EditION: A Collaborative Calligraphic Tool to Manage Virtual Environments

Alfredo Ferreira* Marco Vala Guilherme Raimundo† J.A.Madeiras Pereira
Joaquim A. Jorge Ana Paiva

Department of Computer Science and Engineering
INESC-ID/IST/Technical University of Lisbon

Abstract

Simulating the dynamics of real worlds using Virtual Environments (VEs) is a growing area with many interesting applications. There are tools based on graphical interfaces to create and visualise these VEs, although most of them are specific to a particular type of environment. Following recent developments in calligraphic systems, which have emerged as a viable alternative to conventional direct manipulation interfaces, we devised an approach that takes advantage of these systems to manage the elements in a VE.

In this paper we present a tool that allows users to manage VEs collaboratively using a sketch-based interface together with a simple visual language that generically describes the elements in a VE. To validate our work we did a small test case with the simulation of a world populated by synthetic characters.

1 Introduction

The real world can be seen as a collection of many entities from a simple grain of sand to an intelligent human being. Furthermore, these entities are not static. They are constantly interacting, changing their properties and thus creating an ever mutating universe.

We are often interested on simulating these dynamics in a virtual manner. Maybe we need to better understand the real world such as the movement of crowds in a building on fire or the growth of a plant in a greenhouse due to the manipulation of humidity and temperature. Maybe we just want to impersonate a hero on a quest for entertainment purposes. However, to achieve such goals, we need to have the means to manage these simulations which to some extent is to manipulate and visualise the associated virtual environment (VE).

We use the *ION* framework to handle the simulation details, ensuring that the rules of the world are followed, and propose *EditION* as a calligraphic tool that controls it. *EditION* allows symbolic representation and manipulation of elements in a VE through a sketch based interface based on a visual language for VEs. Moreover, this manipulation is done in a distributed collaborative fashion both before or during the simulation.

The *EditION* tool is an evolution of the preliminary work presented in [9]. It uses a calligraphic interface to take advantage of the way developers sketch VEs in sheets of paper before creating them. To that end we developed a visual language to represent the various elements in a VE which allows users to sketch a symbolic representation of each element directly on the computer, reducing the amount of scripting and/or textual input.

After a short discussion of the related work, we will present an overview of our approach to sketch-based management of VEs. We briefly describe the architecture and the necessary modules. Then we focus on the creation of VEs using sketches, depicting the visual language and the symbol recognition methodology as well as a description of the user interaction with our tool. Finally we present some conclusions and suggest directions for future work.

2 Related Work

Although no calligraphic interfaces had been devised for virtual environment management, several experimental sketch-based systems were developed in recent years for a number of different areas, such as interface design, mechanical systems simulation or control systems analysis. SILK [15] is an interactive tool to sketch interfaces using an electronic pad and stylus. Designers can use SILK to quickly sketch the user interface and, when they are satisfied with the early prototype, produce a complete and operational interface. A similar tool, JavaSketchIt, was presented

*A.Ferreira was supported by the Portuguese Foundation for Science and Technology, grant reference SFRH/BD/17705/2004.

†G.Raimundo was supported by the Portuguese Foundation for Science and Technology, grant reference SFRH/BD/25725/2005.

VL/HCC Workshop: Sketch Tools for Diagramming
Herrsching am Ammersee, Germany
15 September 2008
Editors: Beryl Plimmer & Tracy Hammond

by Caetano *et al.* [6] and generates a Java interface based on hand-drawn compositions of simple geometric shapes. The JavaSketchIt evaluation concluded that users consider their sketch-based system more comfortable, natural and intuitive to use than traditional mouse-based tools.

Alvarado and Davis [1] developed ASSIST, a program that produces simple 2D mechanical devices from hand-drawn sketches. This system performs real-time interpretation, as the sketch is being created, using a set of heuristics to construct a recognition graph containing the likely interpretation of the sketch and selects the best one based on both contextual information and user feedback. A similar approach was followed by Hammond and Davies [11, 12] in Tahuti, their recognition environment for class diagrams in UML.

Hong and Landay [13] developed a Java toolkit to support the creation of pen-based applications. Using this framework, Lin *et al.* [16] created DENIM, a sketch-based system that helps web site designers in the early stages of the design process. SketchySPICE [13] is another program developed using SATIN. It consists in a calligraphic interface for SPICE, a circuit CAD tool developed at University of California at Berkeley. In this editor where users can draw simple circuits gates in two distinct modes. In *immediate* mode recognised sketches are immediately replaced by its formal symbol, while in *deferred* mode recognised objects are left sketchy but the recognised symbols are drawn translucent behind the sketch, in order to give some feedback to users.

More recently, Kara and Stahovich [14] presented the SIM-U-SKETCH, an experimental sketch-based interface for Matlab's Simulink software package, an add-on package for analyzing feedback control system and other similar dynamic systems. With this interface users can sketch functional Simulink models and interact with them, modifying existing objects or add new ones. SIM-U-SKETCH was designed to allow users to draw as they would do on paper, with no constraints imposed by the recognition engine. To that end, this system employs a *recognise on demand* strategy in which the users have to explicitly indicate whenever they want the sketch to be interpreted. Then, once the sketch is finished, it is interpreted by the system in order to become a Simulink model.

Liwicki and Knipping [17] developed a system for recognize sketched logic circuits and graphically simulate them. In this solution even the simulation is controlled through calligraphic interaction. It allows specification of input values by hand-writing the corresponding numbers. However, the sketches were never converted to precise diagram and the simulation is represented over the sketched circuit.

