Why you should empirically evaluate your AI tool: *From SPOSH to yaPOSH*

Jakub Gemrot[1], Martin Černý[1] and Cyril Brom[1]
[1]*Faculty of Mathematics and Physics, Charles University in Prague, Malostranské náměstí 25, Prague 1, Czech Republic*
{*brom, gemrot*}*@ksvi.mff.cuni.cz, cerny.m@gmail.com*

Keywords: Behavior Design, Computer Games, Tool Productivity, User-Study, Agent Languages

Abstract: The autonomous agents community has been developing specific agent-oriented programming languages for more than two decades. Some of the languages have been considered by academia as possible tools for developing artificial intelligence (AI) for non-player characters in computer games. However, as most of the research related to the development of new AI languages within the agent community does not reach production quality, they are seldom adopted by the games industry. As our experience has shown, it is not only the actual language that matters. The toolchain supporting the language and its integration (or lack thereof) with a development environment can make or break the success of the language in practical applications. In this paper, we describe our methodology for evaluating AI languages and associated tools in practice based on controlled experiments with programmers and/or game designers. The methodology is demonstrated on our development and evaluation of SPOSH and yaPOSH high level agent behavior languages. We show that incomplete development support may prevent the tool from giving any benefit to developers at all. We also present our experience from transferring knowledge gained during yaPOSH development to actual AI design for an upcoming AAA game.

1 INTRODUCTION

Modern computer games are becoming more and more complex and game designers seek for a way to bring rich worlds to their audience. This is especially true for role-playing games (RPGs) that are advertised as "large open-worlds" players can explore and interact with. However, the production of such worlds requires a vast amount of authoring time; the time that is spent not only by environment designers and animators but also by non-player-characters (NPCs) designers. NPCs designers are responsible for making those worlds alive by populating them with a wide variety of NPCs. We see the NPC behavior production as one of the bottlenecks in the whole game production process; the large effort associated with developing complex NPCs behavior causes that most NPCs demonstrate only very limited and schematic behavior. Thus players often do not perceive the NPCs as lively inhabitants of the virtual world and are instead left with an impression of indifferent soulless puppets.

Although the demand for better NPC behavior is increasing, the actual NPC behavior in games improves only slowly. In our view, an important reason for that is the lack of proper NPC behavior specification languages and development tools (further referred to uniformly as *tools*). Over the years, academia has proposed many technologies that could be — in theory — applicable to this problem. But the industry is reluctant to adopt those solutions as they are not accompanied with case studies that would shed light on their impact on the game production process. We report on our experience that it is hard to prove that novel/existing AI tools related to the NPC behavior production are of benefits to the industry. We also present a methodology how to (try to) do so. The methodology is based on controlled experiments we have conducted over past four years, in which we studied the productivity of the custom NPC behavior creation tool SPOSH (Bryson, 2001) and its improved version yaPOSH. We briefly present findings of the studies as well.

The main goal of the paper is to introduce the community to practical considerations necessary for proper empirical evaluation of AI tools and to stress the importance of evaluation for practical applicability. It has been noted that even well-established academical tools, that seem intuitively beneficial for AI development, may perform well below a plain programming language (Píbil et al., 2012) (although the

evaluation was non-systematical and purely qualitative).

While the paper is phrased in game development vocabulary, we believe that most of our results are transferrable to other areas of possible industrial AI applications.

The rest of the paper is organized as follows: First we discuss various non-obvious requirements the game industry imposes on AI tools (Section 2) and deal with the notion of productivity (Section 3), then we discuss literature related to tool evaluation (Section 4) leading to an overview of our methodology in Section 5. Section 6 describes user studies we have conducted and Section 7 deals with several important details for successful application of the methodology. Finally, Section 8 discusses the conclusions we have drawn from our experience.

2 NEEDS & FEARS OF THE GAME INDUSTRY

Recently we have had the chance to participate in an AI design process for an upcoming AAA game title. We have successfully transferred many of our ideas on the NPC behavior design to the game industry; ideas that are based on our seven years of experiences from teaching NPC behavior design at our university and on the studies we have conducted. In this section we discuss non-obvious requirements the industry imposes on AI tools.

