An Interaction Design Model for a Petri Net based Behavior Editor

Nuno Barreto,
Licínio Roque
Informatics Eng. Dep.
University of Coimbra
3030-290 Coimbra,
Portugal
nbarreto@student.dei.uc.pt,
lir@dei.uc.pt

Abstract
In the context of Game Design, to define behaviors and choreographies of actors in a game or simulation context can be a challenging and complex programing task. This paper proposes an interaction model for a Petri Net behavior editor to be used as a visual language for game behavior modeling. A proof of concept prototype implementation of the proposed interaction model was created and validated with formal usability lab tests. In spite of some usability issues, most users were able to complete the proposed game behavior definition tasks using the editor interface. We think this provides evidence to reinforce the case that Petri Nets can be used to advantage in the game modeling process and, when coupled with a runtime simulation, can provide an interesting immediate feedback loop for faster design and experimentation.

Author Keywords
Complex Systems; Behavior Modeling; Game Design; Simulation; Petri Nets; Visual Language

ACM Classification Keywords
D.2.2. Software Engineering: Design Tools and Techniques.

General Terms

Design Languages; Digital Games; Experimentation

Introduction

Video-Game development is a multidisciplinary area which can encompass diverse skillsets. One of such skillsets is programming which has an important role in the development as it translates the game concept into an interactive artifact known as a video-game.

Since game programming is what contributes to the creation of aspects such as how game objects (referred from now on as actors) are modeled in an environment and how they interact with each other and with their environment, it becomes necessary to translate every possible course of action as code into actors which can be an error-prone task.

This paper presents an interaction model for an editor that is integrated in a solution proposal used to simplify game programming that is centered around the creation of a visual language, based on Petri Nets. More specifically, a language that is intended to model concurrent actors' behaviors and choreographies (i.e. how the behavior translates into visual and sonic feedback) in the game world as stage.

Petri Nets have been proven accessible and easy to learn by non-programmers and provide an economical way of specifying behaviors in complex systems. As a design tool, they give the advantage of a simple visual language that promotes agile modeling and testing of complex interactive systems.

This paper is structured as follows: the Background section contains state of the art research regarding video game development and the usage of Petri Nets as well as their editors and data structures, the Methodology section illustrates the development approach used for the creation of the solution proposal, the Interaction Model section states the model used and its specification, Evaluation demonstrates the evaluation's results and analysis and finally Conclusions will sum up the results and state some further work.

Background

Developing Video Games

Developing video-games has been made easier with the increasing existence of game engines - a system whose purpose is to abstract common, and sometimes platform dependent, computational game related tasks [28] - and other development tools, leaving developers with the choice of creating their game by either building their own engine using tools such as SDL [20] or XNA [26] or using pre-existing engines including UDK [24] Unity [25] or even CryEngine [7]. Most, if not all, of these tools require some knowledge in programming which imposes a learning barrier to more novice developers and are difficult to maintain, as coding is heavily dependent on a good architectural design to be readable and extensible.

Nevertheless, to counter programming's inherent steep learning curve and maintenance issues, some tools provide visual languages such as Scratch [19], Stencyl [22], ToonTalk [23] and Agent Sheets [2] and some game engines grant their users graphical modeling mechanisms (an example of this is UDK's Kismet, Unity's Mecanim and Cry Engine's Decision Tree Editor) that are utilized to build some aspects of a games (making them somewhat limited because coding is still

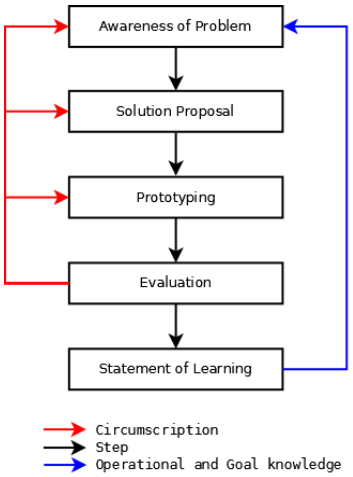

Figure 1. Steps in Design Science Research

required when developing aspects not covered by these mechanisms).

