AN ARCHITECTURE TO ASSIST MULTIMEDIA APPLICATION AUTHORS AND PRESENTATION ENGINE DEVELOPERS

Rodrigo C. M. Santos, Marcio F. Moreno and Luiz Fernando G. Soares

Department of Informatics
Pontifical Catholic University of Rio de Janeiro, PUC-Rio
Rua Marquês de São Vicente, 225 – Rio de Janeiro/RJ – 22453-900 – Brazil
{rsantos, mfmoreno, lfgs}@inf.puc-rio.br

ABSTRACT

This paper presents an architecture for monitoring the presentation of multimedia declarative applications, providing feedback about variables states, object properties, media presentation times, among others. Monitoring tools that follow the proposed architecture are able to detect if visual problems are being caused by programming errors or by player malfunctioning. The architecture presents a communication protocol designed to be independent of the declarative language used in the development of multimedia applications. The main goal is to provide an open and generic architecture that can assist multimedia application authors and presentation engine developers. As an example of the architecture use, the paper also presents a monitoring tool integrated into a graphical user interface developed for the ITU-T reference implementation of the Ginga-NCL middleware.

Index Terms— Multimedia authoring, debugging tools, declarative programming languages, presentation engines, Ginga-NCL

1. INTRODUCTION

Multimedia authoring is far from being considered a solved problem. As Rowe discusses in [1], although significant progress has been made in this field over the last years, there are still open issues to be addressed on this topic (especially regarding the authoring of interactive multimedia applications). In this paper, we focus on the problem of debugging multimedia applications designed using declarative languages.

Interactive multimedia applications are designed using either imperative or declarative languages. The use of imperative languages usually requires an expert programmer, familiar with low-level statements and multimedia drivers and libraries. On the other hand, declarative languages usually have high-level abstractions, allowing programmers that are not familiar with the low-level details to create applications. HTML [2], NCL [3], SMIL [4], SVG [5] and X3D [6] are examples of declarative languages used to create multimedia applications.

When using the imperative approach, there are many widely adopted debugging tools to assist application authoring. However, these tools are still poorly explored when considering the declarative approach.

The development of debugging tools for declarative multimedia languages is challenging mainly for two reasons. First, the imperative concept of algorithmically executing a sequence of statements cannot be applied. The concept of suspending a declarative application, highlighting the code chunk related to that time instant is a known a problem. Second, the verification of the application correctness can depend on the verification of audiovisual outcomes, on unpredictable events, and on the correctness of the employed presentation engine.

To illustrate audiovisual problems, Figure 1(a) presents a snapshot of an application with four media objects in which the banner and flags are rendered on overlapping regions. Because the overlay property value (named zIndex in Figure 1) of the banner object has a greater value than the one of flags, if these two objects run simultaneously, the flags content will not be seen (as illustrated in Figure 1(b)). There is neither a syntactic error nor an error arising from the language semantics. Just running the presentation, an author cannot know if the problem comes from the lack of a statement starting the unseen object, from the lack of the media content, from a visibility property incorrectly set, from the incorrect set of the aforementioned overlay property, or from an implementation error of the presentation engine. Using an appropriated monitoring tool, it is possible to identify the real situation, shown in Figure 1(c), and to check objects' properties during their presentations.

In this paper, we present an architecture suitable for monitoring the presentation of multimedia applications, in particular declarative applications, providing feedback about the states of variables, the object properties, the media presentation times, among others. When using the term *monitoring tool* we refer to tools that are able to monitor (and eventually to change) the state of an execution machine and log/report state changes. Monitoring tools that follow

the proposed architecture are able to detect if presentation problems[1] are caused by programming errors or by player malfunctioning (e.g., synchronization mismatches).

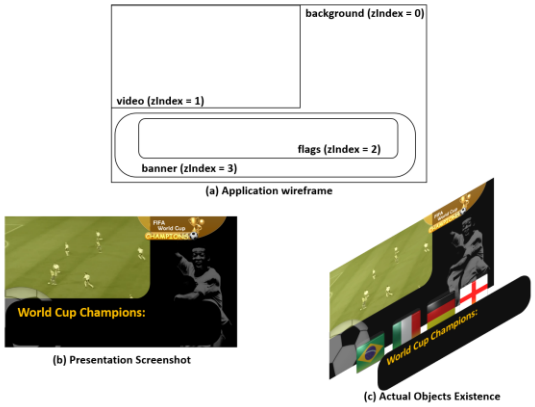

Fig. 1. Media objects with overlapping regions.