Our view of the industry requirements is based mainly on our tight cooperation with a game studio that was newly founded but many of the developers have history in making AAA game titles (Operation Flash Point, Mafia 1 & 2). Although we work with a single game studio, discussions with industry representatives at conferences and experiences of other AI researchers indicate that the requirements are very similar in most of the industry.

Alex J. Champandard pointed out that the industry yields to the argument "You will ship your game three months earlier." (Champandard, 2010). Applying this to our perspective, industry simply needs to produce reliable NPC behaviors quickly. However, as NPC behaviors can be scripted and some tools for behavior specification already exist, the industry is skeptical towards adopting new ideas and related tools.

The first thing to note is that every AI tool related to the game development has to be usable under different use cases; each use case is having its own purpose: 1) reading/understanding how concrete behavior works during the design time, 2) debugging existing behaviors at runtime, 3) extending existing behaviors and finally 4) creating new ones. Those four situations will be referred to as the "game development use cases" (GDUCs).

Secondly, and on a related note, a game development studio does not need only good tools, but also a well thought-out workflow that integrates design, prototyping, development and quality control. If a tool cannot cope with such a workflow, it is not likely to be adopted.

Thirdly, it is strongly desirable to redirect as much development effort as possible to a less qualified workforce. This is also true for NPC behavior design as it is impossible to rely on senior programmers for the behavior production only as they are rather scarce and expensive resource. Thus a tool (as well as the language) should be usable by less experienced programmers or game designers as well.

Generally speaking, development of a computer game is to a large extent a software engineering task. The comprehensibility, maintainability and reusability of the code and data produced play an important role and adoption of complicated algorithms is avoided if it is not having a great impact [1]. Simple solutions are preferred as they facilitate debugging and keep the system predictable.

To conclude, the game studio will typically fear that the new tool will not fit into the game production seamlessly as tools in the academy are not used in the whole context of the game production and thus the negatives might well outweigh the positives.

3 MEASURING PRODUCTIVITY

If the industry is to adopt anything, it needs many proofs. Firstly and the most importantly, a fully working prototype of the technology that proves the industry that they need it must exist. This serves as an opening for the discussion about the time and the memory impacts of the tool during the game runtime. If those are found acceptable, the discussion will turn to the debate about the productivity of the tool. Unless the tool allows the designer to achieve something that was previously unachievable, one has to prove that accomplishing a task using the tool is easier compared to not using any special tool at all or using a different tool.

The term productivity should not be confused with usability. While usability and productivity are strongly related, they are not the same. A tool may

[1] For instance, the first author of this paper was once present at discussion of level designers talking about how high-dynamic-range lighting is great yet extremely hard to tweak as there are a lot of intertwined parameters so it slows the whole level creation process down.

prove to be very productive even though it has multiple usability issues. On the other hand a tool may be perfectly usable but may turn out to bring no practical benefit to the user, because the same task can be easily accomplished with a different workflow. However, an increase in usability almost certainly results in an increase in productivity, as it allows the user to work faster and with less obstacles.

How to formally prove that one tool is more productive than the other in the context of NPC behaviors is hard and rigorous research faces serious methodological as well as practical issues. In contrast to the tool internals, which can often be evaluated by running the tool on a large set of test cases, it is not clear how to evaluate the usability or the productivity of the tool itself. The way users are working with the tool is bound strongly to the tool internals and thus any evaluation effectively evaluates the tool itself, its usability and productivity at the same time. Also the evaluation is strongly influenced by the user's ability to understand and apply the concepts the tool is built upon and its workflow.

4 RELATED WORK: EVALUATING PRODUCTIVITY AND USABILITY OF TOOLS

To our knowledge, the only published evaluation of productivity of an AI tool is a purely qualitative (one subject) case study of using an agent-based language in a non-trivial setting (Píbil et al., 2012). The study notes that the language is in many aspects inferior to plain Java. The ScriptEase AI tool (Cutumisu et al., 2007) was evaluated by letting students design their own interactive story in the Neverwinter Nights computer game. The study however measured only the number of various language constructs the students used and did not focus on usability or productivity, neither was ScriptEase compared to the default tool for the respective environment.