These visual approaches rely on either system block building or classical Artificial Intelligence algorithms and data structures, usually present in behavior modeling, such as Cellular Automata, State Machines, Behavior Trees, Flow Charts and Rule-Based Systems [13]. These visual data structures usually present a readability problem and their maintainability quickly decreases with increased model complexity.

Not as widely used as the aforementioned graphical tools, Petri Nets [15] have already been used to describe some game aspects including their plot as depicted in [6], level sequences as shown in [14] and even entire game models as demonstrated in [8] and [3]. It was showcased in the latter work that Petri Nets appear to be easy to use and learn and support models' validation through simulation. Petri Nets are also extensible and, consequently, have a wide variety of augmentations that add functionalities which, in turn, reduces potential model complexity. Because of their apparent smooth learning curve, validation functionality, extension capability and the demonstrated ability to model several aspects of a video-game, this tool was considered as a candidate for the base of the language described in this paper.

Petri Net Editors

There are several Petri Net editors available that can be used to model interactive systems. These include PIPE [16] Woped [29], Yasper [23] and many others. Since these tools are used to illustrate a wide variety of systems, they are deprived of semantics and their main application is to develop concept diagrams that demonstrate how systems function. As such, simulation functionalities present in these tools are mainly for debugging purposes. However, there are some editors that allow users to add semantics to their Petri Net models. Examples are JFern [9] and JPetriNet [10].

As an attempt to make Petri Net models interoperable and standard, a description language, called Petri Net Markup Language (or PNML) [5], was created. This language is built upon XML and besides describing how Petri Nets are distributed in a specific model, it also allows to add graphical attributes, which provides visual editors the means to render the models, and to append tool-specific attributes that can only be parsed by designated editors.

The editor developed borrows some interface aspects from the tools mentioned above while maintaining simplicity. It also uses PNML to store models persistently in an interoperable way, so that these models can be read on other authoring tools.

Research Methodology

Design Science Research [11] is a methodology that aims to produce a statement of learning as a consequence of research made through design or, in other words, this methodology's objective is to solve problems with the purpose of producing a statement of learning. The Design Science Research, as illustrated in Figure 1, encompasses 5 steps: Awareness of Problem, Solution Proposal, Prototyping, Evaluation and Statement of Learning, each producing its own artifacts. Since this methodology can be used in any area where design is possible, the following clarification of the methodology's steps states example artifacts best suited in this project's context.

The first step, or Awareness of Problem, comprises the definition and identification of a problem. In order to help clarify this definition and identification, State of the Art research was made.

In the following step, Solution Proposal, a suggestion for the problem's solution is created through abduction drawn from the state of the art research made previously. This step outputs interaction models and software architecture.

Prototyping comprehends the actual development. In this stage, software is produced, using the conceptual models created previously. The artifact produced acts as a proof of concept, proving that the proposed solution is possible.

The next step, or Evaluation, attempts to validate the prototype created in the previous stage using an evaluation model. When such model is nonexistent, one is devised alongside other Solution Proposal's artifacts. This step also provides feedback, or circumscription, to the other previous steps which permits an iterative and incremental development (the agile development process is used during the iterations Prototype-Evaluation).

Finally, in the Statement of Learning step, the project is concluded so that knowledge can be produced. Artifacts produced in this stage include concepts, models, methods and prototypes. In our process a Proof of Concept implementation was produced to study the proposed Interaction Model and associated Concepts.

Proposed Interaction Model

As written, the main purpose behind this solution proposal was to develop an easy to use modeling tool, based on the Petri Nets (PN), that could be used, by non-programmers, to define actor behaviors and choreographies in a game/simulation environment, whether running on one machine or on a distributed architecture. Intrinsically, this tool was designed to provide the ensuing quality attributes: portability so that it would not be tied to a particular OS, interoperability which would allow to use the tool with different game/simulation engines, usability to complement Petri Net's accessibility, error recovery due to the fact that designers will be working with a language with a defined syntax and, therefore, they must be warned of syntax errors and how to solve them and scalability relative to the number of actors and/or players.

Given the solution's objectives and non-functional requirements, the proposal was devised to contain the visual language's specification (syntax and semantics), an easy to use visual behavior editor and a language execution engine used to translate the Petri Net models into in-game/simulation actions.