The proposal comprises a communication protocol designed to be independent of the specification language. As a consequence, the architecture addresses an issue of multimedia authoring to which the few existing solutions are proprietary and target a specific language. The main goal is to provide an open and generic architecture that can assist both multimedia application authors and presentation engine developers. As an example of the architecture instantiation, the paper presents a monitoring tool for the NCL language and its presentation environment (Ginga-NCL), part of the ITU-T Ginga middleware Recommendation [7].

The remainder of this paper is organized as follows. Section 2 discusses some related work. Section 3 presents the proposed architecture. Section 4 details the communication protocol used in the architecture. Section 5 introduces the NCL monitoring tool. Finally, Section 6 presents the final remarks and points out some future works.

2. RELATED WORK

Debugging tools provide a means to investigate the internal state of execution engines while running applications step-by-step. Considering software development applying imperative languages, the use of debugging tools such as GDB [8], DDD [9], and CDB [10], is a common sense of a longstanding practice. On the other hand, when considering declarative languages, most of work in the literature focuses only on functional languages [11][12][13][14] and disregards some issues addressed by multimedia presentation languages, among them the important temporal facet.

To detect temporal inconsistencies in multimedia applications, some proposals rely on formal specifications of the temporal models of the used languages. Gaggi and Bossi define a formal semantics for SMIL language inspired by Hoare logic. The formal semantics is based on a set of inferences-rules that express the temporal properties of SMIL elements. Similarly, Picinin Júnior et al [15] propose an approach to verify NCL documents using a formal verification technique based on the Temporal Logic Formula and on the TINA model-checker.

Formal methods are very useful in checking the temporal consistency of a presentation. According to Bouyakoub and Belkhir [16], the complexity of the synchronization models of some multimedia languages, makes it difficult to guarantee the validity of a multimedia scenario using non-formal methods. However, Gaggi and Bossi [17], and also Picinin Júnior et al [15] solutions (and other proposals that employ formal models to check temporal consistency) have some major drawbacks.

First, it must be stressed that to express the whole semantics of a declarative multimedia presentation language in terms of mathematical expressions is almost impossible. The aforementioned works have formalized only a subset of the target languages. Second, the process to convert a source code into a temporary formal model and to verify this model is work intensive. Since authors usually modify the source code several times during the development, the verification process can degrade the authoring tool performance, especially when developing large applications. Third, as the solutions are based on source code checking, they are not suitable to identify execution problems that can affect the audiovisual presentation (for instance, they cannot detect synchronization mismatches caused by network jitters).

The solution presented in this paper does not envision replacing existing verification techniques. Instead, this proposal is complementary. In other words, the proposed architecture do not aim at designing new methods to guarantee the validity of a document, but only in helping authors providing additional facilities to identify problems in multimedia applications.

This trend has recently begun to be considered as a monitoring/debugging feature to be embedded in hypermedia presentation engines. Modern web-browsers have built-in mechanisms that provide web developers access to internal browser state. Recent versions of Chrome, Firefox and Internet Explorer, for instance, allow track down the page layout, set breakpoints in JavaScript (imperative language) code, change variable values, monitor the network traffic, etc. Because the focus of the browsers is on rendering web pages, their debugging features are strongly coupled with HTML language (and other web technologies, like CSS and JavaScript). Moreover, these solutions do not contemplate multimedia presentation problems, like those related to the temporal dimension of presentations. In this paper, these problems are addressed. The proposed architecture is able to consider different programming languages by using a general communication protocol.

[1] In this paper, *application presentation* refer to the presentation of the multimedia application during the authoring phase.

3. THE MONITORING ARCHITECTURE

Figure 2 illustrates the proposed architecture by means of an authoring workflow. At any point of an application specification, authors can submit the code to the presentation engine. During the application running, the presentation engine and the monitoring engine exchange several messages following the protocol detailed in the next section. These messages allow for monitoring the presentation and tracing presentation problems that can eventually occur. When the presentation finishes, the monitoring tool generates a data structure that describes the running steps (the Execution Data in the figure). This data structure is then converted into a human-readable report, in agreement with author preferences, shown to the authors.

