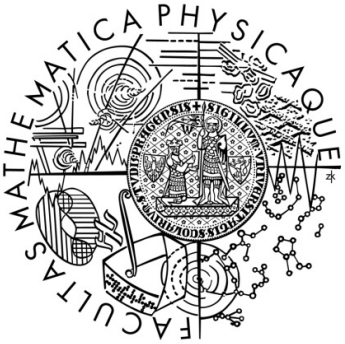
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Martin Černý

Reducing Complexity of AI in Open-World Games by Combining Search-based and Reactive Techniques

Department of Software and Computer Science Education

Supervisor of the doctoral thesis: Mgr. Cyril Brom, Ph. D.

Study programme: Computer Science
Specialization: Theoretical Computer Science

Prague 2016

This thesis is dedicated to my parents who gave me a great start in life and to Antonia for making my present days magnificent.

I want to thank my supervisor Cyril Brom for an extra pair of relentless eyes that caught my errors and made all my writing an order of magnitude better. I would also like to extend my gratitude to all of the friends and colleagues that helped me or supported throughout the period of my studies. I could not have gone this far alone.

Special thanks belong to Warhorse Studios and its director Martin Klíma for making this research possible by their openness to novel approaches and by letting me work in close cooperation with the company.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

Název práce: Snižování komplexity umělé inteligence ve hrách s otevřeným světem pomocí kombinace reaktivních a prohledávacích technik

Autor: Martin Černý

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedoucí doktorské práce: Mgr. Cyril Brom, Ph. D., Katedra softwaru a výuky informatiky

Abstrakt: Hry s otevřeným světem jsou žánrem počítačových her, který hráčům nabízí vysokou míru volnosti pro ovlivňování herního světa. Tato volnost znatelně komplikuje tvorbu umělé inteligence pro tento druh her. V této práci představujeme tři nové techniky, které umožňuji omezit různé druhy komplexity, které vyvstávají při implementaci umělé inteligence pro hry s otevřeným světem. Vyvinuli jsme tzv. behaviorální objekty („behavior objects“) jako nadstavbu nad často používanými chytrými objekty („smart objects“) objekty, navrhli jsme a implementovali metodu pro specifikaci chování z globálního pohledu založenou na splňování omezujících podmínek a ukázali jsme, že techniky prohledávání s protivníkem mohou nahradit složité reaktivní rozhodování v případech, kde je potřeba vzít v úvahu velké množství parametrů. Tyto obecné techniky byly implementovány a vyhodnoceny v prostředí kompletní hry Kingdom Come:Deliverance.

Klíčová slova: umělá inteligence, počítačové hry, výběr akce, reaktivní rozhodování, prohledávání

Title: Reducing Complexity of AI in Open-World Games by Combining Search-based and Reactive Techniques

Author: Martin Černý

Department / Institute: Department of Software and Computer Science Education

Supervisor of the doctoral thesis: Mgr. Cyril Brom, Ph. D., Department of Software and Computer Science Education

Abstract: Open-world computer games present the players with a large degree of freedom to interact with the virtual environment. The increased player freedom makes open-world games a challenging domain for artificial intelligence. In this thesis we present three novel techniques to handle various types of complexity inherent in developing artificial intelligence for open-world games. We developed behavior objects that extend the well-known concept of smart objects and help in structuring codebase for reactive reasoning, we propose and implement constraint satisfaction techniques to specify behavior from a global viewpoint and we have shown how adversarial search techniques can mitigate the need for complex reactive decision mechanisms when a large number of parameters has to be taken into account. The general techniques are implemented and evaluated in the context of a complete open-world game Kingdom Come: Deliverance.

Keywords: artificial intelligence, computer games, action selection, reactive reasoning, search

Contents

1 Introduction 5

1.1 Basic Game AI Concepts 6

1.2 AI Complexity 7

1.3 Goal Statement 11

2 General Analysis 13

2.1 AI Performance 13

2.2 Reactive Techniques in OWGs 14

2.3 Summary 14

3 General Related Work 15

3.1 Software Complexity 15

3.2 AI as an Engineering Problem 15

3.3 Summary 16

4 Evaluation Domain 17

5 Behavior Objects 20

5.1 Motivation for the General Case 20

5.2 Motivation for the Specific Case of KC:D 21

5.3 Related Work 22

5.3.1 Introduction to Smart Objects 23

5.3.2 Intelligent Environments in the Game Industry 23

5.3.3 Intelligent Environments in Academia 25

5.3.4 Other Approaches 26

5.3.5 Summary 27

5.4 Our Solution – Behavior Objects 27

5.4.1 Differences from OOP 28

5.4.2 BO Summary 31

5.5 Implementation 31

5.5.1 Requirements on the AI System 32

5.5.2 Smart Entities 33

5.5.3 SE: Smart Objects 35

5.5.4 SE: Navigation Smart Objects 36

5.5.5 SE: Smart Areas 36

5.5.6 SE: Quest Smart Objects 39

5.5.7 Situations 40

5.6 Evaluation 41
5.6.1 Summary of Evaluation in Previous Work 41
5.6.2 General Observations 42
5.6.3 Qualitative Feedback 43
5.7 Discussion 47
6 Constraint Programming for Global Specification of Behaviors 49
6.1 Motivation for the General Case 50
6.2 Motivation for the Specific Case of KC:D 51
6.3 Related Work 52
6.3.1 NPC Interactions in Crowd Simulation 52
6.3.2 Constraint Programming in Games 53
6.4 Our Solution – Global Specification 53
6.4.1 CSP Overview 54
6.5 Implementation 55
6.5.1 Our CSP Solvers 56
6.6 Evaluation 59
6.6.1 Quantitative Results 59
6.7 Discussion 64
7 Adversarial Search For Handling Large Rule Complexity 65
7.1 Motivation for the General Case 65
7.2 Motivation for the Specific Case of KC:D 67
7.3 Related Work 67
7.3.1 Goal-based Techniques in the Game Industry 67
7.3.2 Goal-based Techniques for Games in Academia 70
7.3.3 Learning Reactive Controllers for Games 70
7.4 Our Solution – Adversarial Search 71
7.4.1 Game Trees for Combat in OWGs 72
7.4.2 Alpha-Beta and its Variants 73
7.4.3 MCTS and its Variants 75
7.5 Implementation 77
7.5.1 Combat Model in Game 77
7.5.2 Combat Model for Search 80
7.5.3 Integrating the Model with the Game 86
7.5.4 Search Implementation 89
7.5.5 Comparing Algorithms 91

7.5.6 Finding Optimal Parameter Values 91
7.6 Evaluation 93
7.6.1 Evaluation by Tournaments in the Simulator 93
7.6.2 Evaluation by Tournaments in the Game 96
7.6.3 Evaluation with Human Users 97
7.7 Discussion 108
8 Discussion 110
8.1 Summary of the Main Contributions 110
8.2 Future Work 111
List of Author's Publications 112
Bibliography 114
List of Tables 129
List of Algorithms 131
List of Figures 132
List of Abbreviations 133
Appendix A – Digital Attachment Contents 134
Appendix B – Evolution Results for Adversarial Search Algorithms 135
Appendix C – Preregistration of the Human Evaluation 141

1 Introduction

A growing number of computer games advertise to feature a "large open world". No strict definition exists as whether a particular world can be considered "large" and "open" but one of the key properties is definitely freedom: In an ideal case, the player is constrained only by the physical laws of the virtual world – they may interact at any time with all the objects and characters in their surrounding and will always get a meaningful feedback from the environment. We will refer to this class of games as open-world games (OWGs).

Contemporary game worlds that are considered large feature a landscape of tens to hundreds of square kilometers. Recent and popular OWGs such as The Witcher 3: Wild Hunt (CD Projekt Red 2015), The Elder Scrolls V: Skyrim (Bethesda Game Studios 2011) or Read Dead Redemption (Rockstar Games 2010) are actually in the lower part of the spectrum because of gameplay considerations – travelling through the world should not take too much time. Such worlds are then populated with dozens of non-player characters (NPCs) that are part of the story of the game and possibly hundreds of NPCs as background cast.

This thesis deals with the development of behaviors for NPCs in OWGs, as there are both theoretical and practical unsolved challenges for applied AI. In general, the NPCs should be *believable*, that is, to appear and behave in a lifelike manner. Believable NPCs enable users to suspend their disbelief by providing a convincing portrayal of the personality the user expects (Loyall 1997). While the visual fidelity of NPCs in contemporary games is spectacular, this is often not the case for NPC behaviors. Limitations of NPC AI are exacerbated in OWGs in particular, since OWGs give the user a large degree of freedom and NPC behaviors thus have to maintain believability even when faced with unpredictable actions of the player.

In game development practice, the goal is to be perceived as intelligent and/or believable, but not necessarily to develop a cognitively plausible model producing the behavior. This is further supported by the fact that the way the AI is presented to the player often makes larger difference than the quality of the actual AI algorithm. This has been nicely demonstrated in (Denisova and Cairns 2015) where the authors show that merely telling people a game features "adaptive AI" improves perceived quality of the AI. There is also anecdotal evidence (Champandard 2007a) that simply increasing enemies' health makes them perceived as more intelligent.

The role of complex AI algorithms in OWGs is further diminished by the fact that in most OWG contexts, NPCs are not required to solve complicated logical problems or to perform a true long-term planning. Therefore almost any relevant behavior can be expressed with a reactive approach. Nevertheless, as game worlds grow and try to capture increasing variety of NPC behavior, increasing effort is required to develop reactive behaviors and the returns are diminishing. This has been partially addressed by both improving the reactive approaches in industry use (e.g., Isla 2005) and by introducing additional goal-based layer to ease authoring by handling the combinatorial explosion of the possible states of the world (Orkin 2006). There is however still large room for improvements.

Despite the overlap between academic AI and game AI, rational behavior as studied by classical AI may be of little advantage in games. It may even be undesirable – the NPCs' behavior should primarily match player's intuition about the world and intentions of the game designer, which may not necessarily align with what is optimal with respect to the actual game mechanics – the AI should not

exploit the mechanics against their spirit. It is also important to limit the intelligence of the NPCs so that the game difficulty is appropriate.

Overall, the initial premise of this thesis is that while intelligence is rarely the goal in itself, it is highly desirable to reduce the complexity and increase manageability of the AI code and thus help to express the design intentions behind the game with less effort. In this view, OWG AI techniques are primarily tools for the game designer and the development team. This thesis shows three new techniques that are promising to become powerful contributions to a designer's toolbox.

The techniques that we implemented advance the state of the art in the development of game AI with the primary aim to better handle the complexity of the game world.

1.1 Basic Game AI Concepts

To make the claims of this thesis explicit we first need to introduce some basic concepts relevant to game AI. In the literature on intelligent agents, the basic problem of agent's AI is called *action selection* (Russel and Norvig 2010) – deciding what to do next. We build upon this notion, but we need to extend it in several ways.

As the NPCs in OWGs need to deal with multiple different sets of tasks (e.g., following a daily routine versus combat behavior), the action selection for an NPC is usually divided into multiple *components*, each adjusted to the specifics of the tasks it deals with. We will assume that a mechanism to choose which component should control the NPC at a given is already provided – it may be, for example a variant of the subsumption architecture (Brooks 1986).

First, we need to distinguish between the implementation of action selection and its manifestation in the virtual environment. In this text, we will use the heavily overloaded term *behavior* to refer solely to the NPC's activity as perceived by the player.

On the implementation side, we will distinguish between an *action* and a *script*. In our view, action is a single command sent to the game engine, e.g., "play an animation", "move to a nearby location", "add an item to inventory", "change a property of the NPC", etc. In general, an action can be described by a single verb and has short duration. Script is then a procedurally defined sequence of actions achieving a meaningful task[1]. A script may issue multiple actions to the game engine during a prolonged period of time, reacting to immediate state of the world. To preserve reactivity, the scripts should generally be interruptible so that if a different script is selected to run, it may quickly replace the currently executing one. In most cases we will abstract from the actual language that the scripts are represented in.

In the following text, we will speak more broadly about *script selection,* which is defined separately per AI component. In this view, we are given a set of already implemented scripts and the goal of script selection is to choose the correct script to execute at the present moment. It is important to note that what constitutes a script in this view is dependent on the level of abstraction we work with. At the lowest level of abstraction, even a single action can be a script (e.g., "play talking animation"). At the highest level, scripts may correspond to very large chunks of NPC behavior (e.g.,

[1] Note that the term "script" is used in many different contexts and its exact meaning varies. The way we define script for the purpose of this thesis may not align with usages of the term in other works.

"perform complete work routine" or "aggressively attack nearest enemy"). Another possible view is that script selection functions are nested. Once first-level script selection functions are defined over low-level actions, we can build second-level selection functions which treat first-level selection functions as atomic scripts. Higher levels of nesting can be introduced by iterating this scheme as necessary. Our focus in this thesis is not fixed on a single level of abstraction. For each of the techniques we present, we assume that we are given a set of "building blocks" – scripts that have already been implemented, be it low level actions in the engine or complex selection functions of their own. Our task is then to build a new script selection function over those building blocks, regardless of the level of abstraction the building blocks represent.

Before delving into more detailed analysis, let us introduce additional game AI concepts that will be indispensable throughout the thesis. From architectural viewpoint, the NPC AI in an OWG may be divided into several main components, each performing script selection in a different context. As fighting enemies is still a major part of most contemporary games, *combat AI* is often the largest AI component. It may be further divided into *enemy AI* that guides NPCs opposing the player and *ally AI* that controls NPCs trying to help the player in a fight. *Non-combat AI* governs the rest of the NPC behavior. It may be further divided into *direct interactions* with the player (e.g., dialogues, barter, …) and *ambient AI* which covers the daily life of the NPCs and other actions they perform on their own. While the aforementioned components are present in the vast majority of OWGs, in a particular game some of the components may not be present at all, the individual components may be further subdivided or other components may be added to suit the needs of the game (e.g., a component that coordinates groups of NPCs). As the needs of various AI components are vastly different, we will always focus on a single AI component rather than on the NPC as a whole. Another reason to focus on components instead of NPCs is that some AI components may be challenging to develop as a whole even if the behavior of any individual NPC would be easy to create. These are the components where the difficulties arise from the interactions among NPCs or simply from the number of different NPCs that need to be covered.

From the implementation viewpoint, the AI techniques used to develop reasoning for NPCs in games can be divided into two large classes: *reactive* and *goal-based,* corresponding to "reflex agent" and "goal-based agent of (Russel and Norvig 2010) respectively. While no exact definitions exist, any mechanism that can be reasonably represented as a set of if-then rules is generally considered reactive, while mechanisms that explicitly represent NPC's goals and project the effects of NPC's decisions on the state of the world are considered goal-based. With the above classifications in mind, let us discuss two large categories of issues in behavior development that manifest to various extent in all AI components: performance and complexity. Performance is the one easier to grasp: script selection in OWGs needs to achieve amortized sub-millisecond performance and thus, goal-based reasoning is used scarcely as it tends to be computationally costly. The analysis of complexity, as a central point of this thesis, however deserves a closer look.

1.2 AI Complexity

This thesis aims to provide OWG developers with new techniques to reduce complexity of AI code. Focusing on any given level of abstraction, complexity of an

AI component corresponds to how difficult it is to realize the script selection in code. To turn this intuitive notion into a more precise working concept that would enable us to analyze complexity in more detail, we introduce a basic formalism for script selection.

For the purpose of this thesis we will define AI component as a set of *sensory inputs* (I), a set of *possible scripts* (S), a set of *script selection functions* (F), a set of NPCs that use the component (N) and a *behavior assignment function* (a). The behavior assignment function connects NPCs with script selection functions, formally, it is a function a: N → F. A single script selection function $f \in$ F, f: I → S represents the instantiated script selection mechanism for a particular NPC[2]. Importantly, we do not analyze AI at the level of individual NPCs but rather look at the whole component. This lets us capture the fact that elements of behaviors are often shared by multiple NPCs, which is important in practical development.

For the sake of our initial analysis, let us assume that every input $i \in$ I is a tuple of values of a fixed set of Boolean, integer and real-valued parameters and every $f \in$ F is realized by a decision tree where each node performs a test on one of the parameters. We further assume that all the trees have the minimal possible number of nodes that can realize the given function.

Now, in a very broad sense we can classify the complexity of the AI component along two axes: *size complexity* – the number of scripts the NPCs choose from (|S|) and *rule complexity* – the maximal number of non-leaf nodes of any of the decision trees ($\max\{|f|;\ f \in \mathrm{F}\}$).

As an example, let us analyze combat AI in a typical shooter game. Here the size of the component is small – there are only few broad types of scripts (attack, retreat, advance, cover an ally, …) each with a handful of concrete variants. The inputs consist of the NPCs health, available weapons and ammo and basic spatial information such as distance to the player and the availability of cover spots. With good design of the fighting mechanics and the levels, even a low rule-complexity AI (< 10 nodes in all script selection functions) can create great gameplay.

In contrast, governing an army of units in a real-time strategy game (RTS), has both large size complexity and large rule complexity. There is huge amount of combinations of allocating the individual units to particular tasks and the script selection function cannot be represented concisely, as witnessed by the difficulties in implementing a good RTS AI and the sheer amount of skills and knowledge human players need to play the game well.

We also see that as the number of input parameters that the NPC needs to take into account grows, the rule complexity necessarily increases. Even if each parameter is checked at most once on every path from the root of the tree to a leaf and a single parameter is relevant only for a small but fixed percentage of all paths from the root of the tree to a leaf (i.e. the tree is at least slightly balanced), the rule complexity grows exponentially with the number of parameters.

This view of complexity lets us categorize all AI components into four loosely defined groups:

[2] For simplicity, we have omitted memory of the NPCs from our analysis, but reading from memory can be easily modeled within this framework as just additional stimuli and writing to memory can be modeled through specialized actions within the scripts.

1) Small size complexity and small rule complexity: this is the situation for most contemporary shooter games. It is easy to create the scripts and the selection functions manually. This case may be considered trivial.
2) Large size complexity and small rule complexity: with growing size complexity, the sheer size of the codebase starts to be an issue. This is typical for ambient AI, where it is useful to have a lot of diverse behaviors, but the actual script selection is very straightforward. Although it is still possible to write all the selection functions manually, good structuring, reuse, and decoupling of the code is necessary for efficient development.
3) Small size complexity and large rule complexity: the individual script selection functions are too big to be efficiently written and maintained manually. Although NPCs in games seldom solve truly difficult problems, this case may arise simply because there are too many parameters the NPC needs to take into account, as in combat AI for games with complex fighting mechanics. The most common approach in the industry is to avoid this case by keeping the rule complexity low and sacrificing the quality of NPCs decisions. Planning-based techniques have also been employed in such cases with notable success (see Table 1). Note that planning and other search-based techniques require small size complexity to be feasible in real-time – small number of possible scripts translates into small branching factor of the search.
4) Large size complexity and large rule complexity: this case typically arises when there is a need to coordinate a group of NPCs in a task that would fall under case 3 if only single NPC participated. Since the scripts now consist of tuples of instructions for all NPCs, the size explodes. This case is not frequently present in contemporary OWG games (in part due to its inherent difficulty). Nevertheless, similar problems have been handled to various extent of success in different game genres, most prominently in both turn-based and real-time strategy games, using mix of search-based approaches and machine learning. An important difference to OWGs is that strategy games generally have lower graphical fidelity and the individual behaviors exhibited by units in strategy games tend to be significantly simpler than those in OWGs, resulting in both easier development of higher-level abstractions of the game and more computing time available for AI.

The categorization is summarized in Figure 1 and Table 1. In this thesis we focus on cases 2 and 3, as handling them to a full extent is beyond the state of the art in contemporary OWGs. Since case 4 is both vastly more difficult and does not naturally arise in OWGs, it is out of scope of this thesis.

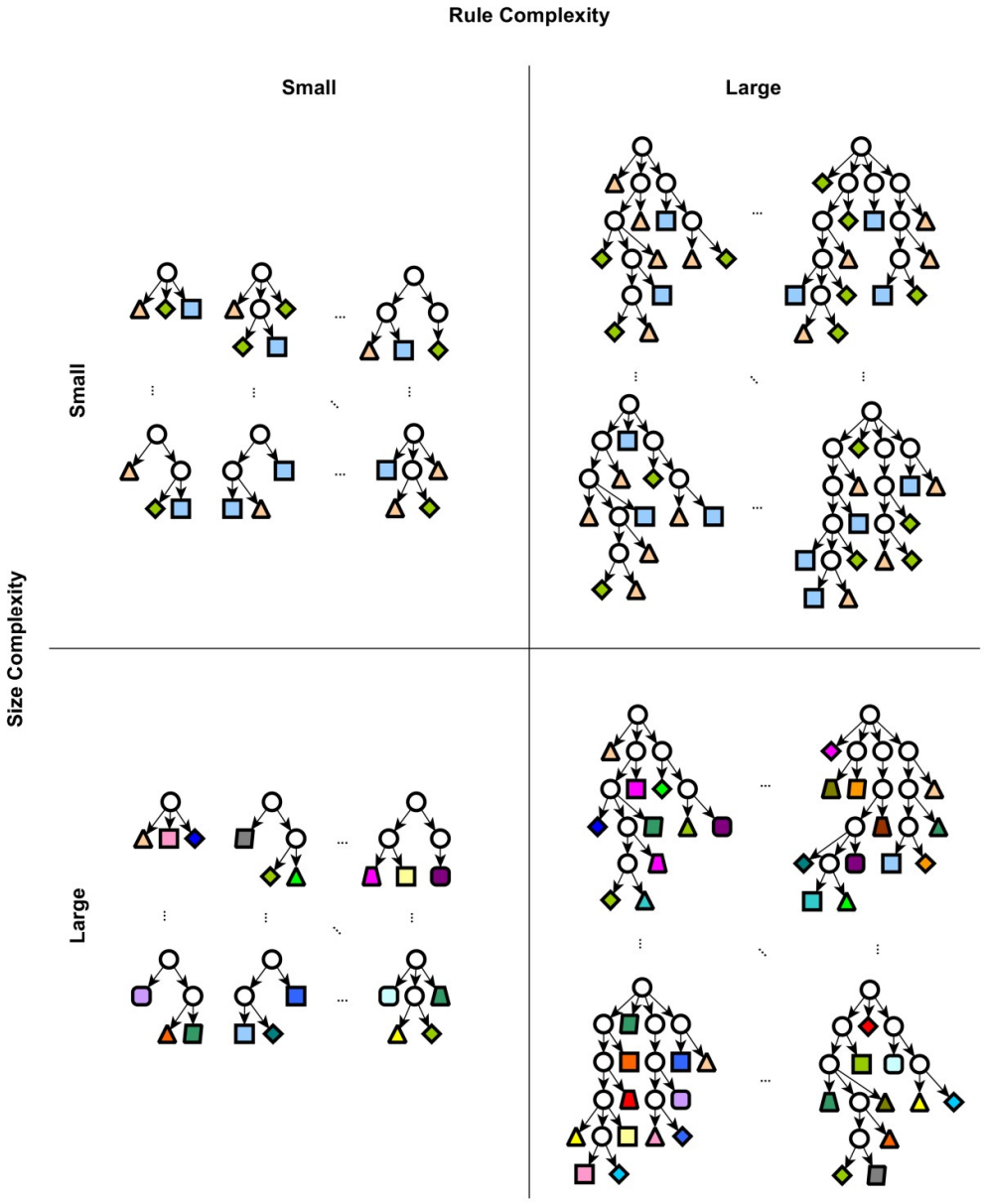


Figure 1: Diagrams of script selection functions AI of components with different types of complexity.
Here we assume that script selection functions are represented by decision trees. White circles represent internal nodes and colored shapes represent various scripts (same shape with the same color is the same script in all selection functions). Rule complexity reflects the number of internal nodes of the selection functions (left to right) while size complexity reflects the number of different scripts the component has to handle (top to bottom).

		Rule complexity: Small	Rule complexity: Large
Size complexity: Small		Basic enemy AI *Half-Life* *Halo 2*	Quality enemy AI *F.E.A.R.* *Killzone II* Ally AI *Bioshock: Infinite*
Size complexity: Large		Ambient AI *The Sims* *Elder Scrolls V: Skyrim* Dialog-handling AI	Strategic squad coordination *Fable Legends* *Frozen Synapse* High-level RTS AI *Planetary Annihilation*

Table 1: Two basic dimensions of OWG AI complexity.
For each type of complexity, the table provides examples of representative AI components and games that contributed significantly to the state of the art and/or provided inspiration for this work (italics). The highlighted areas are dealt with in this thesis.

1.3 Goal Statement

The goal of this thesis is to develop, implement and evaluate new techniques that can be used in AI components with either small rule complexity and large size complexity or large rule complexity and small size complexity. These techniques should enable creation of next generation OWG AI that would be impossible or impractical to achieve with state-of-the-art methods. In particular, there are three subgoals:

1. Implement and evaluate new ways to structure NPC-centric reactive script selection functions when the size complexity is large and rule complexity is small. To address this we devise and implement behavior objects as a parallel to object-oriented programming in game AI (Chapter 5).
2. Implement and evaluate a global approach to script selection for coordinated multi-NPC in-game events in components with large size complexity and small rule complexity. We address this subgoal by using constraint satisfaction techniques to select tuples of NPCs that will enact a designer-specified situation (Chapter 6).
3. Implement and evaluate script selection functions based on adversarial search for OWG AI components with large rule and small size complexity. We address this subgoal by testing AlphaBeta and Monte-Carlo Tree Search in the context of enemy AI for swordfighting scenarios (Chapter 7).

The rest of this thesis is structured as follows: we start with general analysis of requirements imposed on game AI (Chapter 2) and with discussion of related work common to all of the techniques we investigate (Chapter 3). Then we describe the

OWG that we evaluated all of our techniques in (Chapter 4) and present the individual techniques we have developed (Chapters 5 – 7). The thesis concludes with discussion of the overall contribution and considerations for future work (Chapter 8).

2 General Analysis

As we already noted in the introduction, OWG AI is better perceived as a tool that helps the game designers to achieve a desired effect on the player – either directly by helping the NPC to choose an appropriate action, or indirectly by making a desired behavior easier to develop. In both cases, AI development in OWGs is primarily a software engineering task – implementation and integration with the game is at least as important as the choice of the algorithm. The AI technique in question must align with the development lifecycle of the game – in particular it must be amenable to iterative improvements, fine-tuning, testing and debugging. It is further very useful if the way AI is expressed corresponds to the way designers think about the game. While these non-technical requirements are harder to evaluate, they cannot be omitted if the systems we develop are to be truly helpful in game development.

To better frame the technical aspects of our work, this section will give a more detailed attention to the performance requirements of OWGs and to the properties of contemporary formalisms for reactive AI.

2.1 AI Performance

In OWGs, CPU time is a very scarce resource as almost all of the CPU time is dedicated to graphics and physics. Time allowed for AI is usually less than 5 ms per frame (100 - 150 ms per second) on a single core for all NPCs together[3], including not only script selection, but also pathfinding and collision avoidance, which can be costly on its own – see for example (Berg et al. 2011). This means that computationally expensive techniques such as AI planning can be used only for one or very few NPCs at once. Due to this limitations, planning and similar goal-based techniques have been used only for combat AI (Champandard 2013), as only few NPCs are usually engaged in combat at the same time. For other AI components, and often even for combat AI, OWGs use reactive techniques.

We can even see developers abandoning already implemented goal-based approaches in favor of scripts. For example, developers of Just Cause 3 (released in 2015) switched to BTs from a planning approach they used in Just Cause 2, because they needed better performance and more designer control.[4]

In our previous work, we have examined planning approaches in the context of computer games, especially with regard to quality of decisions classical planning can provide and the computational requirements of planning (Černý et al. 2015). In that work, we took strictly the perspective of classical AI and measured how efficient an agent is in reaching its goal in an environment with small size complexity but large rule complexity. The agent also had a complete CPU core at its disposal, which is unrealistic amount of computing time for an OWG. Even in this limited context, the results indicate, that planning can be outperformed by simple reactive reasoning, unless the environment is very hostile to the agent or is relatively static.

[3] Based on our personal experience when cooperating with a game studio and discussions with developers at various conferences.

[4] As explained at time 2:08 of their video development diary dealing with the AI engine https://www.youtube.com/watch?v=GUEwOGo3BFI

We have further compared Hierarchical Task Networks (HTN) planning (Ghallab et al. 2004) to classical planning in game contexts (Černý and Gemrot 2013). While the results show that HTN planning has some benefits over classical planning, they manifest only when a huge amount of control has been delegated to a scripted layer below the planning algorithm, and the decisions the HTN makes are relatively simple and could be probably easily implemented in a reactive way. These are just further confirmations that reactive AI is often the best choice for computer games.

It is worth noting, that even reactive approaches often need to be heavily tuned for performance to satisfy the requirements of games (Canary and Champandard 2014; Mueller and Champandard 2015), further confirming that only very limited space remains for more expensive computations.

2.2 Reactive Techniques in OWGs

The state of the art in reactive OWG AI are – to our knowledge – variants of *behavior trees* (BTs) (Champandard 2007b). The common denominator of all BT approaches is that the script and/or script selection function is represented in a tree structure which is traversed for every update of the NPC to determine a leaf that should be executed. The leaves either directly represent actions in the game engine or contain short programs in a procedural language. The internal nodes then direct how the tree is traversed based on state of the nodes and input from the environment. Conceptually the nodes close to the root correspond to high-level decisions while nodes close to the leaves correspond to low-level decisions.

Other notable reactive techniques in use are finite state machines (Fu and Houlette 2004) and direct use of an interpreted procedural language like Lua (Schuytema and Manyen 2005). In all cases we are aware of, the limitations are similar to those of BTs.

An important benefit of reactive approaches is that they are easy to tweak and provide a lot of control to the game designers – handling a special case means simply adding a branch to the BT or a adding a new node to the state machine. Reactive techniques are also usually easy to understand and reactive scripts or script selection functions can be – with proper tooling support – created even by non-programmers. Ease of use and transparency is thus a vital consideration for OWG AI that all of the techniques we propose have to address.

2.3 Summary

The main difference between OWG AI and "classical" AI as researched in most of academia is that OWG AI is heavily constrained by both design and technical considerations that are not directly related to the AI algorithms themselves. The aim of this work therefore cannot be to solely propose new techniques, but to also devise how the techniques will be integrated in an actual game and how they will meet performance and design requirements that may arise during the development of a game.

3 General Related Work

In this chapter we deal with the research related to the general concept of handling complexity of software systems and engineering aspects of AI. Since the individual techniques we describe in this thesis come from different areas of AI, we defer the discussion of extant approaches to individual types of complexity to the chapters dealing with our particular solutions.

3.1 Software Complexity

In a broad sense, attempts to manage more and more complex programs have been the driving force of the development of modern programming languages and software design methodology. While object-oriented programming (OOP) remains the dominant paradigm, many orthogonal techniques have been combined with OOP and adopted by wider programmers' audience. Notable extensions include generic programming (Musser and Stepanov 1989), aspect-oriented programming (Kiczales et al. 2001) and reflection (Smith 1982).

All those specific techniques have been conceived to let developers realize some of the basic concepts of software design – decomposition, abstraction, modularity and separation of concerns. The tools we use however do not directly help us in choosing the right decompositions and abstractions in our programs. A basic methodology for choosing a good decomposition is *information hiding* (Parnas 1972) – decomposing system in such a way that each of its parts hides certain design decisions from the rest of the system. If the design decisions are properly hidden, changes to those decisions affect the system only locally.