Some work has been done in determining the usability of specific general purpose programming language constructs (Sadowski and Kurniawan, 2011) or to evaluate how quickly novice programmers learn language constructs (Stefik et al., 2011). None of the works known to us deal with evaluating usability or productivity of development tools for a language neither are we aware of a work that would quantitatively evaluate an agent-oriented language. Since our focus is on the tools as well as the specific language, we cannot directly reuse any of the methodologies of the above papers.

An insight to the evaluation problematic can be gained from the academic research on human-computer interaction. A classical paper of Jeffries et al. (Jeffries et al., 1991) compares four different techniques for the usability evaluation of software. They conclude that the best performance is achieved by letting a group of humans test the application in a realistic setting. Among user groups, user-interface (UI) experts following the methodology called heuristic evaluation are better at testing UI than regular software beta testers. Two other methods were also tested — validating the UI against expert-designed guidelines and cognitive walkthrough which is a structured analysis of the program by developers. However, techniques not involving testing with humans fared much worse at determining usability deficiencies.

Further studies confirmed the findings of Jeffries et al. (Karat et al., 1992; Nielsen and Phillips, 1993). A good overview paper on usability testing methods is (Hollingsed and Novick, 2007). This overview suggests that the most cost-effective results are obtained by combining internal evaluation within the development team early in the design process with user studies later on.

Although the aforementioned research is aimed at usability, we think that most of its results generalize to productivity as well, especially the necessity to involve actual human users working on realistic tasks in the evaluation process.

5 METHODOLOGY

Based on insights from the related work and the experience from our studies we propose a general research methodology for comparative controlled experiments that should validate both usability and productivity of AI design tools. Proposed methodology is certainly neither complete nor bullet-proof and is open for comments. We think that the relation between the practice of comparative controlled experiments for AI design tools and the respective methodology is similar to chicken-egg problem. As there is no methodology well thought out to follow, it is hard to conduct experiments yielding fruitful data about the tools productivity within the whole context of the game development; as there is a lack of experiments conducted and presented in papers, it is hard to formulate any methodology whatsoever. Although some of the individual steps might seem obvious, doing everything correctly without any reference methodology is not trivial as we have learned through trial and error. For more experienced experimenters, our methodology could serve as a kind of a checklist for good ex-

periment design.

5.1 Methodology Steps

1. Choose the game AI problem you plan to solve. E.g., behavior specification of game NPCs in the context of first-person shooters (FPS).
2. Think about tasks the team of game developers will face during the problem solving utilizing existing tools (e.g. GDUCs as introduced in Section 2).
3. Find a technology that is promising in solving the problem, e.g. SPOSH (Bryson, 2001) that was successfully used in robotics and shown to be usable for NPCs of FPS games.
4. Implement the tool internals and create a simple UI that should support not only the solving of the problem but (ideally) also the spectrum of tasks from the Step 2.
5. Work with the interface yourself or within the development team to determine the most obvious deficiencies and fix them. Think and stress (at least) different GDUCs that your tool will undergo during the game development: a) reading/understanding of the data it produces (e.g., is your colleague able to quickly grasp what you have done/created with the tool?), b) debugging existing data (e.g., is it easy to diagnose and correct an error you have made?), c) extending the data (e.g., is your colleague able to continue with a half-done work?) and d) creating the data. Some of the structured approaches mentioned in the related work might prove beneficial. Long-term student projects or bachelor theses using the tool have also shown to be useful for initial evaluation.
6. Design a realistic set of tasks that your tool should help in solving but limit their complexity to allow for controlled experiments in a lab. Ideally, users' solutions should yield qualitative data so the solutions can be compared. The set of tasks should sample tasks from the Step 2.
7. Do a "pilot study": solve the task set by yourself and entice a few colleagues to solve it as well in order to predict the time your users will need to solve the task set and to "debug" the experimental setting.
8. Gather two groups of users sampled randomly to solve the task set: one will use your tool the other will use a baseline tool or no specific tool at all (e.g., only standard IDE for underlying programming language the NPCs are implemented in). Since involving actual game development teams is usually not feasible, we did our evaluations with students of AI courses. We think evaluating on students is reasonable, because a) many of our students are of high skill, already working part-time as junior programmers, b) behavior developers in games usually are not programming experts and c) it is accepted practice in other branches of science to work with student subjects (e. g. many psychological experiments are performed with students).
9. Let all users pass a preliminary test that confirms they have a sufficient experience using any tool, platform or game engine they will need to utilize for your tasks.
10. Let each user solve the task set. Measure their time required to solve respective tasks and quality of the solution. Gather their feedback — what was easy and/or difficult to achieve? What features of the tool were used? Distinguish between features the tool offers during the game runtime and the design time carefully.
11. Optionally prepare a second set of tasks and swap groups to mitigate sampling error (e.g., different average programming experience between user groups).
12. Analyze the results and see, which group performed better and what obstacles users faced. Be careful to distinguish between artifacts of the experimental setup (e.g. users became fatigued from the overly long study, computers used for the study were misconfigured), artifacts of the assigned task (e.g. users misunderstood the assignment, users spent too much time figuring out an unobvious trick not related to your tool) and the actual effect of the tool.
13. Analyze source code / data that solves the task set to gain insight which features of your tool were used the most, which were avoided and which were misused. Perform post-hoc (but as soon as possible) interviews with users that misused your tool to gain additional information.
14. If the tool did not improve user performance sufficiently, alter the tool and its user interface to mitigate the difficulties that were faced most often. If necessary, also change the experimental setup so that it really tests user performance with as little noise as possible.
15. Reiterate to the Step 4. Having the same group of users in next experimental run can bring valuable insights about the difference between the two versions of the software, at the cost of not gathering data about the performance of novice users.