Language

The language used in this solution proposal contains a similar syntax to that of Hierarchical Petri Nets [1] with weighted arcs. By utilizing the capability of grouping sub-nets, this language is able to reduce graphical complexity and thus, improve readability. Another important advantage is that it promotes component reutilization, i.e. the same sub-net can be used in different contexts.

Editor's supported actions:

- Add/Remove Language objects (places, transitions and tokens)
- Link places and transitions through arcs
- Create sub-nets
- Edit objects' properties
- Filter sub-nets according to keywords
- Export/Import sub-nets
- Save/Load Petri Nets to/from disk
- Undo/Redo actions

MODEL

A game/simulation model is represented by a root Petri Net that contains a set of Petri Net models, each symbolizing a different actor archetype. It is worth noting that instances of an actor archetype share the same Petri Net model. Every child of the "root" model is an independent net that is associated to an Actor and follows the language syntax detailed earlier. These PNs cannot communicate directly with one another by means of arcs, but only through explicit messages - tokens moving from an output place of a PN into an input place of another PN.

Places can be of four types: Input Places, Output Places, Fused Places or Regular Places (this designation must not be confused with the naming given to places linked to/from a transition on the original Petri Net language). Regular Places share the same meaning as places in the Petri Net language. Input Places act as actor sensors. This means that when a token arrives at these special places, something was perceived by the actor. Output Places, on the other hand, assume the role of announcers, i.e. when they receive a token, it is announced to the game world that something has happened. Furthermore, an Input Place can be used to observe an Output Place. Finally, Fused Places are places inside sub-nets that are linked with outer-net places, acting as proxies for their outer-net counterparts. Whenever an outer-net place receives/loses a token, its Fused Place receives/loses the same one as well. Unlike places, which have four different types, Tokens, just like in the Petri Net language, stand for a condition that was met. Transitions, however, have associated programming scripts that govern actions. When a transition fires, its script is executed, meaning that an action is taking place.

Interface

The interface's design originated from an iterative process. Initially, a paper prototype was constructed. This prototype was then evaluated through user testing so that it could be refined. After several iterations, the prototype was converted into a mockup representation using Balsamiq [4], as illustrated in Figure 2. This proposal is adequate because the interface's viewport provides the necessary information for the simulation and manipulation of the language's constructs in a segmented way. Because of this, users can easily interact with the editor without having to navigate through menus in order to look for actions. Furthermore, the spatial distribution helps organize the information so that users don't feel overwhelmed.

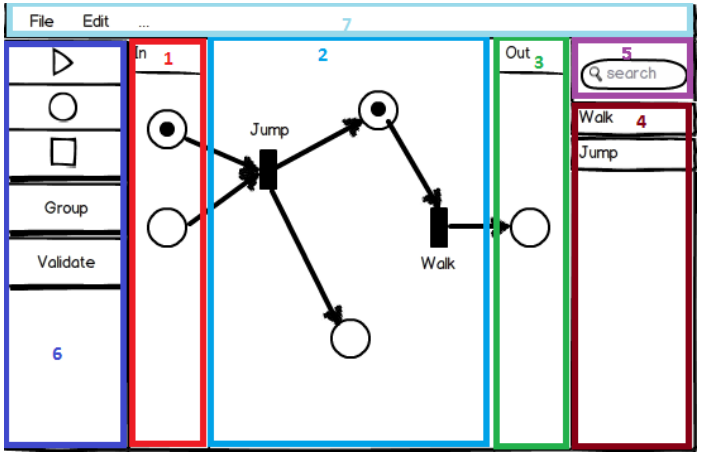

Figure 2. Mockup of the petri net editor's GUI

As can be seen in Figure 2, the editor is divided into 7 panels or menus:

1. **Input Panel**. Here, designers are allowed to put places that represent the actor's inputs. These inputs can range from sensors (vision, hearing, etc…) to messages containing information.
2. **Function Panel**. In this panel, designers can model the actors' logic. As depicted in the figure, this logic can contain both places and transitions.
3. **Output Panel.** Similar to the Input Panel, places set in this panel represent the actor's outputs, or information he transmits to the game world.
4. **Group Panel.** A panel that lists grouped Petri Nets.
5. **Search Panel.** A panel used to filter the Group Panel, through the means of a keyword search.
6. **Button Sidebar.** A sidebar containing the most important action buttons. From top to bottom, these buttons are: Play, to simulate the given petri net; Add Place, as the name indicates, inserts a place onto one of the panels 1 to 3, as given by the cursor; Add Transition, functioning as Add Place but adding a transition instead; Group, used to gather selected sub Petri Nets into one transition; and Validate to check whether the petri net is valid or not.
7. **Menu Bar.** A menu bar where designers can save/load petri nets to/from disk and import/export groups, if implemented.