The complexity-driven evolution of general-purpose programming languages and programming methodology is being mirrored by the evolution of tools to develop AI in games: game AI progress is also motivated by the necessity to handle more complex decision making. The most prominent example is the now-standard paradigm of behavior trees which stems from works such as (Bryson 2001) on trying to express complex behaviors in a concise way and (Isla 2005), where the authors tried to find a way to handle AI complexity in Halo 2.

We see the present work as a continuation of this effort to mitigate complexity and we will show how specific techniques let us hide and isolate design decisions that could otherwise propagate through large portions of the AI system.

3.2 AI as an Engineering Problem

The fact that an AI technique is *in principle* able to mitigate complexity does not automatically ensure it is useful in everyday development. In practice, AI techniques in games cannot be separated from the engineering and maintenance challenges they bring in. This consideration is seldom discussed in separation, and is mostly implicit in the way industrial game AI is deployed or talked about. The only resources known to us, focusing attention explicitly and systematically on the engineering side of game AI was (Champandard 2004) and an invited talk by Squirell Eiserloh at AIIDE 2014 (Carr et al. 2014).

Nevertheless, the engineering aspects of AI have been discussed at least as early as in the era of expert systems (Nilsson 1982). The idea that careful engineering for a

given task is of critical importance is also implicit in the influential works of Brooks – the subsumption architecture (Brooks 1986, 1991) and his arguments against the pure symbolic approach to AI (Brooks 1990). Closely related to NPC AI, the multi-agent systems community has also long recognized the necessity to handle the practicalities of everyday software development (Wooldridge and Jennings 1998).

In a similar vein, researchers from Google acknowledge, that only very small part of real-world machine learning systems are the actual learning algorithms discussed in the research literature – majority of the system is actually "glue code" connecting various algorithms, preprocessing data, etc. (Sculley et al. 2014). To alleviate the difficulties of everyday development, software engineers have created a set of design patterns – "simple and succinct solutions to commonly occurring design problems" (Gamma et al. 1994). While a similarly highly-accepted and generalized set of patterns is yet to be developed for AI, there were some attempts to formalize patterns occurring in specific areas of AI.

Design patterns have been introduced for autonomous agents (Kendall et al. 1998) or robots (Graves and Czarnecki 2000). Walsh (2003) proposes design patterns for constraint satisfaction problems. And, close to our work, Weber et al. (2010) propose "reactive planning idioms" – a set of patterns for designing reactive reasoning in NPCs.

3.3 Summary

Complexity is one of the driving forces of game AI development and there is a lot of inspiration that could be taken from the way general purpose programming languages evolved in response to increasing complexity. At the same time, OWG AI is to a large extent an engineering field that needs to be rooted in practice. Similarly to other fields of AI or software engineering, OWG AI can benefit from establishing patterns and methodologies for effective development.

In this line of thinking, we see the approaches we introduce as a set of design patterns for OWG AI and – to a smaller extent – for game AI in general. Further, we integrate all our techniques in a complete game, letting us to discover, describe and resolve the engineering challenges of applying the techniques in practice.

4 Evaluation Domain

In this work, we propose two techniques that apply to components with large size complexity and low rule complexity and one that applies to components with small rule complexity and low size complexity.

We implement the techniques in the context of an upcoming high-budget OWG Kingdom Come: Deliverance (KC:D) and evaluate them, both in terms of how they are perceived by humans (players or developers) and how much CPU time they require. From the software engineering perspective (as discussed in Section 3.1), we will ask which design decisions can be hidden using our techniques – i.e., which design decisions can be confined to affect only small parts of the AI system, making them easier to change later in development.

The AI system, that forms basis for our evaluation is described in (Plch et al. 2014). In this section we briefly introduce the system[5].

The basic NPC decision making in KC:D is performed by a variant of behavior trees (Champandard 2007b). In plain behavior trees (BTs), the script consists of a tree. The leaves of the tree are scripts and senses while the internal nodes (called *composites*) represent the main structure of the script selection function. Evaluation of a node may return three possible values: `success`, `failure` and `running`. Upon evaluation, scripts return `success` when the NPC has finished the action, `running` if more is to be done and `failure` if the script cannot complete. Senses test a condition in the world and succeed if the condition is true and fail otherwise. Composites are either *selectors* or *sequences*; both evaluate their children in order and when the evaluated child returns `running`, they also return `running`. Selectors return `success` when the first child node succeeds and do not evaluate the rest of the children. Sequences on the other hand need all of their children to succeed in order to return `success`.

In this scheme senses are typically the first children of sequences followed by single other node. Thus the node is evaluated if and only if the sense is active. Selectors then play the role of priority ordering on nodes below them and the highest priority subtree which does not fail is executed. See Figure 2 for an example.

This simple formalism allows for easy coding of quite complex behaviors and variations of BTs have become a de facto industry standard. The actions and senses are directly implemented by programmers in the game engine and thus are quick to evaluate. Another advantage is that subtrees may be easily reused among different scripts. Similar reactive planning approaches have been previously evaluated in academia (Bryson 2001). Further extensions to the formalism including *decorator* nodes (Champandard 2007c) and *parallel* nodes (Champandard 2007d) were proposed.

[5] The contents of this chapter is adapted from (Plch et al. 2014) and (Černý et al. 2016). The AI system was built by Tomáš Plch, Jakub Gemrot, Matěj Marko, Martin Štýs and others at Warhorse studios with only minor contributions from the author of this thesis and as such is not a part of this thesis.

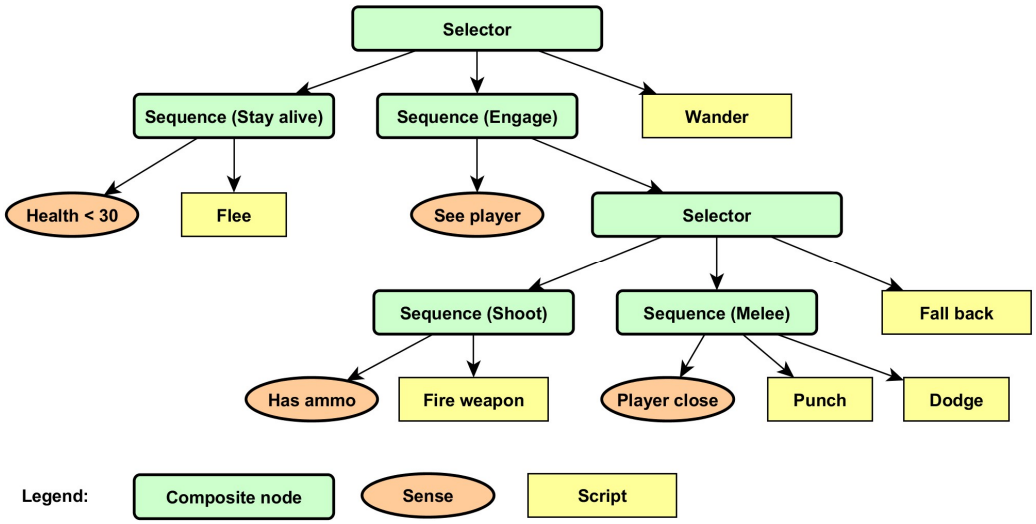


Figure 2: Example behavior tree representing a simple guard behavior for a shooter game. All children are ordered left to right. The first leaf to be evaluated is the "Health < 30" sense. If it succeeds, the "Flee" script will be started (it is in a sequence). If "Flee" returns `success` *or* `running`*, no more nodes will be evaluated. If either health is over 30 or the "Flee" script fails, the "See player" sense will be evaluated and if it succeeds, the selector over "Shoot" and "Melee" sequences will be evaluated etc. Note that a selector represents a prioritized list of alternatives and sequences a list of preconditions and actions that correspond to distinct parts of the script selection function. Sequence names (in parenthesis) have no semantic meaning – they are shown only to make the structure more readable.*

A decorator is a node that has a single child and performs no direct actions on its own, but alters how its child is evaluated. Examples include decorators that, upon evaluation, simply evaluate their child but alter the return value (e.g., turning `failure` into `success`) or decorators that evaluate their subtree only if a condition is met and return a fixed value otherwise.

A parallel node evaluates all its children every time it is evaluated. There are multiple possible ways to aggregate the return values of the children into the return value of the node, which are usually specified as parameters of the parallel node. Common settings include returning `success`/`failure` once a single child succeeds/fails, or running until all children succeed or fail and then returning `success` if all children succeeded and `failure` otherwise. As their name suggests parallel nodes are useful for issuing multiple actions in parallel (e.g., moving and directing gaze at the same time).

In KC:D, the BT formalism is further extended with variables and a custom type system which allows for complex structured types and type inheritance. The execution model of the BT nodes has also been extended to ensure consistent initialization and cleanup of subtrees; in particular the formalism allows a specific cleanup script to be executed when the main script is interrupted. The way a cleanup is expressed in the BT structure is shown in Figure 3.

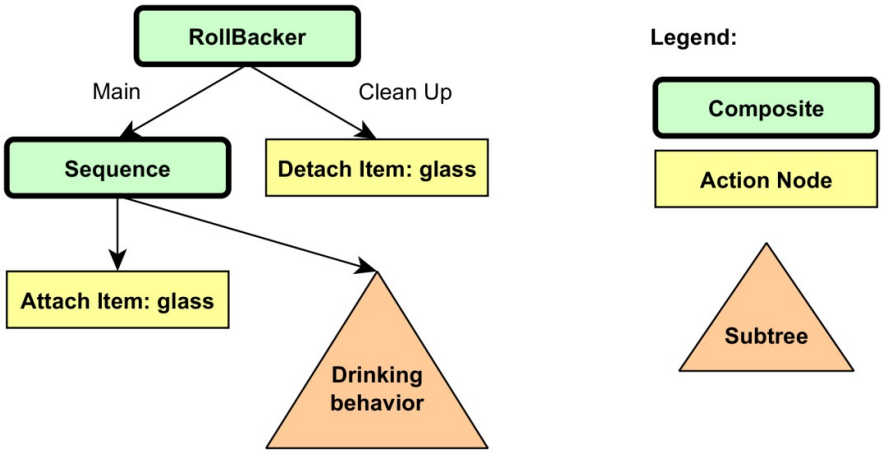


Figure 3: Script cleanup example.
The semantics of the RollBacker node ensure, that once the Main branch starts executing, the Clean Up branch is always executed to completion before control is yielded to another subtree.

The decision logic of every NPC in KC:D is represented as a hierarchy of *subbrains*. Subbrains represent individual components of the NPC logic (ambient AI, combat, quest-related logic, ...) ordered by priority and connected in a manner similar to the subsumption architecture (Brooks 1991). If the subbrain becomes active, it executes a BT associated with it. If a higher-priority subbrain tries to run, the BT of the lower-priority subbrain is stopped, including proper script cleanup. The subbrain priorities are fixed per NPC template and assigned by scripters. If more complex handling is desired, the scripters may create a special "switching" BT running in parallel with the subbrains and enforce activation/deactivation of specific subbrains through dedicated BT nodes.

Other important features of the AI system include a link database and a messaging system. The link database stores named links between game entities and can be queried at runtime with complex queries (e.g., return all objects with "child" link to an object linked to me with a "parent" link). The messaging system manages a list of inboxes for all NPC. Each inbox has its own message type. The system manages thread-safe message queues for each inbox.

Although we implement all the techniques in this thesis within KC:D and thus using BTs, all of the techniques we present generalize in a straightforward manner to AI systems with other architectures common in the game industry. We will discuss those generalizations in the individual chapters dealing with the techniques.

5 Behavior Objects

In this chapter we discuss the motivation, implementation, and evaluation of a new way to structure NPC-centric reactive script selection functions when the size complexity is large, as outlined in Subgoal 1 of this thesis (see Section 1.3). As a consequence, practical implementations will almost exclusively belong to AI components of small rule complexity, to keep the script selection functions amenable to a reactive representation. Table 2 shows our target complexity for this chapter in context.

The overarching theme of this chapter is that some of the lessons from object-oriented programming can be applied to game AI, resulting in what we call *behavior objects*. Behavior objects allow for better structuring of reactive code and thus let the development team to create better, more reliable behaviors in less time.[6]

The rest of this chapter is structured as follows: First, we discuss the motivation for the approach in general (Section 5.1) and for the implementation in KC:D (Section 5.2) followed by a survey of the related work (Section 5.3). We then describe the general concepts of behavior objects (Section 5.4) and dive into the specifics of the KC:D implementation of behavior objects (Section 5.5). The chapter concludes with evaluation of behavior objects in practice (Section 5.6) and a discussion of the results (Section 5.7).

5.1 Motivation for the General Case

In this chapter we focus on ambient AI – the AI component handling the behaviors NPCs perform on their own and that do not require direct interaction with the player. For ambient AI, size complexity is the most pressing issue – there is a large number of behaviors the NPCs may perform and although each individual NPC performs only a handful of scripts and the script selection function for any single NPC would be easy to implement, managing the codebase for a large number of NPCs becomes problematic. As noted in Chapter 1, ambient AI in contemporary games has to be implemented reactively, because of the large number of NPCs that need to be updated in a very short time frame. In practice, many OWGs implement ambient AI by letting NPCs either wander randomly around the game world or stay at a single place and loop an animation.

[6] Some of the systems described in this chapter were implemented by staff at Warhorse (especially Tomáš Plch, Matěj Marko, Martin Štýs) and thus their implementation cannot be considered a part of the thesis. To be specific, smart areas were initially designed and implemented by me and further developed by Warhorse staff. All types of smart objects were implemented by Warhorse staff with little direct input from myself, but based on the initial design and implementation of smart areas. Situations were fully developed by me. The unifying view presented in this chapter as well as all experiments, their interpretation and the text of this chapter are my contribution. This chapter is an adapted version of our paper on this topic (Černý et al. 2016).

		Rule complexity	
		Small	**Large**
Size complexity	**Small**	Basic enemy AI	Quality enemy AI Ally AI
	Large	**Ambient AI** Dialog-handling AI	Strategic squad coordination High-level RTS AI

Table 2: Complexity classification for the techniques in Chapter 5.
This chapter deals primarily with small rule complexity and large size complexity in general. The implementation in KC:D is focused on ambient AI in particular (highlighted). This table is a reiteration of the complexity classification shown in Table 1.

We perceive the complexity of the codebase required to handle varied ambient behaviors as the main obstacle to improving the state of the art. While truly lively and dynamic ambient behaviors could be, in principle, implemented with state of the art technology, the large amount of edge cases and possible interactions between various parts of the ambient AI make the script selection functions interdependent and hard to debug (e.g., what if innkeeper does not arrive to a pub, because a player blocked his way to the pub). Moreover, the improvement in player's experience achievable with better ambient AI is usually modest and may not justify the possibly very large expenses of maintaining the necessary script selection functions. As script selection for ambient AI is limited to reactive techniques for performance reasons, the problem cannot be overcome by planning or similar methods. Improving the way reactive script selection functions are expressed is thus a key to progress in this area.

It is of little surprise that hierarchical decomposition and code reuse are the basic ingredients for a good formalism for expressing script selection functions. To an extent, the contemporary reactive techniques – especially BTs – were made with exactly these goals in mind. But current OWGs seem to stretch state-of-the-art techniques to their practical limits and a new layer of abstraction needs to be placed on top of them to move forward. This is our goal in this chapter.

5.2 Motivation for the Specific Case of KC:D

The general points given in the preceding section arose very concretely during the work on AI system for KC:D. Initially, an augmented variant of BTs was implemented (as described in Section 4) to help in creating lively ambient AI, but a desired level of fidelity of ambient behaviors was still very challenging to achieve, since it was hard to separate individual parts of the code realizing ambient AI from

each other (e.g., work behaviors and sleep behaviors). Therefore new scripting techniques were needed to make the BT codebase manageable. We wanted the new techniques to let the scripters gradually decompose the behaviors into simple elements that can be designed, implemented and tested separately. In many ways, it was the basic software engineering problem: how to structure the AI code to get the best results for a given amount of effort.

The design team of KC:D introduced certain objectives the new technique should fulfill beyond the objectives of BOs in general:

- **O1** Strong guarantees must be made that gameplay-critical behaviors (quests, combat, …) will not be disrupted.
- **O2** The scripts must be interruptible and maintain consistency even on prolonged execution. The game is expected to be played for several dozen hours while all the NPCs are continually simulated, without any reset. In contrast to most contemporary OWGs, the design team of KC:D required that all NPCs are simulated even when the player is not in their proximity.
- **O3** Primary use-case is the ambient AI.
- **O4** The behavior code has to be decoupled from the data in the game world. In particular, using an already defined script in a new context (e.g., adding a new pub to the game world) should be possible without changing any of the code of NPCs that use the script and without modification to the pub logic.
- **O5** Some NPCs should be allowed to behave differently in the same context: e.g., in a pub, rich people behave (and are treated) differently than poor people.
- **O6** Coordination of joint behaviors among agents (a pub brawl, a game of cards …) must be supported.

To an extent, all of these additional objectives can be achieved with state-of-the-art reactive script selection functions, but at the cost of reduced code manageability. This is an exact parallel to traditional programming, where everything that can be implemented using an object-oriented language can be, in principle, written with a language that only supports structured programming or directly in the assembly language. Nevertheless, OOP has great merits for practical development. In a similar vein, our aim is to meet the design objectives while increasing code manageability and promote information hiding to prevent most future changes from affecting large parts of the codebase.

5.3 Related Work

In both academia and industry, a prominent approach to managing behavioral complexity (including, but not limited to ambient AI) is embedding intelligence in the environment. We will start this section by introducing smart objects as the oldest and most widely used example of this approach. Further we will review how smart objects and other elements of intelligent environments have been used in the game industry and academia respectively. We conclude this section with a discussion of other approaches relevant to our work.

Note that most of the approaches we will discuss here are not aimed directly at ambient AI, but rather at behavior development in general.

5.3.1 Introduction to Smart Objects

Smart objects as introduced by (Kallmann and Thalmann 2002) are, to our knowledge, well-established in the game industry, although in a very simplified form. A smart object as used by the industry is typically a graphical entity in the game world that is accompanied by a character animation (or several animations) that should be used when a character desires to use the object. The smart object is also responsible for positioning the character at the exact spot where the animation should be played.

A typical example of a smart object is a lever on the wall. An NPC that wants to change the state of the lever simply fires a "use smart object" action and the smart object takes care of the necessary details. This way, many different levers and switches may be present in the environment, but the AI only needs one action to use them all properly. In other words, smart objects hide animation details from the NPC's script.

Another frequent use are so-called *navigation smart objects*, which are smart objects intended solely for navigating around the environment. A navigation smart object connects a graphical entity in the game world with an entry and an exit point. When an NPC wants to move from the entry point to the exit point, it plays the animation associated with the smart object. This keeps the details of movement hidden from the NPC's script. Navigation smart objects typically connect disjoint areas that could not be traversed by regular navigation – a typical example is jumping over a barrier. Another approach to handle barriers and doors is to embed additional navigation information inside edges of the navigation graph (Reed and Geisler 2004). While this is conceptually similar, it is less flexible as the navigation graph needs to be manually kept consistent with the environment.

Conceptually, smart objects are inspired by the psychological notion of *affordances* (Gibson 1986). The idea is that animals (and humans) do not perceive the environment as physical objects but rather as a set of possibilities the environment affords, e.g., a door is something that may be opened, lever is something to be pulled, barrier is something to jump over, etc.

Kallmann originally proposed smart objects as more complex entities that can provide multiple interacting parts, each with its own location, mechanics, instructions for NPC positioning and optionally also with code the NPC should run to use the given part. Kallmann's smart objects could also run code on their own to alter their internal state. Kallmann's idea is close to our goal, although it misses several important features. Most notably Kallmann does not consider interrupts to the scripts (O2), and it does not support script nesting (important for O4).

5.3.2 Intelligent Environments in the Game Industry

A version of smart objects close to the Kallmann's version has been implemented in The Sims series (Ingebretson and Rebuschatis 2014). The Sims form a very different application than OWGs, because the user is not embodied in the environment and interactions with NPCs and objects are triggered indirectly. The NPCs autonomous decision making consists of selecting an appropriate smart object (NPCs are also smart objects) to satisfy its current needs. The basic units of scripts in The Sims 4 are called "interactions" (e.g., sit down). An interaction consists of animations and changes to state of the NPCs and/or the world (e.g., NPC is now in "seated" pose, chair is occupied). These interactions are then connected to objects in the game (e.g., the same sit down interaction is connected with a chair and a bench). The interaction

may further decompose into atomic "blocks". Those blocks are not interruptible and are always run to completion, but blocks of multiple interactions may be interleaved (e.g., sip a drink – look at TV, cheer – finish the drink) and nested (e.g., cuddling while sitting on a sofa). Thus in The Sims, all objects have the same interface for the NPC and their differences are hidden from the NPC logic.

Our work is the first that we are aware of that implements comparably powerful (complex behaviors, behavior nesting …) smart objects in a game where the user may interact directly with the NPCs, which requires a different approach to O2.

Although our experience indicates that smart objects are used in many first/third person games, there are – as is often the case with game industry – relatively few official sources and very little detail revealed. Among those, smart objects are mentioned in the context of FarCry 4 (Isla 2014), Castlevania: Lord of Shadows (Parera 2013) or F.E.A.R. 2 (Champandard 2011). BioShock:Infinite also has "opportunities" placed in the environment for the sidekick character Elizabeth to interact with.[7] From the little information available, all these implementations do not seem to go much beyond levers and other simple objects and are therefore not suitable for improving manageability in large-scale OWGs.

Notably the S.T.A.L.K.E.R. series extended smart objects to "smart terrains" that provide more long-term behaviors to all NPCs in a specific area (Iassenev 2008). In more recent developments de Sevin et al. (2015) propose "smart zones". Smart zones are a collection of roles NPCs may perform at a given location. Upon arrival to the zone, the NPC is assigned a role and starts performing it. The problem with both of these approaches is that the smart zone/terrain completely takes over the NPC, bypassing any high-level reasoning on the NPCs side. Thus the internals of the smart zone/terrain scripts are not completely hidden to the NPC script, as it needs to consider, whether the smart zone/terrain script may interfere with high-priority behaviors of the NPC. Also, there is no mention on how interruptions to the scripts are handled, so it is unclear, whether interruptions are handled at all. Therefore it is likely to be unsuitable for implementing behaviors of long-lived NPCs with a large palette of possible behaviors.

In a slightly different vein, Skubch (2015) uses "smart locations", which are a collection of assets together with a shared blackboard. Behaviors are then implemented as rules that are triggered by a matching set of facts on the blackboard. These rules then either start scripts for the given NPCs and/or manipulate the contents of the blackboard. Upon arrival of an NPC to the smart location, special rules assign the NPC a role (e.g., a waiter or a guest in a restaurant), and the role is then part of the preconditions of all rules handling the particular role. The advantages of this approach is that proper handling of various amounts of NPCs at the smart location is handled emergently and that such a rule based formalism allows for offline validation of basic properties with a planner (rules that cannot fire, blackboard states that are inconsistent). The downside is that truly complex behaviors are harder to develop as there is no explicit hierarchical structuring of the rules and it is easy to introduce unexpected dependencies into the rule set. Interruptions of scripts are also handled in an ad-hoc manner. The approach worked great for Final Fantasy XV, as the NPCs there are short-lived and the locations are reset frequently, but is not suitable for long-lived NPCs in OWGs.

[7] See an interview with the developers at http://www.youtube.com/watch?v=2viudg2jsE8

Another interesting approach is presented in Hitman: Absolution. Here, the AI uses objects called "situations" to coordinate multiple NPCs (Vehkala 2012). Whenever an NPC deals with an event that requires coordination with others (e.g., the player is trespassing and should be stopped), it subscribes to a corresponding *situation object*. The situation object assigns roles to the subscribed NPCs and alters their knowledge based on events in the game world (e.g., tells the NPC that it is in a trespassing situation, who is the leader of the situation and how aggressively should the NPC react). The NPCs then take that knowledge into account in their own decision making. The drawback of this approach is that every NPC needs to include specific code for every situation it may participate in and thus the situations are not hidden from the main NPC logic. Furthermore, the code for the situation is scattered among multiple NPCs making the technique impractical for large OWG codebases.

With regards to the major available game engines, CryEngine Free SDK seems to have the best support for embedding intelligence in the environment (Crytek 2013). In CryEngine Free SDK a "smart object rule" can be assigned to any entity in the game. The rule consists of a condition and an AI script to be executed, when the condition is met. This approach allows for simple creation of a wide variety of active non-character entities (e.g., landmines, machines), but the approach is more problematic when implementing behaviors for NPCs as the script within the rule executes in parallel with the NPC's logic. In all but the simplest situations the script within the rule would have to manually synchronize/communicate with the NPC's logic to prevent the rule from threatening the consistency of the NPC's state or from interrupting a gameplay-critical behavior, introducing unnecessary coupling of the respective code. There is also no support for communication when multiple NPCs use the same object. In general the "smart object rules" of CryEngine Free SDK are great tools for what CryEngine was intended for – quick action in first-person shooters – but they are not very suitable for ambient AI in complex persistent worlds.

Unity3D (Unity Technologies 2016) and Unreal Engine (Epic Inc. 2016) have no built-in support for embedding intelligence into the environment, although there are AI middleware solutions that provide some support. One example is Autodesk Gameware Navigation[8] that includes support for navigation smart objects (as described above).

5.3.3 Intelligent Environments in Academia

In academia, the concept of smart objects has quickly been extended by crowd simulation research to whole areas. In (Tecchia et al. 2001) the environment is overlaid with a grid, where each cell may dictate a movement behavior for the agents in it. In (Sung et al. 2004) "situation based behavior selection" is presented. The system detects situations in the environment and instructs the agents participating in the situation what should they do. While situation based behavior selection is very general, the situations tested in the paper are mostly triggered by entering a location. In both of those works, the environment is the sole source of behaviors and the behaviors cannot be further decomposed, preventing the implementation of more complex behaviors.

Stocker et al. (Stocker et al. 2010) introduced "Smart Events" – specific objects providing NPCs with ready-made responses to external events. An important problem is that Smart Events provide the same behaviors for all types of characters

[8] http://gameware.autodesk.com/navigation

(conflicting with O5) and do not provide means for coordination among the characters (O6).

The simulation of Shao and Terzopoulos (Shao and Terzopoulos 2005) features autonomous pedestrians in a virtual railway station. Several social behaviors (e.g., buying tickets, spectating an art show) with coordination (e.g., queue at the ticket booth) mediated by specialized environment objects are introduced. However, every character must be explicitly prepared for all the social behaviors it may perform, limiting scalability to OWGs.

Further, the crowd simulation approach cannot be directly translated to OWGs as crowd simulation is intended to be believable from a larger perspective, but does not necessarily retain believability when individual characters are tracked. To quote (Sung et al. 2004): "When we look at a crowd, we care only about what is happening, not who is doing it". In their work this let them adopt a simple probabilistic behavior selection model, which is unsuitable for computer games as it allows a character to, for instance go, to work twice without a break.

Brom et al. (2006) take the idea of smart objects further with "smart materializations". In their work the world is inhabited by agents using the belief-desire-intention (BDI) architecture (Bratman 1987). The only way to act on intentions is to choose a smart materialization which is a script fragment embedded in the environment. The smart materialization may in turn introduce subintentions, which are again resolved in the same manner. For example the character may adopt a "have fun" intention. A pub in the environment would provide a materialization that realizes the "have fun" intention by instructing the agent to go to the pub and adopt subintentions "buy a beer" and "drink a beer". A simple scheme to choose the best materialization among those that achieve the same intention is implemented. This work has provided substantial inspiration for us. While smart materializations have many of the desired properties, they lack the possibility to create scripts or their fragments without any materialization. Also BDI architecture is seldom used as a game AI architecture, probably because it is relatively complex and not well known to the developers.

5.3.4 Other Approaches

Orthogonally to embedding intelligence in the environment (or outside of NPCs in general), Bryson (Bryson 2001) advocated using object-oriented programming principles in behavior design. In her view, every capability of an agent should be represented as an object. Bryson's approach however has no explicit support for agent coordination neither does she outline the use of objects of a finer granularity.

A different approach to modularizing behaviors is provided by ScriptEase (McNaughton et al. 2004). ScriptEase lets users create scripts by using "generative design patterns" which are essentially parameterized code generators. This approach allows for great flexibility as the scripter can make low-level modifications to the generated code but does not account for hierarchical decomposition of complex behaviors.

In a similar vein, (Zhao and Szafron 2014) shows a framework for generating cyclic schedules for ambient AI offline. This approach alleviates the problem of size complexity, as a large codebase is generated from a much smaller (and thus manageable) number of declarative constraints. It however does so at the expense of some runtime flexibility, as the general schedules for NPCs have to be prepared offline and thus cannot directly react to events in the game world.

5.3.5 Summary

While most of the works mentioned in the preceding sections follow the same general direction as we aim for (managing complex code, encapsulation and information hiding), they either do not scale to larger use cases or to highly complex behaviors or they lack support for some specific features we require. Most frequently, the techniques assume the behaviors to be uninterruptible or do not provide any consistency guarantees upon interruption (O2). Most of the techniques also have limited support for coordination and communication (O6).

5.4 Our Solution – Behavior Objects

Behavior objects (BOs) are our attempt at combining the benefits of object-oriented programming (OOP) with the benefits of smart objects and other intelligent environment elements. BOs aspire to be a parallel to OOP in the behavior context and allow scripters to handle complex codebases while maintaining consistency and providing robust support for cooperation and coordination as well as the rest of the objectives O1 - O6.

Let us start by making the parallel to OOP explicit: Objects in OOP consist of code (methods) and data (fields). The code is defined once for a class of objects, while data are specific to object instances. When a method is invoked on an object, it manipulates the object's data to provide a desired result.

BOs consist of code (scripts), data and central decision logic which we call the *brain*. Code and brain is defined in a BO *template*, the data is specific to a BO *instance* (this addresses the objective O4). When a script selection function of an NPC invokes a script provided by a BO, it executes the script in its own context and lets it access the NPC's internal state. The NPC becomes a *holder* of the script. The script however still has access to the BO instance data which provide further context for execution and provide an implicit communication channel to other NPCs using a script from the same BO instance (addressing objective O6). The brain (if present) manages the individual scripts and may actively influence their execution, either by manipulating the BO instance data or by explicit communication with NPCs holding BO's scripts. The BO instance data come in two very different flavors: *environment data*, which are links to entities in the game world, and *state*, which is internal to the object.

A simple example of a BO is a chair with a "sit" script – here the environment data consists only of the chair; the object state is a flag indicating whether the chair is in use, and the script consists of three animations (sitting down; idle while sitting; standing up). The chair has no brain. A complex example is a BO that manages a pub. It contains scripts for guests, the innkeeper and the waitress. The environment data consist of links to chair BO instances (as above) inside the pub and the area the pub covers; the state is a list of orders for drinks. See Figure 4 for a diagram of the situation.

The brain of the pub BO handles requests for seats and drinks and sends messages to guests to inform them where to sit and to the innkeeper and the waitress to instruct them to prepare and deliver drinks respectively. This makes the pub a central point of communication, which allows multiple waitresses/innkeepers to be added to the pub without changing the code. Since the code for guests, the innkeeper and waitress is all in the BO, the development of the communication protocols for seat and order management is greatly simplified as all its uses are from within the BO. To use OOP

terminology, the communications are private to the pub object, hidden from other scripts. Most of the rules of thumb used in object-oriented analysis can be easily translated to the behavior case to help design a good decomposition of behaviors into BOs.