6 THREE USER-STUDIES: FROM SPOSH TO YAPOSH

We now present three user-studies that were conducted to gain insight into the productivity of the academy tool SPOSH in the context of NPC behavior design. The first two studies were done with SPOSH and the last study was done with its improved version yaPOSH. The presented methodology was formed based on our experience gained from the studies.

SPOSH is a dialect of POSH action selection developed by Bryson in late 1990s (Bryson, 2001). POSH specifies clear action selection semantics capturing NPC behavior in a tree. Roughly speaking, every edge in the tree is annotated with a *sense* (or multiple senses) — a condition that must hold in the environment for the edge to be active. The children of nodes are ordered by priority. To determine the action to perform, the highest-priority child of the root node connected by an active edge is chosen. If it is a leaf, an action or an action sequence associated with the leaf is executed, otherwise the node is searched recursively. SPOSH behavior primitives (senses and actions) serve as a communication interface between the SPOSH engine and the rest of the NPC. Senses and actions are user-defined and implemented in the same language as the SPOSH engine (in our case Java). Behavior primitives are implemented as (class) methods of a specific signature. The SPOSH language syntax is Lisp-like. As Lisp-like syntax has been found confusing to our users, we have developed a simple drag & drop graphical editor for SPOSH plans and integrated it into NetBeans Java IDE (see Figure 1) . The same NetBeans Java IDE is used for behavior primitives coding in Java.

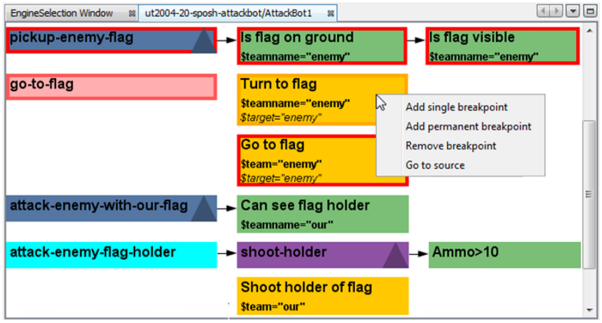

Figure 1: Screenshot of the yaPOSH Plan Debugger.

6.1 Studies Setting

All studies focused on creation of NPC behaviors in game-like tasks. We were interested in comparing performance of users using Java with SPOSH (yaPOSH) tool to users using Java only. Users were given necessary low-level primitives (turn, move, navigate, shoot, speak, do-i-see, do-i-hear, etc.) and they were asked to create high-level plans for the NPC, which should solve game-like tasks. We have been working with three tasks. The first task was to create a simple HunterBot behavior, where an NPC had to explore its environment, gather weapons and hunt down an enemy NPC. The second task was a GuideBot behavior; to search the environment for other friendly NPCs and guide them home. The NPCs were programmed to follow the bot after a request and to stop following whenever they lost visual contact with the bot. The third task was to alter the existing GuideBot behavior and turn it into the GuardBot behavior that protected the friendly NPC while guiding it as hostile NPCs were added to the environment.

