Authoring Communicating Agents in Virtual Environments

Christian Geiger**, Georg Lehrenfeld*, Wolfgang Mueller**

* Heinz Nixdorf Institut, ** C–LAB
Fuerstenallee 11, 33102 Paderborn, Germany
http://www.c-lab.de/vis

September 4, 1998

Abstract

3D-graphics popularity has steadily increased in a number of areas such as entertainment, scientific visualization, simulation, and virtual reality. Despite this rapid growth the authoring of animated 3D objects in virtual environments is still by no means trivial. This article presents new concepts of the animated 3D programming language SAM (Solid Agents in Motion) and its programming environment. In SAM, the main syntactic objects like agents, rules, and messages are represented as 3D objects. The design of a SAM program is supported by a dedicated 3D structure editor. The editor allows the definition and spatial arrangement of SAM agents in a 3D scene by direct manipulation. The paper gives a number of SAM examples, demonstrating the authoring of simple animated virtual 3D scenarios.

1 Introduction

With the increasing availability of 3D-supporting low-cost hardware and software, static and dynamic 3D representations have become the subject of interest for a wide range of applications. Today, 3D applications mainly focus on 3D games, various forms of virtual reality applications, and illustrations for technical presentations.

Currently, authoring of multimedia presentations is supported by various visual approaches based on different paradigms [14]. Authorware or IconAuthor, for instance, define the interaction between objects by icon–based flow diagrams. Other systems like Quest and Apple Media Kit support the navigation through icon (frames) palettes with conceptual links. HyperCard-based systems provide navigation with indexed cards. Popular approaches like Macromedia Director use a cast/score paradigm provided by parallel tracks (time lines) for combining multimedia objects over synchronous time points. These approaches are currently applied to interactive 2D or pseudo-3D applications. For authoring of virtual 3D environments only limited graphical support by diagram editors exists (e.g., CosmoWorlds, Authorizer [12] or work described in [4, 2, 15]. Many of the currently available authoring systems for 3D come only with textual scripting languages like Tcl/Tk or Python [11, 16, 1]. They are all based on indirect manipulation as far as the authoring of their behavior is concerned. The object's interaction is controlled separately by textual or visual 2D means rather than being directly manipulated within the virtual environment.

In this paper, we introduce the 3D programming language SAM (Solid Agents in Motion) and its programming and animation environment. SAM is a parallel, state-oriented, general purpose programming language. SAM agents synchronously communicate by exchanging messages. Agents are specified by the means of production rules each with a condition and a sequence of actions. In SAM, the main syntactic objects are 3D where subelements are partly given in textual form, which are displayed when moving the mouse over the corresponding graphical object. Individual execution steps are animated in 3D by smooth continuous motion and scaling of individual objects. Therefore, 3D agents can be inspected and directly manipulated even during their run time, i.e., animation. Due to the individual application, arbitrary 3D representations can be assigned to agents and messages. Though SAM, in principle, is applicable as a general purpose programming language, in the re-

mainder of this paper we focus on SAM's application for authoring communicating agents in virtual 3D worlds. In the following paper, we first discuss related work in the field of authoring 3D scenarios and direct manipulation of 2D/3D animations. Thereafter we introduce the SAM language and its programming environment and conclude with a few examples.

2 Related Work

Authoring the behavior of virtual 3D objects is currently investigated by few approaches. These approaches mainly concern textual scripting languages and 2D flow diagrams.

Brown and Najork have introduced Oblique as an object–oriented untyped scripting language for their animation system Oblique–3D [11] which supports the creation and animation of graphical objects in 3D scenes. Recently, MacIntyre and Feiner extended Oblique–3D to a distributed 3D graphics library for prototyping distributed virtual environments [9]. Alice [16] uses the object–oriented script language Python. Action routines change the object's internal state. Animation is implemented by the manipulation of list elements. Dive was introduced for the development of distributed virtual worlds [1]. A Dive world is composed of a global world description, Dive objects, views, lights, and actors. The behavior of objects is defined by tcl/tk scripts enhanced by specific Dive commands.