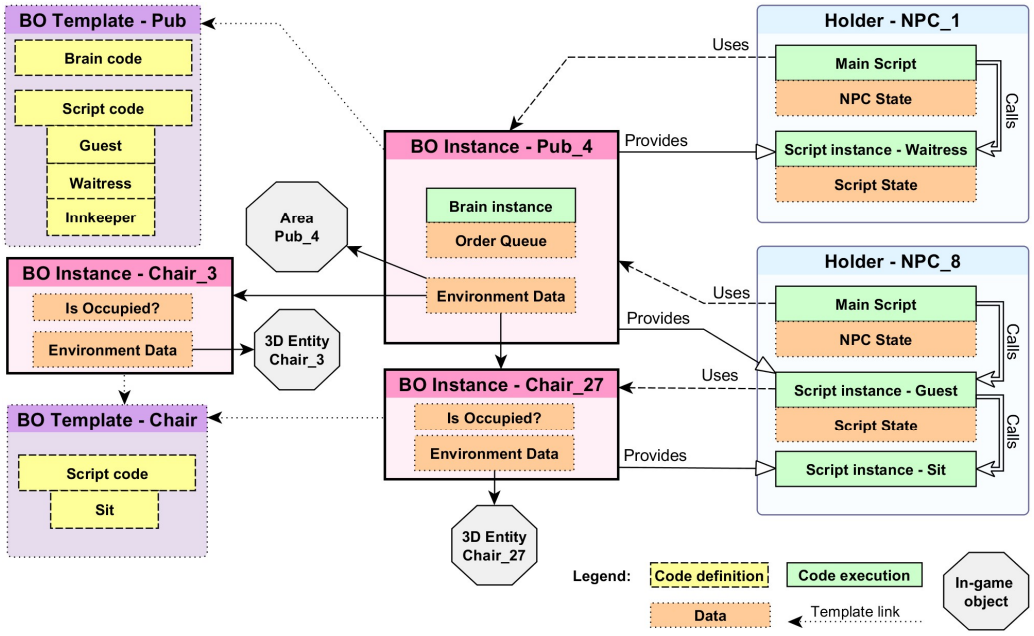


Figure 4: An example usage of behavior objects: a pub with multiple chairs.
The code for individual scripts and central decision logic is provided in BO templates (purple, dotted), that are shared by multiple instances (pink, solid). The instances execute code and encapsulate state and environment data – links to in-game entities and other BOs. NPC_1 represents a waitress – it uses "Waitress" script provided by the pub and manages data used by the script. NPC_8 is a guest in the pub – it uses the "Guest" script provided by the pub. On a finer level of abstraction, the "Guest" script acts as another script selection function which further uses "Sit" script provided by a chair.

5.4.1 Differences from OOP

While OOP is a very general methodology, we have found three notable issues specific to behavior development that preclude direct application of any of the classical implementations of OOP. These issues motivate the main differences between BOs and OOP. First, the state of an NPC is implicitly shared by all scripts the NPC may execute, which complicates encapsulation. Second, scripts representing behaviors have different execution model than programs which needs to be taken into account. Third, execution of scripts is highly parallel – all NPCs and BO brains act like separate threads. This section details these three issues and how we addressed them with BOs.

5.4.1.1 Shared State

The fact that the NPC's state (position, speed, active animation) is shared by multiple scripts makes encapsulation of code more difficult – the shared state makes the currently executing script implicitly dependent on the scripts that were executed previously. A general principle followed by BOs to minimizes issues caused by

sharing state between multiple scripts is to require all scripts to terminate only when all the changes to NPC's state have been fully completed or rolled back and to require all scripts to check the NPC's state anew when they resume execution after an interrupt. This is well supported by the underlying BT formalism.

The reality of the game engine has forced us to make an important exception to the above principle: Since a script may require an extensive computation or data exchange with other NPCs/BOs to determine the next action, the system cannot guarantee that a new script will issue an action on the same frame in which the old script has ended. This would result in movement and animation artifacts where the NPC stays still for one or two frames during a script transition. To remedy this, a script should terminate before its last movement and/or animation action completes. Every script is then required to issue an animation and/or a movement action (or force the NPC to stop) at the beginning of its execution. This way, the transitions are instantaneous and the animation subsystem can take both new and old animations into account when choosing an appropriate transition animation. As almost all scripts start with movement or animation anyway, this approach required very little modification to behavior code and worked reasonably well in practice.

5.4.1.2 Execution Model of Scripts

In OOP, executing a method results in a full change of context – the methods lower on the stack do not influence the execution of the active method. However, inactivating the script selection function of the NPC while a BO script is running is not a viable option for OWGs as the BO script would then need to be able to react to high-priority external events (e.g., combat) on its own, reducing modularity of the code. Thus, in BOs, scripts are *injected*: using a script of a BO keeps the parent script selection function active. In the context of a BT-driven system, using BO results in inserting a new subtree in the BT that drives the NPC and thus the parent tree still influences execution. This becomes crucial, if the injections are nested. In particular a node closer to the root may switch to a different child and terminate the injected script – see Figure 5 for an example. The script may also be injected at a well-defined place in the NPC decision making that is not the part of the main BT. For example the NPC architecture may allow asynchronous execution of a small "event" tree whenever the NPC is damaged. The script then may be injected as this event tree.

We have considered two variants of script injection: Either it is *on-request* – NPC script selection actively requests a script from a BO and is in direct control of the injection (as in the example in Figure 5), or *on-command* – a script is imperatively injected into NPC's script structure based on conditions external to the NPC (e.g., injecting code to handle a combat event). Nevertheless, even the on-command injection still keeps the top-level script selection in-place and it is the top-level script selection which decides when and if at all the injected code is executed (this helps to maintain consistency – objective O2). For most of the use-cases in KC:D, the on-request method is more appropriate: it is the NPC that decides to perform a work behavior and the actual request for the script should be made at the time of such decision. However, some use cases require on-command injection as well.

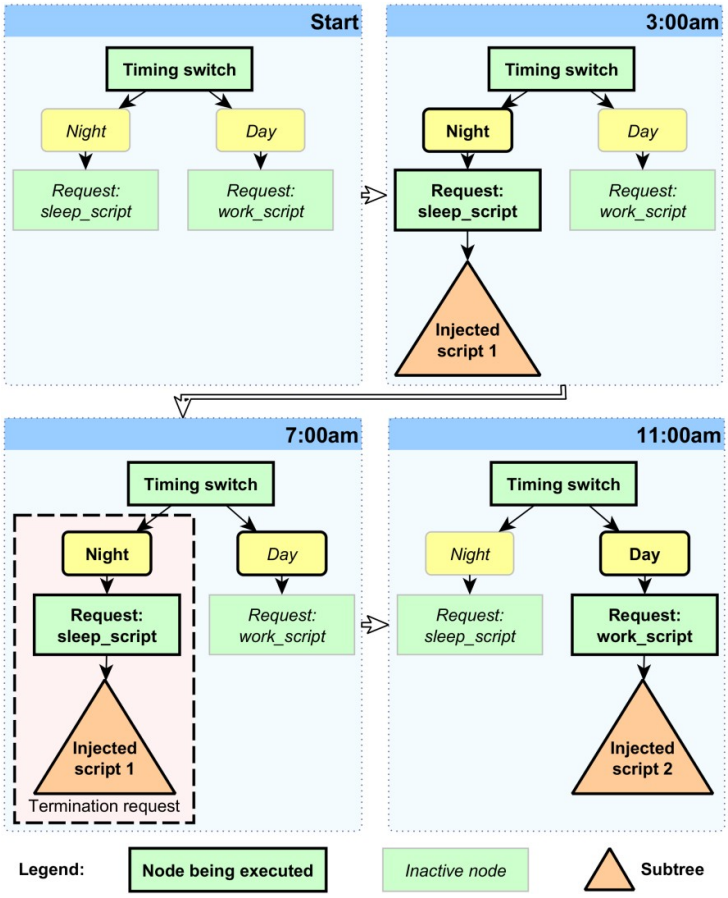


Figure 5: Injecting script into an NPC's main script.
In this case, it is the NPC that actively requests a script of a given type to be injected. Since the higher-priority nodes are still being evaluated when performing the injected script, the high-level NPC decision making may terminate the injected script when necessary. A similar logic handles the ambient AI of NPCs in KC:D.

Note that the injection principle can be applied to formalisms other than BTs. For example, when finite-state machines (FSMs) (Fu and Houlette 2004) are used, a special "use behavior object" state may be expanded to a new FSM prior to transitioning to that state. For a belief-desire-intention architecture, methods to act on certain intentions may be injected, etc. However, BTs naturally provide a very clean support for decomposition and hierarchical structuring of the code which aligns nicely with the BO approach.

5.4.1.3 Parallel Execution

While OOP languages provide mechanisms to handle parallelism, BOs differ in the scale of the problem – effectively every NPC and BO instance acts as a separate thread, and thus parallelism has to be accounted for at the architecture level and not ad-hoc at the code level.

The main problem arising from the parallel nature of game AI is safe and consistent data access and sharing required for coordinated behavior (O6). While it is safe to directly access immutable data belonging to a different thread, access to mutable data needs to be more careful to avoid race conditions. There are multiple solutions to this problem in OOP, but we consider it best to make each thread (NPC,

BO brain) solely responsible for its mutable data. Mutable data of other threads can be accessed only indirectly by sending messages to a) request data from another thread, b) provide data to another thread or c) request a change of data belonging to another thread. The receiver then handles those messages within its own updates. Using the message system as the sole mechanism for sharing mutable data is in most cases sufficient to ensure that the scripts are robust to any possible interleaving with other scripts. In most cases, BO's environment data is immutable at runtime and thus may be directly referenced from anywhere. Internal state on the other hand is almost always mutable and thus cannot be referenced directly from other threads. Since an injected script is executed within the main script selection of an NPC, it has full access to the NPC's internal state but the state of the injected script cannot be directly referenced by the BO's brain and vice versa (see Figure 6).

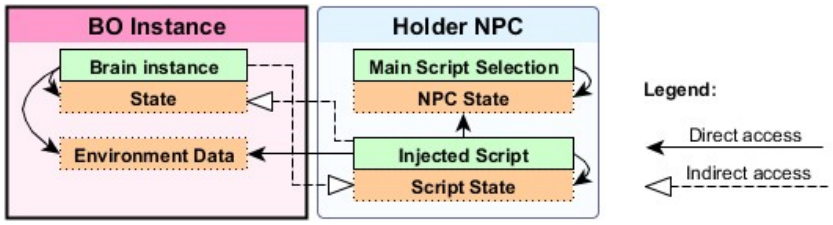


Figure 6: Possible data access between a BO instance and a holder.
In direct access the data may be explicitly referenced, while indirect access requires sending messages to read/write data.

5.4.2 BO Summary

While BOs derive most of their usefulness from the concepts of OOP, they have additional functionality and coding rules specific to behavior development: BOs require all BO scripts to explicitly handle interrupts and always leave the NPC in a well-defined state – even at the cost of not terminating the script immediately after an interrupt. BOs have specific execution model that keeps higher-level logic in place while executing low-level behavior. BOs support parallelism at the architectural level and define clear rules for data access to prevent the most common concurrency issues.

Like OOP, the BO approach is not a silver bullet to solve all behavior design problems, but it has the potential to mitigate complexity and enable scripters to create more lifelike behaviors within a given timeframe.

5.5 Implementation

In this section we describe the implementation of BOs for KC:D. The most common type of BOs in use are smart entities: BOs that embed intelligence into the environment. To both test the versatility of the BO approach and to meet requirements of the design team, four kinds of smart entities were developed: smart objects, navigation smart objects, smart areas and quest smart objects. We also created situations as a very different type of BOs. The motivations for the individual types of BOs are provided along their implementation details in the following sections. Figure 7 shows the relationships between types of BOs that were implemented. As the AI system is primarily based on BTs, BTs are used for both script selection (at multiple levels of abstraction) and for the low level scripts.

Unfortunately, we cannot publish the code for the BO implementation described in this chapter, as it is tightly bound to the underlying proprietary game engine. In addition, the usefulness of the code would be limited, since the implementation of the BOs themselves is straightforward and most of the code relevant to BOs is dedicated to integration of BOs with the virtual world, which is highly case-specific. Furthermore, BOs are mainly a paradigm for behavior development rather than a tool for a specific use case and we consider the paradigm and its evaluation in a full-fledged OWG the contribution of this thesis. The digital attachment however contains several videos, showcasing BOs in the complete game.

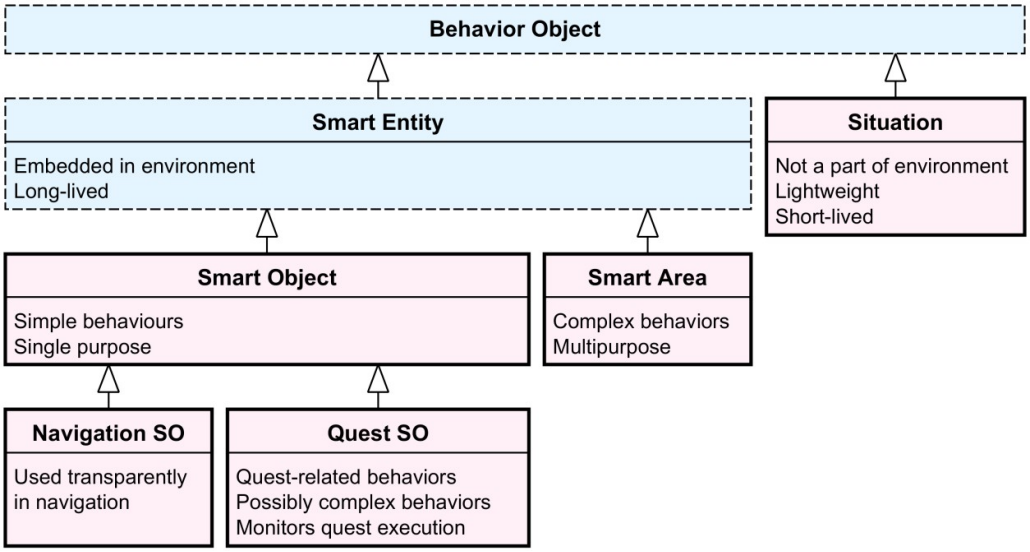


Figure 7: A simple class diagram of various types of behavior objects and their basic properties.
Abstract categories have dashed borders, while full borders correspond to types of BOs that are actually used in the game.

5.5.1 Requirements on the AI System

From the perspective of behavior injection, there are four very important aspects of the AI system in KC:D that we want to highlight (details for all of those features are given in Section 4). In practice, those or similar mechanisms will be needed for most BO implementations. The first one is the ability to execute a specific cleanup script when a script is interrupted. This feature is critical to allow behavior objects and NPCs to maintain consistency (objective O2). The second one is the hierarchy of *subbrains*. The subbrain interaction is important for the situation BOs and to decouple combat logic and ambient AI in order to make sure that combat is always functional (objective O1). The third one is a standalone mechanism for connecting BO instances with their environment data (as seen in Figure 4), which in the case of KC:D is the database of links between game entities (e.g., a pub is connected to all seats within; a quest is connected to an item the player should find). This way instances have specific data, but the data is decoupled from behavior code (objective O4). The fourth requirement is a mechanism for communication between BOs and NPCs – in our case this is the messaging system.

5.5.2 Smart Entities

A *smart entity* (SE) is a behavior object associated with a specific entity in the game 3D world. There two basic types of SEs in KC:D – *smart areas* (s-areas), which are associated with an area in the game world (e.g., a pub), and *smart objects* (s-objects), which are associated with a specific object (e.g., a door). S-objects are further divided into regular, navigation and quest s-objects. In a sense, SEs are a generalization of all of the well-proven techniques that embed intelligence into the environment (see sections 5.3.1 - 5.3.3). SEs are thus designed to be versatile and to be able to take a wide range of roles in the script codebase.

We will first discuss the common properties of SEs and later deal with their differences. Note that while there are huge differences in the way different types of SEs are used in practice, there are only minor differences in their implementation. This provided significant speed-up of development and allowed reuse of the same script development tools. SE contains all the information the NPCs need to behave appropriately in the context of a game entity (e.g., scripts for the innkeeper and for the guests, scripts for opening the door). If necessary, the SE's brain coordinates various NPCs within (e.g., assigns free seats to the guests, chooses NPCs to engage in a brawl, handles queues near the door, …). As such, an SE together with the in-game entity forms a standalone package that may be plugged in to the virtual world and be used by NPCs without modification to NPC code (objective O4). The SE may also disable certain scripts and limit the maximal number of NPCs that may hold a given script at one time.

5.5.2.1 Script Injection for Smart Entities

To ensure that gameplay-critical behavior remains uninterrupted (objective O1), we have decided that the injection should be performed on-request. We have thus added a new BT node that requests a script from SE (further referred to as *request node*).

There are two possible configurations for the request node that are handled in a slightly different way: 1) the designer explicitly states the name of the script that should be requested and 2) the script is left unspecified. In the former case it is checked, whether the SE provides a script with the given name, whether the script is currently enabled and that the maximal number of holders for this script is not exceeded. If any of the conditions is not satisfied, the SE returns failure and the request node fails. In the latter case, the first available script is chosen, failing only if there are no scripts available. This is used especially in the context of s-objects which often provide only one script. Another use is for the special case, when the NPC wants a smart area to just give it any script, which is useful for "idle" script. This is specific to ambient AI (objective O3).

If a more complex logic for requesting scripts is necessary, the NPC requests a high-level script from the SE, whose only task is to perform script selection and requests more specific scripts from the SE. This approach is used primarily when NPCs should behave differently in the same context based on their traits, preferences or knowledge (objective O5).

If the requested script is available, it is instantiated in a data structure called *script descriptor* which is passed to the request node. The script descriptor contains meta data about the script (e.g., when it should be dropped) and an instance of the script (BT) that achieves the behavior, which we call *injected subtree*. The injected subtree is then added as the only child of the request node and the tree continues execution by evaluating the subtree. The injected subtree has access to the NPC's state and data

and thus may modify the behavior appropriately (e.g., a rich guest in a pub orders more expensive food – objective O5).

If needed, the script descriptor contains new message inboxes that should be added to the NPC to allow synchronization and communication (objective O6). The request node is responsible for subsequently removing those inboxes when the script is dropped. The AI system supports nested inbox contexts in order to avoid problems with name clashes, i.e., if the script descriptor contains an inbox which has the same name as one of the inboxes the NPC already possesses, references from scripts defined in the SE will resolve to the newly added inbox while references from outside will resolve to the original inbox. This is necessary to maintain consistency (objective O2).

As synchronized action of multiple holders is often required (objective O6), the descriptor also refers to the SE's local context in which locks are resolved (the context is part of the instance state). This ensures that using a fixed lock name across multiple SE instances is safe. For example, when NPCs sitting around a table (a BO instance) in the pub want to synchronize movement during a toast, they may all explicitly reference "toast" lock. Since the lock name is resolved relative to the BO instance, holders of the same script at another table instance will receive a different lock when referencing a "toast" lock. This improves code readability and prevents the necessity to share a lock explicitly by messages.

As requests may be nested, it is technically possible to request the same script twice from the same object. But this is considered a runtime error, as the expressive power of recursion would do more harm than good in a game setting.

5.5.2.2 Smart Entity's Brain

The basic decision making of the SE is passive: for each script, the SE maintains information whether the script is enabled (i.e., whether new instances of the script may be requested) and the maximum number of instances that may be adopted at the same time. This information is used upon request processing – disabled scripts or scripts where maximum number of adopted instances has been exceeded are not made available for injection.

Some SEs, especially areas, however need to have brains to actively influence the scripts. The brain contains a script that gets updated regularly and may either modify the passive decision making based on external conditions (e.g., disallow "drinking" script in a pub if no innkeeper is present) or it may perform some coordination among script holders inside the area (e.g., instruct a pair of customers to play cards together). The coordination is done by sending messages between the SE and the script holders. Since the NPCs are now controlled by the injected subtrees, the SE can make strong assumptions about NPCs responses to its messages. Even if the NPC terminates the injected subtree, the cleanup logic of the behavior will notify the SE of this fact and allow for recovery. This central control of joint actions is an important aspect of the implementation as it removes the need for NPC negotiation (related to objective O6).

There are special BT nodes specific to the SE brain that enable/disable scripts and that send messages to holders of a certain scripts. The brain BT can access variables containing references to script holders and system data (e.g., what scripts are enabled). The BTs for the NPC scripts then may use a special node to send messages to the SE that the NPC received the script from.

In many scenarios, the SE needs to perform some action whenever an NPC adopts a certain script (e.g., assign a free seat to a customer in a pub) or when an NPC drops

the script (e.g., the innkeeper says goodbye to the leaving guest). To streamline the development in such scenarios and to make the BTs of the SE brain and the scripts more readable, we have introduced event handlers to the SE brain. An event handler is simply a BT that is executed until completion for each instance of an event. All of the SEs implement two events OnAdopt – an NPC adopts a script – and OnDrop – an NPC drops a script. S-areas introduce two more events that fire whether the NPC has requested a script or not: OnEnter – an NPC enters the area – and OnExit – an NPC leaves the area. The SE adds events to an event queue. If the event queue is non-empty upon updating the SE, the handler tree of the event to be processed is updated instead of the main tree. In order to keep handler code simpler and without safety checks and to simplify debugging, the handler trees are executed without interruption. The designers however must make sure that the handler trees complete quickly. The current practice is to only update the state of the SE or send messages inside the handler trees and perform any actual actions on the main tree. To prevent the main tree from starving at least one update to the main tree is guaranteed between two successive events.

5.5.2.3 Linking Data to Smart Entities

As there is an in-game entity (e.g., pub area, chair 3D model) attached to every SE instance, it is possible to use the linking feature of the underlying AI system to connect the instance to its environment data. This is easily done and visualized in the game editor. For example, the pub area has a link labeled “seat” to all chairs available for guests in the particular pub and further labeled links for the beer tap and other notable locations in the area. Upon initialization, the SE gathers the immutable environment data from the links to its internal variables to simplify access.

5.5.3 SE: Smart Objects

S-objects are SEs with the simplest intended use. Their task is primarily to handle short behaviors associated with specific in-game objects (sitting on a chair, opening door, cooking on a fire, …). The environment data of s-objects is therefore usually only the 3D model they are attached to. To reduce system load, majority of s-objects do not have their own brain and act only passively.

Still most of the s-object behaviors cannot be implemented by simply playing an animation, because KC:D aims for high behavioral fidelity. For example, when sitting on a chair, the NPC should move the chair a little away from the table with its hand, go closer to the table and drag the chair back to its original position while sitting down. To properly align the chair with the NPC, it must be first attached to the NPCs hand by its back, then the hand is detached and later the NPCs bottom is attached to the chairs seat. Without the attachment, the chair might easily become slightly out of sync with the NPC producing an eerily looking result – this is a limitation of game engines in general. Since the s-object provides complete code and not only an animation, these issues are handled easily.

Although s-objects are used for the simplest use-cases in KC:D, they are still much more powerful than s-objects in other OWGs that we know of. Apart from the simple uses outlined above, more complex scenarios are supported due to the very general nature of SEs. The most elaborate s-object that has been deployed so far is a bench that allows up to 4 NPCs to sit on it. Since the bench is attached to a table, NPCs cannot stand up directly, but need to move to the end of the bench and then

leave. If an NPC in the middle wants to go away, the NPC on the side stands up, clears the way and then sits back again.

5.5.4 SE: Navigation Smart Objects

Navigation s-objects are an extension of regular s-objects. Navigation s-objects work as a link between two navigable areas that would be disconnected otherwise. The most common ones are doors or barriers that can be jumped over. The purpose of the navigation s-object is to provide a script that the NPC should use to traverse the link. As in the regular s-object case, a more powerful mechanism than just playing animations was needed. A good example is a door: not only does the NPC play an animation, it also must be properly synchronized to the door and, more importantly, a queue of NPCs waiting for the door must be handled reasonably. For this purpose, the doors are linked to nearby places where NPCs should wait for their turn in the door and explicitly manage the queue, including giving way to the player. This central approach was chosen in favor of distributed solution using steerings or similar techniques because given the specifics of the AI and animation systems and various minor design requirements, the central control allows for much better results, although with some extra work.

Navigation s-objects also differ from regular s-objects in the injection method. The navigation s-object behavior is injected on-command – the NPC does not partake in the decision to use the s-object, it is the navigation system that decides that the particular s-object is used during movement. The injected subtree is then inserted as a child of the move node and updated accordingly. Once the injected subtree finishes, the move node resumes its normal execution if the subtree was successful or fail if the subtree failed. The injected subtree is removed from the move node in both cases.

5.5.5 SE: Smart Areas

One of the limitations of smart objects is that not all behaviors can be meaningfully attached to an object in the game. Many higher-level behaviors are better viewed as attached to an area (pub, forest, city, …). We have therefore developed smart areas (s-areas) which are smart entities connected to whole areas in the game world. As s-areas usually represent higher-level behaviors, they often delegate the low-level functionality to s-objects. In the pub example, the chairs or benches in the pub area are s-objects that provide “sit” script that is then requested from the “guest” script. A typical s-area thus has a large amount of environment data which consist mostly of smart objects it uses.

5.5.5.1 Script Requests and Smart Area Hierarchy

In contrast to s-objects, where the NPC has to possess an explicit reference to an s-object instance while requesting a script, an s-area may be used implicitly as “the s-area I am currently in”. While in some cases it turned out to be more useful to use s-areas with explicit references as well, implicit area referencing is useful when the NPC wants to perform any particular script from a larger group of scripts. A typical example is “relax” behavior: the NPC wants to perform any relaxing activity that is available in the area it is in (e.g., drinking or dancing in a pub, idly resting at a field or watching comedians in the city center). It is actually beneficial, if the relaxing activity is different when requested repeatedly. We call behaviors where an NPC is

not bound to a particular area *general*. This is in contrast with *specific* behaviors, such as "work" where the NPC has a specific place where it works and this should be the same every time it works and thus an explicit reference should be used.

There is however a catch in using general behaviors: to reference an s-area implicitly, the NPC must be inside the area. But how does the NPC know, where a pub is if all it requires is to relax? As mentioned in the design objectives, the pub location should not be hardcoded in the NPC's script. The solution was to introduce parent-child relationship between s-areas and make the whole city an s-area and make the pub its child. Now the city (the city designer) knows the locations of all pubs and other relaxing areas within. The NPC thus requests a "relax" script implicitly and the city s-area gives it a BT that consists of a sequence of a move node that moves the NPC to the pub and a request node that requests a "drinking" script in the pub.

However it later proved necessary to involve higher-level areas in specific behavior execution as well. The reason behind this is that the s-area should be able to control or modify all movement of NPCs within its bounds (e.g., make the NPC pick-up a torch when moving at night). A high-level s-area is a good place to store this kind of behavior modifications as it applies globally to all movements within the area. The code for this movement behavior is generic and not bound to any particular script (the target location is passed to the injected subtree through a shared variable, because BTs in KC:D currently do not support parameters to script requests). An example of injection of this kind of movement script is shown in Figure 8.

In the SE implementation we evaluated for this thesis, sleeping, eating and most of working behaviors are specific, but the pastime behaviors (fun, prayer, …) and some non-distinctive working behaviors (e.g., fishing or hunting) are general.

A different problem we aimed to solve with s-area hierarchy is that an NPC that is currently in a pub may decide it wants to pray, but the pub should not be required to know of all churches in the city. It is thus a good idea to ask the city in such a case. For this reason, if the current s-area cannot provide any applicable script, the request node asks the parent s-area. To avoid confusion, the scripters have adopted the practice that scripts in leaf areas have distinct names from scripts in the parent area and for general behaviors, only the scripts from the higher-level areas are requested. This way, the higher-level areas always take part in decisions about general behaviors that take place within its bounds and may for example balance the amount of NPCs in individual pubs in the city.

There is however one possible exception, when defining the same script in both child and parent areas might be desirable. This would be the case with general behaviors like "go to toilet", where the NPC should stay in the same s-area, if it may perform the behavior there, but should be able to ask a parent area, if this is not possible. As of now, no such behavior that the design team would want to have implemented in the game has been identified.

We have also introduced a default top-level area, covering the entire game world. This way general behaviors can be requested anywhere on the game map and the default area is able to guide the NPC to an s-area that provides a suitable script.

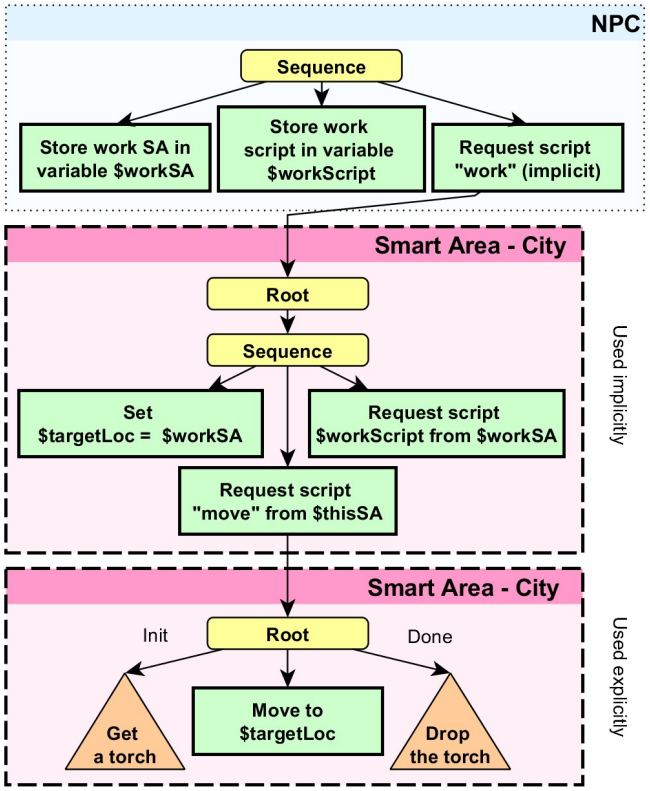


Figure 8: Handling specific behaviors at a high-level smart area to control movement within the area.
The $thisSA variable is provided by the system and refers to the s-area that provided the script. Note that when s-area scripts are nested, $thisSA will refer to a different value in different subtrees.

5.5.5.2 Using Smart Objects inside Smart Areas

One of the interesting problems the scripters tried to solve was how to properly use s-objects, in particular chairs, inside s-areas: i.e., how to best compose script selection functions expressed with BOs. The typical problem is as follows: the pub s-area wants the NPC to sit down, wait for a beer and drink it, then stand up. While sitting down and standing up should be delegated to a s-object, it is necessary that the script in between remains controlled by the s-area while still letting the s-object make sure, that the NPC does not remain seated if the script is terminated.

To keep the s-object in control of init/done scripts, the solution that was adopted is that the s-area script requests the s-object script which in turn requests a "private" s-area script that expects the NPC to be seated. The name of the private script is passed through a shared variable. Technically, the chair script should work as a decorator node over the private s-area script. An example of the setup is given in Figure 9.

