
DEF BernsteinBase(n :: IsInt) =
AA:(Bernstein:n):(0..n);
DEF Bezier(ControlPoints :: IsSeq) =
(CONS~AA:(+~AA:*~TRANS)~DISTL):
  ⟨BernsteinBase:degree,
  (AA:(AA:K)~TRANS):ControlPoints⟩
WHERE
  degree = LEN : ControlPoints − 1
END;
```

A.5. Segment CS curves

The symbolic description of the CS paths of the lamp's segments is shown here. The example given in the paper is so completely reproducible.

```
DEF CSpath_0_1 = AA : ((Bezier~AA : XCAT) : ⟨
  ⟨100,0,convert : ⟨0,0,0,0⟩⟩,
  ⟨150,0,convert : ⟨30,30,0,−10⟩⟩,
  ⟨200,50,convert : ⟨ − 150,−20,90,0⟩⟩,
  ⟨200,100,convert : ⟨90 + 180,−60,105,60⟩⟩
⟩);
DEF CSpath_1_2 = AA : ((Bezier~AA : XCAT) : ⟨
  ⟨200,100,convert : ⟨90 + 180,−60,105,60⟩⟩,
  ⟨200,150,convert : ⟨60,−30,80,20⟩⟩,
  ⟨150,200,convert : ⟨20,−20,60,90⟩⟩,
  ⟨100,200,convert:⟨ − 90 + 180,
  − 20,30,30⟩⟩
⟩);
DEF CSpath_2_3 = AA : ((Bezier~AA : XCAT) : ⟨
  ⟨100,200,convert:⟨ − 40 + 180,
  − 20,30,30⟩⟩,
  ⟨50,200,convert : ⟨ − 20,30,0,−10⟩⟩,
  ⟨0,150,convert : ⟨ − 40,−20,90,0⟩⟩,
  ⟨0,100,convert : ⟨90 + 180,−30,55,40⟩⟩
⟩);
DEF CSpath_3_4 = AA : ((Bezier~AA : XCAT) : ⟨
  ⟨0,100,convert : ⟨90 + 180,−30,55,40⟩⟩,
  ⟨0,50,convert : ⟨ − 120,30,0,−10⟩⟩,
  ⟨0,20,convert : ⟨ − 60,−20,−55,20⟩⟩,
  ⟨0,0,convert : ⟨0,−0,0,0⟩⟩
⟩);
DEF CSpath_5_6 = AA : ((Bezier~AA : XCAT) : ⟨
  ⟨100,−100,convert : ⟨0,60,−60,90⟩⟩,
  ⟨100,−50,convert : ⟨30,0,0,−10⟩⟩,
  ⟨150,0,convert : ⟨ − 150,−20,90,0⟩⟩,
  ⟨150,100,convert : ⟨90,−90,145,45⟩⟩
⟩);
DEF CSpath_6_7 = AA : ((Bezier~AA : XCAT) : ⟨
  ⟨150,−100,convert : ⟨90,−90,145,60⟩⟩,
  ⟨150,150,convert : ⟨60,−60,80,80⟩⟩,
  ⟨250,250,convert : ⟨20,60,−60,90⟩⟩,
  ⟨100,250,convert : ⟨ − 40,45,−130,90⟩⟩
⟩);
DEF CSpath_7_8 = AA : ((Bezier~AA : XCAT) : ⟨
  ⟨100,250,convert : ⟨ − 40,45,−130,130⟩⟩,
  ⟨ − 50,250,convert : ⟨ − 20,−30,20,0⟩⟩,
  ⟨100,100,convert : ⟨ − 40,20,−90,−50⟩⟩,
  ⟨50,0,convert : ⟨90,45,−145,80⟩⟩
⟩);
```

References

[1] Alias Wavefront (Silicon Graphics). Maya. http://www.aw.sgi.com/entertainment/solutions/complete/index.html

[2] Alias Wavefront (Silicon Graphics). Power Animator. http://www.aw.sgi.com/pages/home/pages/products/pages/poweranimator

[3] Bajaj CL, Anupam V. SHASTRA—an architecture for development of collaborative applications. International Journal of Intelligent and Cooperative Information Systems 1994;3(2):155–172.

[4] Bajaj CL, Cutchin S. The GATI client–server animation toolkit. In: Thalmann N, Thalmann D, editors. Proceedings of Computer Graphics International, CGI93. Communicating with Virtual Worlds, Berlin: Springer, 1993. pp. 413–423.

[5] Backus J, Williams JH, Wimmers EL. An introduction to the

programming language FL. In: Turner DA, editor. Research topics in functional programming, 1990.
[6] Electric Image. http://www.electricimage.com/product/ei/index.html
[7] Foley J, van Dam A, Feiner A, Hughes J. Computer graphics: principles and practice, 2. Reading, MA: Addison Wesley, 1993.
[8] Gleicher M. Retargeting motion to new characters. ACM Siggraph 98 Conference Proceedings, Annual Conference Series, Reading, MA: Addison Wesley, 1998. pp. 33–42.
[9] Grzeszczuk R, Terzopoulos D, Hinton G. NeuroAnimator: fast neural network emulation and control of physics-based models. ACM Siggraph 98 Conference Proceedings, Annual Conference Series, Reading, MA: Addison Wesley, 1998. pp. 9–20.
[10] Kelley JE, Walker MR. Critical path planning and scheduling. Proceedings of the Eastern Joint Computer Conference, 1959.
[11] Kinetics. 3D Studio Max. http://www.ktx.com/3dsmaxr3/
[12] NewTek. LightWave. http://www.newtek.com/products/frameset_lightwave.html
[13] Paoluzzi A, Pascucci V, Vicentino M. Geometric programming. A programming approach to geometric design. ACM Transactions on Graphics 1995;14(3):266–306.
[14] Paoluzzi A, D'Ambrogio A. A programming approach for complex animations. Part II. Reconstruction of a real disaster. Computer Aided Design 1999;31:711.
[15] Pascucci V, Ferrucci V, Paoluzzi A. Dimension independent convex-cell based HPC: skeletons and product. International Journal of Shape Modeling 1996;2(1):37–67.
[16] Paoluzzi A, Francesi S, Portuesi S, Vicentino M. Rapid development of VRML content via geometric programming. Fifth Eurographics Workshop on Virtual Environments. Vienna, Austria, 31 May–1 June 1999.
[17] Pausch R, Burnette T, Capeheart AC, Conway M, Cosgrove D, DeLine R, Durbin J, Gossweiler R, Koga S, White J. Alice: rapid prototyping system for virtual reality. IEEE Computer Graphics and Applications 1995;15(3):8–11.
[18] Richter-Gebert J, Kortenkamp UH. The interactive geometry software Cinderella (Interactive geometry on computers, New York: Springer, 1999.
[19] Side Effects Software. Houdini. http://www.sidefx.com/product/index.html
[20] SoftImage. SoftImage V3.8. http://www.softimage.com/Products/3D/default.htm
[21] Watters A, van Rossum G, Ahlstrom J. Internet programming with Python, New York: MIS Press/Henry Holt publishers, 1996.
[22] Wernecke J. The inventor mentor, New York: Addison Wesley, 1994.