Only few systems support a visual specification of 3D animations. DIAL (described in [4]) retains features of linear list notations and displays the sequence of events as a series of parallel rows of marks. Actions are specified by textual means. S-Dynamics (also in [4]) combines parametric descriptions of actions with a bar chart visualization. Complex actions are represented as hierarchical charts. In MAM/VRS [2] structure and behavior is symmetrically defined in two separate graphs. The behavioral nodes of the behavior graph are related to geometry nodes via constraints. MAM/VRS uses an object–oriented extension of Tcl/Tk for manipulation as well as a visual notation for constructing 3D widgets. Stevens et. al have introduced a toolkit architecture for 3D–widgets and interactive illustrations supporting manipulation of 3D primitives through a visual 2D language [15]. For our own animation library AAL [5] we introduced a visual notation for objects with keyframe animations [3] based on concepts of Pictorial Janus, a visual logic programming language[8]. For direct manipulation of animated objects we built a prototype 3D-editor directly supporting the design of animated objects.

To our knowledge, in 2D/3D animation, only some approaches support direct behavioral manipulation of animated objects and only a real 3D visual programming languages exist. In Repenning's Agentsheets [13] a designer creates a visual language by defining the look (depiction) and behavior of agents. The look is defined by the use of a depiction editor. An agent interacts on a grid explicitly with other spatially related agents. Depictions actually display the different states of the agent. Agents are defined in terms of graphical rewrite rules. The condition of a rule is given by the depiction of the neighbors representing their current states. The next state can be simply defined by discretely changing their depiction. This also defines a basic animation where the current state of the program is given by the discrete change of the current depiction. Kahn's ToonTalk [7] is a completely visual 2.5D programming environment and language for kids which combines programming with video-games character animation. The programmer, in form of a woman or man, freely walks or flies a helicopter through a city. A set of houses represent the program. A ToonTalk program is a network of concurrently communicating agents exchanging messages. Agents are basically defined in terms of guarded rules through teaching robots. Spawning and deleting agents correspond to building and exploding of houses. Sending a message is executed when passing a box to a bird. Each bird is associated with a nest. When receiving a message, the birds carries it to its nest. Unfortunately, due to fixed representation of ToonTalk characters the tool is not applicable for general purpose programming in other domains. Cube [10] is a 3D data-flow language based on higher order horn logic while 3D-visulan is a 3D variant of Bitpict [17].

Like in Agentsheets and ToonTalk, in SAM the programmer does have to manage different environments and languages when specifying a program and its animation. He/she directly manipulates the program objects in 3D without leaving the environment. In contrast to all of the above 3D scripting languages, SAM can be regarded as a very high-level language with respect to its visual extent (3D keyframe animation). Like in Agentsheets, the SAM user can easily author domain-oriented animation for end users.

3 Visual Programming in 4D

SAM (Solid Agents in Motion) is an animated 3D visual programming language with textual parts.

SAM is a synchronous parallel, typed, and state-oriented programming language and an advanced programming environment.

3.1 SAM Language

A SAM program is a set of agents which communicate by exchanging messages. An agent is defined by a set of production rules which execute a sequence of actions. SAM agents and messages can have both an abstract and a concrete 3D representation, i.e., arbitrary forms which directly correspond to the objects' meaning within the program.

The main SAM objects are 3D. These are messages, agents with ports, and rules with precondition and actions. The individual values of messages as well as the expressions and instructions in conditions and actions are specified as text. When moving the mouse over the individual graphical objects the object becomes transparent and displays the enclosed textual identifier, expression, or value. Agents are 3D objects with