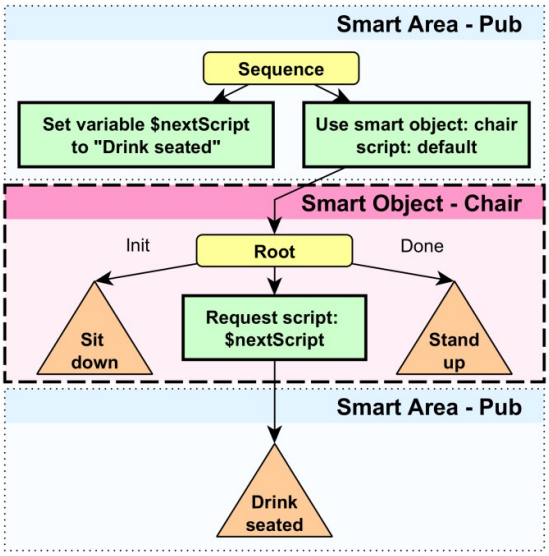


Figure 9: Using smart object to "decorate" a script in a smart area.
In the actual implementation, the node structure at the s-object script root is a bit more complicated, but is conceptually equivalent to the structure in the figure.

5.5.6 SE: Quest Smart Objects

Quests can be divided into two categories with regard to the way they affect NPCs: quests that do not require NPCs to change their behavior are called *behavior-preserving* and those that do are *behavior-changing*. An example of a behavior-preserving quest is a guard asking the player to kill bandits living in the woods. The bandits behave the same way they would without the quest (attack the player when they see them) and the guard keeps guarding the village. The only change is in dialog options the guard provides – once the bandits have been killed a new dialogue to congratulate the player on success in the quest is activated.

The quest system in KC:D is event-based and handles behavior-preserving quests very easily. A quest is composed of a series of steps. Individual steps of an active quest listen to events in the game (player picking up items, killing enemies…). These events then trigger progression of the quest to next steps. The quest steps are also bound to dialogues that the NPCs use and allow/disallow various dialog options.

Behavior-changing quests are however more demanding and handling behavior-changing quests is the main motivation for quest s-objects. It is not desirable for NPCs participating in a quest to completely abandon their daily cycles and stand at one place, waiting for the player. Instead, behavior changing quests may directly modify daycycles of the participating NPCs. Initially a new type of BO was intended to encapsulate scripts related to a given quest, but to save development time, it was decided to use *quest smart objects* to handle this task. A behavior-changing quest delegates execution of some of its steps to a quest s-object. The quest s-object then notifies the quest of completion/failure of the assignment by the player. This communication is done through the message system.

The quest smart objects are technically the same as regular s-objects which allowed us to directly reuse code for both the game and the editor, but they are used very differently. The most notable difference between regular s-objects and quest s-objects is that quest s-objects are connected to *quest anchors* – game entities not visible in the game. Quest anchor's only function is to connect the quest s-object to

its environment data. Quest s-objects also always have brains that guide the execution of the quest step(s). The quest s-object then may instruct the NPC to exchange a part of its daycycle with a behavior requested from the quest s-object.

If the quest requires the NPC to change its behavior completely, regardless of the day cycle, the appropriate approach would be to introduce new higher-priority quest subbrain to the NPC. The quest s-object would then activate the quest subbrain which will in turn request a script from the quest s-object. However all of the quests implemented so far are designed to keep at least the basic daycycle intact (in particular let the NPC sleep at nights).

5.5.7 Situations

While SEs are powerful and versatile, they are not well suited for small and short-lived events. To enrich the world with such short events, we designed situations as a different type of BO. A situation encapsulates a short coordinated behavior involving multiple NPCs. Typical examples of situations in the context of KC:D are two villagers pausing for a small talk, a collective dance in the pub or a brawl. An important aspect of situations in KC:D is that they serve mostly as "eye candy", i.e., they should not significantly alter the state of the game world. This is important because it lets the AI system run situations without considering their consequences for the current state of the game. Situations have deliberately very low priority so that any "important" behavior always overrides the situation.

Technically, situations are run within a specific situation subbrain of the NPC, which has a higher priority than ambient AI, but lower priority than any other subbrain. Thus if only an ambient behavior is being performed, and the NPC should start performing a situation, the ambient AI is suspended and the script provided by the situation is started. This could still lead to undesirable results (such as an NPC starting to dance in the middle of a conversation). To keep the ambient AI in some control over situation execution, the NPC has to explicitly subscribe to the situation system. This is achieved by decorating a subtree of the ambient AI script with a special node that subscribes the NPC when the execution of the subtree starts and unsubscribes when the execution is finished. This makes it possible for situations to be developed almost independently of the rest of the AI code, as the potential negative interactions with other scripts are minimized by design (protecting gameplay-critical behaviors – objective O1).

A situation template describes several roles, each providing a script for one of the NPCs that participate in the situation. Roles also have associated conditions that an NPC must satisfy to take the given role (e.g., to engage as an aggressor in a pub brawl, the NPC must be drunk). The situation templates are connected to s-area templates to allow for area-specific situations. Once a component called situation manager decides that a particular situation template should be instantiated, it tries to find suitable NPCs using constraint satisfaction techniques. After the NPCs are chosen, an instance of the situation is created and the scripts are injected on-command as the main tree for the situation subbrain making the subbrain active.[9] More details on the inner workings of the situation manager are given in Chapter 6. Here we are interested only with how situations encapsulate a set of scripts.

[9] To avoid multithreading issues and inconsistencies, the script is sent to the situation subbrain as a request for injection. The situation subbrain then injects the script during its next update.

If any of the chosen NPCs cannot execute the situation or terminates the situation prematurely (e.g., because a higher-priority subbrain becomes active), all other participants also abort the situation. Once again, the clean-up scripts are guaranteed to be executed, keeping the system in a consistent state (objective O2). After all participants finish their scripts, the situation instance is destroyed.

As a behavior object, situation is lightweight compared to SEs. This is mostly because situations are much more specific than behaviors provided by SEs and that situation instances are short-lived. In particular, situations do not have their own brain, as central decision making is usually not necessary and in the rare cases when it is, one of the holders may handle the central logic.

For coordination purposes (objective O6), all participants are given explicit references to all other participants and a local synchronization context is maintained for the situation. The situation also provides the participants with up-to-date information on the state of the other participants, especially if they already started the given script or if they dropped the script and thus may no longer be expected to cooperate.

5.6 Evaluation

The evaluation in this chapter consists of two main parts: qualitative observations gathered in 16 months since the scripters first used smart areas for development and two rounds of semi-structured interviews we performed with scripters. In our previous work, we have also performed quantitative evaluation, which will be discussed before the results in this thesis.

5.6.1 Summary of Evaluation in Previous Work

In our previous work, we have performed quantitative evaluation of scripters' performance using BTs and using BTs with s-areas (Černý et al. 2014). While part of the measured metrics showed statistically significant differences, the sample size and the scope of the tasks assigned to the participants was limited and it was concluded that the "data provide some support that s-areas are better, when modifications are frequent – which is the case in real development – but the results are not clear and further research is needed."

Qualitative feedback was also gathered, including the fact that "subjects were relatively quick at understanding code created with s-areas. Judged by the researcher monitoring experiment progress, the subjects using s-areas had no trouble finding the code for a particular behavior [in contrast to subjects not using s-areas]."

We have also tested the computational load of the AI system – including SEs – imposes on the CPU and found that it is fast enough for production (Plch et al. 2014). The system consumes less than 1 ms on average with 30 complex NPCs running and less than 2 ms with 300 simple NPCs running. However, the large amount of s-objects turned out to be a bottleneck. In response the KC:D team decided to reduce the number of s-objects that have their own brains and to update those that have brain less frequently to ensure swift execution even with larger worlds. BO instances also share pools of instantiated BTs and inboxes to reduce memory footprint while keeping the performance benefit of preallocated and preconstructed objects.

5.6.2 General Observations

In general, the virtual world works well using BOs and the structuring of scripts into objects enables scripters to concentrate on the individual aspects of the world (pub, shop, church, …) while only minor problems arise during integration of the individual objects into the world (objective O4). As an example, one of the scripters was tasked with adding ambient AI to a freshly created village. This required placing approximately 60 s-areas and 200 s-objects in the world, linking them to environment data and performing basic tests. The scripter completed all those tasks by himself in two days.

The scripters have been able to implement behaviors that – to our knowledge – have not been present in any commercial game, for example realistic door and seating logic and complex interactions with items in a pub. Those behaviors were included in the public release of a beta version of the game. S-areas and all types of s-objects are considered stable and have been deployed in a public beta version of the game. The situation system is still in preliminary use but unlikely to change significantly.

We have also noted that s-areas, quest s-objects and situations closely correspond to the way game designers think about the world: it is natural for them to describe the behaviors that NPCs should manifest in a pub separately of other behaviors the NPC perform in their daily cycles.

So far, over 30 types of s-areas and over 40 types of s-objects of release quality have been developed and released in the public beta version. Nine situations were developed to test the situation system, but these situations will be subject to heavy changes before inclusion in a release build. Fifteen quests have been released in the public beta version and multiple others have been developed in release quality.

We received mixed feedback to the fact that s-areas have strict boundaries. Boundaries introduce issues to handle when the movement to the area fails for some reason or produces unnecessary movement, in case the s-area instructs the NPC to leave the area (e.g., to gather wood outside the area). On the other hand, strict boundaries are beneficial from debugging perspective – one can be sure, that since the NPC is outside the area, it cannot receive a script from it.

Another lesson learnt is that it is vital to keep the s-object scripts small and focused on a single task while providing detailed control to the parent s-area. An example that used to be problematic is feeding fire in a house. In an initial implementation, the house s-area told the NPC to use a fire s-object. If the fire s-object realized there is no wood, it instructed the NPC to use an s-object representing a pile of wood outside the s-area. Now the s-area believed that someone was performing fire feeding script and should be done quickly, but in fact, the NPC was outside the area on a much lengthier task. The current implementation is that if there is no wood, the fire feeding script fails. The s-area is notified of the reason for the failure and assigns a “find wood” script to the NPC. This way the s-area is more aware of what is going on and may react to individual events.

The above example also illustrates where the system evolved to: the individual scripts are kept small and are hierarchically requested from a large number of BOs. One further example is the pub. The pub directs NPCs to table s-objects which in turn delegate the actual sitting to several attached chair s-objects. The table also manages a bowl s-object that manages pieces of chicken (also s-objects). To eat, the NPCs thus request eating script from the table, which requests a script from the bowl, which requests a script from the chicken pieces. This arrangement is instructive and scripters are happy that building a new pub can be done by simply arranging the

premade s-objects and connecting them with links. Furthermore, any element can be replaced without changes to the others (e.g., a plate instead of a bowl, a piece of pork instead of chicken) and the individual scripts are easy to debug. The downside is that the abundance of s-objects is taxing on the system by both the need to manage the s-objects and by making the NPCs trees deeper and thus slower to evaluate. However, this load seems manageable so far. Another performance related technicality it that it was very important to pool and reuse the injected subtrees instances instead of creating and destroying them on the fly.

5.6.3 Qualitative Feedback

We performed two rounds of semi-structured interviews with all 6 scripters involved KC:D development in the first half of 2015. Except for the technical design lead, these are all of the KC:D development team that used BOs on a daily basis at the time of the interviews. We have chosen a qualitative approach because there are few scripters in the company and thus quantitative conclusions would be weak. The experiences of the individual scripters are also not comparable, as the scripters specialize and solve very different problems. Recruiting external subjects to extend the sample size is not practical, as a large amount of knowledge has to be mastered, before a user is able to deal with tasks at least remotely connected to actual practice. Structured interviews have their limitations, nevertheless they have given us valuable insights for further development of our variant of BOs and we consider them of interest to anyone trying to implement their own BO variant.

Shneiderman and Plaisant (2005) recognize five basic usability measures: time to learn, speed of performance, rate of errors by users, retention over time and subjective satisfaction. We focused on subjective user satisfaction as this is the only category where we can, to some extent, separate the effects of using BOs from the features and quirks of the underlying AI system and BT implementation.

The first round of the interviews consisted of broadly formulated questions on the general usage of the AI system (including BOs), while the second round had more focused questions linked to the design objectives of BOs.

5.6.3.1 First round of interviews

The interviews in the first round consisted of nine questions and took 30 - 60 minutes. Table 3 shows the questions and the information we expected to gather from the answers. In general, we tried not to mention BOs in the questions to make scripters more likely to report when they used alternative solutions and to prevent bias. We also wanted to gather feedback for other parts of the AI system than BOs. While we asked the scripters to report on the AI system as a whole, we expected all the answers to reflect on BOs to an extent, as the vast majority of in-game behaviors are built with BOs.

Q1	What were the tasks you worked on recently? *Frame the interview and provide source for specific examples for the rest of the interview.*
Q2	What activity consumes the most of your development time? *Discover the main bottlenecks for production.*
Q3	Give an example of a code segment/snippet that is often repeated across behaviors and has to be copied each time and a segment that is well reused across behaviors. *Discover a situation where BOs are not applicable in practice, although they should be in theory. Understand the potential for AI code reuse.*
Q4	Describe the process of implementing a behavior from a design request to the final code. *Discover how BOs fit (or do not fit) in the overall production pipeline.*
Q5	How would your behavior code change if you could only use plain tree injection (without BOs). *Understand what features of BOs are considered important.*
Q6	What was the most complex/difficult task you have worked on in this company? *The most challenging tasks are likely to demonstrate the full power (or lack thereof) of a system.*
Q7	Describe the process of resolving an issue reported by the QA department. *Discover whether BOs help/hinder debugging.*
Q8	What do you dislike about the scripting tools? *Gather all the problems scripters face when writing code.*
Q9	Describe your ideal scripting tool. *Gather constructive suggestions and let the scripters compare BOs to hypothetical alternatives.*

Table 3: Questions in the first round of interviews.
The reason why we included the individual questions are shown in italics.

First, we will focus on issues with the system, which were mostly reported for Questions 3, 8 and 9. The scripters reported a number of usability problems with the underlying AI system, especially with the BT editor and debugger, but only three concerns that could be linked to BOs were raised. The most common issues related to BOs (mentioned by four scripters) were the usability problems inherent in debugging large trees (e.g., "the trees do not fit well on a single screen"). This is mainly an issue with the BT editor in KC:D, but is related to BOs, because every injection adds depth to the tree and use cases in KC:D require a large number of injections. Second came the necessity to use global variables to parameterize injected scripts (four mentions by two scripters, see Figure 8 for an example). One scripter also mentioned that he dislikes that all s-objects need to be connected to an in-game entity, although for some quest s-objects there is no natural connection. Overall, BOs are seldom the source of frustration of scripters, although they are used on a daily basis. The answers also support our previous work (Gemrot et al. 2014) where we show, that quality tooling support is vital for a technique to succeed in practice.

Except for a few references to usability issues with the AI system, Questions 4, 6 and 7 did not provide any valuable insight into BO usage. The other questions however conveyed some interesting feedback.

The most time-consuming activities (Q2) were debugging in general and updating code after a backwards incompatible change has been made to the underlying AI system (both mentioned by four scripters). Good debugging support is thus vital to a success of a tool. It is also obviously beneficial, although not always possible in

practice, to change the underlying AI system as little as possible during production. One scripter reported development of synchronized behaviors as the most time consuming and one reported that he spends most time in figuring out, how exactly should the relatively broad requirements from game designers be implemented at the low level.

Examples given for good encapsulation (Q3) were very specific to KC:D and do not provide a valuable insight into BO usage. However, seven examples were given of frequently copy-pasted code. All of those were small snippets consisting of up to eight nodes and were not suitable candidates for BO-based implementation (e.g., searching the link network for a useful object, aligning animations to game entities). While this means that BOs let the scripters reuse larger code structures without problems, it also indicates a room for improvement of the AI system: creating a reusable BT snippet should be made easy, especially it should be straightforward to pass data (parameters) to an injected tree.

Best insights into BOs were provided by Q5. Scripters reported that without BOs they would reimplement: the ability to connect behavior code and data (four mentions); a local communication hub/an entity that handles messages related to a given context (three mentions); a central logic (brain) for a set of scripts (two mentions) and a container of related scripts (one mention). One scripter also mentioned that BOs help him write consistent code and another stated that he "would implement something very similar". We see that the defining properties of BOs (connecting code and data and a centralized point for coordination) were indeed perceived as important.

5.6.3.2 Second round of interviews

The second round of interviews consisted of five questions which aimed to elicit feedback on how do BOs fulfill the design objectives of the system (see Table 4) and took 10 - 30 minutes.

Q10 was bound to objectives O1 and O2. Only one scripter reported that he has written complex code with interruptions in mind. He has been responsible for making scripts work when interrupted with a dialogue. While it was not hard to let the NPC finish an uninterruptible task prior to dialogue, main difficulties stemmed from the fact, that the script has to resume to the point where the dialogue started, while some of the NPC's state resides only in the animation system, inaccessible to the BT. In particular, an animation may be queued for execution, but not actually started when the dialogue is invoked (see Section 5.4.1.1 for the reasoning behind this).

To remedy this, an improvement in the animation handling was implemented, letting the scripters to directly access animation state and to create "lambda BTs" – a BT counterpart to lambda functions in classical programming languages. Lambda BTs are subtrees that are attached to events in the animation system. These subtrees then get executed regardless of the progress in the main BTs and can send messages that are handled in an appropriate moment by the main BT.

Q10	When writing code, do you take into account the possibility of interruption by quest/combat? How? *(O1 and O2)*
Q11	Is there a difference in using BTs and BOs in quest logic and in ambient AI? *(O3)*
Q12	What are the necessary steps to place a new instance of an s-area/s-object in the game world? *(O4)*
Q13	Have you implemented any behavior where the attributes of an NPC would change the way the NPC behaves in a given context? *(O5)*
Q14	What was the most difficult synchronization/coordination task you implemented? Why? *(O6)*

Table 4: Questions in second round of interviews.
The design objectives that motivated the questions are shown in italics.

The same scripter and three other colleagues have implemented simpler interruption-aware code that handled halting of the subtree (stopping the script without the need to resume to the original state). Two of those reported that it was easy and one other reported that BTs support halting well.

We see that the system demonstrates capability to properly handle "hard" interruptions when the NPC discards the running script completely, while further refinements are necessary to the "soft" interruptions where the script is expected to maintain its state after the interruption has finished.

Q11 was intended mainly to check whether our focus on ambient AI (objective O3) has not introduced problems in quest handling. This does not seem to be the case as no scripter reported notable problems with implementing quest behaviors. The only problem that was mentioned was the fact that quest logic intersects with multiple systems with overlapping capabilities: quest s-objects, the dialog system and the quest system (see Section 5.5.6 for details). The consequences are twofold: 1) writing quests requires the scripter to interact with several different user interfaces and 2) there are multiple ways to distribute the quest logic among the systems. The current consensus is that when a quest uses an s-object (i.e. when the quest alters behaviors of NPCs), then all of the quest logic is implemented within the s-object and the other systems only pass messages to the s-object. Other than that, three scripters considered quest behaviors to be very similar to ambient AI and two scripters considered quest behaviors to be simpler in general than ambient AI. One scripter has not implemented any quest yet. Although we did not ask directly about quest s-objects, two scripters said that quest scripts differ in that quest s-objects serve as a central entity to coordinate the quest. This indicates that quest s-objects do their job well.

For Q12, all scripters reported that to create a new instance of a BO, they never needed to do more than link the BO to the appropriate environment data. Two scripters explicitly said that the process was quick, while two reported on usability issues with the linking system. This shows that behavior code is well decoupled from data and that objective O4 was fulfilled.

With regards to different behaviors of NPCs based on their attributes (objective O5, Q13), only one scripter has already implemented such a behavior. This was a military camp, where soldiers are assigned different work tasks based on their rank. He did not report any problems in achieving this, but further investigation would still be needed to verify that this use case is well supported. As for synchronization and coordination (objective O6, Q14), all scripters encountered tasks that required

explicit synchronization of NPCs, but two only in a very simple context. Only one scripter built synchronization outside the scope of a BO and he referred to this case as the most difficult to handle. Another scripter explicitly mentioned that s-areas were helpful for coordination. Three scripters considered synchronization to be non-problematic, while two reported usability issues with debugging and implementing synchronized behaviors. One scripter also reported that he considered parallel behaviors challenging in principle. Two scripters reported usability issues with the message system that make writing message-oriented code tedious. Two scripters described the need to reduce the scope of possible NPC states when coordinating behaviors for quests – when NPCs need to cooperate on a quest, they are usually instructed to stay at a well-defined place and perform only very simple activities so that other NPCs can make simplifying assumptions on their state.

One scripter reported a performance issue that arose while he was implementing advanced door handling behavior where NPCs form a queue, but there are dynamically changing priorities for NPC ordering (e.g., if the door is locked, an NPC that has a key is given priority). This resulted in multiple rounds of messages being exchanged between NPCs. As two-way communication cannot be performed within a single frame and there was a relatively lot of computation involved between the messaging, the system exhibited visible lag when many NPCs tried to use the same door at once. This can be resolved by both simplifying the code and by giving larger time budget to evaluate trees of s-objects that are heavily used.

In general the data indicates that synchronization and coordination is handled by BOs in a satisfactory manner, although improvements can be made, especially in tools and usability.

5.7 Discussion

In this chapter we have described behavior objects as a tool to manage complexity of reactive decision making in OWGs. BOs become useful when the size complexity is large and it is thus necessary to partition the behaviors into pieces that may be developed and tested independently. BOs allow for easy development of reactive script selection functions at multiple levels of abstraction while maintaining consistency of the scripts. BOs let us efficiently hide *how* behaviors are implemented from the script selection function deciding *what* behaviors should be performed.

We have shown five use cases (smart areas, smart objects, navigation smart objects, quest smart objects and situations) for BOs within the AI system for an upcoming AAA role-playing OWG. The behavior object concept has withstood field testing and was deployed in multiple forms within a public beta version of the game. Qualitative feedback and lessons learned during the implementation were presented. The available feedback suggests that BOs are a suitable approach for managing complexity in NPC scripts, fulfilling all design requirements.

While the initial development of smart entities and situations was driven simply by the needs of the AI system, we have noticed the similarity of the concepts to object-oriented programming. We have established the connection between behavior objects and OOP explicitly, as it helped us drive further development of the implementation in KC:D and provided inspiration. We believe that inspiration by OOP can be useful for the next generation of game AI and lead to dramatic improvements in code manageability, as OOP has done for classical programming. Our implementation is based on behavior trees, but BOs should be usable in all

reactive action-selection mechanisms that are in frequent industry use. Since the BO concept is not bound to any specific AI technology, but is rather a form of imposing abstractions in the code, it is applicable to a wide range of use cases in OWGs or in game AI in general.

6 Constraint Programming for Global Specification of Behaviors

Behavior objects, as presented in the previous chapter are useful for structuring AI code around higher-level concepts, but may still be insufficient in certain cases. One particular dimension for improvement is to allow for more global connections of code and content. This was emphasized in the Subgoal 2 of this thesis (see Section 1.3). In particular, we are interested in a global approach to script selection for coordinated multi-NPC in-game events in AI components with large size complexity. We address this subgoal by using constraint satisfaction techniques to select tuples of NPCs that will enact a designer-specified situation.[10]

In relation to BOs, it is important to note that keeping the connections between code and content local was one of the main motivations for BOs. This is definitely a good thing and we need to keep this locality for most cases. What we aim for in this chapter is to find a mechanism that would allow us to impose a more global view on script selection wherever it is necessary, while building on top of BOs and maintaining local (and thus easier to manage) connections everywhere else. We will show that constraint satisfaction problems (CSPs) are a viable formalism for declaratively specifying script selection in a more global context. Similarly to the previous chapter, this chapter focuses on AI components with small rule complexity, but large size complexity (see Table 5).

We will first describe what exactly we mean by "global connections" and motivate both the general approach (Section 6.1) and the particular use case we aim to solve (Section 6.2.) Then we discuss related work (Section 6.3) and describe our solution to the general problem (Section 6.4.). We further describe a particular implementation of global specification in KC:D (Section 6.5), evaluate it (Section 6.6) and summarize the results (Section 6.7).

[10] This chapter is based on the paper (Černý· et al. 2014), a large portion of this chapter has been directly copied from the paper, with only minor adjustments.

		Rule complexity	
		Small	**Large**
Size complexity	**Small**	Basic enemy AI	Quality enemy AI Ally AI
	Large	**Ambient AI** Dialog-handling AI	Strategic squad coordination High-level RTS AI

Table 5: Complexity classification for the techniques in Chapter 6.
This chapter deals primarily with small rule complexity and large size complexity in general, the implementation in KC:D is focused on ambient AI in particular (highlighted). This table is a reiteration of the complexity classification from Table 1.

6.1 Motivation for the General Case

Let us start with an example of script selection in a BO-driven AI component: An NPC, currently in a city, wants to relax – at the highest-level script selection, it requests a "relax" script from the city BO. The script provided by the city BO enumerates child BOs suitable for relaxing and chooses a pub BO, instructing the NPC to go to the pub and request a "drink" script there. The "drink" script than instructs the NPC to use a "sit" script from a chair BO and makes the NPC interact properly with objects representing glasses to drink from, which are connected to a table BO.

We see that a connection between an NPC and a piece of code or a game object is made by a series of decisions that are local to the NPC or the BO the NPC uses. This paradigm however prevents us from choosing scripts from a more global viewpoint.

A typical case where a global view is useful is when specifying script selection functions for an AI component responsible for randomly generated events or optional side-quests. For example a designer may provide requirements of the following kind:

1. "for a robbery event to occur, we need an NPC that is rich and a pair of thugs", or
2. "in a pub, two drunk NPCs may get into a fight over who gets the attention of a third, handsome NPC", or
3. "an optional side-quest will ask the player to carry an expensive item stored at a city controlled by enemy forces to a city controlled by a friendly army".

A typical approach to handle such generated events in contemporary games is to spawn new NPCs/objects fulfilling the requirements as needed and despawn them

once the event ends. This can however damage immersion, as the player may notice that the events are not tied to the state of the game world and that the events never concern the NPCs that are long-lived.

However, if the NPCs/objects that are already present should participate in the events, there are other difficulties. A direct approach is to handle the event requirements locally (within the thug script selection function or a brain of the BO controlling the pub or the city). Unfortunately, this quickly leads to problems once the requirements become non-trivial – should the pub keep count of all the drunk NPCs and handsome NPCs present? How does a city know which items are stored within it? And how does it discover an enemy city? Handling all those small tasks inflates the size of the codebase and hinders manageability.

While we obviously can incorporate code for handling those events into NPCs/BOs script selection, checking more complex requirements becomes verbose and plagues the script selection functions of the NPCs and BOs with code unrelated to their main function. It is also not optimal from the performance perspective if all cities or all thug NPCs try to find a suitable match on their own, possibly repeating many checks. At this point, creating a new AI component that performs script selection with a well-separated global view may become very useful.

6.2 Motivation for the Specific Case of KC:D

The BO system described in the previous chapter proved to enable scripters to create a large palette of ambient behaviors with little effort. Nevertheless the system primarily works by sending NPCs to various locations where they perform a behavior and then move to another location. For a truly lively world, the NPCs should also exhibit non-trivial behaviors while "in transit". For the use case in KC:D we therefore aim for a subsystem that is able to add variety to the NPCs movement across the virtual world with focus on interactions between NPCs. Literature on crowd simulation provides inspiration for good implementation of low-level behaviors and interaction of the NPCs (gaze control, collision avoidance, …), but there are fewer guidelines how to introduce higher level interaction (e.g., small talk, petty crime). At the same time, design and computing power restrictions severely limit the acceptable complexity of NPCs.

To enrich the simulation of the virtual world without the need to increase complexity of the individual NPCs, we propose what we call *situations*. A situation is a specific type of event as discussed in the previous section. It is a short scripted scene that involves multiple interacting NPCs. All of the situations that were of interest to the design team involved 2-5 participants. The situations span from very simple and frequent (e.g., NPCs greeting each other in various ways) to complicated and infrequent (e.g., people gathering to perform a collective dance). The situations have to be decoupled from the AI of the individual NPCs, so that adding new situations does not require any code changes on the NPC side. Also the individual situations have to be well encapsulated pieces of AI code to make them easy to modify and debug. Note that the requirements on NPCs to enact a situation naturally take a form of constraints on the NPCs or their relations (e.g., the situation needs a beggar and a rich man who are close to each other). This makes constraint satisfaction problems (Dechter 2003) a promising formalism to work with.

The downside of the global approach to script selection is the necessity to search for suitable tuples of NPCs. In the case of KC:D, situations are expected to be

scheduled at relatively long intervals (several situations per minute at maximum), which gives some leeway, but as a purely decorative element, it still needs to not be very taxing. An individual search is required to not take more than 1 ms in the worst case and the average should be lower than 100 μs.

Apart from choosing good algorithms, a huge performance gain may be achieved by limiting the scope of the problem. Although the world of KC:D should feature hundreds of NPCs, there are multiple considerations that will reduce the number of NPCs involved in search and thus let us meet the runtime requirements. Most importantly, situations are, in the case of KC:D, only "eye candy" so there is no need to consider NPCs that are too far from the player and cannot become visible to the player for the duration of the situation. Next, most of the situations make sense only in certain areas (e.g., the "Beggar" situation is tied to a city) and thus only NPCs that are present in the area may be considered. Finally, some of the NPCs will not participate in situations at all because they execute a more important script. This way we will never search more than a few dozen candidate NPCs.

It is further vital, that situations do not disrupt any important game mechanic (quests, combat, …) or threaten the consistency of ambient behavior scripts in any way. In particular, situations should not change the state of the world in any significant way.

With respect to the complexity classification presented in section 1.2, the situation-controlling AI component will have low rule complexity (there are few requirements to satisfy per situation and they are easy to express) and high size complexity (we usually want to have a large pool of situations) – similarly to the case of behavior objects.

6.3 Related Work

There are two broad areas of related work: crowd simulation research on development of complex NPC interactions and the use of CSPs in games in general.

6.3.1 NPC Interactions in Crowd Simulation

Pedica & Vilhjálmsson (Pedica and Vilhjálmsson 2009) pioneered the area of social interactions in virtual crowds. However, their work focuses on relatively low-level social behavior (attention, gaze and positioning) rather than on higher-level behaviors.

In the CAROSA framework (Li and Allbeck 2011) interactions between characters may emerge from agent-centric decision making. Although little detail is provided, the main mechanism seems to involve characters switching behaviors in response to their needs and in response to another character with a given need nearby. This way the interaction code has to be split among several parts of the system – the code that triggers the need for interaction, the code that ensures proper role switching in the other character once the initiator approaches and the code for the interacting roles, making the approach unnecessarily complicated and reducing the amount of design decisions that are hidden from the rest of the AI code.

Also in other contexts, it has been noted that complex short-time events are hard to generate emergently from local rules and work better when scripted (Shoulson et al. 2013). A need for a compromise between local and global control in agents is also discussed in (Russell 2008), giving support to the idea of separate local control (e.g., with BOs) and global specification as in our approach.

6.3.2 Constraint Programming in Games

Closest to our focus, The Sims: Medieval feature a central entity that selects individual NPCs to perform a specific role in the game based on simple constraints (Graham 2011). However, tuples of NPCs are not searched for.

Most applications of the CSP formalism in computer games are in procedural generation of game content. This is a very natural application, as valid content can often be concisely described by a set of constraints that have to be met.

Most CSP-based content generators are fully autonomous. Marouene, Paul & Vincent (2011) use CSPs to place objects in a 3D virtual environment. Horswill & Foged (2012) populate a map in a game with items and enemies to fulfil various constraints. Everyday Genius: SquareLogic uses technique very close to CSP to generate numerical puzzles (Sturtevant et al. 2014).