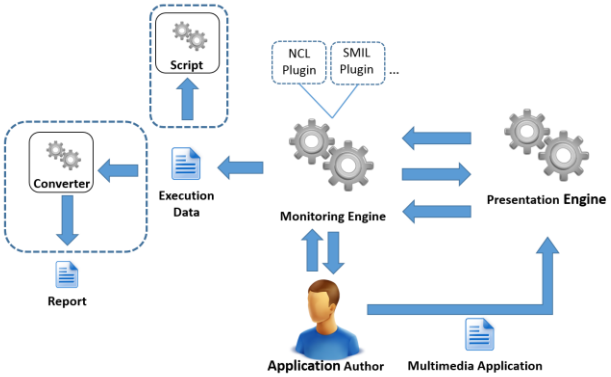

Fig. 2. Monitoring workflow.

The Execution Data could also be used to validate conformance test cases of the presentation engine. A scripting procedure would compare the Execution Data with the expected Execution Data previously generated by human means or produced using some reference implementation.

The communication protocol allows for not only monitoring functionalities, but also enabling users to get and set values to objects' properties and system variables. As discussed in the next section, this would allow for the simulation of different execution scenarios. This feature is useful both for application authors and for presentation engine developers. The former is assisted with application bug fixes and the latter with the evaluation of the presentation engine correctness.

Using presentation engine messages, the monitoring engine trusts in the presentation engine correctness. However, a non-conforming player can violate language semantic rules. Hence, the architecture integrates plugin extensions that implement the conceptual models of the target-languages to validate actions performed by the players.

4. THE COMMUNICATION PROTOCOL

The communication protocol has four sets of messages, as shown in Figure 3: i) messages sent from presentation engine to notify the monitoring engine about internal actions performed on media objects (notification messages); ii) messages sent from monitoring engine to get or set media objects' property values, and the responses sent from the presentation engine (get/set messages); iii) messages sent from monitoring engine to emulate input events (input messages); iv) messages containing arbitrary commands (command messages).

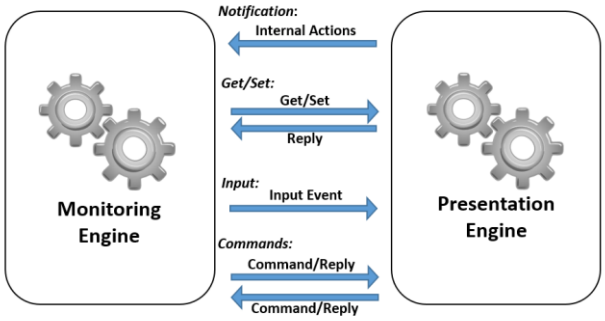

Fig. 3. Types of protocol messages.

Each message is prefixed with the string "PE" (Presentation Engine) or "ME" (Monitoring Engine), which identifies the message sender. Listing 1 presents the message syntaxes.

Notification:
1. PE::{status}::{action}::{id}[::{triggerEvent}::{triggerId}::{time}]

Get/Set:
1. ME::{method}::{scope}.{var}[::{value}]
2. PE::reply::{status}[::{value}]

Input:
1. ME::ie::{type}::{params}

Commands:
1. {Prefix}::cmd::{args}
2. {Prefix}::cmd::reply::{args}

Listing 1. Template for the protocol messages.

4.1. Notification Messages

Notification messages are sent to notify that some internal action has been performed on a media object (e.g., starting the object presentation) by the presentation engine.

The status field indicates whether the action is succeeded or failed: 0 means success and 1 indicates an error. An example of an unsuccessful action could be the start of a media object in a remote device that is unreachable by the presentation engine.

The action field identifies the action type performed (assuming a successful try). Table 1 presents the action identifiers and their semantics. Finally, the id field identifies the media object target of the action.

Table 1. Action identifiers.

Action	Description of the notification
startApp	An application presentation has been started.
stopApp	An application presentation has been finished.
start	A media object presentation has been started.
stop	A media object presentation has been finished.
pause	A media object presentation has been paused.
resume	A media object presentation has been resumed.
select	A media object has been selected.

Multimedia languages are usually based on the causal/constraint paradigm to relate media objects in a presentation. Using these languages, authors can express the relationships among objects through causal or constraint sentences. In causal sentences, conditions have to be satisfied to trigger actions, for example: "the end of 'x' causes the start of 'y'". In this case, the presentation engine should notify not only the action performed, but also the condition that has triggered it. Thus, the triggerEvent field identifies the condition that has triggered the action, while the triggerId field identifies the object related to this condition. Table 2 illustrates the condition identifiers and their semantics. In addition, the presentation engine should also notify the instant the action should occur (according to the application specification). The time field notifies this correct time instant.