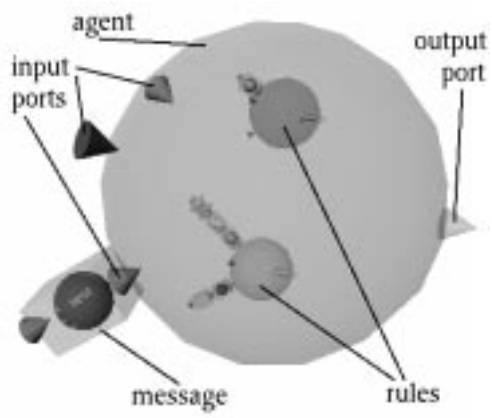

Figure 1: Abstract Presentation of an Agent

an arbitrary number of input and output ports at its outside. Figure 1 shows its abstract representation. The agent receives and sends messages via these ports. The different ports can be roughly distinguished by their different colors. Input and output ports can be distinguished by the direction of the representing cone, i.e., whether the cone's peak points to the agent or to another agent. In their abstract representation, agents are initially represented as transparent spheres with ports at the outside and a set of rules enclosed. An agent has an initial state. The state can be explicitly changed by a specific action when executing a rule.

The three different basic types of messages can be distinguished by their different shapes, i.e., sphere, diamond, and cylinder. In its abstract form, a message is given as a transparent box with two cones at its outside for connecting to other messages or agents. Different data types are represented by diamond (integer), sphere (string), or cylinder (arbitrary datatype). A message carries the (textual) information of its value and its sender and receiver. Figure 1 shows a message of type string currently connected to the port of an agent.

Rules are located inside the agent's object. Rules are copies of the agent's graphical representation with objects representing the precondition at their outside. A sequence of actions given by "sliced" pieces of a tube are enclosed in the rule's body. Each rule corresponds to a production rule of form

$$IF(C_1\&...\&C_n)THEN\,A_1;...;A_m$$

with condition C_i and actions A_j. Figure 2 gives an example of a rule with different conditional values at the upper and lower port and three actions. Conditions may be specific values, types, any item, no item or "don't care". The small ring inside the rule is the scanner which marks action A_1. Currently supported actions are listed in the following table but the SAM language can be easily extended by new actions or data types.

Action	Description
SEND	send message to agent
SENDTOALL	broadcast
REPLY	send answer
CHSTAT	change the agent's state
CPAGENT	create a copy of this agent
DESTROY	delete this agent
MOVE	move agent (position constant)
MOVEI	move agent (position variable)
ROTATE	rotate agent
START	start external process
KILL	kill external process
PLAY	play sound
SCALE	scale agent
COLOR	change color
MOVIE	animated texture (mpeg file)

The SAM interpreter iteratively executes two phases. In the first phase all agent check the conditions of their rules. The first rule which matches a specified state S and another condition C is selected and its actions are executed in sequential order. When executing a send command the message is not immediately sent but temporarily scheduled at the output port. Other actions like moving, rotating the agent are immediately executed. The second phase begins when all agents have completed the execution of their actions. Then all messages are sent to the specified destination. Once all messages have arrived at their

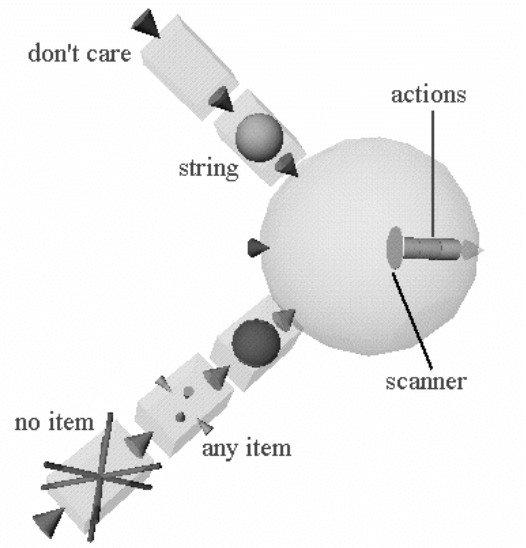

Figure 2: SAM Rule

destination the first phase triggers the pattern matching again, etc. The execution of abstract SAM programs is visualized by smooth and continuous animation. When a rule of an agent is selected by pattern matching it is enlarged to the size of the agent and the currently executed action is marked by a scanner.