Apart from fully autonomous generators, Tanagra (Smith et al. 2011) is a mixed-initiative tool for platformer level design. It uses CSP to ensure that the level under consideration is playable.

Some generators also use constraints expressed as answer-set programs to generate both content (Smith et al. 2012) and whole rulesets of games (Smith and Mateas 2010).

CSPs have been used also for automatic camera control (Bourne et al. 2008; Ali and Goodwin 2008).

While all the above approaches have provided inspiration and proven that CSPs are viable technology for game development, none of the works considers using CSPs to encode global specifications of behaviors.

6.4 Our Solution – Global Specification

As noted in the motivation for this chapter, it is cumbersome to find good matches for events or side-quests locally from within the script selection of the participants. What we want to achieve is a mechanism for a declarative *global specification* of script selection. Global specification means that we simply specify some conditions for a script/a set of scripts to start along with requirements on entities that will enact the scripts and the system selects the participants automatically. While this requires a list of all entities in the game or all entities in a given location to be available, such lists are usually already present in the game and if not, they are easy to create and maintain.

We further note that the requirements imposed on participants can be modelled naturally as constraints on individual objects and their relations, exactly as in constraint satisfaction problems (CSPs) (Dechter 2003). Using CSPs in this context lets us build upon a large body of previous research. Importantly, CSPs relevant to games will have few variables (designers are unlikely to design a quest/situation/… with more than several participants). And such small CSPs can be solved very quickly using state-of-the art CSP techniques, allowing us to meet the runtime requirements of the game.

In this context, CSPs are not limited to events or side-quests – CSPs can for example model complex goal conditions or trigger an action from an AI director of the player's experience. In general, a CSP-based approach allows us to partially overcome our inability to predict or even control emergent global patterns of behaviors in the game world. CSPs can serve as a bridge that let us search for a global structure in the current world state and exploit that structure as it arises. We

can then implement mechanisms that incorporate the result in the NPCs script selection without direct changes to the NPC code, maintaining a clean separation between local and global viewpoints.

Returning to the parallel between BOs and object-oriented programming (see Section 5.4), global specification is loosely related to aspect-oriented programming (AOP) (Kiczales et al. 1997, 2001). AOP introduces *aspects* which are single entities that implement a functionality that is shared by a large number of objects/methods, but does not align well with the object decomposition (e.g., logging, transaction handling, …). Aspects let the logging or transaction handling functionality to be defined in one place and thus easily modified and reused. In this view, global specification provides a single place to define scripts that may be adopted while the NPC executes any of a large number of unrelated BO-provided scripts.

6.4.1 CSP Overview

A Constraint Satisfaction Problem (CSP) (Dechter 2003) consists of a finite set of variables, where each variable has a finite set of possible values, called a domain. In our case, each variable corresponds to an entity the global specification scheme requests and the domains are lists of all entities of the relevant type (NPCs, items, locations, …). The values that can be assigned to variables are restricted by constraints, where each constraint is defined over a subset of variables and implicitly defines a subset of the Cartesian product of variables' domains (allowed tuples). A binary constraint can for example express the maximal distance between two entities and a unary constraint may express required NPC properties for a given role. A solution to a CSP is an instantiation of all the variables satisfying all the constraints.

CSP is NP-complete so the search is exponential in the worst case, but we expect the search to be relatively "easy" in an actual game implementation – only a small number of variables will be required and there usually will be a lot of solutions or there will be no solution and very few "almost solutions", letting us prune the possibilities quickly. In other words, most global specifications will not be contrived counterexamples to the heuristics/pruning methods we will use.

The mainstream approach to solve CSPs is based on a combination of *backtracking search* and *inference*. Backtracking search repeatedly selects a variable for instantiation and then selects a value to be assigned to that variable. After each variable instantiation, constraints are tested against the partial solution. If the constraints fail, the search *backtracks*: tries another value for the current variable. If there are no more values for the current variable, the backtrack returns to the variable assigned previously. The search terminates once all variables have been instantiated (success) or when there are no more values to try for the first variable (failure).

When there are multiple constraints affecting only small number of variables, backtrack may be caused by conflicts with variables that have been assigned "long ago". *Backjumping* detects such situations and backtracks to the conflicting variable directly without trying different values of the irrelevant variables.

Inference techniques reduce search space by propagating the partial instantiation to other variables. One of the basic inference techniques is *forward checking*. It means that values violating any constraint (with the currently instantiated variables) are removed from domains of not-yet instantiated variables. For example, if we assign an NPC to a given role and there is a maximal-distance constraint for another role, then all NPCs that are too far are removed from that role's domain. If any

domain becomes empty then we backtrack immediately. Upon backtrack the domains of unassigned variables are restored to the state prior to the instantiation.

Since unary constraints are independent of assignment of other variables, values that violate them may be removed from the domains prior to search. This is called *node consistency* and it is actually a weaker form of forward checking, where only unary constraints are considered. There are more advanced inference techniques such as *arc consistency* where all the constraints between unassigned variables are taken into account. Such techniques are useful for hard combinatorial problems, which is not our case.

6.5 Implementation

The situations that were needed to enrich the virtual world in KC:D can be implemented as globally-specified events. Each NPC that should interact in the situation takes a *role*. Each role imposes requirements for an NPC to be allowed to take it. For example a situation called "Beggar", where a rich man shows contempt for a poor beggar and gives him some money has two roles: the beggar and the benefactor. Only poor NPCs may take the role of the beggar and only rich ones the role of the benefactor. Moreover the two NPCs must be close to each other to enact the situation promptly and safely.

The system assures that only NPCs satisfying all the requirements enact the situation. It further tries to provide variety by randomizing which of the acceptable NPC combinations are actually selected for the particular situation instance.

The central component of the situation system is the situation manager, which decides what situations should be enacted, when they should start and which NPCs should participate. Situation scheduling is driven by designer-specified minimal and maximal intervals between successive executions. Once a situation is scheduled for execution, the manager searches for suitable NPCs based on constraints associated with the situation. As in the "Beggar" example, there are unary constraints restricting the holders of individual roles by various traits of the NPCs (abilities, occupation, social status etc.) and n-ary constraints such as maximal distance between the NPCs or requirement for visual contact between the NPCs. Furthermore, each NPC keeps track of situations it has been involved in recently and thus designer-specified intervals between successive executions of situations with the same NPC can be enforced.

The full lifecycle of a situation, once the role holders are decided, was already given in Section 5.5.7. To quickly reiterate the main points: situations are a type of a behavior object, i.e. the code and data for a given situation are defined in one place. The NPCs have to explicitly subscribe to the situation system, letting scripters to ensure, that an NPC does not participate in a situation when it is performing a critical task. The situation scripts are injected within the participating NPC's script selection function through a special subbrain that, once the situation is started, takes priority over the active ambient subbrain, but has lower priority than combat and other subbrains. If any of the NPCs leave the situation (e.g., their script fails or the situation subbrain is suspended due to a higher-priority subbrain being switched in), all NPCs terminate their situation script and the situation itself is discarded. All the above precautions were made because situations in KC:D are only decorative and should not prevent any important scripts from executing. On the other hand, situations should be designed in a way that interrupting them causes no problems –

especially, situations should not change the state of the game world in any important way.

6.5.1 Our CSP Solvers

We tested a set of CSP solvers based on backtracking. The chosen solver is invoked directly by the situation manager whenever a situation is scheduled and is updated independently of the individual NPCs. Since our CSPs are small, it did not make sense to implement a sophisticated CSP solver with complex inference – the overhead would easily cancel out the slight gain in search performance. Instead, we were adding inference techniques one by one to see, when the performance stops improving.

We started with the basic backtracking algorithm (see Algorithm 1). Since variety is important, the solver should not return the same solution upon repeated execution with the same data (if there are multiple solutions). Thus the instantiation at each level starts at a random element of the domain.

```
procedure BackTracking(X:variables, V:assignment, C:constraints)
        if X={} then return V
        x ← head(X)
        for each value h from x.domain do
                if constraints C are consistent with V ∪ {x/h} then
                        R ← BackTracking(X – {x}, V ∪ {x/h}, C)
                        if R ≠ fail then return R
        end for
        return fail
end BackTracking
```

Algorithm 1: Plain backtracking.
To initiate the recursion, the algorithm is called as BackTracking(X, {}, C). Algorithm pseudocode adapted from (Barták 2015).

Then we added node consistency (NC), i.e. all NPCs that did not match the unary conditions were removed from the domains before the search. Surprisingly initial results showed this approach to be often worse than plain backtracking. Closer analysis revealed that on the smaller domains the time spent in evaluating all the unary constraints dominated the time in the actual search by factor of two to ten. Thus we introduced lazy NC evaluation: the unary constraints are tested only on the values that are actually searched, but if the condition is not met, the value is removed from the domain permanently, i.e. it is not reinserted upon backtrack (see Algorithm 2). While lazy approaches to consistency are known in the literature (Schiex et al. 1996) we are not aware of any applications to node consistency.

```
procedure BT-LazyNC(X:variables, V:assignment, U: unary constraints,
    C:constraints)
        if X={} then return V
        x ← head(X)
        for each value h from x.domain do
                if not x.domain.tested[h]
                  && h inconsistent with any u ∈ U then
                        x.domain ← x.domain – {h}
                        if x.domain is empty then abort
                        x.domain.tested[h] ← true
                end
                if constraints C are consistent with V ∪ {x/h} then
                        R ← BT-LazyNC(X – {x}, V ∪ {x/h}, U, C)
                        if R ≠ fail then return R
        end for
        return fail
end
```

Algorithm 2: Bactracking with lazy node consistency.
The algorithm is a slight modification of the basic backtracking algorithm, the additions are highlighted.

Last we added forward checking both with and without LazyNC (FC and LazyNC-FC, see Algorithm 3). With a straightforward implementation where contents of domains (arrays of NPCs) were simply copied prior to instantiation of variables and restored upon backtrack, the algorithm fared worse than lazy NC only. An implementation trick was needed to make the algorithm competitive: Pruned values were not removed from the domains, but kept at the beginning of the domain and an internal pointer to the first domain element that was not pruned was kept. This way, only the pointer needed to be changed on backtrack and still permanent removal of values from the domain due to lazy NC was possible in constant time (copying the last element over the removed element). See Figure 10 for a diagram.

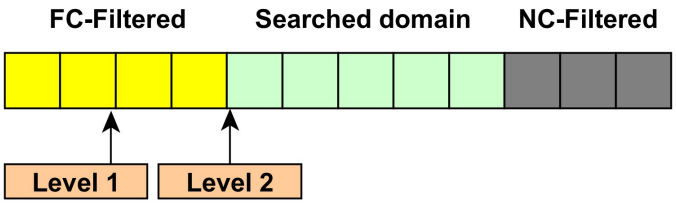


Figure 10: Domain contents while using forward checking with lazy node consistency.
The elements filtered by lazy NC are moved to the end of the vector, and the size of the vector is permanently decreased. The elements filtered by FC are moved to the start of the searched domain and the searched domain start at current level (here level 2) is increased. Upon backtrack, the searched domain start is restored to the value at the previous search level (choicepoint), returning all FC-filtered elements from the last search level to the domain.

```
procedure ForwardCheck(X:variables, V:assignment, U: unary constraints,
    C:constraints)
        for each x from X
                for each value h from x.domain do
                    if UseLazyNC and h is not consistent with any u ∈ U then
                            permanently remove h from x.domain
                            if x.domain is empty then return fail
                    else if constraints C are not consistent with V ∪ {x/h} then
                            temporarily remove h from x.domain
                            if x.domain is empty then return fail
                    end
                end for
        end for
        return true
end

procedure BT-FC(X:variables, V:assignment, U: unary constraints, C:constraints)
        if X={} then return V
        if not UseLazyNC
                remove values of all variables inconsistent with U
                if any domain is empty then return fail
        end
         create choicepoint
        if ForwardCheck(X, V, U, C) = fail then
                backtrack
                return fail
        end
        x ← head(X)
        for each value h from x.domain do
                if constraints C are consistent with V ∪ {x/h} then
                        R ← BT-FC(X – {x}, V ∪ {x/h}, U, C)
                        if R ≠ fail then return R
                end
        end for
        backtrack
        return fail
end
```

Algorithm 3: Backtracking with forward checking.
UseLazyNC is a Boolean parameter specifying whether unary constraints are enforced eagerly (at the beginning of the search) or lazily during the search. The differences from the plain backtracking algorithm have been highlighted. Algorithm pseudocode (except for the incorporation of Lazy-NC) adapted from (Barták 2015).

We did not implement arc consistency (AC), because it requires keeping explicit track of pairs of values in unassigned variables. Since NC was already slow and allocating and updating linear amount of space was troublesome for FC, it was not likely that working with quadratic amounts of data could improve performance.

We have not implemented backjumping, because in all situations the designers brought up, there were only one or two non-unary constraints both involving all

variables. Thus most backtracks were caused by assignment of variables "close" to the current variable and it was not likely that significant benefit could be gained.

To summarize, we implemented and tested five variants of backtracking solver – plain, with eager NC, with lazy NC, with eager NC and FC, with lazy NC and FC.

In addition to explicit constraints given by the designers, an implicit all-different constraint (the same NPC cannot take more than one role) is encoded in the solver algorithms. The solvers are implemented in C++, the code of the solvers can be found in the digital attachment to this thesis.

6.6 Evaluation

To evaluate the situation system, we first gathered informal qualitative feedback and then performed qualitative evaluation to assess computational requirements of the system. At the time of the evaluation (April 2014), the development team included 6 scripters and 6 designers. All of the scripters had limited programming experience, while designers had close to none. Except for one scripter with mathematical modelling background, both designers and scripters had no experience in CSP or similar formalisms. Nevertheless, the idea of situations was relatively clear to both designers and scripters.

The designers found it natural to think in terms of situations, although there was some confusion on the capabilities of the system and its intended use. Most often, designers would propose situations that could break if a participant leaves it due to a higher priority event. To give an example: *"NPC falls into a trapping pit and someone comes to help"* – if the rescuer is disturbed, the NPC is stuck in the pit.

Another frequently mentioned issue was the impossibility to choose and constrain non-NPC game entities as a part of the search (e.g., find an NPC and a nearby water source where he could drink). This was left out as future work, once the system is proven suitable in practice.

Describing situations in terms of conditions on role holders was a straightforward idea for the scripters, but they had some difficulty to decide what should be modelled by unary constraints and what should be an n-ary constraint. For example, in the "Beggar" situation, the first idea was to not constrain the richness of the benefactor in a unary condition, but create a binary condition "benefactor richer than beggar", which would likely result in longer search times.

6.6.1 Quantitative Results

To field test the situation system, the design team of KC:D has proposed several situations that were implemented to test the system[11]:

- **Beggar**. A rich NPC passes a beggar and gives him money. The beggar displays gratitude.
- **Payment**. A peasant meets a rich man. The rich one demands money to settle a loan. The peasant is reluctant, but finally pays the requested sum.
- **Small talk**. Two peasants meet on a corner of the street and discuss the weather briefly.

[11] The actual scripts for the situations were developed by Martin Antoš and are thus not part of this thesis.

- **Lively argument**. Three peasants meet to argue about trade, taxes and the heir of the throne.
- **Dance**. A musician starts playing music on the street. Four peasants gather around and perform a group dance.
- **Difficult**. A non-natural representation of the "Dance" situation to test a slightly more extreme scenario (see below for details).

Formally, all of the situations have constraints on the occupation/social class of the participants, and distance/visibility constraints among all pairs of the participants. The "Difficult" situation is an exception as only 4 pairs of NPCs are affected by binary distance constraints. Still the constraint graph is connected and thus there are implicit "transitive" distance constraints over all NPC pairs, but those implicit constraints are inaccessible to the search algorithm and violations of those constraints are discovered only after assigning NPCs to the "intermediary" roles (see Figure 11).

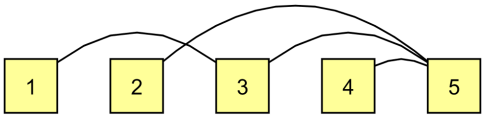


Figure 11: Distance constraint structure for the "Difficult" situation.
Note that while there is no explicit constraint connecting roles 1 and 4, we see that an NPC holding role 1 cannot be too far from the NPC holding role 4 (at most 2 times the maximum distance between the constrained pairs). Violations of this implicit constraint are however discovered by the solver only after assigning an NPC to role 5. Similar implicit constraints hold between other pairs of roles not connected by an explicit constraint.

The "Small talk", "Lively argument" and "Dance" further require all NPCs to be close to one of designer-marked areas suitable for situation enactment. "Payment" has a special condition for the peasant to check whether it makes sense for him to pay a loan. This condition is expressed as a short Lua (Schuytema and Manyen 2005) program and we expect it to be very costly to evaluate.

As discussed in Section 6.2, situations will not be scheduled in areas far away from the player and only a part of the NPCs will be subscribed to the situation system in any given moment, limiting the maximal number of candidate NPCs.

We created two scenarios. Scenario 1 models the expected workload and involves 50 NPCs: 2 musicians, 6 rich, 8 beggars and 34 peasants. The NPCs moved around the game world, sometimes performing a non-interruptible script; in effect 28 NPCs were registered for situations in an average search task (standard deviation: 4.7, max: 49, min: 14) – a screenshot of the setup is shown in Figure 12. Scenario 2 tests a heavier, unrealistic load to check the scalability of the system. It involves a larger world and 300 NPCs, divided in the same proportions as in Scenario 1, with 193 registered for situations on average (standard deviation: 24, max: 270, min: 149). Tests were run on an Intel i5-3470 quad core processor @ 3.2 GHz, with 8GB RAM. To reduce measurement noise, the solvers were run synchronously with the game engine and the engine was forced into single-threaded execution for the purpose of the test. In both scenarios, the situations (as described above) were scheduled on average once every 20 seconds.

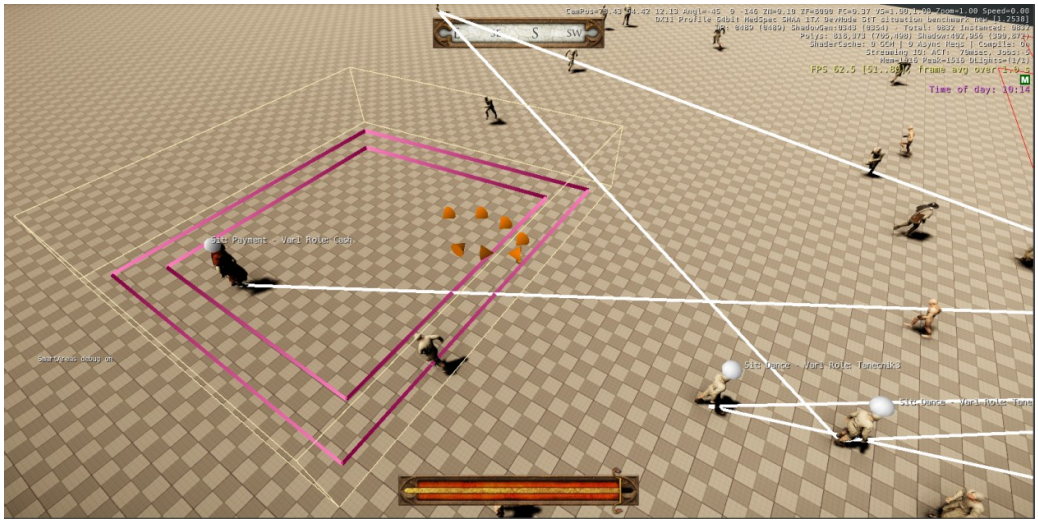


Figure 12: A screenshot from the quantitative evaluation setup.
White lines connect NPCs selected to engage in a situation, purple rectangles and orange cones indicate area and exact positions of NPCs that would perform the Dance and Small talk situations.

Both scenarios were run until at least 200 instances of all situations were executed. During the experiment all situations were run with the same frequency. All five backtracking solver variants solved exactly the same CSP instances. The full dataset of the results can be found in the digital attachment to this thesis.

Table 6 shows the ratio of successful searches in both scenarios. Since the search was never interrupted prematurely, failed searches represent problem instances that had no solution. The table shows that we tested both situations that always had a solution and those where solutions were rare and those in between, although the number of solvable assignments grew with more NPCs available.

	Scenario 1			Scenario 2		
Situation	**Success**	**Fails**	**%**	**Success**	**Fails**	**%**
Beggar	215	2	99%	222	0	100%
Small Talk	101	116	47%	205	25	89%
Payment	98	123	44%	157	69	69%
Argument	56	161	26%	172	52	77%
Dance	24	187	11%	93	134	41%
Difficult	45	166	21%	168	60	74%

Table 6: Number of successful and failed searches.

Situation	Back	NC	LazyNC	NC-FC	LazyNC-FC
Beggar	5 (10)	16 (4)	***3 (2)***	27 (5)	15 (2)
Small Talk	18 (10)	19 (5)	***14 (7)***	29 (7)	25 (5)
Payment	95 (52)	117 (27)	***87 (42)***	161 (37)	92 (41)
Argument	30 (22)	27 (9)	***20 (9)***	38 (10)	36 (8)
Dance	22 (25)	22 (19)	***17 (11)***	35 (21)	42 (8)
Difficult	245 (618)	38 (11)	***21 (08)***	51 (12)	55 (12)

Table 7: Average search times and standard deviations for individual situations and algorithms in Scenario 1.
Standard deviations are given in brackets. Best times for each row are highlighted. All times in µs. Scenario 1 consisted of 50 NPCs, ~28 registered at a time.

Situation	Back	NC	LazyNC	NC-FC	LazyNC-FC
Beggar	125	34	***12***	47	33
Small Talk	60	43	39	61	***36***
Payment	331	258	***205***	329	248
Argument	123	***55***	60	81	60
Dance	154	79	***55***	110	80
Difficult	4869	82	***53***	92	93

Table 8: Maximum search times for individual situations and algorithms in Scenario 1.
Best times for each row are highlighted. All times in µs. Scenario 1 consisted of 50 NPCs, ~28 registered at a time.

Situation	Back	NC	LazyNC	NC-FC	LazyNC-FC
Beggar	6 (3)	130 (18)	**4 (3)**	164 (26)	31 (4)
Small Talk	95 (92)	158 (20)	69 (45)	196 (27)	**52 (29)**
Payment	539 (396)	875 (173)	**466 (335)**	1104 (208)	482 (359)
Argument	213 (242)	235 (37)	119 (73)	271 (41)	**96 (59)**
Dance	435 (700)	306 (149)	179 (124)	345 (155)	**149 (67)**
Difficult	14ms (38ms)	366 (66)	**89 (38)**	408 (66)	136 (66)

Table 9: Average search times and standard deviations for individual situations and algorithms in Scenario 2.
Standard deviations are given in brackets. Best times for each row are highlighted. Unless explicitly stated, all times in µs. Scenario 2 consisted of 300 NPCs, ~193 registered at a time.

Situation	Back	NC	LazyNC	NC-FC	LazyNC-FC
Beggar	19	208	**16**	408	54
Small Talk	594	232	183	286	**171**
Payment	1816	1702	1586	1958	**1579**
Argument	1239	415	366	423	**305**
Dance	5557	584	600	686	**462**
Difficult	273ms	584	**246**	577	326

Table 10: Maximum search times for individual situations and algorithms in Scenario 2. Best times for each row are highlighted. Unless explicitly stated, all times in µs. Scenario 1 consisted of 300 NPCs, ~193 registered at a time.

The timing results for Scenario 1 are given in Table 7 and Table 8. The most important finding is that LazyNC performed the best for all situations in Scenario 1 and safely fulfilled the required time limits (recall the requirements: 100 µs on average and 1 ms in the worst case). The problem instances are just too small for FC to make a difference, although the LazyNC-FC variant fulfils the time limits as well. Solving "Payment" takes a remarkably long time, because it involves execution of an interpreted Lua function to evaluate a unary constraint which is costly. Interpreted constraints are discouraged for production use and should serve as a prototyping tool only. Still LazyNC and LazyNC-FC stayed within performance bounds.

In Scenario 2 (see Table 9 and Table 10) LazyNC still performs very well for all the 2-NPC situations. This is not surprising, as FC needs to scan the whole domain of the second NPC with every assignment of the first NPC during inference and then scans the remaining values again during search while LazyNC visits all the domain values at most once. In this regard it is unexpected that LazyNC-FC is better in the "Small Talk" situation. The reason is probably that the binary distance constraint is faster to evaluate than the unary constraint that checks whether the NPC is close to a suitable area, which is worst-case linear in the number of nearby areas (up to 5 in our scenarios). Some optimization of this condition might be useful.

The FC technique performs better also at the 3-NPC "Argument" and 5-NPC "Dance" where finally the domains are large enough for pruning to have notable effect. While in Scenario 1 the "Difficult" situation took more time on average than the "Dance" situation, most solvers solved the "Difficult" situation faster in Scenario 2. This is most likely due to the fact that in Scenario 2, "Difficult" had solutions much more often than "Dance" and successful searches were generally much faster than failed ones.

Combination of LazyNC for 2-NPC situations and LazyNC-FC for multiple NPCs situations satisfies the runtime constraints even upon heavy load, except for the "Dance" situation, where LazyNC-FC exceeds the average time limit by 49µs and "Payment" (which takes long due to the interpreted constraint). If the "Dance" situation is not scheduled frequently, the system as a whole would be fast enough. Notably, the search for NPCs is on the same time scale as pathfinding – instances in Scenario 1 are comparable to finding shorter paths and those in Scenario 2 are comparable to finding paths across large parts of the game world. This justifies the computing requirements of situation search, as pathfinding is one of the most frequent AI tasks, while situations should be relatively rare.

Further analyzing the successful and failed runs separately shows some deviations from the average case, but the overall picture remains the same. In general, plain backtracking ranks better in successful runs than on average, but is much worse in failed runs, while eager NC and NC-FC ranks better in unsuccessful ones than on average. This is most likely an intrinsic property of the algorithms: If there are many solutions, backtracking is likely to find one and it avoids the overhead of the other algorithms. Eager NC on the other hand is a pessimistic technique which fails quickly if there are no or very few values satisfying the unary constraints.

6.7 Discussion

We have introduced a system for global script selection for AI components with small rule but large size complexity. We have shown an implementation of this system that enriches a simulated world with short pre-scripted situations. Our algorithm uses CSP to search for appropriate NPCs and to keep the script selection function of the situation AI component hidden from the rest of the AI code. We have shown that the system meets functional and runtime requirements of a commercial game. The system is currently being evaluated for production use in KC:D and the initial response from scripters and designers was positive.

In recent years the advances in hardware allowed classical AI techniques such as planning to become part of mainstream game AI. We show that CSP may as well follow. In our view application areas of CSPs in games are open: CSP makes it possible to decouple various parts of the game and find appropriate connections (NPCs to situation role in our case) at runtime, maintaining an easily describable, consistent global view of the game world. It also has numerous applications in both offline and online procedural content generation. Instantiating side quest templates or detecting important abstract game events might be among the interesting future applications. CSP also maintains a high level of designer control – undesirable solutions are easy to remove by adding new constraints. We hope that this example will promote the use of CSPs in games, as even simple and fast-to-implement algorithms have satisfactory performance.

In further development of the system, we have introduced “passive” and “optional” roles. Passive NPCs are constrained in the search, but do not receive a script; they serve as a target of actions only (e.g., a situation where an NPC comments on another NPCs work). Optional roles on the other hand receive scripts, but the search may be successful, even when no NPCs are found for the role.

Future work includes extending the situation system to search for items, locations and other non-NPC entities in the game world as a part of the situation search.

7 Adversarial Search For Handling Large Rule Complexity

In this chapter we will deal with a different type of complexity than in the previous two: we will discuss the case of small size complexity, but large rule complexity. As script selection functions with large rule complexity are impractical to code by hand, this chapter will not be concerned with ways to structure code, but with a goal-based approach to choosing the right script.

We will start by discussing the motivation for the general approach (Section 7.1) and for the particular implementation for KC:D (Section 7.2) followed by related work (Section 7.3) and description of our method in general (Section 7.4). Then we discuss the actual implementation (Section 7.5) and its evaluation (Section 7.6). We end the chapter with some concluding remarks (Section 7.7).

		Rule complexity	
		Small	**Large**
Size complexity	**Small**	Basic enemy AI	**Quality enemy AI** Ally AI
	Large	Ambient AI Dialog-handling AI	Strategic squad coordination High-level RTS AI

Table 11: Complexity classification for the techniques in Chapter 7.
This chapter deals primarily with large rule complexity and small size complexity in general, our implementation is focused on enemy AI in particular (highlighted). This table is a reiteration of the complexity classification from Table 1.

7.1 Motivation for the General Case

A typical AI component that may exhibit small size complexity but large rule complexity is combat AI – there are relatively few basic actions (attack, retreat, advance, take cover …), that can be combined in many ways to create a varying and challenging gameplay. As noted in Section 1.2, large rule complexity may be either inherent in the combat mechanics when the combat is more strategic (as in the Fallout series) or it may arise simply due to the necessity to take a large number of parameters into account when making a decision. The latter case arises quite frequently in games with role-playing mechanics as those typically include a large

number of parameters that influence the combat, often in an indirect, interwoven manner. Most of the parameters in combat in OWGs take the form of *stats* – the numerical properties assigned to characters and items in the game that reflect their various qualities (health, agility, attack, defense, …).

To avoid dealing with large rule complexity, combat AI in contemporary OWGs is often purely reactive with small rule complexity. Although reactive approaches can provide a reasonable combat AI, they do not scale well once more intelligent behaviors are required. Also NPCs with different properties require different reactive scripts, increasing the size of the codebase. At this point a goal-based approach may both allow for more intelligent behavior and reduce the size of the codebase, as a single algorithm may produce very different behaviors based on the actual stats of the individual NPCs.

There are however significant downsides shared by many goal-based approaches that have to be addressed before they may be deployed in an actual game. The two most important are the high computational requirements and the black-box nature of deliberation.

As for the computational requirements, one of the simplest goal-based reasoning techniques, classical planning, is already PSPACE-complete in theory and very demanding in practice. During our evaluation of state-of-the-art planners in game environments, we estimated that "The limits of real-time applicability (planning faster than 1s) of contemporary planners are somewhere above one hundred atoms and two hundred grounded actions" (Černý et al. 2015). Note that these are still relatively small domains and the budget of 1 second is unrealistically high for most real-time games. While planning performance can be hugely improved by writing an optimized planner for the particular game domain, classical planning does not capture many concepts of typical OWG combat AI – in particular stochasticity and the need to consider the actions of the adversary – which further inflate the search space. On the other hand, the computational requirements can be further reduced, because OWG AI does not usually need optimal actions, only actions that are "good enough". All in all, any goal-based technique to be used in games has to be heavily optimized and make it possible to do trade-offs between optimality and efficiency.

A much harder problem to address is the "black box" behavior of most goal-based techniques. Black box in this context means, that the algorithms receive the current state of the world and choose a script to execute, but expose little or no information on why the script was chosen. Without understanding why a given script was chosen, it is hard to debug and tune the script selection. This is in stark contrast to reactive approaches, where introspection is usually very easy and tuning the script selection for a specific case is simply a matter of providing a condition recognizing the case. It is also harder to incorporate design goals into a problem specification for a goal-based technique. Any goal-based technique to be used in games thus has to support debugging and allow designers to provide "manual overrides" for specific cases.