Despite their apparent similarity, distinct approaches and strategies are used in the systems referred above. We studied the advantages and drawbacks of these methodologies and used them to devise a collaborative calligraphic interface to create and manage VEs in the context of *ION* framework.

3 Overview

Usually agent frameworks do not offer native tools to create agent-based applications. Among positive exceptions are ZEUS Agent Building Toolkit [19], depicted in Figure 1, and AgentFactory [7] which have tools to generate starting scripts for creating agents. Agent Academy [18] has a tool to parameterize and launch agent-based applications using previously defined agent types. Agent Society Configuration Manager and Launcher (ASCML) [5] is a tool for the Java Agent Development Framework (JADE) which facilitates the configuration and deployment of agent societies. NetLogo [21] is a programmable modeling environment, featuring hundreds of independent agents, where modelers can give instructions using text-based input.

But all these tools lack some interactivity. Most use a script language and a command interpreter that parses and executes scripts. Even the solutions which provide graphical interfaces are quite limited and dependent on textual input.

In order to overcome the problem stated above, we present *EditION*, a solution to collaboratively create and manage VEs defined in the *ION* framework. With *EditION Editor*, users can sketch world elements or command gestures in order to create and control the VE. By providing immediate visual feedback to users of what is happening in the world, it allows high level debugging.

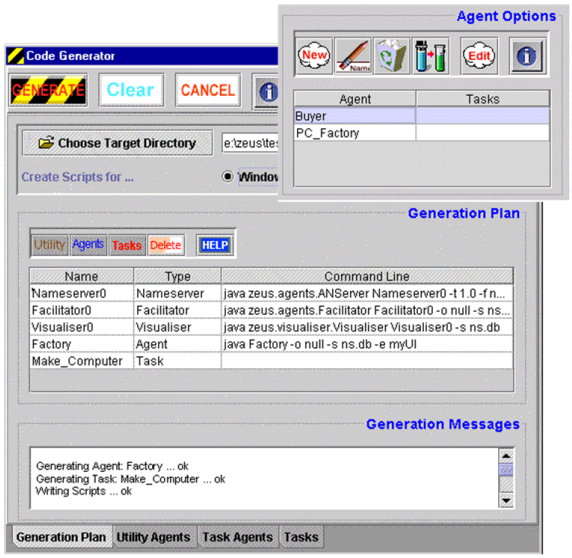

Figure 1. Zeus agent building toolkit

3.1 Architecture

Although *EditION* should work closely with the *ION* framework, they must be independent. Indeed, the framework core handles the VE and provides a generic API for external control, which can be done by different clients, ranging from a simple textual console to our calligraphic tool. Next we will describe the architecture we propose for such solution.

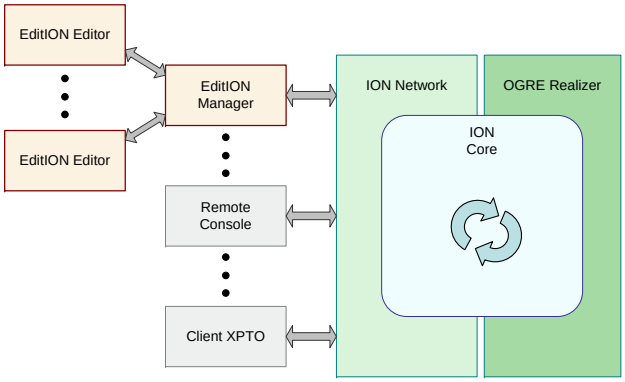

Figure 2. Architecture overview

Figure 2 depicts the overall architecture of the developed system. The actual simulation process takes place at the *ION.Core* module. *ION.Network* supplies an access point where all *ION.Core* functionality is available to potential remote clients. One of these is the *EditION Manager* which handles all spatial and visual representation issues in order to cope with distribution. Users can control the world through *EditION Editor*, a calligraphic tool connected to *EditION Manager*. On the right of Figure 2 we have *OGRE Realizer* which is a module used for the visualisation of 3D worlds.

3.2 The ION framework

The *ION* framework aims at providing a means to simulate dynamic environments. To achieve this goal it identifies five basic elements: *Entities*, *Properties*, *Actions*, *Groups* and *Events*. *Entities* populate the simulation universe, they can have *Properties* and change the world through the use of their *Actions*. Also, *Entities* can be connected through the use of *Groups*.

The simulation is processed in a discreet step manner. All the universe manipulations are made so that it is assured that at a given simulation step the same information is available for all elements. Let us take as an example Craig Reynolds simulation of flocking behaviour Boids [20]. Imagine a flock of birds that fly in an open space. In the *ION* framework each bird would be represented as an *Entity* that has a given position *Property* and that can fly around through the *Action* move. In this example the movement of a bird depends on the positions of the birds in the vicinity. This is true for all moving birds.In other words, when the move *Action* sets the new position for the bird it will influence the movement of the other birds in the simulation. The *ION* framework assures that for a given simulation step all changes to the world only are taken when all actions have been performed. So, the position of a bird will only be changed after all new positions are computed and the information available to all birds will be the same no matter the order by which their movement is computed.

When the universe changes, such as when the value of a *Property* is modified or an *Action* is started or stopped, an *Event* is created. This *Event* is then propagated to whomever registered to be notified of such an *Event*.

The *ION.Network* module provides a way to manipulate the simulated universe from a remote access point. This is accomplished through a star network topology.