3.2 SAM Programming Environment

Visual programming becomes only effective if dedicated tools for program development are available. Editors and debuggers for direct manipulation are paramount components for visual programming environments. SAM comes with a dedicated graphical editor for creating and manipulating SAM programs and a visualizer for observing the execution of animated programs. The editor saves the logical structure and the graphics in separate files. The visualizer can load those files and perform an animation without additional information (cf. Figure 3). Both tools are implemented on top of our animation library AAL (Animated Agent Layer) which provides high–level animation functions. The complete environment is implemented on SGI with C++ and OpenInventor. We first introduce the basic concepts of AAL. Thereafter, we introduce the basic functions of the SAM graphical entry and visualizer.

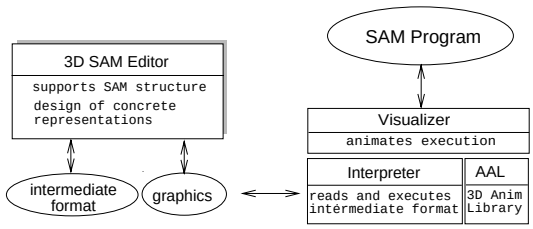

Figure 3: SAM Tool Environment

3.2.1 AAL

AAL is a system for prototyping of 4D scenes and provides layers of different abstractions with predefined functionality of elements [5]. The lowest level *A1: Open Inventor* is provided by a subset of the 3D graphics library Open Inventor. This subset should ensure that AAL applications run efficiently even on mid range hardware. Level *A2: Static Objects and Abstract Animations* distinguishes two sorts of elements: static objects for the description of scene elements and (abstracts) animation methods which provide animation basics. The group of static elements covers objects for the description of 3D scenes. For prototyping basic primitives like cone, sphere, cube, cylinder and external models are sufficient. Properties like size, position, orientation or material are attributes of the corresponding object. Predefined methods exist which can modify the object's attributes. In contrast to the scene graph based approach of most current graphic libraries, static AAL objects are completely object-oriented.

The *A2* class for abstract animation methods supports parametric keyframe animations and algorithmic animations. Methods for keyframing support translation, rotation, scaling, material change and animated textures. An animation is specified in time or with a given speed and may use linear or accelerated / decelerate interpolations. While keyframing declaratively specifies an animation algorithmic animation methods describe the dynamic in an operational way by continously modifying object properties. We defined commonly used algorithmic frames like repetition, inversion, pendulum or shuttle as animation skeletons which can be instantiated with a keyframe animation method.

The level *A3: Animated Objects* combines the level *A2* objects to animated 3D objects. Based on level *A2* classes level *A3* distinguishes AnimatedShape, AnimatedLight, and AnimatedViewer and each animated object possesses a predefined set of inherent anima-

tion methods. Animation methods can be executed sequential or parallel. Basic animations are combined to complex operations. Sequential execution of animations is defined through animation lists. The elements of a list are animation commands. Multiple lists allow a parallel execution of commands. This highest level *A4: Animated Agents* covers interactions and relations between objects by perceiving their environment and the ability to autonomously plan actions. Animated objects become intelligent animated objects. Behavior and strategies of agents and their interactions or relations are modeled by means of rules or constraints. We do not describe these functions in further detail since they are not important for the remainder of this article.