A further downside of a black-box approach is that it makes it hard to access and expose internal state of the NPCs decision making. Exposing internal state is important, because in some cases it is hard for the player to distinguish between a rich and well-motivated behavior and random actions. This is greatly alleviated if the inner state of the NPC is communicated to the player. A nice documented example of this phenomena is given in (Orkin 2006). Orkin describes how NPCs in F.E.A.R. let the player know whenever the planner decides it is optimal to perform an action that may be considered stupid (e.g., not taking cover) by shouting out that they have no better option. Further, squad coordination is displayed by NPCs asking each other

questions or giving orders. We therefore require our solution to expose at least some details about the NPC's state.

7.2 Motivation for the Specific Case of KC:D

KC:D involves melee combat with swords and other medieval weapons. The combat was designed to allow for rich combinations of fighting style, weaponry and armor composition. There are over 40 parameters per NPC that meaningfully affect the combat mechanics, making it difficult to script a behavior that takes them all appropriately into account.

At the same time, the design team of KC:D wanted the combat to be a relatively rare but challenging experience. The obvious caveat is that more challenging does not necessarily mean more fun and that combat difficulty could easily be increased by simply increasing the opponent's stats. But the design asked for the strongest opponents to be difficult to conquer – at least in part – because they behave intelligently.

The large parameter space and demand for high-quality decisions makes a goal-based approach desirable. Since the combat is largely influenced by chance events, and the result of most actions is contingent on the actions of the opponent, the system should explicitly take chance and opponent decisions into account.

The system however cannot take full control of the NPC, as scripters sometimes need a very tight control over combat behavior (especially in connection with quests). Therefore the component must be well integrated with the BTs used in KC:D and allow scripters to invoke it only when there is no gameplay-critical action to be performed. The system should thus act as more of an advisor on what to do in a typical situation than a complete controller of the NPC.

For the sake of the investigation in this thesis, we limit our focus to one-on-one combat as one-on-one scenarios are easier to formalize and thus better suited for initial exploration of possible solutions.

7.3 Related Work

While goal-based and search-based techniques have been employed since the early research on AI for board games, their introduction to real-time games took a lot longer. In this section we will first discuss the current usage of goal-based techniques in the game industry and then review recent advances made in academia that have not yet been adopted by the industry. Last we note recent progress on tackling high rule complexity by developing agents that learn to play the game.

7.3.1 Goal-based Techniques in the Game Industry

The use of goal-based techniques to reduce complexity in NPC decision making in commercial real-time games can be first traced to the use of planning in F.E.A.R. (Orkin 2006). F.E.A.R. introduced GOAP (Goal-oriented action planning), which is a planning system derived from STRIPS (Fikes and Nilsson 1971) with multiple adjustments for use in a game. This pioneering work also makes explicit claim that planning is intended to primarily reduce complexity and increase manageability of code rather than to increase perceived intelligence. Further the GOAP system used in F.E.A.R. allowed the designers to decouple abstract actions (e.g., "attack", "retreat")

from their actual implementation. So although a beast would have the same "attack" action as a soldier, both would run very different scripts to execute it in the world.

Planning with Hierarchical task networks (HTN) formalism (Ghallab et al. 2004) has also been used in games. To our knowledge, it was brought to mainstream games by Killzone 2 and 3 (Champandard et al. 2009; Straatman et al. 2013). In Killzone 2 and 3, HTN is used to both select actions of the individual NPCs and to assign roles within squads of enemies.

Planning techniques derived from both the GOAP and HTN formalisms have since become commonplace in games. There are hints that HTN planning is favored in new games, because it leaves the designers with more control than GOAP (see Champandard 2013).

A common property of all planning implementations in games is that they do not explicitly model opponent's decisions and actions. They mostly assume that the world is static except for the NPC/squad of NPCs the system plans for and represents enemy threats by adding high cost to actions where the NPC can be hurt. The obvious reason is performance: even for a static world, the search is very expensive and considering enemy actions would effectively square the branching factor and preclude guiding the search with a heuristic function. Also, planning techniques have proven to work reasonably well, once good cost functions and heuristics are implemented.

More recently, Monte-Carlo tree search (MCTS) (Browne et al. 2012) has been employed in strategy games, but in both cases we are aware of it was employed as an optimization technique, without explicitly considering opponent's actions. We will discuss the following applications in greater detail, as their description is not freely available (unlike the rest of the works discussed in this chapter).

In particular, the recent iterations of the Total War strategy series have used a two-layered MCTS to choose its actions for its turn-based campaign gameplay (Gosling and Andruszkiewicz 2014; Andruszkiewicz 2015). Although the setup is turn-based, the game mechanics are much more complex than in most board games and the implementation imposes very strict time limits, making the work relevant to real-time approaches. Here, a high-level MCTS chooses a set of non-conflicting goals and a lower-level search then handles the tactical aspect, giving orders to individual units. Although the campaign part of Total War is not real-time, the AI still needs to make its decisions relatively quickly, enforcing some interesting optimizations. First, the search looks only one turn ahead and instead of explicitly considering enemy's actions it uses a simple worst-case heuristic: every army/settlement that can be reached in the next turn by enemy units of total strength larger than strength of defenders will be destroyed. Second, it heavily pre-caches pathfinding results for the units and involves aggressive pruning of the search tree. Third, it ignores randomness present in the game: actions with probability above a certain threshold are considered to be automatically successful, others as failed. If any of the actions that were expected to succeed fails during execution, the AI creates a new plan. To preserve as much opportunities as possible for the eventual replanning, the actions that may fail are executed as early in the turn as possible.

An application of non-adversarial MCTS planning to a real-time squad coordination was done for Fable Legends strategy game (Mountain 2015). Here, the motivation was once again the complexity of the problem – a reactive approach would not scale to the complex gameplay. The authors also note that since MCTS runs over an abstracted model of the game, it automatically adapts to changes in game design – only the abstract model needs to be updated. The algorithm assumes

that the opponent's squad remains static and tries to find sequence of script assignments to all units, simulating the combat until a defeat/victory. Once the scripts are chosen, they are realized in-game as behavior trees, allowing for certain runtime flexibility. Considering only the basic scripts, the branching factor is approximately 40, which prevented the algorithm from finding good solutions at the beginning of a skirmish where there are a lot of possibilities, while only few actions have direct impact on health or other important properties and terminal states lie in large depth. So at the beginning of a skirmish, they choose macro actions that represent more long-term strategies to be assigned to NPCs.

An interesting problem the developers of Fable Legends had to deal with was that once the AI faces an almost certain defeat, all action assignments have almost the same value and thus the AI starts behaving randomly. To counter that, if there are multiple actions with value close to the maximal value, they choose among those top-valued actions using designer-specified priorities. Since attacking has high priority, squad facing a certain defeat will preferably attack the opponent, making him weaker for the next skirmish and also appear as deciding to perform a desperate last attack.

To optimize the search, the system precomputes all paths and visibility over a simplified map of the world and there is a minor pruning of the search based on designer-specified rules. In the end, the system is able to perform 10 000 iterations in approximately 5ms when running on its own and 10 000 iterations in 5 frames (~150ms) when running within the game. Depending on the dynamics of the current situation, the time to make decisions is kept roughly between 5 frames and one second.

Another interesting feature of the Fable Legends AI is its debugging support. To understand the algorithm, a designer may stop the game, let the search run multiple times from the current state and then explore statistics over the set of results (i.e. in this case AI will choose to attack in 80% of the searches).

The main difference between the present work and the above uses of MCTS is that we explicitly model the opponent and chance events.

The only use of a truly adversarial search schema in the game industry we are aware of is in Frozen Synapse (Hardingham 2012). While Frozen Synapse is turn-based at the highest level, it shares many features with its real-time counterparts. In Frozen Synapse, each side controls a small squad of soldiers in a continuous space with obstacles that provide cover. Each side assigns orders to all of its soldiers for the next 5 seconds. Once all orders are placed, the environment is simulated in continuous time and the orders are executed simultaneously without further input from the players. The environment is fully deterministic.

The AI in Frozen Synapse works in two phases: strategy generation and strategy selection. The strategy generation phase consists of a set of steps, each creating multiple candidate best-response strategies for one side. Initially, it is assumed that the opponent soldiers choose an empty strategy (soldiers remain stationary). Several different best responses to this strategy are found using a classical search algorithm over a simplified version of the environment (a hand-crafted scoring function is used to evaluate the final state). Then a set of best-response strategies for the opponent against each of the strategies from the previous step is found. These steps are repeated for a given number of iterations and a method for discarding some of the strategies is implemented to prevent explosion of the number of strategies. At the end of the generation phase, we have a set of candidate strategies for both sides. The strategy selection phase than consists of running pairs of the AI's and opponent's

strategies against each other. The original pool of strategies is enriched with combinations of good performing strategies (part of soldiers get orders from one strategy, the rest from a different strategy) in a manner similar to a genetic algorithm. The AI then selects the strategy that wins over the largest fraction of opponent's strategies.

While the approach in Frozen Synapse is interesting, it targets a very different scenario that focuses on squad coordination and is not real-time.

7.3.2 Goal-based Techniques for Games in Academia

Unlike the game industry, there is a considerable amount of academic works on using adversarial goal-based techniques in real-time games, mostly focusing on real-time strategy (RTS) games.

A Monte-Carlo tree search variant called upper confidence bound on trees (UCT) has been employed for tactical coordination of units in an RTS game Wargus (Balla and Fern 2009). Later, variant of Alpha-Beta was developed for combat scenarios of up to 8 vs 8 units in Starcraft (Churchill et al. 2012) – the work augmented Alpha-Beta to handle actions of varying durations and to handle simultaneous decisions. The authors then improved on this approach to tackle scenarios with up to 50 vs 50 units, by limiting the search to assignments of longer-duration scripts and playing out the whole scenario with the assigned scripts – i.e. actions for both players consist of assignment of a script to each unit and only 2-moves look-ahead is performed (Churchill and Buro 2013).

Assigning scripts instead of actions proved to be useful to reduce branching factor. To allow for longer look-ahead the branching factor was further reduced by clustering the units and giving all units in a cluster the same script (Justesen et al. 2014). Nevertheless, the technique still focused only on combat scenarios. Most recently, (Barriga et al. 2015) show a more general approach called "Puppet search," where a whole game of Starcraft is represented as a game tree where the choice points are either a selection of a script (scripts control the game fully in this case, not only a single unit as in the above works) or further choice points exposed as needed by the scripts themselves. For example, a choice point may let the player choose either a "defensive" script, that moves all units into base and builds defensive buildings or an "all-out attack" script that orders all units to attack. The "all-out attack" script may then expose a choice point to decide which of the enemy bases to attack first.

In a related line of research, (Buro and Ontañón 2015) tested an adversarial HTN approach to finding scripts at multiple levels of abstraction to play a full Starcraft game.

While all of the above works use algorithms that provided inspiration for our approach, RTS games are a very different use case than OWGs and thus the techniques cannot be applied directly.

7.3.3 Learning Reactive Controllers for Games

A complementary approach to handle large rule complexity is to still use reactive representations, but learn or evolve the actual reactive script instead of coding it by hand. A very basic approach in this regard is dynamic scripting. In dynamic scripting a simple ranking-based script selection function over hand-crafted scripts is learnt online to adapt to changing player's strategy (Spronck et al. 2006). Dynamic scripting has been applied in multiple game scenarios (e.g., Policarpo et al. 2010;

Ludwig and Farley 2009; Toubman et al. 2014), including an OWG (Ludwig and Farley 2008).

Earlier works on evolution in game context have seen evolutions of rule-based script selection functions (Small and Congdon 2009) or fuzzy finite state machines (Esparcia-Alcazar et al. 2010) for a shooter game Unreal Tournament 2004. Later, attempts have then been made at evolving neural network controllers for bots in a shooter game (Schrum and Miikkulainen 2010; Schrum et al. 2012; Van Hoorn et al. 2009).

A two-level learning approach has been employed for capture-the-flag scenario in a shooter game (Hefny et al. 2008) – reinforcement learning is employed for a commander of the squad, while neural networks, trained on human play-traces are used to learn a low-level script selection for movement and shooting for individual NPCs. More recently, (Wang and Tan 2015) use reinforcement learning to learn a neural network controller for a shooter game.

A more thorough review of uses of machine learning in various game genres is given in (Muñoz-Avila et al. 2013).

Recently, huge progress has been made on agents that can learn to play video games on their own from purely visual input, using various neural network architectures. It has been shown that computers can learn to play some Atari 2600 games better than human experts (Mnih et al. 2015), reasonable success has also been achieved in car racing environment and with navigating a maze in 3D, similar to the Wolfenstein game (Mnih et al. 2016). Attempts have even been made at learning to play text-based adventure games (He et al. 2015).

There are two general downsides to most learning and evolutionary approaches to game AI. The foremost is that very few learning and evolutionary approaches result in controllers that are measurably better than what could be achieved within a few weeks of hand-coding. Since implementing a learning algorithm represents considerable effort of a skilled programmer, the trade-off is often unfavorable. The second large downside is that most learning and evolutionary representations that are capable of representing at least a mildly competent agent (especially variants of neural networks) are completely opaque from a debugging perspective.

7.4 Our Solution – Adversarial Search

As testified by the academic research on RTS games, modern adversarial search techniques are close to being applicable to real-time games. Search can thus possibly replace components with large rule complexity. Although it is possible to envision other use-cases, adversarial search is mostly suited for combat AI. We will thus conduct further discussion only in the scope of combat AI.

Similarly to F.E.A.R. one of our primary aims is to reduce complexity: once there is a huge number of parameters that have impact on what the optimal action is (e.g., armor and weapon stats of both combatants and multiple other RPG stats in the case of KC:D), it is difficult to create a robust scripted behavior. Adversarial search also allows for some interesting tweaks to the behavior – the AI can for example be made risk seeking by adjusting the way chance nodes are handled (it would assign higher weight to favorable outcomes). Other possible changes include an optimistic setting where the model used in search assumes higher damage dealt to opponent and lower damage received, or an aggressive setting where the evaluation function puts more weight on opponent loosing health than on the NPC retaining health. Tuning in the

opposite direction will make the NPC more risk averse, pessimistic or defensive respectively.

Further, search (unlike most learning approaches) is amenable to at least partial debugging solutions – at the very least, the search tree may be visualized and by inspecting the states at individual nodes, a designer may gain insight on why an action was chosen.

Search algorithms also allow for a limited exposure of NPCs internal state and some interesting tweaking. Important data that can be communicated to player can be extracted from the search tree and node values computed by the search algorithm. For example, the value of the root node may be useful to decorate NPCs combat behavior – a high value at the beginning of the combat justifies trash talk on the NPCs part, if the value decreases throughout the combat, it signals that the NPC has underestimated the opponent (or has bad luck) and it should replace trash talk by acknowledging the opponent's strength. If the state value drops too low, the NPC may decide to flee or beg for mercy. As a different example, analysis of node values further down the tree may reveal what action the AI fears the most the opponent will do, letting the NPC to swear or scream in terror, if the opponent chooses this action.

Using adversarial search however requires the AI component to feature small rule complexity (to keep the branching factor manageable) and the relevant game mechanics have to allow for an abstract model that both maintains reasonable accuracy and can be evaluated very quickly. Since planning techniques have very similar requirements and they have been employed in multiple games, we can conclude that the only factors that prevent adversarial search to be employed more often are either design decisions (planning is good enough for the particular case) or speed of execution.

7.4.1 Game Trees for Combat in OWGs

In OWGs, there are rarely more than two sides to a combat scenario and these two sides have usually strictly opposing goals. This enables us to model the combat accurately as a two-player zero-sum game. To prevent confusion between the two meanings of the word player, we will use player to denote the human interacting with the computer and use the term *side* to refer to the participants in the abstract representation of the combat scenario. We will use the common terminology and refer to the sides of the combat as *min* and *max*, where max is the NPC performing the search and trying to maximize its reward. The nodes in the tree that represent choices for the individual sides are *min nodes* and *max nodes*.

The simple game trees that represent zero-sum board games are however no longer sufficient and need to be slightly extended. Foremost, OWGs are not turn-based, actions are executed in real-time and have durations. Following Churchill et al. (2012) we assume actions are not interruptible (which is not always the case, but is often a reasonable simplification), letting us to place choice points for either side whenever the currently executed action ends. It naturally follows, that the children of a min node now do not have to be max nodes, and that all children of a node may not be all of the same type. Note however, that this irregularity of the game tree does not affect the way state value is propagated from the terminal states to the root. Considering the ideal case, when we can construct the whole tree up to terminal states, the value of a min node is still determined as the minimum of the values of child nodes and the value of a max node is still the maximum of the values of the child nodes.

Further, both sides may be able to act at the same time, giving rise to *simultaneous nodes*, where both sides pick actions independently, without knowledge of the choice of the opponent. The children of a simultaneous node thus correspond to pairs of actions. In most cases, the root node of the search tree will be a simultaneous node. With simultaneous nodes, the game is no longer sequential – its extensive form representation now requires imperfect information (although to a limited extent). For a more thorough game-theoretic discussion of this type of games see for example (Bošanský 2014), here we will focus only on the main properties. Simultaneous nodes are conceptually very different from max or min nodes. For optimal decisions in a simultaneous node we need to compute the Nash equilibrium of a game where the payoff matrix is given by the values of the children. Although we are considering only zero-sum games, finding the exact value of a Nash equilibrium translates to solving a linear program of a size proportional to the number of actions available, which may be very costly (average running time of all known algorithms for linear programming is a higher-order polynomial of the program size). The value of the simultaneous node is then determined as the expected reward when both sides follow their equilibrium strategy. However, as we will see in the following sections, approximating Nash equilibria with a computationally less expensive method is often sufficient for a good level of play.

Last, games contain events that are governed by randomness, so we need *chance nodes*, where each child is assigned a probability and the value of the node corresponds to expected value after the random effect, i.e. an average of the values of the children weighed by their probabilities. Chance nodes are also useful for representing mechanics that are deterministic in the actual game but are difficult or too expensive to compute in the abstract model (e.g., results of a physical simulation).

In the following two subsections, we will dig into details of the two most common algorithms for adversarial tree search: Alpha-Beta and Monte-Carlo Tree Search (MCTS) and discuss how simultaneous decisions and chance nodes can be handled in both algorithms. These algorithms and their extensions are state of the art in real-time search-based reasoning and will be implemented for our use case in KC:D. Further, we expect the algorithms with the extensions discussed below to generalize well to other domains and use cases.

7.4.2 Alpha-Beta and its Variants

The Alpha-Beta algorithm (Russel and Norvig 2010) dates back to early AI attempts to make strong AI agents for classical board games and eventually succeeded to win over human champions in Checkers, Chess and many other board games.

The quality of the decisions made by the algorithm depends critically on the quality of the heuristic function used to evaluate leaf nodes.

Two of the most common improvements to Alpha-Beta are *transposition table* and *iterative deepening*. Transposition table (Greenblatt et al. 1967; Marsland 1986) serves as a simple memory for states encountered previously during the search and enables us to reuse computation if a state is encountered multiple times during the search. Once a value is determined for a node in the search tree or the value of the node is proven to lie outside the alpha-beta interval, this knowledge is stored in a hashtable along with the height (distance to the leaf nodes) the value was computed at and the move with the best value found. Then, when a new node is expanded, and the state can be found in the transposition table there are two basic cases. If the

stored value was computed for the same or larger height then the search directly updates the alpha-beta interval with the value stored, possibly causing immediate pruning. Otherwise, the stored best move is tried first, as it is expected to quickly shrink the alpha-beta interval.

There are many variants of transposition table implementations, differing in details. For example, since the size of the table is limited, there needs to be a policy to handle collisions, i.e. how to decide whether to overwrite an older entry that would come at the same place as a newly stored entry. The options here include "always overwrite", "never overwrite", "keep the node where the search evaluated more nodes downstream", "keep the node with larger height" and many others. The actual hash function used is also of great importance as most implementations do not check for equality of the actual positions but only for equality of the hash. Since checking position equality is usually costly, this greatly improves performance in exchange for a small probability of error when two different positions have the same hash.

Alpha-Beta is typically run in an iterative deepening scheme, i.e. searches are run with increasing depth, until the time runs out and the result of the deepest fully completed search is returned. Iterative deepening gives the algorithm the anytime property – it can be terminated at any time and always yield the best estimate available at the time. It also has a small cost – the time spent in search is dominated by the time spent in the deepest search and so searching the lower depths has small penalty. In connection with a transposition table, the scheme is even more powerful, as the transposition table keeps some of the information from the lower depth searches and improves move ordering in the following iterations. With a transposition table, iterative deepening to a given depth may actually be faster than a single search of the same depth.

As for handling chance nodes in Alpha-Beta, the most basic method is to evaluate all children of the chance node and propagate the average of their values weighed by their probability as in ExpectiMiniMax (Russel and Norvig 2010). If the values of the subtrees can be bounded, it is possible to apply *-minimax (Ballard 1983). *-minimax prunes the search once the value of the node will lie outside the alpha-beta interval even if all the remaining children have the extreme values. Methods to use transposition table information in chance nodes have also been proposed (Veness and Blair 2007).

There are two families of approaches to handling simultaneous decisions in variants of Alpha-Beta: those that are guaranteed to find the optimal Nash equilibrium strategy and those that give up optimality guarantees in exchange for increased speed. To the best of our knowledge, state of the art optimal solvers for games with perfect information and simultaneous decisions are those developed by Bošanský in his thesis (2014) and the SMAB algorithm (Saffidine et al. 2012). Nevertheless, all those algorithms need to solve linear programs for each simultaneous node and thus the running time of the algorithms is on the order of seconds or even much larger for games with branching factors 6 or more. The fastest result reported is for Goofspiel with four cards (maximal branching factor 4, 4 steps look-ahead) where the solving time is reported at around 300ms in the work of Bošanský (our guess from a graph given in the thesis) and 12 ms for SMAB. While 12 ms would be acceptable in many use cases, computer games typically require both higher branching factor and longer look-ahead than Goofspiel with four cards. Further, computer games scenarios rarely have the high level of interdependence between simultaneous decisions manifested in Goofspiel, so trading optimality for

higher speed or longer look-ahead would usually be advantageous. Another practical disadvantage is the high implementation cost of the optimal algorithms.

The non-optimal but faster approach is based on serializing the simultaneous decisions into sequential decisions with delayed effects – the actions are chosen sequentially, but the effects of the actions are applied simultaneously once both sides have chosen an action. This was first suggested for an algorithm called RAB (randomized Alpha-Beta), where the results of multiple random serializations are averaged to approximate the true value of the game (Kovarsky and Buro 2005). For real-time performance in more complex domains, only a single such serialization is chosen (Churchill et al. 2012). The latter work was evaluated in the context of small combat scenarios in the strategy game Starcraft and the algorithm was shown to yield good results in under 5ms, making the approach useful also for our case. It is important to note that part of the reason this simplified approach is successful is that scenarios arising in computer games usually do not have strong interdependence between the simultaneous moves and thus the value of the sequential game is often a good approximation of the true value and it can be obtained with much smaller computational expenses.

Other notable approaches to serializing simultaneous decisions include taking the "paranoid strategy" and always make the max side move first, providing a lower bound on the value of the state (Schiffel and Thielscher 2007) or modelling opponent's decisions in a simultaneous node as uniformly random (Kuhlmann and Stone 2006).

7.4.3 MCTS and its Variants

Although Alpha-Beta proved to be successful in many board games, other games – most notably Go – remained very difficult for Alpha-Beta and alternative algorithms have been considered. MCTS was brought to mainstream attention in game AI through a variant called upper-confidence bound on trees (UCT) designed for a Go playing program Crazy Stone, which proved to be very successful (Coulom 2006). The MCTS algorithm starts with a tree consisting only of the initial state and then iteratively and asymmetrically builds the search tree. MCTS consists of four steps which are iterated until the available time or memory is exhausted:

1. *Selection* – starting at the root node, the algorithm selects the "most promising" action to take. If the node has a child corresponding to the selected action the procedure repeats for the child. Selection ends once it selects an action for which there is no corresponding child in the tree. The method to determine what "most promising" means is the main dimension along which variants of MCTS differ.
2. *Expansion* – a new child, corresponding to the selected action is added to the tree.
3. *Playout* (also called *Default policy*) – the game is simulated from the new child using a fixed policy for both sides until a terminal state is reached. A common form of the default policy is to simply choose actions uniformly at random.
4. *Backpropagation* – the result of the playout is propagated from the newly expanded child up until the root node is reached, updating the statistics that affect the selection phase.

The most important feature of MCTS is that it explores the tree asymmetrically, allocating more time to the branches that seem promising. Another important feature

is that MCTS does not depend on a heuristic function to assess the values of leaf nodes. It is the game rules themselves that provide estimates of node value – following the idea that in a good position there are more ways to win than to lose which should be picked up by the repeated playouts.

UCT is the most frequently used variant of MCTS. It was innovative by using upper-confidence bound formula UCB1 originally developed for k-armed bandit problems (Auer· et al. 2002) in the selection phase to choose the best nodes in MCTS. The algorithm selects the node that maximizes the value:

$$UCB1(i) = \frac{s_i}{n_i} + c\sqrt{\frac{\ln n_{parent}}{n_i}}$$

Where s_i is the sum of rewards encountered so far in the subtree rooted at child i, n_i is the number of visits to child i and n_{parent} is the number of visits to the parent node. c is a parameter of the algorithm called *exploration factor* – higher c results in more exploration, as the second term in the formula is larger for children that are under-explored. Lower c on the other hand results in more exploitation as more effort is dedicated to the nodes whose past results were promising. In theory, $c = 2$ ensures asymptotically lowest possible expected regret if the rewards at terminal nodes are identically and independently distributed in the interval [0, 1] (Kocsis et al. 2006). Since this assumption is hard to check and is often violated in practice, the value of c is usually determined empirically to maximize performance in the actual settings. Ties among children are broken randomly and children that have never been explored ($n_i = 0$) are considered to have a UCB value of ∞, so unexplored children are always selected first.

For general games, the current understanding is that Alpha-Beta performs better than MCTS for games with a small branching factor and where good heuristics are available. If heuristics are not available or are hard to design (as in Go) or the branching factor is large, MCTS becomes preferable. This is underlined by the success of MCTS-based agents in general game playing competitions (Genesereth and Björnsson 2013). Recently, an algorithm combining MCTS and learned policies managed to beat the human champion in Go (Silver et al. 2016).

In MCTS, a straightforward handling of chance nodes is to handle them the same as other nodes in the tree, except that in the selection and playout phases the probability distribution of the children is sampled to choose the node to select/move to play. This corresponds to averaging over all possible determinizations of the game weighed by their probability. When the computational cost of sampling a node is high or the branching of the chance nodes makes the tree unmanageably large, this direct approach becomes impractical. Multiple methods to limit the exploration to only a sample of the possible determinizations have been proposed (Lanctot et al. 2013; Bjarnason et al. 2009).

As in Alpha-Beta, the simplest way to handle simultaneous decisions is to serialize them and use plain UCT. This has the same disadvantages as serializing decisions in Alpha-Beta, but is easy to implement and fast. A simple improvement is to represent the simultaneous decision as a single node in the tree where children correspond to joint actions, but have each side maintain separate reward sums and visit counts for their own actions. In the selection phase, each side selects an action that maximizes the UCB1 value over their reward estimates independently. This algorithm is called decoupled UCT. The advantage of decoupled UCT is that simultaneous decisions are handled symmetrically for both sides. Nevertheless,

decoupled UCT algorithm does not, in general, converge to Nash equilibrium (Shafiei et al. 2009).

However, using a different formula in the selection phase, in particular Exp3 (Auer et al. 2002) or regret matching (Hart and Mas-Colell 2000) results in an algorithm that provably converges to the Nash equilibrium (Lisý et al. 2013), although at a higher computational cost per evaluation. Another MCTS-based algorithm with convergence guarantees is Online Outcome Sampling (OOS) (Lanctot · et al. 2013), which uses a formula similar to regret matching but fixes a strategy of one of the sides for each iteration of the algorithm.

In initial empirical evaluation in the context of Goofspiel, decoupled UCT was found to be worse than Exp3, OOS and regret matching (Lanctot · et al. 2013). Further work then evaluated multiple variants of simultaneous move handling methods in UCT in a set of nine games and found decoupled UCT to be best overall with serialization second (Tak et al. 2014). The likely reason is that MCTS variants with convergence guarantees are computationally more expensive then serialized or decoupled UCT, while most actual games are not "hard" in the sense that Nash equilibrium strategy is vastly different from both minimax and maximin serializations.

7.5 Implementation

As with the other techniques in this thesis we implemented adversarial search in KC:D. In particular, we tested search-based controllers for enemy AI in one-on-one swordfighting scenarios. In this section, we will first detail the combat mechanics of KC:D and how we abstracted them into a game tree. We will note how we made the abstract representation match the actual game as closely as possible and how the search-based controller is integrated in the game. Then we discuss the implementation of the actual search algorithms we tested and the section concludes with description of our experiments to find optimal parameter values for our algorithms.

7.5.1 Combat Model in Game

There are four main groups of stats that govern combat in KC:D: health, stamina, attack and defense. When health drops to 0, the NPC dies. Health starts at 100 and cannot be recovered during combat. Stamina represents the energy the NPC has to perform actions and is very important for combat. It ranges from 0 to 110 and every combat action except for walking consumes part of stamina. If the NPC does not perform any stamina-consuming actions for two seconds, the stamina slowly replenishes. The rate of stamina recovery depends on the stance of the NPC (completely at rest, walking, in combat guard – in order of decreasing stamina recovery) and on multiple additional stats of the NPC.

Attack stats represent the ability to cause damage to opponent's stamina and health. Most weapons support two or three attack types (e.g., slashing and stabbing with a sword). *Attack value* is attached to each attack type of a weapon and is further modified by several stats of the NPC. *Defense value* is attached to both weapon and every piece of armor the NPC wears. Armor may have different defense values against different attack types (e.g., chainmail is very effective against slashing, less effective against stabbing and of little use against blunt force). The body of the

character is divided into 6 parts and 36 subparts and different types of armor cover different subparts.

When attacking, the character chooses the type of attack (e.g., slash or stab) and a broad zone of attack (head, upper left, upper right, lower left, lower right and center – not all zones are available for all types of attacks). Once the attack action is issued, the combat system selects an appropriate animation based on the context (obstacles, distance to opponent ...). The animation then controls the movement of the sword.

The opponent can defend himself in three ways: blocking, perfect blocking and dodging. Blocking is the simplest and least effective way of defense: if the defending character is in blocking state when the attack reaches a *critical point* (usually just after the weapon starts moving towards the character), the attack is always blocked by weapon and the defender stamina is reduced – the larger the ratio of attack to the weapon's defense, the larger damage to stamina. Although it is possible to be blocking for long periods of time, this is not the optimal strategy, because unless the attack is very weak, the defender loses much more stamina than the attacker consumed for the attack.

If the defender's stamina drops to zero as a result of the block, the block is broken and the defender is dazed for several seconds and unable to perform any action, letting the attacker score some unblocked hits. Furthermore, after a block, there is always a short time window where the attacker can issue another attack command, while the defender cannot counterattack.