Proof of concept implementation

The editor's backend was built using Java due to its portability and the interface was made with Java's GUI API: Swing. Initially, the editor was meant to be based on the JFern Editor because it provided JFern's Petri Net threaded simulation mechanism, data structure and PNML exporting/importing API as well as views and controllers for their visual representation. This idea was discarded because the GUI's code was poorly documented, confusing and some of its classes were not made available as source code. Consequently, only the simulator, data structure and PNML parser were used. The reason behind using JFern is that besides offering the previously mentioned set of tools, it was the only tool from the ones researched that allowed to introduce code to be executed when a transition fires, thus reducing some programming effort when developing the editor's execution mechanism. This tool was modified to provide some additional attributes to the Petri Net's objects.

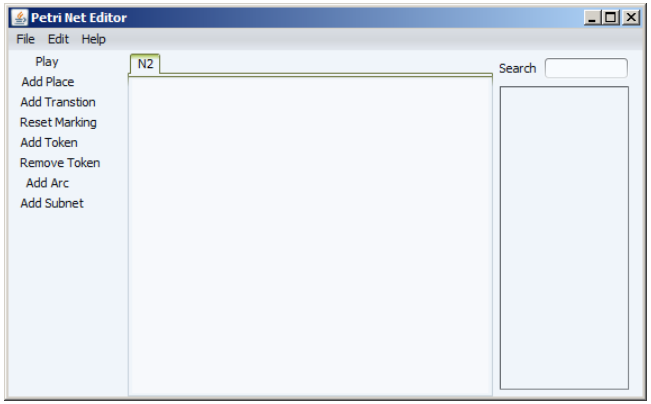

Figure 3. Initial version of the editor

Evaluation

Evaluations were concentrated on the interface as it was a crucial part of the application and encompassed most of the required attributes and objectives devised for the project. The type of tests chosen to evaluate the

Description of the test's task-list:

1. Read crash course and explore the editor for 5 minutes.
2. Read the game's design doc and start a new project.
3. Build a chronometer mechanism.
4. Create a score update mechanic.
5. Build the player's navigation system.
6. Create the player's shooting mechanism.
7. Make a bot spawning mechanic.
8. Devise the enemies' AI.
9. Integrate the score update with a local scoring system.
10. Make a winner announcement system.
11. Save the project.
12. Open a project and answer some questions regarding the language.

interface were usability tests [21]. The main goal of these tests was to verify how users experienced the creation of video game definitions using the editor.

Test Setup

As previously written, the application's interface was evaluated by means of formal usability lab tests. These tests had an expected time of completion of approximately 1h30m, however subjects were free to surpass this schedule. They consisted in individual sessions where each voluntary tester was prompted to setup and define the behaviors in a video-game using the thesis' application in conjunction with the Unity game engine and pre-existing graphical assets (level and character 3D models). During those sessions, testers were accompanied by an evaluator, whose job was to clarify any rising questions and to take notes of events that could happen during the test. In order to help document any event that might escape note taking, audio was recorded.

Each session followed a predefined script. Primarily, test subjects were introduced to the project's context and test objectives. Secondly, they were asked demographic questions for later analysis of the population performing the test; the actual test started afterwards, when testers were given a document with information regarding Petri Nets and were encouraged to explore the interface for 5 minutes, after which they were given the game's design document and a list of tasks that contributed for the creation of said game. After each task, test subjects estimated its difficulty in a scale of 1 to 5. Subsequently they were interviewed to detail their overall user experience and performance and were requested to list the top 5 best and worst aspects of the interface, according to their opinion.