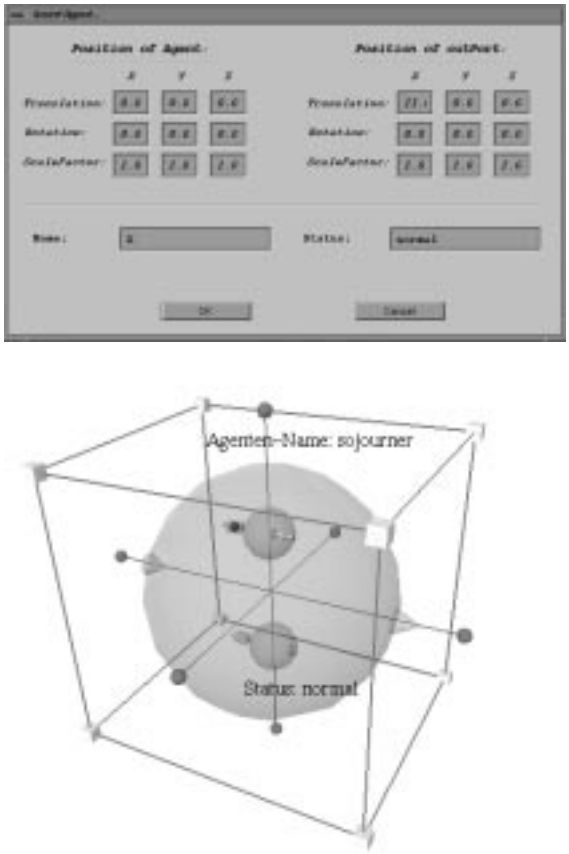

Figure 4: Specifying an Agent

3.2.2 SAM Editor

The SAM graphical editor provides a combination of form–based capture with direct manipulation of 3D objects. When creating an agent the user first inserts relevant data in a form. Thereafter, the graphical object in its abstract representation can be adjusted by the means of a transformer such as it is shown in Figure 4. Graphical detail can be directly modified with the transformer enclosing the object. The SAM editor provides the user with a set of predefined 3D-primitives for designing concrete SAM objects, standard editor functionality (e.g.,copy, paste, group, etc) and a set of functions for precise relative positioning. These functions include the snapping to sides, edges and points of a selected target object and the approximation of coordinate and rotation values.

While through the form the user can specify the approximate location exact positioning can be performed by means of direct manipulation in 3D. In the abstract presentation non-graphical data like identifier and current status are given as a texual anontations close to the object. The values can be modified through double–click on the corresponding 3D object.

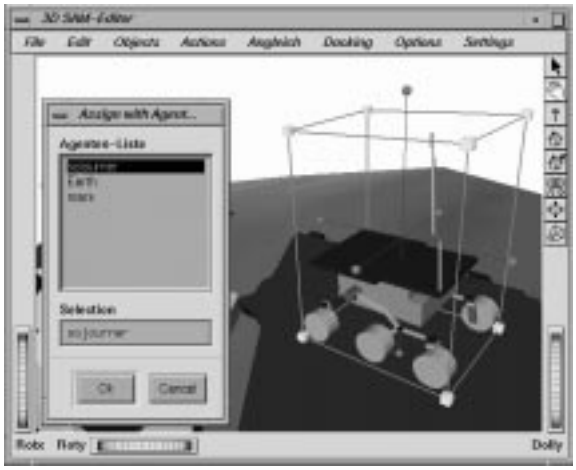

Figure 5: Assigning a Concrete Representation

After having created an agent the user can assign a concrete representation to it. For this the concrete representation can be selected from a list of available OpenInventor files (cf. Figure 5) or the user can design his own simple models. Messages are correspondingly created and assigned, thus not further outlined here.

Once at least one agent has been created, the user can select this agent and specify a rule for it. For this purpose a form opens for specifying the condition. After closing the form the graphical rule object can be moved to its required location within the agent's body.

After selecting the graphical rule object an arbitrary number of actions can be assigned to the rule's

Figure 6: Specifying a Rule

body. For each action there is a predefined form for entering the individual data. Figure 6 gives the form for defining the rotation of an agent.

3.2.3 SAM Visualizer

Having specified the program and its input data the visualizer can execute the program which is visualized by an animation of the abstract and concrete representation of the program. At any time during animation, the user can switch between the two. In the abstract view the user can observe the animation of the individual computation steps like selecting rules, consuming and producing messages, as well as the execution of the individual actions. For this purpose abstract 3D objects are semi–transparent. This representation which is dedicated to programming experts helps in functional debugging and allows the easy detection of faulty behavior, i.e., incorrect actions and their order of execution as well as incorrectly duplicated or lost messages. The animation of the concrete representation represents agents and messages as application-specific 3D objects and allows only to graphically observe the exchange of messages.

4 Examples

We demonstrate SAM by three simple examples. The first illustrates a simple producer–consumer example. The second example models the interaction of sojourner and path finder during their latest mars mission. The final example gives the reaction of pendulae on impulses.