Table 2. Condition identifiers.

Condition	Description
onBegin	When object presentation begins.
onEnd	When object presentation ends.
onAbort	When object presentation aborts.
onPause	When object presentation pauses.
onResume	When object presentation resumes.
onSelect	When object presentation is selected.
onAttribution	When an object property has its value changed

4.2. Get/Set Messages

Get/Set messages allow authors to get or set values of properties. Getting a property value can help authors to better understand the presentation engine internal functioning – and possibly improve their knowledge about the language. Similar features are found in debugging tools for imperative languages, in which one can get values stored in variables.

Setting new property values is useful to test the application behavior in face of different execution scenarios. As an example, consider an application that can present different contents depending on user locations.

The first get/set message syntax in Listing 1 defines messages sent by the Monitoring Engine. The method field holds the value get or set. The scope identifies whether the property is global (a system property) or relative (local) to a media object. For the first case, the value is system; for the second, the value is the media object identifier. The var field defines which property is the target. The value field should be used just in the case of the set method. It holds the value to be set to the property.

On receiving get/set messages, the presentation engine must reply, notifying whether it was able to meet the request or not. There are two types of replies, one indicating success and other indicating failure. For a get request, a success reply indicates that the presentation engine is sending back the value of the property. For a set request, a success reply indicates that the presentation engine has set the value of the property.

There are some cases for which the presentation engine cannot meet the request. For example, if the author is requesting the value of a nonexistent property or if (s)he is trying to change the value of a read-only property (e.g., a property that holds the amount of available memory). The second get/set message syntax in is used to send messages in which the status field indicates whether the presentation engine has been able to meet the request or not (0 indicating success and 1 indicating failure).

4.3. Input Messages

Input messages allow authors to emulate input events. They are designed mainly because some multimedia applications are developed in authoring platforms different from the ones they will be presented. A typical example is the development of interactive TV applications, which are usually developed in a regular computer, but are intended to be executed in Digital TV receivers with remote control or any other more sophisticate input devices, like gesture sensors, etc. By emulating input events, the Monitoring Engine can provide authors the possibility of testing their applications without needing specialized devices.

In input message syntax the type field identifies the input event type, and the params field provides the parameters to describe the event.

Input events are divided into the following categories: keyPress, keyRelease, mousePress, mouseRelease, gesture, voice, and sensor. Each input event type has its own set of parameters. Table 3 summarizes these types and their parameters.

Table 3. Input event types and parameters.

Type	Parameters
keyPress	*{ [Modifiers,] key}*
keyRelease	*{ [Modifiers,] key}*
mousePress	*{X, Y, button}*
mouseRelease	*{X, Y, button}*
gesture	Gesture description.
voice	Recognized sentence.
sensor	Sensor specific parameters.

A gesture input event emulates a multi-touch gesture on a screen, or else a body gesture. The parameter for this type of event is a description using some gesture description language, e.g., Gesture Markup Language (GML) [18],

Gesture Description Language (GDL) [19], etc. The voice event is designed to test applications that are intended to run in devices that have voice recognition capabilities. The parameter of this event is a string that emulates the sentence recognized. There can also be input events coming from other sensor types. The parameters for these events need to be defined depending on the sensor type.

4.4. Command Messages

Command messages allow authors to send arbitrary strings to a particular presentation engine. As a concrete example, consider the NCL language and its Ginga-NCL presentation environment. Ginga-NCL has an API that allows for receiving editing commands during application runtime [20]. Authors can send editing commands to emulate the application behavior placing editing commands in the arg field.

5. THE MONITORING ENGINE

As an example of use, the proposed architecture has been instantiated and integrated into a graphical user interface developed for the ITU-T Ginga middleware reference implementation, named GingaGUI. To enable the communication between both engines, following the protocol previously described, changes were made in the Ginga open source code.

The Monitoring Engine maintains a global clock that starts at the beginning of the application presentation, and a local clock for each media object, started when the object presentation starts. The global clock is useful to check whether media objects are starting, pausing, resuming or ending in correct time instants (according to the application specification). Upon receiving messages notifying that some action occurred in any media object, the Monitoring Engine compares the received time with the current global time. If the difference between these times is greater than a previously defined threshold, the Monitoring Engine reports that a possible[2] synchronization problem can be occurred.