The defending character may also try to perform a *perfect block* – that is to start blocking in a very small time window that follows the critical point. The length of the perfect block window depends on the relative skills of the combatants and the type of attack. For NPCs that try to perform a perfect block, the same stats affect the probability that they will succeed. A successful perfect block costs the defender no stamina and also gives him a short time window for a *riposte* – a strong counter attack that cannot be blocked by a simple block, but only with a perfect block. On the other hand, a failed perfect block results in an unblocked hit.

Dodging is activated if the defender starts moving in a small time window that starts slightly before the critical point. The length of the time window/probability of success for dodging is computed from different character stats than for perfect block and is usually longer. Dodging acts very similarly to perfect block, but it does not let the defender to riposte. Dodging also involves character movement which may be beneficial for tactical reasons (maneuvering the opponent to a constrained space). Dodging also has the advantage that the defender may be in a blocking state while attempting to dodge. If this is the case, a failed dodge will result in a blocked hit, which is almost always better than an unblocked hit.

To summarize, the difference between dodging and perfect block is that a successful perfect block gives higher advantage than a successful dodge (perfect block allows for riposte), while dodge is usually a safer choice (longer time window /higher probability of success, failure may not result in an unblocked hit).

In all cases of successful defense, the combat system ensures that the animations of the attacker and defender are properly synchronized and result in a physical collision of their swords or in a clear miss.

If the defender does not block or fails to issue a perfect block/dodge, the attacker's sword follows the trajectory outlined in the attack animation. If the sword collides with the opponent's body, the opponent is hit. The actual damage done to the character depends on the ratio of attack value and the summed defense values of all armor parts that cover the body subpart where the sword hit the character. A hit

results in damage done to character's stamina and possibly also to character's health. Except for the strongest hits only little damage is done to character's health and the majority of the damage is done to stamina. However, if the character's stamina is not large enough to absorb the whole damage from the hit, the remaining damage translates directly to health damage. Strong hits (attack larger than 2.5 times defense), do considerable damage to character's health even if his stamina is not depleted by the attack, but those are almost impossible to achieve against armored opponents. Very weak hits (attack less than 0.75 times defense) result in almost no stamina or health loss. The body part that was hit also affects the final damage – hits to the head result in larger damage, while hits to body members result in reduced damage.

A successful hit also gives a short time window, when the attacker may start another attack (if he has enough stamina) while the defender cannot issue any attack actions. If the attacker is able to chain attacks precisely he may execute a *combo*. A combo is a designer-specified sequence of attack types and zones and if the attacker executes those attacks successfully with good timing (always issuing the attack action at the first possible moment) and the defender does not perform a dodge or a perfect block, the last attack turns into a special animation and deals the defender extra damage.

Another important mechanic is *feigning*. If the attacker manages to change attack zone in a short time window between the moment the attack command is issued and the start of the actual attack animation and the defender is holding a normal block, the block will not automatically adjust to the new attack zone and the attack will hit. The defender may however counter by *reblocking* – releasing the block button and pushing it again immediately, which makes the block adjust to the new attack zone. Once again, AI has a probability for both performing a feign and successfully reblocking if the player feigns.

Every block also damages the weapon and every hit damages the armor, decreasing their defense and/or attack values. All the time the characters can also move, stepping in and out of reach of opponent's weapons. As different weapons and attack types have different maximal distances and every piece of armor slows the characters down, movement adds another level of complexity.

The stamina is the key resource for combat, it acts as both a shield and energy for attacks and the character needs to weigh the benefits of consuming stamina for a chain of attacks, which may defeat the opponent, with the risk of exposing himself to a riposte while his stamina is low. Simple blocking is often useful if the attacker is low on stamina and will not be able to continue with further attacks after his attack is blocked, while a risky perfect block may tip the balance in favor of the currently weaker combatant. When character's health decreases, his maximal stamina also decreases, but the decrease is non-linear and relatively slow (at 50 health maximum is 87 stamina, at 10 health maximum is 57 stamina).

The character may also repeatedly aim for the same part of opponent's body, damaging a fragile piece of the opponent's armor and increasing the effectivity of his further attacks. A heavily armored opponent may easily endure several unblocked hits from a light weapon in a row while a fast but lightly armored character has to keep a safe distance and try to replenish his stamina even after a minor hit. Note that except for the most unbalanced cases, it is impossible to win a fight without taking risks – a series of successful perfect blocks and ripostes may defeat even a much stronger opponent.

Repeated hits in the same body part may also result in bleeding – the character starts to slowly lose health over time, the actual speed depends on the severity of the bleeding.

The game features one other melee combat mechanic: *clinch*. Clinch is a specific combat state the occurs when the characters move to close together – their swords become interlocked and the only possibility is to perform a physical action (e.g., a kick or a punch), which may decrease opponent's stamina and moves the characters further apart and terminates the clinch. In the game, clinch is simply a race as to who first pushes an attack button once the clinch starts. The AI calculates a random delay for its attack in clinch based on various stats of the NPC.

The game currently features fist combat, broadswords, short swords, sabres, maces and shields. Short sword may be used together with a shield, while broadsword needs both hands of the NPC. All weapons follow the same basic mechanics and differ only in parameters and animations.

Including armor, over 40 parameters per NPC affect the combat mechanics. This is where a search-based approach should be useful – due to the large number of parameters involved and to the large number of weapon and armor combinations, scripted combat decisions cannot be tailored to all possible cases and have to necessarily compromise between optimality and code complexity.

The combat mechanic also has the desirable property that after important events (hit, block …), there are short time windows (about 100 ms long) where the animations are fully controlling both characters and neither of them can perform any action. These windows are very suitable to performing search for a next action. We thus consider 100 ms the longest time a search may take to be useful in the game.

Note also that the combat mechanic has only limited interactions with the obstacles in the 3D world. For example, when the combatant is close to a wall, certain attack zones may be disabled (e.g., when the sword would have to move through wall to perform the associated animation). Obstacles may also limit the possible directions for dodging. Nevertheless, combat most often happens at open spaces, as obstacles are very hard to handle properly on the animation and AI side, so the game is designed to generally avoid combat in crammed spaces.

7.5.2 Combat Model for Search

As the combat mechanics rely heavily on animations and physics, reusing the code that executes the combat in the game as a model for the search would be both difficult and computationally very expensive. To make search possible, we need an abstract representation that maintains the important aspects of the mechanics but can evaluate millions of actions per second. Such model will necessarily reimplement a portion of the combat mechanics, which is one of the disadvantages of using search in this scenario. Since the combat is very fast-paced and a different choice at a single time point can make the difference between winning and losing the fight, we wanted the model to stay very close to the game mechanics and represent individual actions the combatants can perform. In this section we describe the model that proved most useful for our implementation.

One of the biggest simplifications we made is that we reduce movement of the combatants to only one dimension. This is because in one-on-one combat, positioning plays a secondary role – the primary purpose of movement is to move in or out of range for attack, which is mostly captured by the 1D model. The 1D movement space is bounded to model that the combatant may be pushed to a wall.

The model cannot capture limitations to combat actions formed by obstacles and it cannot capture complex evasion/pursuit in a maze-like environment. Although obstacles can in principle have big influence on combat (e.g., good movement through a maze may give the combatant enough time to replenish stamina without letting its opponent attack), this is seldom encountered in the actual game.

Further, the model assumes that all the equipment of both combatants is fixed throughout the combat, although combatants are allowed to change weapons in the middle of the fight. The model also assumes that all character parameters are fixed, in particular we ignore the possibility of increasing skill mid-combat and of time-related changes in character parameters (e.g., if the character is hungry, his maximal stamina slowly decreases over time). In all those cases, our model works with a snapshot of the world as it was at the beginning of the search. Note however, that all those effects are very rare in combat and once they take place, they are taken into account in the next search invocation. To improve performance, all values derived from character properties that are assumed fixed are precomputed (e.g., the total armor for each body subpart).

The last major mechanic we do not model is combos, as they are difficult to capture in a simplified way and it is difficult to obtain the exact data to model them – combos are declared procedurally as preconditions for special combat animations which can be evaluated only against the full game state.

Other than that, we model – to various levels of fidelity – all aspects of the combat as discussed in the previous section. All events related to physics and reaction times are modelled as chance events where either the probabilities of success are a direct input to the search (e.g., the probability of a successful perfect block) or are determined by the stats of the combatants (e.g., the average time to reaction in clinch). To model the player, the parameters may be deduced from the player's combat history combined with a baseline guess given by game designers.

From the perspective of a search-based AI, a favorable feature of the combat mechanics is that on many occasions, a single combatant has the attack initiative, while the other character can only decide on a way to defend. Those moments can be adequately represented as the attacker deciding first on his attack and the defender deciding afterwards. This greatly reduces the number of nodes in the search tree that represent simultaneous decisions. Further, the attack and defense decisions are already discrete, we only need to discretize the movement of the NPCs and the flow of time.

7.5.2.1 Actions

After thorough experimentation with multiple variants, we arrived at the following set of actions for the abstract model:

- AttackNow – if the enemy is within the attack range, attack immediately, if not, approach the enemy and attack once he is in range. The action is parametrized by the type of attack the character performs.
- AttackOnApproach – do not move, but attack immediately if the enemy moves within attack range. The action is parametrized by the type of attack the character performs.
- TryPerfectBlock – if the enemy starts an attack, attempt a perfect block.
- TryDodge – if the enemy starts an attack, attempt to dodge.
- SweetSpot – move towards the optimal attack distance (given as input to the algorithm). If the enemy starts an attack, attempt a perfect block.

- SweetSpot & Block – move towards the optimal attack distance. The combatant holds normal block while moving.
- StepBack – increase distance from the enemy. If the enemy starts an attack, attempt a perfect block.
- StepBack & Block – increase distance from the enemy. The combatant holds normal block while moving.
- Nothing – do not do anything. If the enemy starts an attack, attempt a perfect block.
- Nothing & Block – do not do anything except holding normal block.

Note that all actions that do not explicitly include attack or a defensive action include an attempt at perfect block. This was introduced to both mimic the way players play the game (unless explicitly deciding to do something else, players almost always attempt to perfect block whenever they see the opponent attacking) and to reduce the number of occasions when the AI would receive a hit without doing anything to counter it, which players perceived as stupid. As a result, TryPerfectBlock executes the same as Nothing, but we kept those actions separate and use TryPerfectBlock when the AI is making a defensive decision and Nothing otherwise, which makes for a slightly more comprehensible visualization of the search tree for debugging purposes.

The SweetSpot & Block and StepBack & Block actions were also added later in the development to let the AI move and defend itself at the same time, which once again reduced the number of unblocked hits the AI received and allowed for smoother risk balancing by the AI. Keep in mind however, that normal blocking is often not the best choice as it lets the attacker keep initiative, so StepBack without normal block (but with an implicit attempt at perfect block) may be a better defensive decision in certain contexts.

Initially, we also had only one attack action. We tested both having the AI to always attack immediately (as in AttackNow) and letting AI attack only when the enemy is within range (as in AttackOnApproach). The former resulted in the AI performing a lot of "preemptive strikes" as doing anything other than attacking meant letting the opponent to gain attack initiative. The latter option led to a very defensive behavior as the AI could approach the opponent only with the SweetSpot action, which gave the opponent the possibility to always attack first, and was thus avoided. However, once we let the AI use both actions, its strength increased considerably.

When a combatant attacks, the duration of the attack and the defensive response is estimated from the real duration of the attacks in the game. If neither side attacks, the time moves by a fixed step, given as a parameter to the algorithm. This time step then determines the distance the combatants move with a single action. In all our experiments, the default time step was 0.5s.

7.5.2.2 State

Since most of the parameters influencing combat are assumed fixed in the model, the state of our model is formed by only:

- Position of the combatants in the 1D space.
- Health, stamina and the remaining time until stamina starts to replenish for both combatants.
- Health of all pieces of armor for both combatants.
- The action taken by the combatants at the latest decision point.

- Which combatant has the attack initiative (if any).
- The type of search node.

The state contains the action taken in the last decision point because many actions result in chance events that determine the success of the action and therefore the action chosen needs to be retrieved in later chance nodes.

The attack initiative models the situations where one of the players can issue actions, while the other still recovers from a hit or from opponent's perfect block. The side having the attack initiative is the *attacker* while the other side is the *defender*.

The type of search node represents the general state of the combat and together with the attack initiative it implies the type of the node from the perspective of the search algorithm (simultaneous decision, min decision, max decision or chance). We have the following types of nodes (see Figure 13 for a diagram):

- Decision nodes
  - Simultaneous – Both sides can perform all actions which are then evaluated simultaneously.
  - Attacker – The attacker side decides whether to attack or not. In this node, the attacker chooses only the type of attack, not the individual zone. There are two special cases for this node:
    - Riposte – The attacker may perform a riposte. If he succeeds, a defense with a normal block will be useless.
    - Broken block – the last attack resulted in broken block of the defender. The defender thus cannot react with a defensive action and will certainly be hit by the attack.
  - Defender – The defender chooses how to react to an attack. This node is only encountered after the attacker decides to attack in the attacker node.
  - Attack Zone – The attacker chooses the zone for the attack. This node was introduced to reduce the branching factor of the search. Note that unless the attack is a hit, it does not matter what zone the attacker chose, only the type of the attack. So we defer the choice of an attack zone to a separate decision node. This node is reached only in the branches when the attack resulted in a hit.
- Chance nodes
  - Both attacking – This node is encountered when both sides choose to attack in the simultaneous node (if both of them can actually attack). It randomly chooses which of the combatants hits first. The probability of hitting first is derived from the difference of the durations of the attack action. Since the first hit invalidates the attack of the opponent, it is impossible for both combatants to score a hit at the same time.
  - Armor zone – Once an attack is determined to be successful and once the attacker has chosen the attack zone, a probability distribution is sampled to determine which part of the body was actually hit and corresponding damage is dealt to the defender.
  - Dodge – If the defender chooses to attempt dodge, this node determines the actual outcome – either a) the dodge is successful, or b) it is not successful, but the defender has been able to perform

a normal block, or c) it is completely unsuccessful and the defender receives an unblocked hit.

- Perfect block – Determines whether an attempt at perfect block has been successful. This node is encountered whenever the defender does not use a normal block and is not attacking.
- Riposte – If the defender successfully performs a perfect block and decides to attack in the riposte decision, this node determines whether he is successful at executing the riposte. Since the only possible defense against a riposte is a perfect block, successful riposte transitions to perfect block chance for the defender. Unsuccessful riposte behaves as a normal attack from the attack decision.
- Feign and reblock – If the defender decides to perform a normal block (or performs a normal block as a result of the dodge chance), it is still possible that he gets hit, if the attacker executes a successful feign. The defender can however react to a feign by quickly reblocking. This node gathers these two events. The possible results are a successful block (either the feign has not been successful or the reblock has been successful) or an unsuccessful block with a changed attack zone. For simplicity, it is assumed that the new attack zone is chosen randomly, and so this node has one child for each attack zone. In both cases, the attacker retains attack initiative.
- Clinch – This node models the clinch mechanic – it is encountered whenever the combatants get too close to each other. It randomly chooses which player will first react and thus gain attack initiative. The probability is determined by the average time for clinch reaction of the combatants.

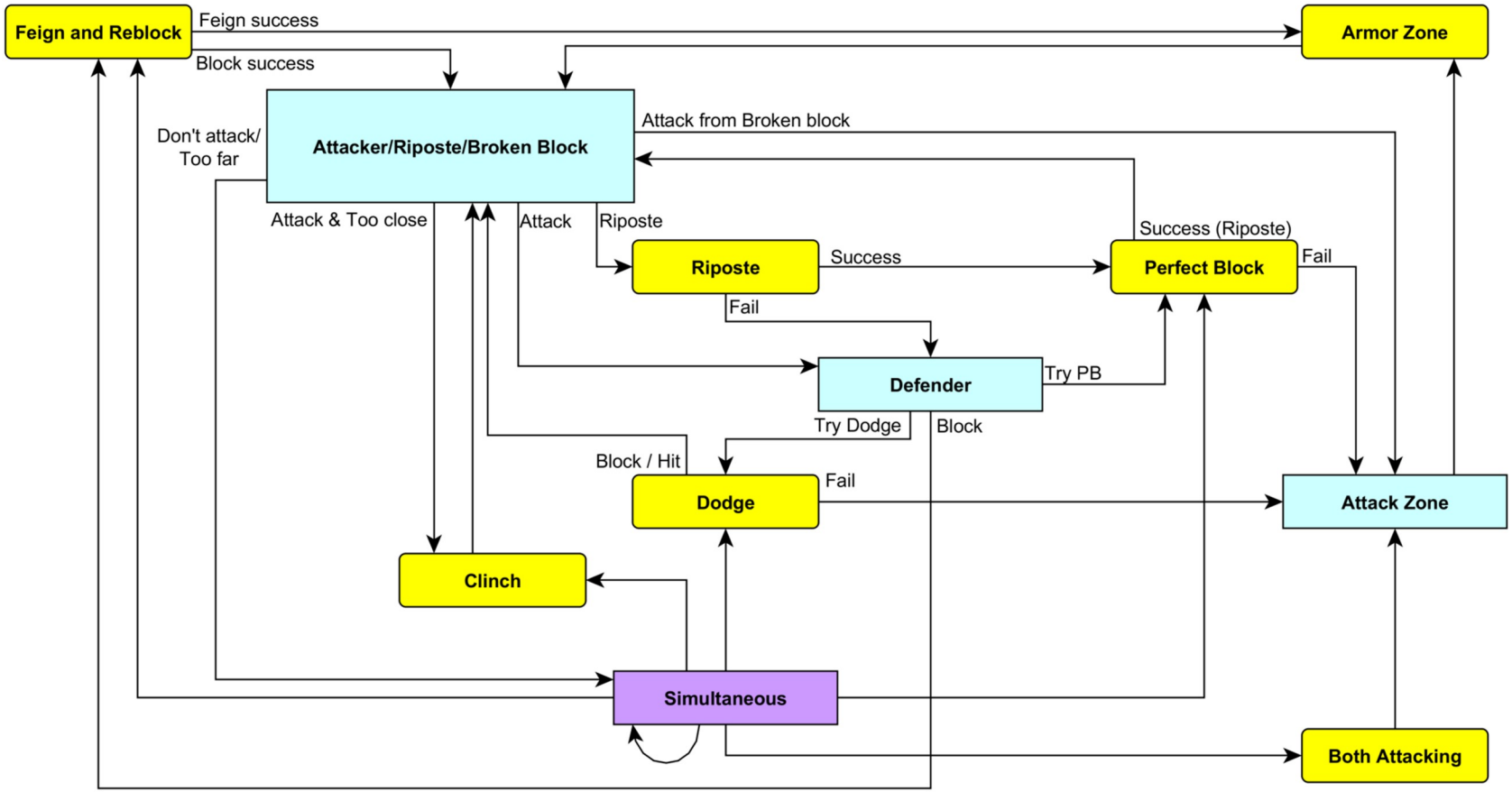


Figure 13: A transition diagram of the node types in the abstract combat model.
Chance nodes are displayed in yellow, decision nodes are blue and the special simultaneous decision is violet.

7.5.2.3 Extracting Abstract Model Parameters

Apart from parameters that are directly represented in the game mechanics (e.g., attack values, health), our model needs additional parameters to estimate the mechanics that, in the actual game, are governed by animations and physics. These are the durations of attack actions and the actual body subparts that are hit with successful attacks at various zones. To estimate the parameters, we have setup a large arena with several dozens of fighting pairs of NPCs (see Figure 14) and measured all attacks and the zones that were hit. The scenario was run for a day, giving us 62 083 attack and hit combinations. Initially, we experimented with fitting linear models to the data taking various features into account (e.g., distance between the combatants, difference in elevation), but those additional features provided little improvement. In the end, we simply counted the number of times a particular subpart was hit as a result of an attack of a given type and zone. To keep the branching factor of the corresponding chance nodes reasonable, we then removed all subparts that were hit in less than 5% of the cases and used the normalized counts of the hits as the probability distribution. The attack durations are then simply the average durations of the individual attack types.

Figure 14: Data gathering scenario for the abstract combat model.

7.5.3 Integrating the Model with the Game

A correct integration of deliberative reasoning with an abstract model in a real-time environment is a challenging task that may largely affect the practical performance of the system. There are three basic aspects of the integration:

1. Translating the actual game state into the abstract representation.
2. Enacting the decisions made by the system in the virtual world.

3. Meta-reasoning about the system – deciding when a new round of reasoning should be performed or when do the decisions made in a previous reasoning run become invalid.
4. Debugging support.

In most cases, including our model, translating the game state into the abstract representation is relatively straightforward – almost all variables in our model can be directly extracted from the game state. As our model is one-dimensional, we simply project the positions of the two combatants on the line connecting them and query the environment for obstacles on that line. We assume that the distance to obstacles on that line is representative of the general amount of free space available for the maneuvering of the combatants.

In the framework of script selection, enacting actions in the world simply means running a script. While the actions represented in the abstract model are relatively low-level, the scripts handling their execution need to also cover the aspects of the combat that are not represented by the model – in particular, the scripts are responsible for handling combos and incorporate 2D obstacle information into movement.

The meta-reasoning in our system continuously tracks the combat state and executes the search every time the combat state changes significantly or an important action is completed (e.g., when either combatant was hit or when the NPC successfully performs a perfect block). As we already noted, most of those situations are followed by a short time when neither combatant can perform an action, which lets us to enact the decision without any delay. Further, the system monitors the current attack initiative and prepares the root node of the search tree accordingly (e.g., when the opponent has started an attack, the search is initialized to a Defender decision node; after the NPC performs a perfect block, the search starts with a Riposte decision node). The system also estimates the time that is available for decision making in time-critical contexts (e.g., when deciding on a defensive actions when the attack is already in progress) and may shorten the time allocated for search to make sure the decision can be enacted safely. A new search is also started after a given time interval has elapsed, even if no significant changes have happened. We have achieved the best results with setting the maximal interval between searches to 500 ms.

It is also important that our model does not gain complete control over the NPC. Instead, the model is incorporated in the larger context of script selection and thus a higher-level decision may decide to ignore the results of the search or not run the reasoning at all. This is mostly to allow scripters to retain detailed control over the NPCs actions, if it is required by game design.

On the implementation level, the search algorithm is represented by a decorator BT node which makes the results of the latest search run available to the child subtree in a variable. This enables the meta-reasoning system to get updates frequently while letting the scripter using the system implement and control how exactly should the actions be handled.

The debugging of the model is facilitated by storing the complete trees for recently run searches and letting the user to explore the tree to learn why a given decision was made (see Figure 15). The user can also visualize the state represented by a given node in the stored tree directly in the game editor to be able to compare it to the actual state of the game (see Figure 16).

Search: NPC1 @ 4.6 s	0	Chosen: Attack... 0.52	#Steps: 11644, ...
AttackNow: 0-?	1	Simultaneous, ... 0.515489	#Visits: 1734
AttackNow: 1-?	1	Simultaneous, ... 0.516598	#Visits: 1917
AttackNow: 0-?	1	BothAttackingc... 0.508881	#Visits: 1706
40% Chance	0	ZoneDecision, ... 0.533433	#Visits: 129
59% Chance	1	ZoneDecision, ... 0.492877	#Visits: 169
AttackNow: 0-0	1	ArmorZoneCha... 0.492582	#Visits: 83
AttackNow: 0-1	1	ArmorZoneCha... 0.492978	#Visits: 85
15% Chance	1	Attacker, Attac... 0.464369	#Visits: 10
15% Chance	1	Attacker, Attac... 0.495710	#Visits: 15
53% Chance	1	Attacker, Attac... 0.500833	#Visits: 45
15% Chance	1	Attacker, Attac... 0.487141	#Visits: 14
AttackNow: 1-?	1	BothAttackingc... 0.510727	#Visits: 1454
TryPBlock	1	PerfectBlockCh... 0.512284	#Visits: 1284
30% Chance	1	RiposteDecisio... 0.495841	#Visits: 72
70% Chance	0	ZoneDecision, ... 0.521409	#Visits: 143
AttackNow: 1-0	0	ArmorZoneCha... 0.525791	#Visits: 30
AttackNow: 1-1	0	ArmorZoneCha... 0.523511	#Visits: 29
AttackNow: 1-2	0	ArmorZoneCha... 0.515381	#Visits: 26
AttackNow: 1-3	0	ArmorZoneCha... 0.514417	#Visits: 26
AttackNow: 1-4	0	ArmorZoneCha... 0.526410	#Visits: 31
TryDodge	1	DodgeChance, ... 0.511787	#Visits: 1334
SweetSpot	1	PerfectBlockCh... 0.512277	#Visits: 1285
SweetSpot + Block	1	FeignAndReblo... 0.514427	#Visits: 1095
StepBack	1	PerfectBlockCh... 0.512771	#Visits: 1239
StepBack + Block	1	FeignAndReblo... 0.513961	#Visits: 1138
Nothing + Block	1	FeignAndReblo... 0.514333	#Visits: 1103
TryPBlock	1	Simultaneous, ... 0.510649	#Visits: 1167
TryDodge	1	Simultaneous, ... 0.510319	#Visits: 1139
SweetSpot	1	Simultaneous, ... 0.511164	#Visits: 1218
SweetSpot + Block	1	Simultaneous, ... 0.509987	#Visits: 1111
StepBack	1	Simultaneous, ... 0.510693	#Visits: 1170
StepBack + Block	1	Simultaneous, ... 0.509964	#Visits: 1110
Nothing + Block	1	Simultaneous, ... 0.509598	#Visits: 1080

Figure 15: Search tree visualization in level editor.
The color of the nodes represents who is deciding at the given node (blue: max side, green: min side). The intensity of the color corresponds to the node's preference among siblings (node value in AlphaBeta, number of visits for UCT and DUCT).

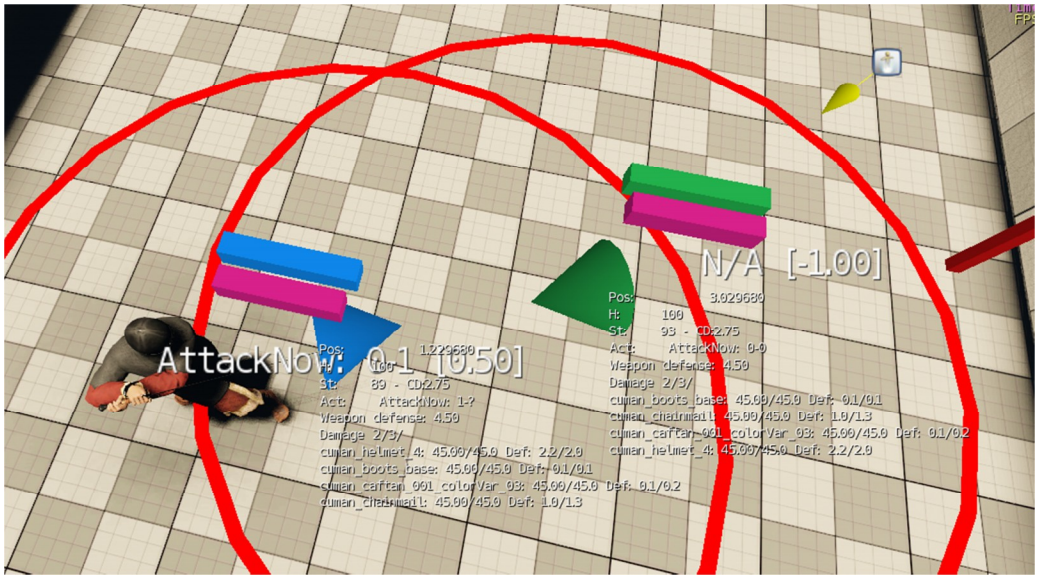


Figure 16: Visualizing the abstract model state by overlaying it with the game world.
A possible state envisioned by the search algorithm is displayed over the actual state of the game. The cones represent the positions of the combatants in the hypothetical state, the bars represent health and stamina (purple) and the red circles are the attack ranges of the combatants. The red prism on the right represents the limit of the 1D space of the model. Text provides further details of the state (chosen action, armor stats, ...).

7.5.4 Search Implementation

The search algorithms and the abstract combat model are implemented in C++ and are intended to be run in a separate thread. Implementation of the model and all search algorithms has been heavily optimized with regards to best practices for C++ programming: high cache locality and cheap state copy was achieved by storing all data in a contiguous segment of memory, no dynamic allocation is performed in the main loops of the algorithms and allocation on stack is kept minimal. Since copying the state is – in our case – cheap while evaluating the effects of actions may be costly (multiple floating point operations), the search algorithms copy the state when branching instead of doing and undoing actions on a single state copy as is common in Go or chess. Our abstract model can evaluate approximately 10^7 moves per second (copying the state included) on a higher-tier PC (Intel Core i5 3470 @ 3.20 GHz). We have implemented three search algorithms that form the state of the art in online adversarial search: Alpha-Beta, UCT and Decoupled UCT (DUCT).

In our version of the Alpha-Beta, we handle chance nodes with *-minimax (Ballard 1983) and randomly serialize simultaneous decisions. We have also implemented transposition tables, which slightly improved the performance, but the gain was very tenuous and we therefore did not implement transposition tables for chance nodes. Alpha-Beta is run in an iterative deepening scheme until the allocated time runs out. We used a class of evaluation functions that return difference in health for terminal nodes and a weighed sum of difference in health and difference in stamina for non-terminal nodes. The weight of the health difference in non-terminal nodes is always below one, as it represents the penalty for uncertainty (the same difference in health gives a higher value, if min is dead and lower value if max is dead). Our version of the Alpha-Beta algorithm thus has four parameters: the time limit, the size of the transposition table and the two weights for the evaluation function for non-terminal nodes.

Our implementation of UCT and DUCT are very similar and so we will first speak about their shared properties. An issue for direct application of any MCTS-based algorithm to our abstract combat model is that the length of the combat is not limited, and in extreme cases it may not even be possible to end the combat even if both sides cooperated (with high armor, high stamina cost for attack and weak attacks). Therefore, we setup a limit on the maximal length of a playout. If the playout exceeds this limit, the value of the playout is determined by evaluating the state reached. For both terminal and non-terminal states, we use the same evaluation function as for the Alpha-Beta algorithm (a weighed sum of difference in health and difference in stamina).

In both UCT and DUCT we handle chance nodes straightforwardly – all possible outcomes are added as children of the chance node and in the selection phase, the probability distribution is sampled to determine the next node.

For sampling the chance nodes in the selection phase and in playouts, we use the Xorshift128+ random number generator (Vigna 2014), as its inclusion significantly improved the number of steps the algorithms could perform within the time limit over using the default generator employed in KC:D (a Mersenne twister) while also providing stronger guarantees on statistical quality.

Since all memory is preallocated, including an array to store the nodes of the search tree built by UCT and UCT, the maximal number of nodes has to be known

during algorithm initialization. If the algorithm runs out of preallocated memory for nodes, it continues to run playouts from the leaf nodes, but does not expand new nodes. In practice we however set the memory limit large enough to prevent the algorithm from consuming all of the preallocated memory.

As we expect the algorithms to be run for a very short time, we do not need to be conservative in memory usage, which has let us to implement a wasteful but faster storage scheme for the search tree stored by the UCT and DUCT algorithms. In particular, we store all children of a node in a contiguous segment of the preallocated array storing the nodes. This improves cache locality in the selection phase and makes the individual nodes smaller (only the index of the first child and the number of children have to be stored). On the other hand, it means that the memory for all children of a node needs to be reserved once any of the children is expanded. Since most potential children are never expanded (simply because trees in general have much more leaves than internal nodes), the memory requirements of the algorithm are several times larger than with a more compact storage.