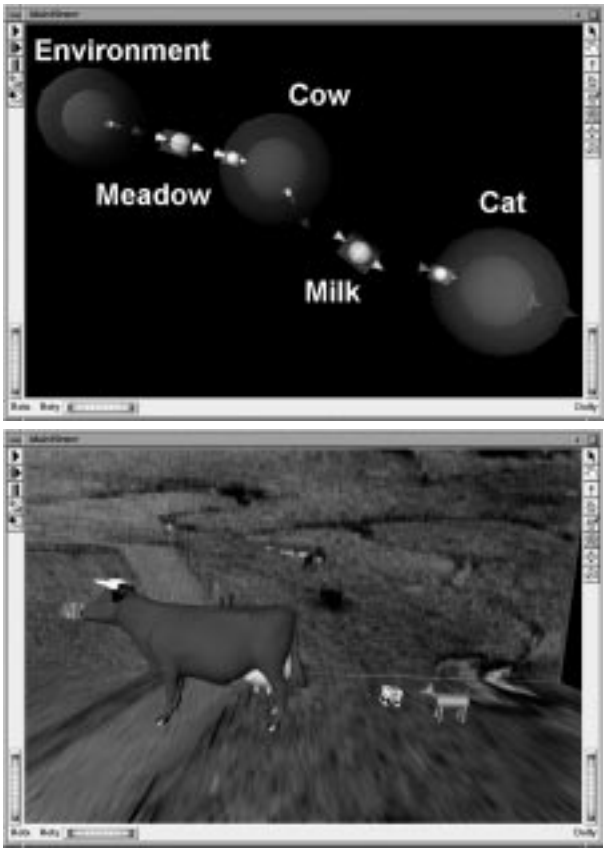

Figure 7: SAM Wildlife

4.1 Producer-Consumer

We first define a simple interaction of a producer with a consumer. Consider for this scenario a very simplified wildlife with the following behavior: *meadow feeds cow, cow eats hay and produces milk* and *cat consumes milk*. In this scenario we have 3 agents: environment (meadow), cow, and cat as it is shown in the abstract model of Figure 7 from left to right. In the concrete presentation below the meadow is represented by the texture in the background. The screenshots give a snapshot of their animations at the same time. The meadow has one rule for generating bunches of hay. The rule produces one bunch in each SAM execution cycle. That explains why we have chosen a synchronous execution model. The cow agent has one simple rule with one action for producing milk when

receiving a message. The cat is specified by one rule which consumes one portion of milk in each execution cycle.

4.2 Mars Mission

In the mars mission scenario we have three different agents: earth, path finder (space ship), and sojourner (robot vehicle) with following behavior:

1. earth sends request for photo
2. path finder forwards request to sojourner
3. sojouner moves to rock, takes a photo, returns photo to path finder
4. path finder forwards photo to earth