Studies were always done as a part of the semestral test of the course on intelligent virtual agents; during the course, students were taught how to create NPCs behaviors both in plain Java and using our SPOSH (yaPOSH) tool.

Users were given 3.5 hours to finish a single task. We currently measure the productivity by the total time a subject needs to accomplish the tasks. Future work should aim on the question how to distinguishing between different development use cases during the subject's work, as they are often mixed together.

6.2 The 1st Study — Java vs. SPOSH + Java

The first study (removed) was a pilot where we have designed a comparative controlled experiment in order to answer a general question about the SPOSH tool suitability for the specification of NPC behaviors. Its theme was focused on the subjective language preference not the actual productivity of the tool. The hypothesis was that the Java IDE NetBeans featuring graphical SPOSH plan editor plugin will prove to be better than NetBeans alone. The pilot consisted of two tasks; HunterBot and GuideBot. 30 students have participated in the study. The main findings were:

(a) SPOSH was favored for the HunterBot task, but not for the GuideBot task (quantitative answers).

(b) Students argued that GuideBot task could have been easily solved with event-driven approach, which could have been easily captured in Java but not in SPOSH (qualitative answers).

(c) As no user of any group had a problem finishing both tasks, it raised questions about tasks complexity and relevance of users opinions.

Students' comments revealed that their language preference correlated with the difficulty they had with its actual use. The HunterBot behavior was reported to be better expressible within SPOSH than the GuideBot behavior and thus the preference for the SPOSH language differed between these two task. Although SPOSH is a tool specifically designed for NPC behavior development, the study did not indicate that SPOSH is generally preferred than plain Java.

6.3 The 2nd Study — Java vs. SPOSH + Java

The second study (removed) used GuideBot and GuardBot tasks. This time we introduced a twist into the setting for the second task; all users received implementation of GuideBot from someone else and they were asked to extend it into GuardBot. This was intended to raise difficulty of the second task and to test the tool under different GDUC; users were forced to read alien code and extend it. The study was hard for students, but brought crucial data. Only two out of 22 students were able to finish the second GuardBot task. Qualitative data were also more fruitful, as users were commenting on the code of someone else. The necessity to alter existing (not always well thought-out) behavior plan/code revealed strong and weak points of SPOSH.

However, the study's findings regarding productivity of the tools used were rather inconclusive: a) The average time required to solve the GuideBot task by subjects in both groups was almost the same: 2:42 hours (sd = 28 minutes) for Java and 2:50 hours (sd = 33 minutes) for SPOSH. b) It was impossible to extract data from the GuardBot task as it was finished by only two students. But the study still brought some interesting insights:

(a) High-level SPOSH plans helped users to quickly grasp the general idea behind the code created by someone else. Understanding behavior written in plain Java was found to be more difficult.

(b) Both user groups had similar opinions about the suitability of their tool for the assignment.

(c) There was a shift of users' opinions to dislike SPOSH after they failed the 2nd task.

(d) As SPOSH constructs lacked parametrization, the amount of logic present in the plan was limited.

(e) SPOSH users complained about the complexity of custom actions they implemented in Java. The complexity was imposed by the SPOSH engine implementation that forced users to track the action state (action initialization / execution / finalization) by themselves.

(f) Usability issues of SPOSH plan editor were reported; most notably a problem with the plan debugging that relied on text logs only.

Despite our intuitions about SPOSH, the study results once again did not show that SPOSH is more productive than plain Java. Nevertheless, it revealed an area where SPOSH (if improved) can beat Java and thus its features may be relevant to the industry.

6.4 The 3rd Study —Java vs. SPOSH + Java

Based on the findings from the 2nd study, we altered SPOSH and created yaPOSH. Changes: 1) yaPOSH plan primitives were made parameterizable. 2) New yaPOSH debugger was created that allowed users to place breakpoints on plan nodes. 3) yaPOSH engine was improved to track the state of executed actions.