3.3 The EditION Manager

Connected to the *ION.Network* module, the *EditION Manager* provides distribution functionality to calligraphic editors. To that end, it must handle not only an image of the symbolic representation of the VE used by *ION.Core*, but also its schematic counterpart, the diagram representation of that world used by *EditION Editor*. As depicted in Figure 3, *EditION Manager* stores a *World Representation* containing all relevant information about the VE, as seen by *ION*, and corresponding diagram, as seen by *EditION*.

Besides guaranteeing a correct synchronisation of these two representations, the *EditION Manager* must handle collaborative management of the schematic representation, since several instances of *EditION Editor* can edit the diagram at the same time. Since collaboration itself is not

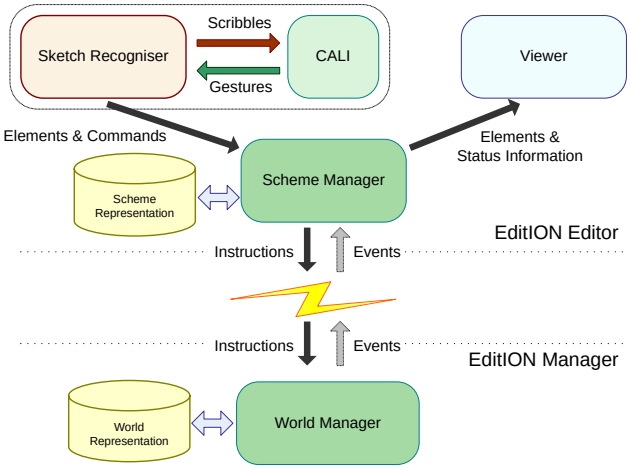

Figure 3. EditION architecture

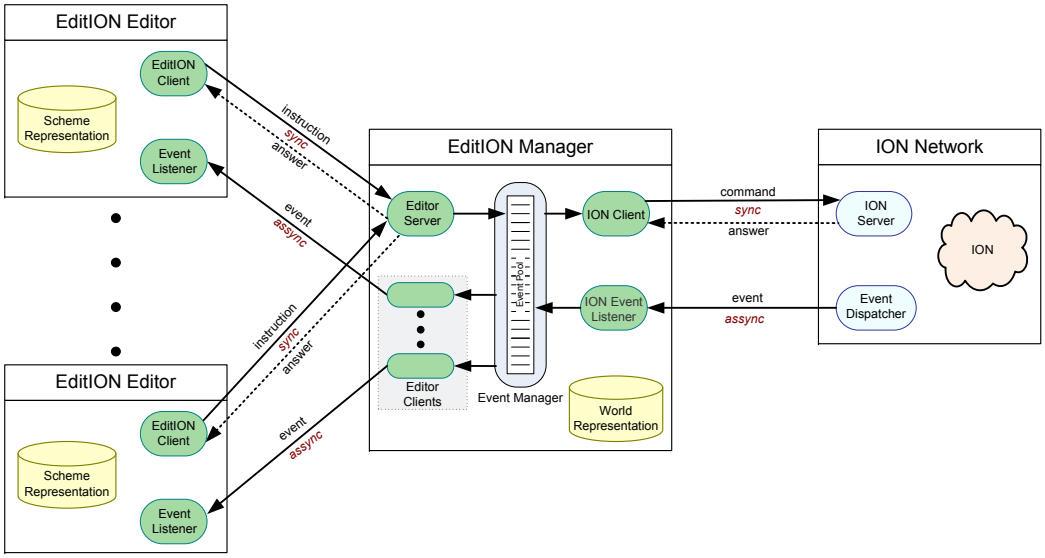

Figure 4. Communication details

in the scope of this paper, here we will just briefly describe how it was implemented, omitting an extensive presentation of this topic.

To implement the collaborative features of proposed system, a rigid communication protocol was established, with centralised control in the *EditION Manager*. In short, it receives messages from editors, commonly generated upon user intervention. Parsed and validated, the received information is then broadcasted to other editors and forwarded to *ION* framework, after proper processing.

In the basis of these collaborative features is the communication system upon which the proposed solution was developed. This system relies on a simple message exchange over TCP/IP connections. As depicted in Figure 4, each editor has a client that send instructions to the manager and a event listener that receives events from the manager. Every time a user changes something on his editor or something happens in the ION network, the event manager updates its world representation and disseminates the corresponding events to all editors and, if necessary, to the ION network. This solution proved to be fast and effective for our purposes, allowing real-time collaborative management of world simulations.

3.4 The EditION Editor

To create and manage the VE users interact with the proposed solution through *EditION Editor*. It includes a calligraphic interface where users sketch a symbolic representation of the VE, receiving proper visual feedback. Unlike other approaches, like SketchySPICE, where users can select in which mode they draw, only the *immediate* mode makes sense in *EditION*. Therefore, the sketch is interpreted and validated as it is being drawn, allowing on-the-fly creation instead of having to draw the entire diagram prior to its interpretation.

Therefore, real time recognition of gestures must be performed as well as syntactic validation of sketches and corresponding context. This means that the whole process should be, above all, efficient. Following this premise, we focused on devising a simple solution that provides good results. The *EditION Editor* tool can be divided into four distinct modules, illustrated in Figure 3 and described below.

Scribbles drawn by users are processed by the Sketch Recogniser module, which uses CALI [10] library to recognise them as shapes or commands. We choose this recogniser due to its simplicity, comparing for instance with the powerful sketch recognition engine proposed by Alvarado and Davis [2], and due to the fact that it completely fulfils our needs. Based on feedback provided by CALI, which classifies submitted scribbles according to a predefined gesture set, Sketch Recogniser then identifies if recognised gesture is an element of the world or a command, sending the corresponding information to the Scheme Manager module.