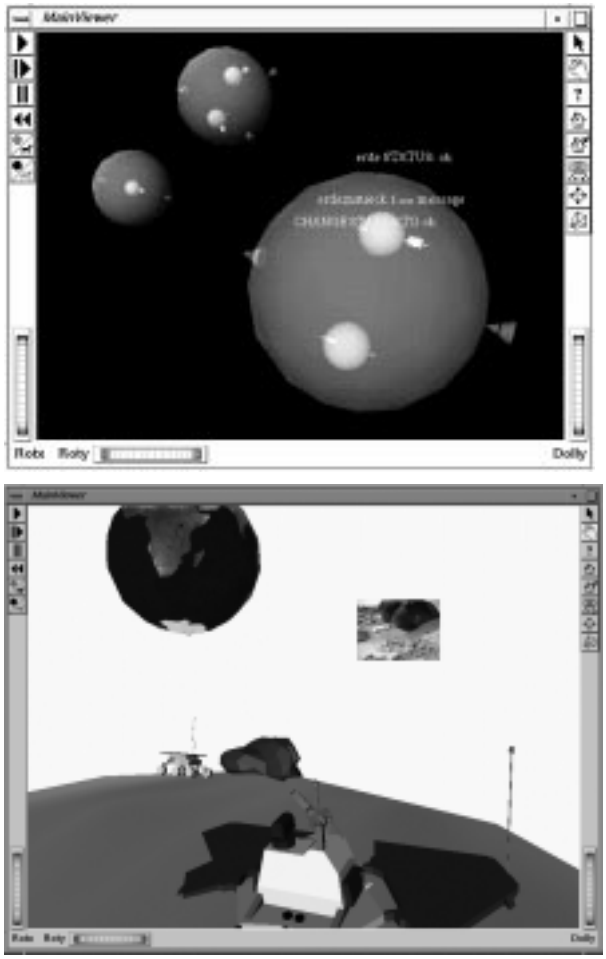

Figure 8: SAM Mars Mission

Figure 8 gives the abstract and concrete presentation of that scenario. The abstract presentation shows the earth agent in the front. That agent has two rules. One for sending a request and another for receiving photos. Sojouner on the left has one rule with 6 actions: rotate to direction of the rock, move to rock, produce photo, rotate to direction of path finder, move to path finder, and send photo to path finder. Path finder has two rules. One rule for forwarding messages in each direction.

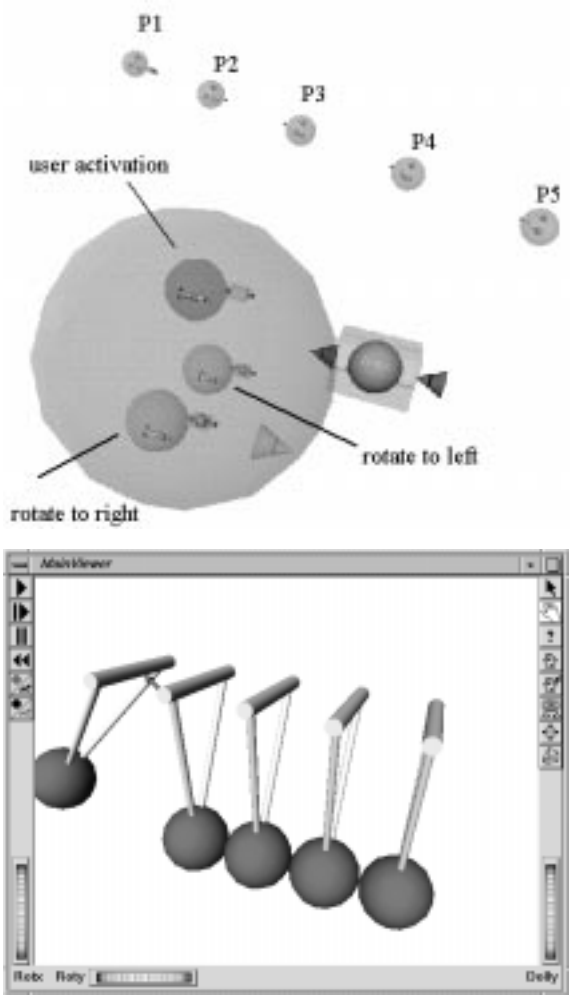

Figure 9: SAM Pendulae

4.3 Pendulum

The final example demonstrates the interaction of 5 pendulum agents. They all have the same behavior when receiving an impulse. First, the pendulum rotates around the mounting point to the top, then it rotates back and sends the impulse back. To ensure

a proper animation the first pendulum P1 sends its impulse to the last pendulum P5, P2 to P4 and P3 to itself. The initial impulse is generated by the user by clicking on the corresponding agent. This animation behavior is a simplification of the real behavior of the pendulae. In SAM, impulses are modeled as messages. In the concrete presentation of Figure 9 the message has the form of a tiny arrow currently attached to the mounting point of the leftmost agent. Each agent has a rule with 3 actions: decelerated rotation to the top, accelerated rotation back, and the generation of a message. In each agent three rules exist for the initial activation by the user and for the rotation of the pendulum to the left and to the right.