The third study (not published yet) used the same setup as the second one. Given the same setting, we decided to have only single group of users who were using yaPOSH+Java. We recruited 18 students different from those that participated in the second study.

Results of the study are encouraging:

(a) All students successfully completed both tasks. Students finished the first task in 1:29 hours (sd = 31 minutes) on average and the second task in 3:15 (sd = 47 minutes).

(b) All yaPOSH changes were picked by almost all users, especially yaPOSH debugger and plan constructs parameterization that allowed pushing more logic into yaPOSH plans in contrast to the SPOSH.

(c) Users shifted their opinions to like yaPOSH more after they finished the 2nd task.

6.5 Discussion

The results of the 3rd study show how proper tool support can boost productivity. Students from the 3rd study using yaPOSH have finished the first task around 1.8 times faster than students from the 2nd study (both tools counted, as the times were very similar). Additionally, all students from the 3rd study were able to finish the second task in the given timeframe. Unfortunately, due to the nature of studies, it is hard to isolate which new feature contributed the most to this improvement. We believe it is a strong evidence that standard programming languages (such as C++/Java or Lua) are not sufficient as a tool for NPC behavior creation. However one can object that changes done in SPOSH might have helped specifically in GuideBot and GuardBot tasks (even though

we did not design yaPOSH this way and we are using it in different contexts as well) and thus the productivity improvement might not generalize to other tasks. But even if this was the case, our results would still be applicable to industrial practice as game studios are always adapting existing tools and engines to suit the needs of the particular project they are working on. Therefore, we suggest that AI experts should be part of the game design team since the early preproduction stage in order to have the time to tailor and test the AI tools before the production stage starts.

This necessity is even greater considering the results of our first two studies: a well thought-out tool which has taken a significant amount of time to develop (SPOSH) failed to outperform the tool which was already available (Java).

7 METHODOLOGY DETAILED

Based on experience from conducted studies, we now discuss some of methodology steps in a greater detail to provide a cook-book of useful ingredients for a good experimental design.

7.1 Designing Task Sets

Choosing right tasks for the evaluation (Step 4) is critical. Simple tasks might not let users use the full power of the tool or provide only small differences between the groups, e.g. the 1st study.

A task too complex may uncover weaknesses of tools used (e.g. the 2nd study), but such study will not provide relevant data about the tool productivity (e.g. students failed to finish the 2nd task of the 2nd study). When multiple tasks can be assigned, it is a good idea to choose them from the whole spectrum. Another alternative (yet to be tested) is to design a lot of simple tasks and ask users to complete as many as they can in a given time frame.

Various tool advantages and disadvantages manifest themselves under different use-cases, e.g., the 2nd tasks of the 2nd and the 3rd study. Thus the task set should be designed to test how users work with the tool under different GDUCs. If possible, each task should be designed to target a single use-case (complex tasks can be broken down to several steps, each step analyzed separately) and the whole task set should cover them all.

It is also important to minimize the amount of work needed to solve the task that does not directly involve your tool, e. g., when testing a behavior tree debugger, the user should not be forced to fix errors that stem from the inaccurate NPC navigation.

The scale of the tasks is also important. If the task set can be completed within hours, it allows for a controlled experiment in a lab, which lets the researcher to gather data without much noise and to monitor individual user progress. On the other hand, such a time frame often precludes realistic problems as they frequently cannot be solved that quickly. In our research we have focused on controlled experiments, but the length of the experiments necessary for proper evaluation was on the border of manageability; in 8 hours long experiment, users became seriously affected by fatigue throughout its second half. It is always beneficial to let a few colleagues to solve the experiment tasks during a pilot study. We have observed, that (the mix of bachelor and master) students required 2-4 times more time to complete given tasks than our colleagues experienced in behavior development.

7.2 The Users

Recruiting users to test the tool is probably the most difficult part of the study. Obviously the more users the better, but even in case of a few users, methods of qualitative research allow the researcher to extract valuable data from the test group. In an academic setting we have found it useful to let students of a relevant course test the tool as a part of their semestral evaluation test.

Unless very focused test group is available (such as a development team from a specific studio) it is necessary to account for differences in skills and background knowledge of individual users. Pre-test questionnaires determining relevant previous experience of the users are very useful to this end. If possible, a within-subject experiment design also helps to alleviate those issues.