The game that test subjects were supposed to create was, as stated, described in a pre-made design document and detailed in a task-list. This document defined the context of the game, its rules, actors and sensors and scripts that were available to them. The task-list helped guide the users in the completion of the game by dividing it into tasks. The first half of the list contained a step-by-step guide while the second half was only comprised of objectives. This way, testers, during the first half, could learn the basics of the application as well as its quirks. Overall, the game consisted in a competitive first person shooter where players and AI-controlled bots had to toss balls at each other in order to increase their team's score. The following screenshot illustrates the game, as made by one of the test subjects.

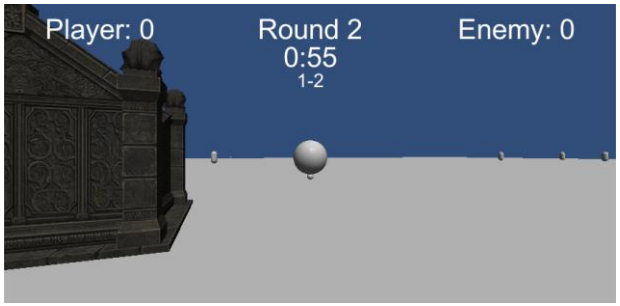

Figure 4. Screenshot of the test game "Spheres of Steel" as created by one of the subjects

Tests were performed using 11 subjects [17] and the data collected from the recorded audio, demographic questionnaire, interviews and notes, was categorized into three classes: demographic information, user performance and usability issues. Although 11 testers participated, only 10 completed the development of the

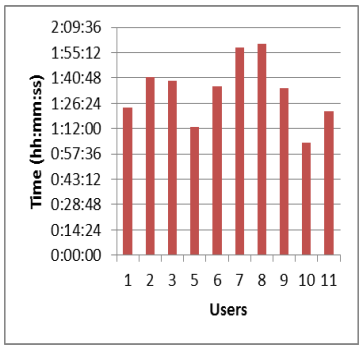

Figure 5. Total Time per User for the complete exercise.

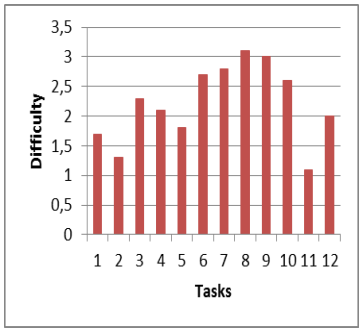

Figure 6. Average Perceived Difficulty per Task (scale 1-5)

game and, therefore, the information aggregated from the subject who had to abandon the test midway, due to personal reasons, was only used in the demographic information and usability issues as there was not enough information necessary to compile in the user performance category.

Demographic questionnaires required subjects to state their age, sex and highest academic degree. They also inquired users to rate their experience, in a scale of 0 to 2 – 0 meaning never heard of the term and 2 denoting highly proficient - in textual programming (TP), visual programming (VP) and game development (GD). The reason for this is that textual programming introduces people to algorithms; Experience in visual programming would make the subjects used to the mannerism required to manipulate a visual language; and experience in game development would make users accustomed to the steps involved in creating a game.

The population sample is composed of 2 females and 9 males, with an average age of 26. Their qualifications range from Bsc student to Phd student. From this information, it can be deducted that the highest degree achieved by the test subjects range from High School to a Msc coinciding with the academic education that game designers often have. The average experience in textual and visual programming and game development is, as self-stated, 1.18/2, 0.45/2 and 0.72/2 respectively. This means that subjects are familiar but not proficient in textual programming, barely know about visual programming but have a little knowledge of game development. Overall, the subjects constituting the population sample were selected in a manner that allowed for a heterogeneous sample, in means of qualifications and experience levels. This way, in theory, it would increase the amount of issues found by subjects.

Results and Analysis

USER EXPERIENCE

In this context, user performance consists in the overall time subjects took to complete the game's definition and each individual task and their relative perception of the difficulty of every task. This was extrapolated from the audio recordings and ratings that testers gave after finishing their tasks.

From the data presented Figure 5, it was concluded that on average, testers completed the test in 1h33m, only 3m above the expected time, and their perception of the test's difficulty was, on average, 2/5 – this was derived from the values available in Figure 6. This means that users thought the test they made, while using the application, was easy. Nevertheless, only 1 out of 10 subjects did not require the evaluator's assistance.