The Scheme Manager module can be considered as the core of `EditION` tool. It handles the diagram representing the world, performing syntactic and basic semantic validation. To that end, it applies a set of predefined grammatical rules to guarantee the correctness of the scheme. The scheme is stored as a directed graph, in which the nodes represent the elements and the edges represent the connectors. This way, common graph manipulation techniques could be used to manage the scheme and navigate through it. At this level just essential information about the world is stored,

since for viewing and editing purposes it can be considered as a diagram.

The *Viewer* module encapsulates the output details of our approach. It is responsible for providing visual feedback to the user, by selecting the graphical shapes displayed for each element and its properties according to the current state. Based on information received from the *Scheme Manager*, the viewer determines which shape must be drawn, its position and colour. Moreover, it manages the way messages are shown to users and how long they remain in the screen.

The communication with the *EditION Manager* application is done trough sockets and is controlled by the *Scheme Manager* module. It includes an event listener triggered by messages sent from *ION Framework*. Moreover, any new element or command generated by the Sketch Recogniser originates, after being validated, an instruction sent to *ION Framework*, which performs the complete validation and provides proper feedback.

4 Sketching Virtual Environments

Developers often start by drawing VEs on paper. Then, they generally use script-based tools to specify the world in the framework. Even when these tools have graphical interfaces with mouse interaction, they are greatly dependent on textual commands. Following the recent developments in pen and sketch-based interfaces, `EditION` is an alternative to standard mouse-based tools to create and manage VEs.

4.1 Visual Language

In order to produce an easy to use tool we start by identifying shapes commonly used by developers when they sketch a diagram of a VE on paper. We select a set of four symbols to represent the different elements of a VE, based on its visual similarity with the hand-drawn elements, usually sketched by developers when schematically representing their VEs. As depicted in Figure 5, *entities* are represented by a rectangle, *actions* by a small square, *properties* by a small circle and *groups* by a lozenge.

To allow easier identification, we give a different colour to each shape. Moreover, element colour changes according to element state to convey even more information about the world. For instance, when a property changes the corresponding visual element gets darker and slowly returns to the original colour. This way users can easily understand what is happening in the world. However, selected colours are just development options, lacking some tests to validate them or changing to a more appropriate palette.

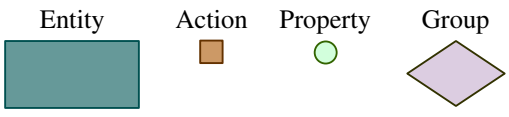

Figure 5. Symbols for virtual environment elements.

4.2 Scribble Recognition

As expected in a calligraphic tool, the interaction with `editION` is usually made through pen-based hardware such as a TabletPC or a digitising tablet. After capturing the scribbles drawn by the user, these are recognised, interpreted and validated to become useful. Thus, the whole process of handling the user input and producing corresponding output both to the user and to the framework, plays a major role in our calligraphic tool. This task can be divided in two distinct components: symbol recognition and sketch analysis.

In order to provide on-line recognition of sketches, our approach processes the scribbles as they are being sketched and not the entire sketch on demand, as performed by SILK and SIM-U-SKETCH. These scribbles are clusters of strokes drawn by the user which are submitted to a recognition process when the user's pauses are longer than a given time between strokes. To improve efficiency, older strokes are discarded if not included in a recognised symbol during a certain period of time, giving scribbles a decaying behaviour.

All combinations of strokes that compose the scribble

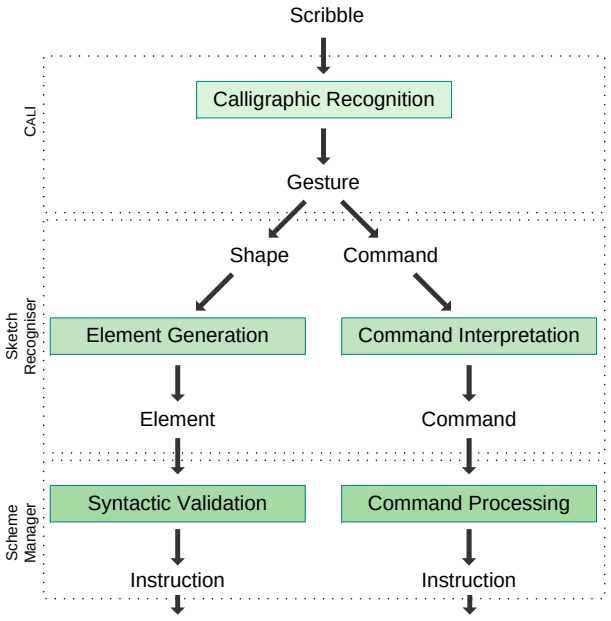

Figure 6. Recognition strategy

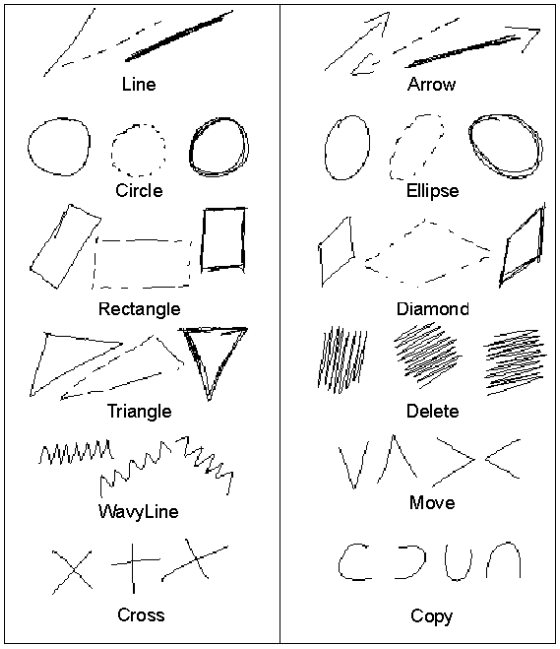

Figure 7. Gestures recognised by CALI

are submitted to the recognition mechanism, starting with combinations that contain all the strokes and finishing with single strokes if no symbol were recognised before. In order to recognise scribbles drawn by users and convert them into instructions for *ION*, we use a multi-level recognition and parsing strategy, outlined in Figure 6. This strategy is divided in several steps, detailed below.