The Monitoring Engine's graphical interface has been designed to provide a visual overview of the presentation on a timeline. The use of timeline gives a good overview of the relative synchronization among media objects. Figure 4 presents a screenshot of a presentation example. In the visual interface, the "drible" object has started 5 seconds after its correct time instant, the "photo" object 8 seconds after, and the "icon" object 7 seconds after. These problems are highlighted with a bold border on Figure 4. In the figure, thumbnails are presented below the timeline, showing screenshots of the presentation at each notification message received.

[2]Possible because the detected synchronization mismatch can be false, since the problem can be the delay of the notification and not the command reaction time.

Thumbnails can also be useful to test different versions of synchronous presentation engines against a conformance test suite. Assuming that a compliant engine exists, thumbnails generated from a test suite on this engine can be used as the comparative basis (e.g., through image processing or human analysis).

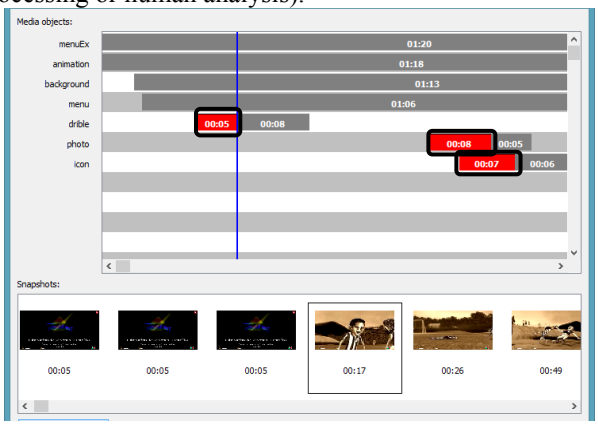

Fig. 4. Monitoring Engine screenshot.

To allow for sending messages to Ginga presentation engine, simulating input events or the execution of specific NCL[3] commands, the graphical interface has an input box on which these messages can be typed. In the current implementation, the entire message syntax must be observed, in agreement with the protocol specification presented in Section 4. In Figure 5 a snapshot of GingaGUI shows the input box

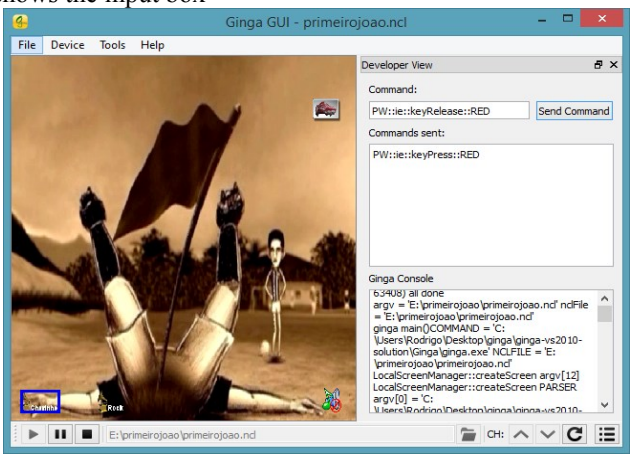

Fig. 5. Sending commands to Ginga.

6. FINAL REMARKS AND FUTURE WORKS

Despite many efforts focusing on the design of authoring tools, multimedia programming is still considered difficult and with several open issues. The architecture presented in this paper addresses one of these issues proposing a

[3]NCL (Nested Context Language) is the declarative language supported by Ginga.

Monitoring Engine to assist authors of declarative multimedia applications.

Evaluation tests have been conducted with a small group of NCL programmers. Their reports are very positive attesting the usefulness of the approach in reducing the debugging time. The gains have been more significant in large applications with dozens of relationships between media objects.

Presentation engine developers have also reported the usefulness of the proposed architecture to test the correctness and performance of their implementations. Likewise thumbnails generated as output can be used in comparisons with outputs coming from a reference implementation in conformance tests, they can be used to evaluate synchronization issues to detect whether the presentation engine implementation has the necessary performance (i.e., if the synchronous hypothesis can be assumed). Performance tests have been made with Ginga reference implementation running on machines with low memory footprint and low CPU power. In some tests, synchronization problems have been forced and detected. For the great majority of applications the synchronous hypothesis could be assumed, since content pre-fetching is performed. The few exceptions come from distributed applications targeting multiple presentation devices connected by some communication means.