USABILITY ISSUES

The notes and interviews originated a list of usability issues. These issues, after compiled and normalized, were categorized according to their importance [17], occurrence frequency, type and occurrence by task and by user. From the usability tests, 406 occurrences, distributed across 88 different event classes, were found. There were only three importance levels given to issues: High, Medium and Low. These levels were attributed according to the issue's degree of prevention in completing a certain task. Each level corresponded to a number: High corresponded to 1, Medium to 0.66 and Low to 0.33.

In total, there were 9 types used to classify the issues [12]. These types were Functional Error (FE), Affordance (A), Feedback (FB), Perception of System State (PSS), Naming Interpretation (NI), Instruction Interpretation (II), Representation Interpretation (RI), Mappings (M) and Domain Knowledge (DK). It was assessed that most events lie on the category of Mappings. This means that during the tests, most recorded events were comprised of discrepancies between their users' intentions and the interface's available actions.

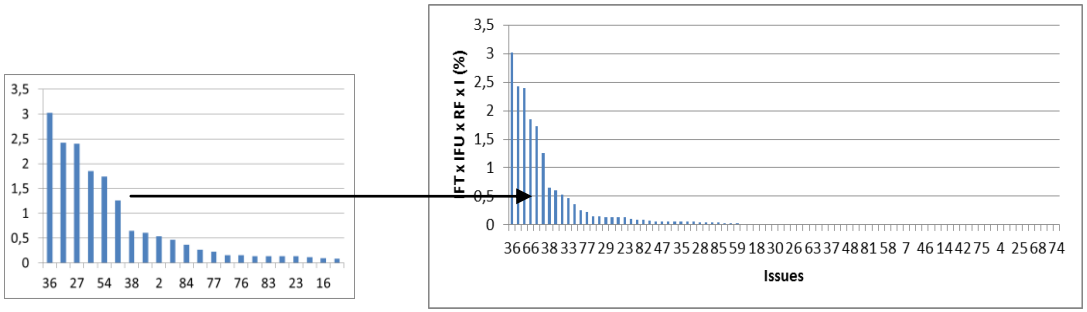

Figure 7. Priority Level per Issue

USABILITY CORRECTIONS

We used a metric to help identify the most critical problems sorted accordingly to their potential impact, to elaborate a correction plan. This metric consisted in a two part algorithm. In the first part, a value, referred to as priority level, was attributed to each issue by calculating the arithmetic product between the its frequency per user (IFU) and per task (IFT), its relative frequency (RF) and its importance (I). The formula is given by the expression $Priority_i = IFT_i \times IFU_i \times RF_i \times I_i$.

This results in a value ranging from 0 to 1 because all variables were normalized beforehand. By multiplying these factors, it is assured that, for instance, issues that appeared frequently during tasks, were encountered by most users and tasks, and hindered the completion of said tasks are given more priority than issues that, for example, were not as frequent or important. A chart detailing the priority levels per problem, sorted by value, is presented in Figure 7.

The revised interface is illustrated in the screenshot in Figure 8.

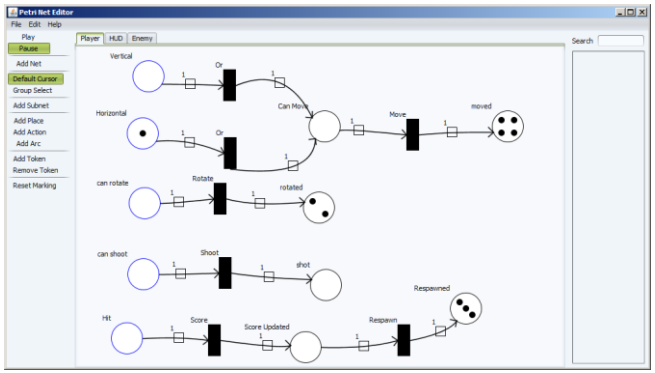

Figure 8. Screenshot of the editor interface (post usability corrections)

The most notorious differences between Figure 3 and Figure 8 rely on the tool bar. In it, some toggle buttons were introduce to provide a better indication of the system's state. Another difference is that the buttons were sorted and grouped with separators to avoid button pressing mistakes and to decrease the time it took an user to look for a specific button. Finally, some label names were changed to avoid confusion

Conclusions

In sum, this paper introduced an interaction design model for a behavior editor used to edit constructs of a language, based on Petri Nets, intended to define actor behaviors and choreographies. This design was instantiated as a proof of concept and validated through formal usability lab tests.