We start by performing the calligraphic recognition of submitted scribbles. To that end, we use CALI, the fast, simple and compact scribble recogniser used in JavaSketchIt. It combines temporal adjacency, fuzzy logic and geometric features to classify scribbles. CALI identifies shapes of different sizes and rotated at arbitrary angles, drawn with dashed, continuous strokes or overlapping lines. It detects not only the most common shapes in drawing such as triangles, lines, rectangles, circles, diamonds and ellipses, using multiple strokes, but also other useful shapes such as arrows, crossing lines or wavy lines, as depicted in figure 7.

Comparing the visual language defined previously with the set of gestures recognised by CALI, we easily identify a subset of this that will be used to specify elements. Then we can use some more gestures to specify commands necessary to manage the VE elements. The complete subset of gestures we use is depicted in Figure 8. The scribbles to represent the elements were obviously selected from the already defined visual language. Additionally, pre-specified gestures represent the "delete", "start", "stop" and "change" commands.

Generally considered as an advantage, the rotation and

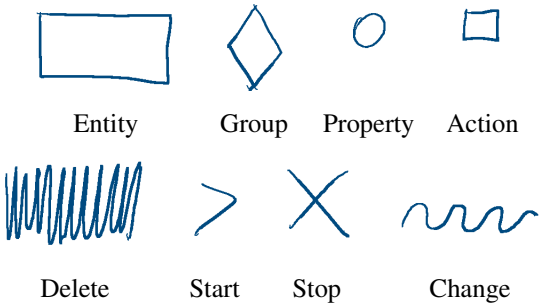

Figure 8. Gestures for elements and commands.

size independence of CALI recogniser is not useful in our approach. Thus, we need to carry out additional computation to determine scribble orientation and size. With this additional information and based on output provided by CALI, the second calligraphic recognition level performs gesture identification and yields two categories of gestures: shape gestures and command gestures. To perform gesture identification we apply the grammar presented in Figure 9. This set of simple rules provides an efficient manner not only to determine if the scribble is a command or a shape, but also to identify the command or element corresponding to a given scribble.

The second level of the recognition strategy, gesture identification, is the application of the grammar mentioned above. This process transforms gestures recognised by CALI into elements or commands for the Scheme Manager.

GESTURE-IDENTIFICATION-GRAMMAR(S)::=
valid_gesture $\rightarrow$ shape — command
shape $\rightarrow$ entity — action — group — property — connector
command $\rightarrow$ delete — start — stop — change
entity $\rightarrow$ $Gesture(S, \textit{RECTANGLE})$ &
$SizeWithin(S, \tau_{max^E}, \tau_{min^E})$ & $AspectRatio(S, 4, 3)$
action $\rightarrow$ $Gesture(S, \textit{RECTANGLE})$ &
$SizeUnder(S, \tau_A)$ & $AspectRatio(S, 1, 1)$
group $\rightarrow$ $Gesture(S, \textit{DIAMOND})$ &
$SizeWithin(S, \tau_{max^G}, \tau_{min^G})$
property $\rightarrow$ $Gesture(S, \textit{CIRCLE})$ & $SizeUnder(S, \tau_P)$
connector $\rightarrow$ $Gesture(S, \textit{LINE})$
delete $\rightarrow$ $Gesture(S, \textit{DELETE})$
start $\rightarrow$ $Gesture(S, \textit{MOVE})$
stop $\rightarrow$ $Gesture(S, \textit{CROSS})$
change $\rightarrow$ $Gesture(S, \textit{WAVYLINE})$
$Gesture(sc, t)$ $\rightarrow$ Scribble *sc* recognized by CALI as t
$SizeWithin(sc, t_u, t_l)$ $\rightarrow$ Size of scribble *sc* within t_u and t_l
$SizeUnder(sc, t)$ $\rightarrow$ Size of scribble *sc* is below t
$AspectRatio(sc, w, h)$ $\rightarrow$ Aspect ratio of scribble *sc* $\simeq$ *w:h*

Figure 9. Grammar for gesture identification.

In this process, rectangles are transformed into entities or actions, depending on their geometric properties, diamonds into groups, circles into properties and lines into connectors. On the other hand, pre-specified command gestures are transformed into ”delete”, ”start”, ”stop” and ”change” commands. After identifying the category to which the detected gesture belongs, we follow distinct parsing paths for shapes and commands.

4.3 Sketch Analysis and Validation

When a scribble is identified as an element, it goes through validation. To that end, context information is used to verify if such element makes sense in the current scheme. In the case of a connector, such information is also valuable to determine if it is a simple connector or a bind. The simple connector links an action or property to it's manager while a bind connects an entity with a group. If the newly created element passes the syntactic validation, the corresponding instruction is created and sent to the *EditION Manager*.