The proposed communication protocol between the monitoring and presentation engines allows for decoupling their implementations. A future work is to use the Monitoring Engine with other presentation engines.

Another future work is to improve the robustness of the communication protocol. An ongoing implementation uses messages with timestamps to guarantee their correct sequencing and validity.

Although the architecture allows for plugins validating actions performed by presentation engines, our current implementation does not implement any plugin. An ongoing work is the development of an NCL plugin.

Still another future work has been already started to use the Ginga Monitoring Engine in the ITU-T Conformance Test Suit of the H. 761 Recommendation [7] for IPTV services.

7. REFERENCES

[1] L. A. Rowe, “Looking forward 10 years to multimedia successes,” ACM Trans. Multimed. Comput. Commun. Appl., vol. 9, no. 1s, pp. 1–7, Oct. 2013.

[2] W3C, “HTML5 - A vocabulary and associated APIs for HTML and XHTML,” 2014. [Online]. Available: http://www.w3.org/TR/html5/.

[3] L. F. G. Soares and G. A. F. Lima, NCL Handbook. Rio de Janeiro, Brazil: Department of Informatics PUC-Rio, 2013.

[4] D. C. A. Bulterman and L. W. Rutledge, SMIL 3.0 - Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books, 2nd ed. Springer, 2009, p. 507.

[5] W3C, “Scalable Vector Graphics (SVG) 1.1 (Second Edition),” 2011. [Online]. Available: http://www.w3.org/TR/SVG/.

[6] ISO/IEC IS 19775-1:2013, “X3D Architecture and base components V3.” .

[7] ITU-T, “H.761 : Nested context language (NCL) and Ginga-NCL,” 2011. [Online]. Available: https://www.itu.int/rec/T-REC-H.761-201106-I/en.

[8] R. Stallman, R. Pesch, S. Shebs, and M. . Free Software Foundation (Cambridge, Debugging with GDB: the GNU source-level debugger. Boston, MA: Free Software Foundation, 2002.

[9]A. Zeller and D. Lütkehaus, “DDD---a free graphical front-end for UNIX debuggers,” ACM SIGPLAN Not., vol. 31, no. 1, pp. 22–27, Jan. 1996.

[10] Microsoft, “Debugging Using CDB and NTSD.” .

[11] B. Pope and L. Naish, “Practical aspects of declarative debugging in Haskell 98,” 2003, pp. 230–240.

[12] R. Caballero and E. Martin-Martin, “A Declarative Debugger for Sequential Erlang Programs,” in Tests and Proofs, vol. 7942, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[13] A. Riesco, A. Verdejo, and N. Martí-Oliet, “A Complete Declarative Debugger for Maude,” in Algebraic Methodology and Software Technology, vol. 6486, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[14] Josep Francesc Silva, “Debugging techniques for declarative languages: Profiling, program slicing and algorithmic debugging,” PhD Thesis, Universidad Politécnica de Valencia, Valencia, 2007.

[15] D. Picinin, J.-M. Farines, and C. Koliver, “An approach to verify live NCL applications,” 2012, p. 223.

[16] S. Bouyakoub and A. Belkhir, “SMIL builder: An incremental authoring tool for SMIL Documents,” ACM Trans. Multimed. Comput. Commun. Appl., vol. 7, no. 1, pp. 1–30, Jan. 2011.

[17] O. Gaggi and A. Bossi, “Analysis and verification of SMIL documents,” Multimed. Syst., vol. 17, no. 6, pp. 487–506, Nov. 2011.

[18] Ideum, “GestureML,” 2013. [Online]. Available: http://www.gestureml.org. [Accessed: 24-Nov-2014].

[19] T. Hachaj and M. R. Ogiela, “Rule-based approach to recognizing human body poses and gestures in real time,” Multimed. Syst., vol. 20, no. 1, pp. 81–99, Feb. 2014.

[20] R. M. de Resende Costa, M. F. Moreno, R. F. Rodrigues, and L. F. G. Soares, “Live Editing of Hypermedia Documents,” in Proceedings of the 2006 ACM Symposium on Document Engineering, 2006, pp. 165–172.