5 Conclusion

We have presented a new approach for authoring the behavior of virtual 3D scenarios by direct manipulation. For this we have introduced the visual 3D programming language SAM and its programming environment. The examples given in this paper demonstrate how SAM supports the prototyping of virtual 3D scenarios. The produce-consumer example was developed within 1 hour and results in 30 lines of the textual intermediate code. The mars mission example took approximately one day (170 lines of code). Most of the effort went into the design of the concrete representation. Other SAM examples include simple algortihms like factorial or sorting and small virtual world with flowers and bees where agents (flowers) are created and destroyed.

SAM provides an abstract 3D view for authoring the behavior of 3D objects. This view supports the first visual inspection of the program when checking the I/O relations of rules within an agent by direct animation of their execution. For the easy migration to an advanced end-user representation the visual representation of each agent can be easily changed to an application–specific representation by simply assigning external OpenInventor files to messages and agents.

Though SAM is presented here as a 3D language for authoring animated 3D objects it has been designed for general purpose programming. The underlying synchronous execution model has been derived from synchronous programming languages like Esterel or Lustre. Thus, next investigations have to demonstrate how SAM can be applied to prototyping of more advanced reactive systems. Moreover, the SAM programming environment will serve as a testbed for our ongoing work in animated 3D user interfaces [6].

References

[1] C. Carlson and O. Hagsand. Dive- a multi user virtual reality system. In *IEEE Annual Virtual Reality Symposium*, 1993.

[2] J. Doellner and K. Hinrichs. Interactive, animated widgets. In *Computer Graphics International*, June 22–26, Hannover, Germany 1998.

[3] M. Duecker, Ch. Geiger, R. Hunstock, and W. Mueller. Visual-textual prototyping of 4d scenes. In *Proc. 1997 IEEE Symposium Visual Languages*, Capri, Italy, Sept 1997.

[4] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice (2nd Ed.)*. Addison-Wesley, 1996.

[5] C. Geiger. Prototyping of interactive animated 3d objects. In *Proc. of the Simulation and Visualization 1998*, Magdeburg, Germany, March. 1998.

[6] C. Geiger and V. Paelke. Enhancing 3d user interfaces with animation principles encapsulated in agents. In *ECAI '98 workshop "Combining AI and Graphics for the Interface of the Future*, Brighton, UK, August 1998.

[7] K. Kahn. Drawing on napkins, video-game animation, and other ways to program computers. *Communications of the ACM*, Vol. 39, No. 8:49–59, Aug 1996.

[8] K. Kahn and V.A. Saraswat. Complete Visualization of Concurrent Programs and their Execution. In *1990 IEEE Workshop on Visual Languages*, pages 7 – 15, Skoje, IL, October 1990.

[9] B. MacIntyre and S. Feiner. A distributed 3d graphics library. In *SIGGRAPH98*, Seattle, WA., July. 1998.

[10] M. Najork and S. Kaplan. The cube language. In *Proc. 1991 IEEE Workshop Visual Languages*, pages 218–224, Kobe, Japan, 1991.

[11] Marc A. Najork and Marc H. Brown. Oblique-3d: A high-level, fast-turnaround 3d animation system. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):175–194, June 1995.

[12] Cybelius Home Page. http://www.cybelius.com. Technical Report Authorizer 1.0, Cybelius Software, 1998.

[13] A. Repenning and T. Summer. Agentsheets: A medium for creating domain–oriented visual languages. *IEEE Computer*, 28(3), March 1995.

[14] J. Siglar. Multimedia authoring systems faq, 7 June, 1998. www.tiac.net/users/jasiglar/MMASFAQ.HTML.

[15] M. Stevens, R. Zeleznik, and J. Hughes. An Architecture for an Extendible 3D Interface Toolkit. Technical report, Dept. for Computer Science, Brown University, April 1994.

[16] UVA User Interface Group. Rapid prototyping for virtual reality. *VR Blackboard, IEEE Computer Graphics and Applications*, 1995.

[17] K. Yamamoto. 3d-visulan: A 3d programming language for 3d applications. In *Pacific Workshop on Distributed Multimedia Systems*, 1996.

We thank Waldemar Rosenbach, Ralf Wegener and Rolf Zelder for implementing the SAM system and Volker Paelke and Sophie O'Halloran for proof reading.