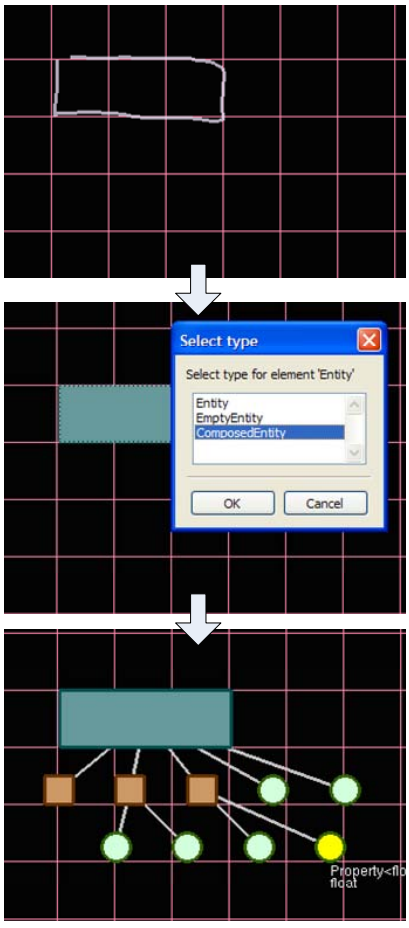

Figure 10. Creating an entity with EditION

Even after passing the syntactic validation performed by Scheme Manager the element, further syntactic validation is needed as well as semantic validation. Thus, the new element is marked as ”not validated” until its creation were broadcasted by *EditION Manager* to other *EditION* editors and sent to the *ION* framework for further validation. According to the result of syntactic and semantic validation performed by the *ION* framework, corresponding events are transmitted back to the *EditION Manager* and then broadcasted to all *EditION* editors. Such events provide information that will be used to give visual feedback to the user, either by replacing the sketch by the corresponding element on the screen, showing an meaningful error message or asking more details about the recognised element.

For instance, in Figure 10 is shown the process of creating an entity in three steps. First, the user sketches the corresponding gesture, which is recognised as an *Entity*. This information is then sent to the *EditION Manager* that validates such instruction and sends information about possible types of pre-defined entities. The user should then select the type of entity from a list and the result of this choice is transmitted to the *EditION Manager*, which validates the creation of the given entity and provides the proper feedback and took the necessary measures to create it. Among these measures are the dissemination of the creation command by all connected editors and to the *ION Network*. This command includes the specification of the complete set of elements that correspond to the created entity.

If a scribble is identified as a command, the context is analysed to verify its validity. It uses information extracted from context to produce an instruction to send to the *EditION Manager*. As for the elements, the command is broadcasted to other *EditION* editors and sent *ION* framework. After validating and processing the command, the framework will transmit the corresponding events. Based on these events the scheme will be updated and the user will receive proper feedback.

5 Editing and Debbuging with EditION

Currently, users sketch their VEs in sheets of paper before coding it into the framework. In *EditION* we take advantage of the users' ability to draw VEs with a pen to automate the boring and time consuming task of writing unnecessary lines of code. Therefore, users can sketch the world in the *EditION* calligraphic interface using a pen-based digitizer and it will be automatically created in the framework.

Since we perform on-the-fly gesture recognition, the sketch is interpreted and validated as it is being drawn. Moreover, the on-line connection with the framework allows *EditION* to provide immediate feedback to all users connected to the framework. To that end, syntactic and semantic verification of the sketches are performed while the

world is being constructed. Thus, it is no longer necessary to design the complete world to check if any errors exist, as it usually happens in other tools.

Thus, to create a VE with *EditION* the user sketches each element at a time using single or multi-stroke scribbles. The scribble is immediately interpreted by the recogniser and its creation is communicated to other editors working in this collaborative environment. When it is interpreted by the framework as a valid element, the corresponding formal symbol replaces the sketch in all editors.

Besides elements, the user can also sketch commands. These are also interpreted by the recogniser and, if as they are validated and executed by the framework, the corresponding action immediately takes place and the drawing area is updated accordingly in all editors.

Moreover, if an external application or an internal event of the framework changes the status of the VE in the *ION*, such change is immediately reflected in the editors if needed.

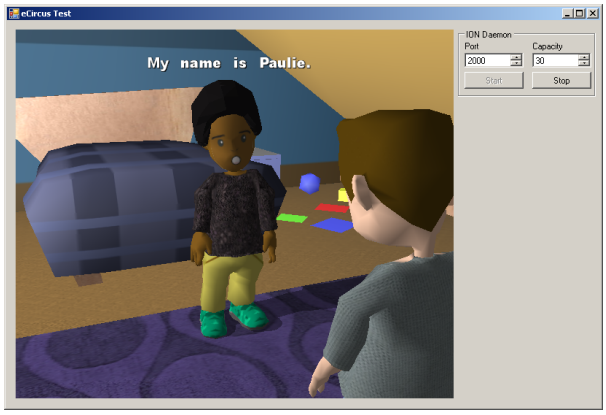

Figure 11. Virtual environment

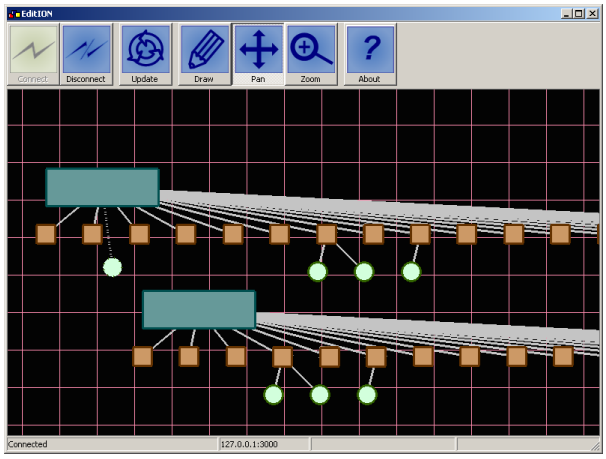

Figure 12. Partial symbolic view

Additionally, to allow proper debugging it is possible to control the program flow from the editor by performing step-by-step executions in the framework. This way it is possible to debug a VE in run-time. Thanks to the collaborative feature of *EditION*, it is easy to have multiple users in distinct computers analysing diferent views of the world symbolic representation during the debug operation.