The users should also be already familiar with the tool they will test so that they understand all of its features necessary for solving the task. This is especially the case if they are already proficient with a baseline tool. One should bear in mind that a tool might be of different value to novice users and to advanced users and — if possible — experiments should be conducted with both groups present.

7.3 Collecting Data

There are two major classes of data to collect: objective and subjective. The objective data include the actual performance of the user at the tasks, especially the time it took the user to finish individual tasks, logs of user activity (even screen capture) allowing for deeper analysis of the resulting creations of the users. This is where there is the largest benefit of controlled

experiments — they allow to gather much more objective data than field studies.

Subjective data include the individual user experiences. The most convenient way to gather them is through questionnaires and by structured interviews with the users. While objective data are usually good at showing "what" happened, the subjective data often help to clarify "why" it happened or to filter out biased users. If the user group is large enough, all of the data should be statistically analyzed.

So far, we do not have other metric for the tool productivity than the time required to solve the tasks.

8 CONCLUSIONS

We have presented our position on the AI tool design, in particular we have stressed the importance of user evaluation of the tools. We believe that comparative controlled experiments should become the main academic approach how to demonstrate AI tools usefulness to the game industry and how to drive the tool improvements.

In general, designing proper comparative experiments for usability and productivity evaluation is tricky as there are many factors that affect user performance, which are difficult to control. Since this research area is relatively new, there are no definitive "best practices" to follow. In this paper we have presented what we consider a candidate for such "best practice" in the case of languages and tools for design of game AI. We believe that our experience is to a large extent transferable to other real world applications of AI and supporting tools.

The experience we have gathered throughout the development of SPOSH and yaPOSH have let us design a new tool for developing NPCs behavior that has been adopted for real-world deployment in the design process of an upcoming AAA computer game. Although the project leads were initially skeptical about the idea, thanks to our studies, we were able to show that we know how to create a practical tool. The tool itself was ultimately received very well by both the project leads and its users (i. e. scripters) and has fully replaced their previous behavior design solution.

ACKNOWLEDGEMENTS

Human data were collected with APA principles in mind. This research is supported by the Czech Science Foundation under the contract P103/10/1287 (GACR), by student grant GA UK No. 655012/2012/A-INF/MFF and partially supported by SVV project number 267 314.

REFERENCES

Bryson, J. J. (2001). *Intelligence by design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agent*. PhD thesis, MIT, Department of EECS, Cambridge, MA.

Champandard, A. J. (2010). Finding a better way to Mordor. Presentation, CIG 2010. `http://vimeo.com/14390998` Accessed 2014-01-05.

Cutumisu, M., Onuczko, C., McNaughton, M., Roy, T., Schaeffer, J., Schumacher, A., Siegel, J., Szafron, D., Waugh, K., Carbonaro, M., et al. (2007). ScriptEase: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32 – 58.

Hollingsed, T. and Novick, D. G. (2007). Usability inspection methods after 15 years of research and practice. In *Proceedings of the 25th annual ACM international conference on Design of communication*, pages 249–255. ACM.

Jeffries, R., Miller, J. R., Wharton, C., and Uyeda, K. (1991). User interface evaluation in the real world: a comparison of four techniques. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 119–124. ACM.

Karat, C.-M., Campbell, R., and Fiegel, T. (1992). Comparison of empirical testing and walkthrough methods in user interface evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 397–404. ACM.

Nielsen, J. and Phillips, V. L. (1993). Estimating the relative usability of two interfaces: Heuristic, formal, and empirical methods compared. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 214–221. ACM.

Píbil, R., Novák, P., Brom, C., and Gemrot, J. (2012). Notes on pragmatic agent-programming with Jason. In *Programming Multi-Agent Systems*, volume LNCS 7217, pages 58–73. Springer.

Sadowski, C. and Kurniawan, S. (2011). Heuristic evaluation of programming language features: two parallel programming case studies. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 9–14. ACM.

Stefik, A., Siebert, S., Stefik, M., and Slattery, K. (2011). An empirical comparison of the accuracy rates of novices using the Quorum, Perl, and Randomo programming languages. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 3–8. ACM.