We concluded that in spite of pending usability issues most users were able to complete the proposed game behavior definition tasks using the PN editor interface. We think this provides a good evidence to reinforce the case that PNs can be used to advantage in the game modeling process and, when coupled with a runtime simulation, can provide an interesting immediate feedback loop for fast design experimentation.

For future work, it would be interesting to develop a look and feel more appealing to the editor's target audience, as this attribute was not considered during the development of the editor, since the main focus was to create a prototype to illustrate the interaction design model.

Acknowledgements

We would like to thank all the support from lab colleagues for the help they provided through the development of this editor, and the test volunteers for their availability to help validate this work.

References

[1] Aalst, W. Hierarchical Petri-Nets. Eindhoven University of Technology, 2011. http://cpntools.org/_media/book/hcpn.pdf .

[2] AgentSheets. http://www.agentsheets.com/ .

[3] Araújo, M., & Licinio, R. Modeling Games with Petri Nets. In Proc. Digital Games Research Association (DiGRA) on Innovation in Games, Play, Practice and Theory (2009).

[4] Balsamiq. http://www.balsamiq.com/ .

[5] Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., . . . Weber, M. The Petri Net Markup Language: Concepts Technology and Tools. Applications and Theory of Petri Nets (2003), 483-505.

[6] Brom, C., & Abonyi, A. Petri Nets for Game Plot. In Proc. AISB on Narrative AI and Games workshop (2006).

[7] CryEngine. http://mycryengine.com/ .

[8] Dormans, J. Machinations Framework. http://www.jorisdormans.nl/machinations/wiki/index.php?title=Machinations_Framework .

[9] JFern. http://sourceforge.net/projects/jfern/ .

[10] JPetriNet. http://jpetrinet.sourceforge.net/ .

[11] Hevner, A., & Chatterjee, S. The General Design Cycle. Design Research in Information Systems: Theory and Practice (2010), 26-27.

[12] Hourcade, J. Usability Principles. University of Iowa, 2006. https://www.cs.uiowa.edu/~hourcade/classes/fa06hci/lecture2.pdf .

[13] Milligton, I., & Funge, J. Artificial Intelligence for games. Morgan Kaufmann, Burlington, USA, 2009.

[14] Natkin, S., Vega, L., & Grünvogal, S. A new methodology for Spatiotemporal Game Design. In Proc. CGAIDE'2004, Fifth Game-On International Conference on Computer Games: Artificial Intelligence (2004).

[15] Petri, C., & Reisig, W. (2008). Petri Net. http://www.scholarpedia.org/article/Petri_net .

[16] PIPE2. http://pipe2.sourceforge.net/ .

[17] Sauro, J. 10 Things to Know about Usability Problems. http://www.measuringusability.com/blog/usability-problems.php .

[18] Sauro, J. Applying the Pareto Principle to the User Experience. http://www.measuringusability.com/blog/pareto-ux.php .

[19] Scratch. http://scratch.mit.edu/ .

[20] SDL. http://www.libsdl.org/ .

[21] Sharp, H., Rogers, Y., & Preece, J. Interaction Design: beyond human-computer interaction. Wiley, USA, 2002.

[22] Stencyl. http://www.stencyl.com/ .

[23] ToonTalk. http://www.toontalk.com/ .

[24] UDK. http://www.unrealengine.com/udk/ .

[25] Unity 3D. http://unity3d.com/ .

[26] XNA Game Studio. http://msdn.microsoft.com/en-us/library/bb200104.aspx .

[27] Yasper. http://www.yasper.org/ .

[28] Ward, J. What is a Game Engine? http://www.gamecareerguide.com/features/529/what_is_a_game_.php .

[29] WoPeD. http://woped.ba-karlsruhe.de/ .