To test presented solution, we performed a small study case based in *FearNot!* [3, 4] which is a computer application developed to tackle and eventually help to reduce bullying problems in schools. Thus, the overall objective of the development of *FearNot!*, was to build an anti-bullying demonstrator in which children, of age 8 to 12, experience a virtual scenario where they can witness (from a third-person perspective) bullying situations.

Through the use of two *EditION* editors, in separate computers, we have set up the simulation universe by creating two *Entities* that represented a bully and a victim character. We then staged a bullying episode by changing *Properties* values, such as who should the bully hit, and by starting *Actions* such as the crying of the victim. As the simulation took place the *Ogre.Realizer* module functioned as a view of the spatial elements of the world (Figure 11). On the other hand, *EditION Editor* provided not only an efficient manner by which to interact with the world but also offered a symbolic view (Figure 12) of the simulation. Namely, aspects that were not easily grasped, or even visible, by the *Ogre.Realizer* module such as the internal emotional state of a character were clearly depicted by EditION.

The diagram depicted in the screenshot of *Edition* shown in Figure 12 represents the two characters seen in Figure 11. Each rectangle corresponds to a character (an agent) in the simulation, while the squares represents the actions associated to that agents and the circles represent the properties of that actions. Properties of the characters are also represented as circles and indeed exist in the depicted scenario but are oustide the illustrated view. Through simple gestures, presented in Figure 8, the user can control the status of the world, namely by changing properties or triggering actions.

To appraise the proposed system on a collaborative environment we install it on lab equipped with a large screen display, the LEMe wall [8]. Then we asked two users of the *FearNot!* application to test it. Using TabletPCs, each user controlled two entities of the world simulation through *EditION Editor*. Besides having the TabletPC screen where each one can see a symbolic view of the world that best suits his purposes, in this particular collaborative environment the proposed system provides an additional symbolic view of the world, as well as a 3D view of the simulation (see Figure 13). Thanks to proposed collaborative methodology changes made by any user are immediately displayed in every views, allowing a realtime collaborative interaction

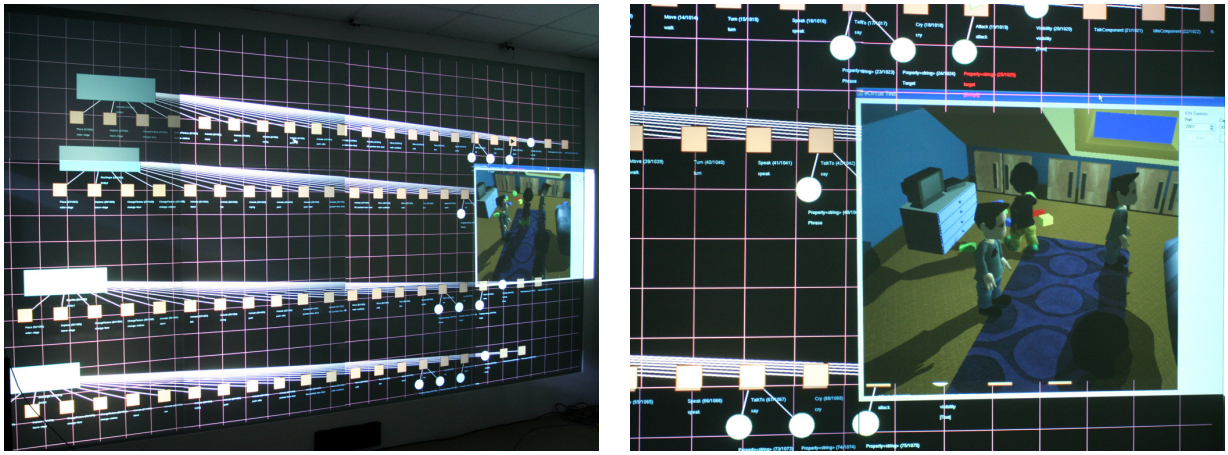

Figure 13. Collaborative usage of EditION on a large screen display

with the simulation.

Compared with a user evaluation, the test described in previous paragraph was quite informal, focused mainly on the technical details of the pre-defined tasks. Nevertheless, we manage to obtain qualitative feedback from the users. Both considered the system a good approach for the problem they face when designing VEs, as well as controlling and debugging simulations in these virtual environments.

6 Conclusions and Future Work

We proposed a collaborative calligraphic solution as an alternative approach to manage VEs. Instead of writing code or dragging and dropping elements from toolbars and menus, *EditION* is closer to traditional paper-and-pencil methods where the user simply sketches the elements he wants to create in the VE. Such sketches are interpreted and validated as they are being drawn, allowing an immediate feedback which prevents errors or incoherence.

The user can also change elements dynamically while the simulation is running. *EditION* uses a visual language to represent each element in the VE and this symbolic representation changes as the VE evolves during the simulation. This is really useful to visually debug and easily understand what is indeed happening in the virtual environment.

Moreover, *EditION* also allows collaborative management, meaning that several users can sketch and manipulate elements at the same time. Collaboration is very important not only for creating the VE in less time, but also to allow multiple users to interact with the simulation at the same time.

However, there is still room for improvements. The generation and constant update of the symbolic representation as the simulation goes creates several visualisation problems related with spatial constraints. Elements should not overlap and they should be distributed in a readable and understandable manner. Currently, *EditION* has some problems when we have too many elements being displayed at the same time. Clearly, We need more advanced methods to achieve an efficient spatial distribution.

The visual language can also be extended to include more visual elements, namely the propagation of events as well as other dynamic processes that occur inside the simulation. This could not only create a richer view of the simulation, but also make the visual debug capabilities mentioned before even more efficient. Additionally, the input of textual information can be further enhanced by adding hand-writing recognition to complement the sketch-based control of the simulation.

In conclusion, *EditION* brings some new contributions to the area of simulation mainly because it is able to display what is happening inside the simulation rather than just the results. Furthermore, it can display this information visually using a symbolic representation that can be manipulated offstage or during the simulation by multiple users at the same time in a collaborative manner. We feel this is the right path to what will be the tools of the future when we talk about managing virtual environments.

References

[1] C. Alvarado and R. Davis. Resolving ambiguities to create a natural sketch based interface. In *Proceedings of IJCAI-2001*, August 2001.

[2] C. Alvarado and R. Davis. Dynamically constructed bayes nets for multi-domain sketch understanding. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 32, New York, NY, USA, 2006. ACM Press.

[3] R. Aylett, S. Louchart, J. Dias, A. Paiva, and M. Vala. Fearnot! - an experiment in emergent narrative. In *Pro-*

ceedings of Fifth International Working Conference on Intelligent Virtual Agents, IVA 2005, pages 305–316, 2005.

[4] R. Aylett, S. Louchart, J. Dias, A. Paiva, and M. Vala. Unscripted narrative for affectively driven characters. *IEEE Computer Graphics and Applications*, 26(2):42–52, 2006.

[5] L. Braubach, A. Pokahr, D. Bade, K.-H. Krempels, and W. Lamersdorf. Deployment of distributed multi-agent systems. In F. Z. Marie-Pierre Gleizes, Andrea Omicini, editor, *5th International Workshop on Engineering Societies in the Agents World*, pages 261–276. Springer-Verlag, Berlin Heidelberg, 8 2005.

[6] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge. Javasketchit: Issues in sketching the look of user interfaces. In *Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding*, pages 9–14, Palo Alto, USA, Mar. 2002.

[7] R. W. Collier. *"Agent Factory: A Framework for the Engineering of Agent-Oriented Applications"*. PhD thesis, "University College Dublin", 2001.

[8] B. R. de Araújo, T. Guerreiro, R. J. J. Costa, J. A. P. Jorge, and J. ao António Madeiras Pereira. Leme wall: Desenvolvendo um sistema de multi-projecção, October 2005.

[9] A. Ferreira, M. Vala, J. M. Pereira, J. A. Jorge, and A. Paiva. A calligraphic interface for managing agents. In *Proceedings of the 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2006.

[10] M. J. Fonseca, C. Pimentel, and J. A. Jorge. CALI: An Online Scribble Recognizer for Calligraphic Interfaces. In *Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding*, pages 51–58, Palo Alto, USA, Mar. 2002.

[11] T. Hammond and A. Davis. Tahuti: A geometrical sketch recognition system for uml class diagrams. In *AAAI Spring Symposium on Sketch Understanding*, pages 59–68. AAAI Press, 2002.

[12] T. Hammond and R. Davis. Tahuti: a geometrical sketch recognition system for uml class diagrams. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 25, New York, NY, USA, 2006. ACM.

[13] J. I. Hong and J. A. Landay. Satin: a toolkit for informal ink-based applications. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 63–72, New York, NY, USA, 2000. ACM Press.

[14] L. B. Kara and T. F. Stahovich. Sim-u-sketch: a sketch-based interface for simulink. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 354–357, New York, NY, USA, 2004. ACM Press.

[15] J. A. Landay and B. A. Myers. Sketching interfaces: Toward more human interface design. *IEEE Computer*, 34(2):56–64, 2001.

[16] J. Lin, M. W. Newman, J. I. Hong, and J. A. Landay. DENIM: finding a tighter fit between tools and practice for web site design. In *CHI*, pages 510–517, 2000.

[17] M. Liwicki and L. Knipping. *Recognizing and Simulating Sketched Logic Circuits*, volume 3683/2005 of *Lecture Notes in Computer Science*, pages 588–594. Springer, Berlin Heidelberg, 2005.

[18] P. A. Mitkas, D. Kehagias, A. L. Symeonidis, and I. N. Athanasiadis. "a framework for constructing multi-agent applications and training intelligent agents". In *Proceedings of the 4th Int. Workshop on Agent-Oriented Software Engineering (AOSE-2003)*, pages 96–109, 2003.

[19] H. Nwana, D. Ndumu, L. Lee, and J. Collis. Zeus: a toolkit and approach for building distributed multi-agent systems. In *Proceedings of the 3rd conference on Autonomous Agents*, pages 360–361. ACM Press, 1999.

[20] C. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *SIGGRAPH '87 Conference Proceedings*, pages 25–34, 1987.

[21] U. Wilensky. Netlogo. Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston, IL., ¡http://ccl.northwestern.edu/netlogo¿, 1999.