The only difference between UCT and DUCT is in handling simultaneous decisions. In UCT we handle simultaneous nodes by randomly serializing the decisions. In DUCT we use the decoupled approach - each player selects an action that maximizes the UCB1 value over their reward estimates independently. To keep the implementation simple, we emulate this behavior by two levels of branching in the tree, where all the nodes on the second level share the same visit count and total score statistic.

Our versions of both UCT and DUCT have five parameters: the time limit, the exploration factor for the UCB1 formula, the maximal length of a playout (referred to as *playout steps*) and the two weights for the evaluation function of non-terminal nodes.

We have also used two baseline controllers during evaluation:

- Random. Uses the same model as the search algorithm (in particular, it uses the same move generator), but chooses actions uniformly at random.
- Default AI. This is the combat AI developed for the public beta of KC:D by the scripters at Warhorse.

While the random controller is very straightforward, the default AI deserves a bit more attention. The default AI is composed of multiple choices, where each choice is realized by sampling a discrete probabilistic distribution. These distributions then form the parameters of the default AI. The individual choices are activated by different stimuli and result in selecting a script. Once a script is chosen, there is no more randomness and the character is always successful at executing the script. This means that the probability distributions combine two types of information: what actions does the AI prefer for the given choice and how likely is the AI to successfully execute those actions. There are the following choices the default AI handles:

- Defensive choice – performed when the opponent starts an attack. The possible scripts to execute are perfect block, dodge, normal block and nothing.
- Idle choice – the default choice performed when no other choice should be active. The possible scripts to execute is moving closer or farther from the player, moving sideways, moving aggressively forward to enter clinch and starting an attack (triggers an attack choice).

- Attack choice – this choice is active when the NPC chose to attack in the idle choice or after a successful attack has been performed. The scripts available represent the individual types of attack or stopping the attack (triggers the idle choice).

For all choices except the defensive choice we used the same parameters as was used in the alpha version of the game. For the defensive choice, we forced the parameters to match closely the defensive capabilities of the search-based AI. We used the default AI only for tests in the game, as the default AI cannot be run without the whole game running.

7.5.5 Comparing Algorithms

Initially, we compared the algorithms by running a multitude of randomly generated scenarios and simply counting the wins for both algorithms. This however proved very ineffective, as most scenarios were unbalanced. Since the combat mechanics favor stats (or skill) over intelligence, this imbalance frequently affected the outcome of the scenario much more than the difference in the quality of the algorithms.

To alleviate this issue, we ran each scenario twice with the algorithms swapping sides and compared the difference: if one of the algorithms won both variants, it scored a win. If both algorithms won one variant of the scenario, the one that won with more health was awarded a win. If the difference in health was very close, we also took into account the amount of time it took the algorithms to win (shorter time is better) if even the time was close, the scenario was declared a draw and both algorithms were awarded half a win. We did not run the same scenario more than twice (once for each ordering of the agents), as we wanted to explore a large number of possible scenarios and running the same scenario multiple times reduced the effect of chance almost the same as running multiple scenarios twice.

7.5.6 Finding Optimal Parameter Values

As discussed in the previous section, the algorithms we developed depend on a set of parameters as discussed in Section 7.5.4 and summarized in Table 12. We chose four values for possible time limits for the algorithms: 1ms, 20ms, 50ms and 100ms (we noted earlier that 100ms is the maximal lag that still lets the NPC react in time to most events). The configuration of the other parameters is more difficult. To get the best possible results, we tried to find optimal parameters through simple genetic algorithm (GA). We used population size of 30 and tournament selection based on 4 duels (as described in the previous section) which was determined in early experiments as a reasonable compromise between stability and exploration. We used one GA thread per core and we allocated up to two weeks of computing time on a single machine per algorithm to this task. This resulted in two runs of the GA for 200 generations for each of the algorithms and time limits.

Alpha-Beta		UCT/DUCT	
Parameter	**Possible Values**	**Parameter**	**Possible Values**
Time limit	{1, 20, 50, 100}	Time limit	{1, 20, 50, 100}
TT Size	$\{2^k; k = 1 .. 20\}$	Exploration factor	[0.01; 4]
		Playout steps	[0 – 200]
Health weight	[0 – 1]	Health weight	[0 – 1]
Stamina weight	[0 – 1]	Stamina weight	[0 – 1]

Table 12: Summary of parameters of the search algorithms.

Since we select algorithms by tournament, there is no global fitness that would let us choose the "best" algorithm. Instead we assume, that the best algorithm will have the most copies towards the end of the GA. Inspecting the evolution gave support for this approach: in most generations there was a single individual with many copies while most other individuals had only one copy. This overrepresented individual usually dominated for several generations until a more fit individual was found which then quickly became the new dominating individual.

For each algorithm and time limit, we took all individuals from the last 50 generations of each of the GA runs and found the median and mode for each of the individual parameters across those individuals, as well as the complete genome that had the most copies. This gave us 3 candidates for best parameters from each run. Those candidates are shown in Appendix B. We then ran 500 duels for each pair of the candidates and then chose the candidate that outperformed the most opponents.

The resulting best parameters are shown in Tables 13 – 15. Note that transposition table (TT) proved to be of little significance for AlphaBeta as its size widely varies between the best individuals. Further, the exploration factors for both UCT and DUCT are very low – the optimal variants are thus quite greedy. The maximal number of playout steps is also relatively low, preventing reaching the end of combat in most playouts. We tested whether this is not an artifact, but all the variants indeed outperform their counterparts with significantly less (or zero) playout steps or a higher exploration factor. Stamina also plays only a minor role in all of the evaluation functions, but is more informative in UCT and DUCT. We assume that this is because the situations when stamina is critical are less frequent than situations when the combatant with lower stamina can step back and replenish the stamina while keeping safe distance from the opponent.

Time Limit	Health Weight	Stamina Weight	TT Size
1 ms	0.80	0.03	131072
20 ms	0.77	0.02	0
50 ms	0.56	0.01	8092
100 ms	0.68	0.01	4096

Table 13: Best evolved parameters for Alpha-Beta.

Time Limit	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
1 ms	0.13	6	0.77	0.03
20 ms	0.31	15	0.88	0.11
50 ms	0.16	14	0.70	0.09
100 ms	0.25	11	0.77	0.01

Table 14: Best evolved parameters for plain UCT.

Time Limit	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
1 ms	0.07	6	0.91	0.11
20 ms	0.10	8	0.82	0.09
50 ms	0.30	21	0.84	0.10
100 ms	0.47	18	0.83	0.07

Table 15: Best evolved parameters for decoupled UCT.

7.6 Evaluation

We have performed three separate rounds of evaluation. First, we ran a large number of duels in the simulator between the algorithms and the random baseline. Then we pitted a subset of the algorithms and the default AI against each other in the actual game and last we performed a human study where players fought against two variants of the algorithms and the default AI provided in the alpha version of KC:D. The raw data for all experiments as well as R scripts to analyze them can be found in the digital attachment to this thesis.

In the following text, we will use the term *time variant* for agents that use the same algorithm but with different maximum computing time and *algorithm variant* when speaking about agents with the same maximum computing time but different algorithm (limited to the search-based agents only, i.e. Alpha-Beta, UCT and DUCT).

7.6.1 Evaluation by Tournaments in the Simulator

In this evaluation we first run a separate tournament per-algorithm (AlphaBeta, UCT, DUCT) among the 1 ms, 20 ms, 50 ms and 100 ms time variants of the algorithm, including the random baseline as a reference. Then we ran tournaments among all algorithm variants for each value of the time limit. Each duel consisted of 500

random scenarios evaluated according to the method described in section 7.5.5 (each scenario ran twice with the agents changing sides). Note that with 500 duels, a win rate above 0.545 or below 0.455 is statistically significant at the 0.05 level (calculated using prop.test in the R statistical software (R Core Team 2015)). As we do not use the results to confirm or reject any hypotheses, we did not correct for multiple comparison, we simply see this interval as a rough indication of what constitutes a meaningful difference. The results of the individual tournaments are shown in Tables 16 - 22.

	AB-1	**AB-20**	**AB-50**	**AB-100**	**Random**
AB-1		0.464	0.471	0.436	0.909
AB-20	0.536		0.498	0.508	0.910
AB-50	0.529	0.502		0.489	0.894
AB-100	0.564	0.492	0.511		0.913
Random	0.091	0.090	0.106	0.087	

Table 16: Results of tournament among all time variants of the Alpha-Beta algorithm and the random baseline.

	UCT-1	**UCT-20**	**UCT-50**	**UCT-100**	**Random**
UCT-1		0.432	0.419	0.395	0.957
UCT-20	0.568		0.476	0.444	0.965
UCT-50	0.581	0.524		0.469	0.946
UCT-100	0.605	0.556	0.531		0.978
Random	0.043	0.035	0.054	0.022	

Table 17: Results of tournament among all time variants of the UCT algorithm and the random baseline.

	DUCT-1	**DUCT-20**	**DUCT-50**	**DUCT-100**	**Random**
DUCT-1		0.410	0.406	0.395	0.911
DUCT-20	0.590		0.466	0.510	0.935
DUCT-50	0.594	0.534		0.494	0.951
DUCT-100	0.605	0.490	0.506		0.956
Random	0.089	0.065	0.049	0.044	

Table 18: Results of tournament among all time variants of the DUCT algorithm and the random baseline.

	AB-1	UCT-1	DUCT-1
AB-1		0.485	0.491
UCT-1	0.515		0.514
DUCT-1	0.509	0.486	

Table 19: Results of tournament among all algorithm variants with 1ms computation time.

	AB-20	UCT-20	DUCT-20
AB-20		0.465	0.464
UCT-20	0.535		0.453
DUCT-20	0.536	0.547	

Table 20: Results of tournament among all algorithm variants with 20ms computation time.

	AB-50	UCT-50	DUCT-50
AB-50		0.510	0.473
UCT-50	0.490		0.490
DUCT-50	0.527	0.510	

Table 21: Results of tournament among all algorithm variants with 50ms computation time.

	AB-100	UCT-100	DUCT-100
AB-100		0.468	0.476
UCT-100	0.532		0.508
DUCT-100	0.524	0.492	

Table 22: Results of tournament among all algorithm variants with 100ms computation time.

Note that while the differences between the random baseline and all the algorithms or between the 1 ms and longer (20 ms, 50 ms and 100 ms) time-variants of the same algorithm are generally large (except for AB, where only the difference between 1 ms and 100 ms variant is large), the individual algorithms fare very similarly. This suggests that the combat is not "hard" in the game-theoretic sense, i.e. the gains from more intelligent behavior quickly diminish. In particular, the performance of 50ms and 100ms variants of all algorithms is very close, indicating that the benefits of added computing time are small beyond 50ms. Among the algorithms, UCT and DUCT generally outperform AB, but with a very small margin. UCT and DUCT are however not distinguishable in this comparison.

Although all of the differences could probably be made statistically significant with more duels performed, it is of little practical interest: if a difference between two algorithms is not distinguishable with 500 duels, it is unlikely to affect the player's experience much as the player is expected to encounter much fewer opponents throughout the whole game.

Generally, all of the algorithms and time variants demonstrated little differences in performance (except the random baseline), probably due to the nature of the combat model. The only conclusions that can be made with some certainty is that Alpha-Beta is slightly weaker overall than UCT and DUCT and that the performance gain of allocating more than 20 ms of computational time for the algorithms is small.

7.6.2 Evaluation by Tournaments in the Game

We further run a setup similar to the evaluation in the simulator (tournaments among all time variants of a given algorithm and among all algorithm variants for a given computing time), but in the full game environment. We restricted the test to a subset of the algorithms, as runs in the game required much more time than the runs in the simulator. In particular, we excluded the 100ms variants from further evaluation as adding more time does not create a practical difference while keeping the computing time low increases reactivity of the system. We have also excluded the random baseline from further evaluation as it is clearly outperformed by even the 1ms variant of all algorithms. We also included the default AI in the tournaments.

Each duel consisted of 75 random scenarios evaluated according to the method described in section 7.5.5 (each scenario ran twice with the agents changing sides). Note that with 75 duels, a win rate above 0.62 or below 0.38 is statistically significant at the 0.05 level (calculated using prop.test in the R statistical software (R Core Team 2015)). The results are shown in Tables 23 - 28.

	AB-1	**AB-20**	**AB-50**	**Default**
AB-1		0.32	0.43	0.63
AB-20	0.68		0.53	0.71
AB-50	0.57	0.47		0.75
Default	0.37	0.29	0.25	

Table 23: Results of in-game tournament among all time variants of the Alpha-Beta algorithm and the default AI.

	UCT-1	**UCT-20**	**UCT-50**	**Default**
UCT-1		0.28	0.28	0.96
UCT-20	0.72		0.48	0.96
UCT-50	0.72	0.52		0.95
Default	0.04	0.04	0.05	

Table 24: Results of in-game tournament among all time variants of the UCT algorithm and the default AI.

	DUCT-1	**DUCT-20**	**DUCT-50**	**Default**
DUCT-1		0.33	0.36	1.00
DUCT-20	0.67		0.52	0.96
DUCT-50	0.64	0.48		0.97
Default	0.00	0.04	0.03	

Table 25: Results of tournament among all time variants of the DUCT algorithm and the default AI.

	AB-1	UCT-1	DUCT-1
AB-1		0.45	0.41
UCT-1	0.55		0.52
DUCT-1	0.59	0.48	

Table 26: Results of in-game tournament among all algorithm variants with 1ms computation time.

	AB-20	UCT-20	DUCT-20
AB-20		0.39	0.40
UCT-20	0.61		0.49
DUCT-20	0.60	0.51	

Table 27: Results of in-game tournament among all algorithm variants with 20ms computation time.

	AB-50	UCT-50	DUCT-50
AB-50		0.33	0.35
UCT-50	0.67		0.57
DUCT-50	0.65	0.43	

Table 28: Results of in-game tournament among all algorithm variants with 50ms computation time.

The results show very similar structure to the results of duels performed in the simulator, in particular that the variants of AB are consistently worse than both the corresponding UCT and DUCT variants, but most of the differences could easily be noise. The similarity of results in the game to the results in the simulator suggests that the abstract model does indeed match the actual game mechanics. We further see that all algorithm variants perform much better than the default AI, which once again indicates that the abstract model is useful in the actual game.

Although the differences in algorithm performance in the simulator and in the game do not show any strong differences between the individual algorithms, we decided to favor DUCT for further evaluation as it felt the best during initial testing with humans. In particular, since DUCT does not expect the opponent to be able to react to the NPCs move in the root node (if the root node is simultaneous), it was able to prevent some awkward situations encountered by UCT and AB when facing a human player. Especially, both UCT and AB often waited for opponent's attack even when they were in a better situation than the opponent, as they expected the opponent to choose the best defense once the NPC chose attack type. But DUCT was able to model that the opponent cannot react to the exact action the NPC chooses and attacked in that situation. While DUCT may not exhibit the optimal behavior, this trait was of big practical importance as it did not let the combat to stall. We therefore chose DUCT as the preferred algorithm for further testing with humans.

7.6.3 Evaluation with Human Users

Although comparing algorithms against each other is important, the bottom line is how players interact with NPCs using the given algorithm. To measure this, we ran a human study. Since the whole development of the AI focused on difficulty (making the AI hard to win against) and not on perceived fun, the main research questions of

the human study is to what extent do different time variants of a search-based algorithm influence the combat difficulty for human players. In the experiment, fun is only a secondary variable.

7.6.3.1 Participants

We have recruited 25 subjects – 2 female and 23 male. The age varied between 15 and 30 (median 23). Twelve subjects (48%) had completed a university degree (either Bc. or MSc.), twelve subjects (48%) have completed high school and were studying university, one subject (the youngest one) had completed only elementary school and was studying high school. Majority of the subjects were studying/have studied science or technology (72%). Only a minority of the subjects (16%) were not playing games regularly (reported less than one hour per week spent playing games on average), while 44% of subjects reported playing games for more than 10 hours a week on average. The subjects reported experience with wide range of games of all genres without any genre being clearly more prevalent.

7.6.3.2 Procedure

The core of the experiment is that players play a set of duels against three types of opponents: the default AI and a weak and a strong version of the search-based AI. Based on the results in sections 7.6.1 and 7.6.2 we chose to use DUCT-50 as the strong variant (note that little performance was gained beyond 50ms) and DUCT-1 as the weaker variant. We chose a big difference in time limit to maximize the difference in difficulty, as power analysis showed that the sample size we planned for could only detect a relatively large difference.

The opponents were color-coded so the subjects could find specific strategies against individual opponents, but they did not know which algorithm the opponents use. The color-to-algorithm mapping is randomized for each subject. We then evaluate both objective difficulty and self-reported difficulty and fun. The objective difficulty consisted of *player wins* – the fraction of the duels the player won against the algorithm and average difference in health between the subject and the opponent at the end of the combat (referred to as the *health value*). For simplicity, all fights are performed with the same weapon (broad sword). The protocol was tested and improved in pilot studies with 8 subjects in total.

Let us now delve into the details of the experiment. First, we let the players get comfortable with the combat mechanics – the players complete the in-game combat tutorial and then fight in the arena used for the main round of experiments against an opponent controlled by the random baseline. The latter set of fights had the exact same stats and equipment for both the subject and the NPC as in the actual experiment. During this training phase, the experiment administrator had a list of mechanics that have to be explicitly mentioned and explained to the subject. Moreover, the administrator answered to any inquiries raised by the subject in connection with the combat mechanics. The training phase was ended after the subject declared they feel comfortable with the controls and game mechanics. The subjects could choose to control the game with mouse and keyboard or an Xbox controller. The training phase took between 30 and 50 minutes.

After the training phase, the subjects filled in the *pre-questionnaire* consisting of the Flow Short Scale (Engeser and Rheinberg 2008) and self-reported assessment of the subject's understanding of the combat mechanics and his ability to use the chosen control scheme. The subjects also completed a short *understanding test* with questions testing the subject's knowledge of the core combat mechanics. Once the

subjects completed the test, the correct answers were revealed by the experiment administrator and the administrator re-explained the mechanics that were not correctly understood by the subject. After this point the administrator provided no more feedback to the subject.

The body of the experiment consisted of two almost identical parts. In each part, the subject fought 15 duels (5 with each opponent) in fully randomized order. For one part the opponent had heavy armor and slightly lower skill (chance for perfect block and dodge) while for the other part the opponent had light armor and slightly higher skill. The order of the parts was randomized. The duels were run in a flat enclosed arena (see Figure 17).

In the experiments, the defense skills of the player are set relatively high, which results in a relatively long time slots for perfect block and dodge. The motivation was to let subjects with slower reflexes and lower gaming experience to be able to perform perfect block or dodge with non-negligible probability. Otherwise, subjects with lower skill would effectively have limited number of defensive choices, reducing the effect of conscious intelligent choices on the outcome of the duel. The exact numbers were tweaked during the pilot studies.

After each part, the subjects completed the *mid-questionnaire* where they rated the overall performance of the individual opponents. The two parts took between 30 and 60 minutes in total, including the questionnaires.

Since the pilot showed that some subjects had difficulty recalling the individual opponents when completing the mid-questionnaire, we added a short *immediate questionnaire*. The participants were asked to fill the questionnaire twice for each opponent in each part (the duels after which the questionnaire appeared were chosen randomly). In contrast with the mid-questionnaire, the participants were instructed to report the perceived difficulty of the single duel that preceded the immediate questionnaire. Further testing in the pilot showed that the immediate questionnaires are filled out quickly and the subjects did not consider them intrusive.

Finally, the subjects completed a *post-questionnaire* investigating the gaming experience of the subject and the demographic information reported above.

The most important experimental condition is the algorithm controlling the opponent. The equipment condition (differing in the two parts of the experiment) is only auxiliary.

7.6.3.3 Questionnaires

The most important variables measured by the questionnaires are subjective difficulty (measured by both immediate and mid-questionnaire) and subjective fun (measured by mid-questionnaire). In the mid-questionnaire the difficulty is measured using a 4-item 7-point Likert scale separately for each of the colors assigned to the opponents. The items in the *mid-difficulty scale* were:

1. For me, <color> was a difficult opponent.
2. I felt helpless against the <color> opponent.
3. The <color> opponent took advantage of the mistakes I made.
4. The <color> opponent made mistakes I could take advantage of.

Here, item 4 was coded negatively.

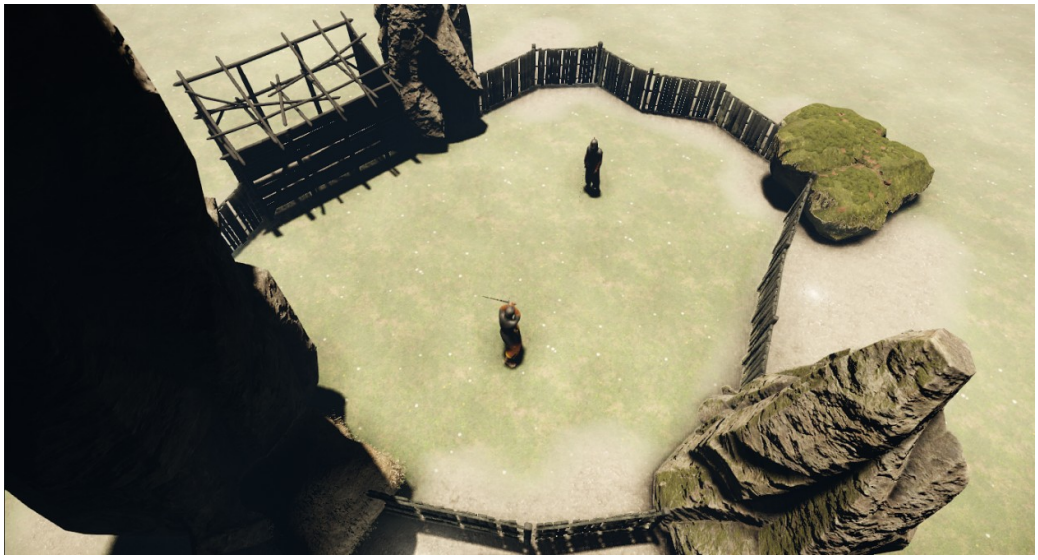


Figure 17: The arena used for experiments with human subjects.

The mid-difficulty was complemented by a 2-item 7-point Likert scale that was the sole contents of the immediate questionnaire. When filling in the immediate questionnaire the subjects were instructed to reflect only on the duel preceding the questionnaire. The items of the *immediate difficulty scale* were the two most salient items from the mid-difficulty scale (as observed in the pilot):

1. For me, the enemy was a difficult opponent.
2. The enemy made mistakes I could take advantage of.

Item 2 was coded negatively.

Fun was measured only in the mid-questionnaire with the following 4-item 7-point Likert *fun scale* (once again separately for each opponent color):

1. Fighting the <color> opponent was fun.
2. I enjoyed fighting the <color> opponent.
3. Fighting the <color> opponent was boring.
4. Fights with the <color> opponent were dull.

Items 3 and 4 were coded negatively. For control, the mid-questionnaire also contained open-ended questions to describe the individual opponents and the tactics the subjects used against them and questions for explicitly comparing all three pairs of opponents. To monitor the overall mental state of the subjects, the mid-questionnaire also contained the flow scale, the enjoyment and competence scales of the Intrinsic motivation inventory (Ryan 1982; McAuley et al. 1989) and a 6-item, 7-point Likert *self-assessment scale* (item 1 was coded negatively):

1. While playing, I just randomly mashed buttons.
2. I think I can be successful in fighting the computer.
3. I think I understand how the combat in the game works.
4. During combat, I was aware of what I am doing and I planning ahead.
5. During combat, when I realize, what I want to do, it is easy for me to press the right buttons at the right time.
6. During combat, I knew exactly what I wanted to do in combat.

The pre-questionnaire was used to check whether the subject is not tired after the training phase of the experiment, it contained the flow scale, a 4-item subset of the self-assessment scale (items 1 and 4 excluded) and several additional questions to check the attitude of the subject toward the experiment.

The questionnaires were filled in on a computer using the Google Forms service. The full questionnaires as given to the users can be found in the digital attachment to this thesis.

7.6.3.4 Analysis

To reliability of the custom scales described in the previous section was checked using Cronbach's α (Cronbach 1951). The α ranges from 0 to 1 and $\alpha > 0.8$ is generally considered reliable. We calculated α using the alpha function in the R package psych (Revelle 2015). The mid-difficulty and fun scales had high internal reliability ($\alpha = 0.88$ and 0.90 respectively), while the immediate difficulty scale (consisting of only two items) had slightly worse, but acceptable reliability ($\alpha = 0.83$). The self-assessment scale also had good reliability ($\alpha = 0.89$). In all cases, the reliability decreased with the exclusion of any of the items.

We have determined that any subject scoring less or equal to 2 (out of 7) on the self-assesment scale should be excluded. Nevertheless no subject has reached the exclusion criteria in either part of the study (the observed minimum was 2.33).

The main dependent variables of interest are the objective difficulty measures (fraction of player wins and mean health value) and mean of the subjective difficulty measures (mid-difficulty and immediate difficulty). The last dependent variable – mean fun – is only auxiliary. The player win variable ranges from 0 to 1, the health value variable ranges from -100 to 100, all of the subjective scales range from 1 to 7.

As not all of the variables we measure are normally distributed, we report median and median absolute difference (MAD) as descriptive statistics.

We expect there to be large differences in player skill, so our analysis focuses primarily on the within-subject difference of the dependent variables for each pair of algorithms. In other words, for each subject and pair of algorithms we subtract the respective dependent variables. In accordance with the methodology of "new statistics" (Cumming 2014) we inspect the mean and 95% confidence interval for all of those differences. As all of the differences were normally-distributed, we calculate the confidence interval using the t.test method in the R statistical software (R Core Team 2015). In our analysis, the mean differences are the effect size.

The design of the study and the statistical procedure to evaluate the main results was preregistered with the AsPredicted authority as study No. 518. The preregistered information is available both as an appendix to this thesis and online[12]. The data were collected between March 14th and April 2nd 2016[13]. All data as well as R scripts to perform all the analyses we report are available in the digital attachment to this thesis.

[12] https://aspredicted.org/public/206052895.pdf

[13] The first day of data collection coincides with the date of the preregistration, but the final version of the preregistration report was submitted before any data were collected.

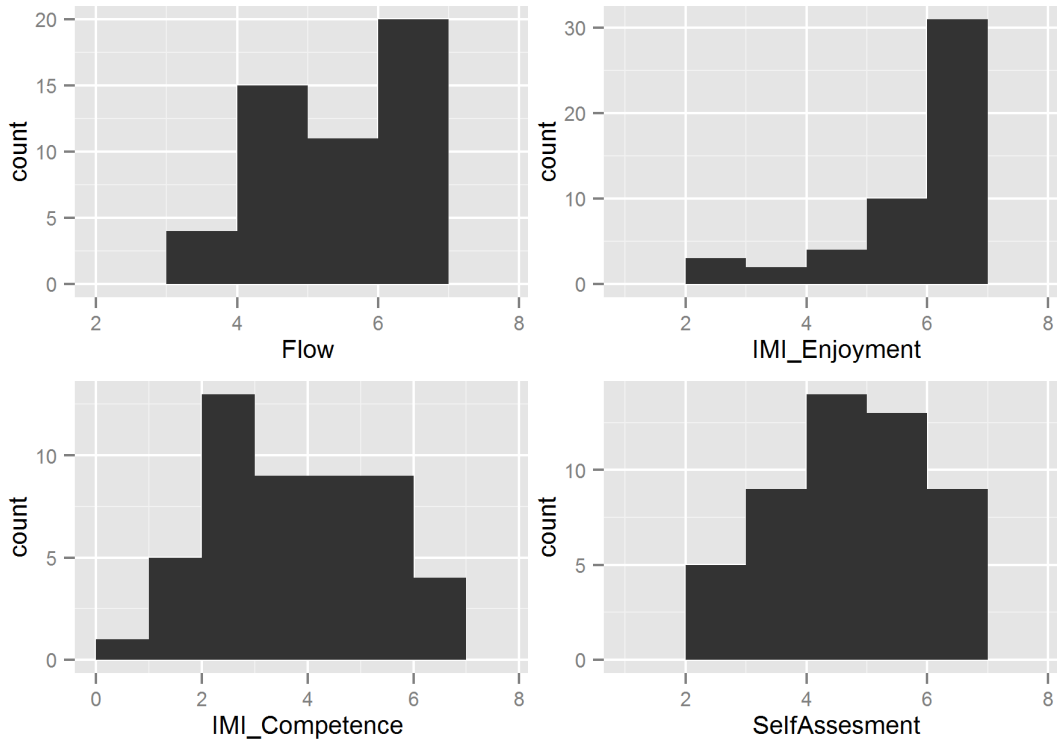


Figure 18: Histograms of reported psychometric values.
Each participant is counted twice - once for each part.

7.6.3.5 Results Inspection

Before analyzing the main effects in this study, we first inspect the data to check for possible confounding factors. First, we were interested to what extent does the immediate difficulty reflect whether the subject has won the last duel. The correlation between the immediate difficulty scale and the result of the duel and health value respectively is -0.67 and -0.75 so it is likely that the outcome of the duel is not the sole predictor of the reported difficulty. The correlation is negative as high score in the difficulty scale represents difficult matches while high score in player win or health value corresponds to good outcomes for the subject.

We also note that players had generally reported high flow and high enjoyment (median 5.75 ± 1.26 and 6.4 ± 0.89 respectively[14]) while the reported competence and self-assessment are much more spread out towards the low values (median 3.67 ± 1.73 and 4.92 ± 1.6 respectively). See Figure 18 for histograms[15]. We thus note that while subjects enjoyed dueling the NPCs, the mechanics were difficult for most of them.

To check how the experience of the subjects evolved over time, we examine the within-subject difference between the values reported for the individual phases. The result is that most measures have increased but very slightly (median differences – flow: 0.1 ± 0.44, enjoyment: 0 ± 0.59, competence: 0.33 ± 0.49, self-assessment: 0.33 ± 0.49). This is further supported by the small within-subject differences

[14] As described in Section 7.6.3.4, we report median and median absolute difference, as some of the variables are not normally distributed.

[15] All graphs were created using the ggplot2 package (Wickham 2009).

between individual phases in the main dependent variables (median differences – player win: 0.07 ± 0.2, health value: 6.8 ± 41, difficulty immediate: -0.08 ± 0.61, difficulty mid: -0.17 ± 0.61, fun: 0.08 ± 0.61). We can thus conclude that the users have generally not been increasingly bored or frustrated by the experiment, neither has their skill substantially changed throughout the experiment (the effects of increased experience and fatigue were either low or mostly cancelled each other out).

7.6.3.6 Algorithm Comparison

We start the algorithm comparison informally by inspecting the boxplots of the main dependent variables (see Figure 19). While all four dependent variables visually show the same ordering of all algorithms, with DUCT-50 as the most difficult and the default AI as the least difficult, we need to be careful to quantify this difference correctly. First we should note that not all of the main dependent variables are normally distributed: the most extreme case is the health value which is bimodal even when averaged per player and algorithm. Overall player win is also not normally distributed (see Figure 20) Second important thing visible in Figure 20 is that there are large differences in player performance: some players won over 20 of the 30 duels, while many won less than 5. However, examining the within-subject differences per algorithm pair gives us a set of approximately normally-distributed values (see Figure 21).

We then inspect the mean and 95% confidence interval for all of those differences (see Table 29 and Figure 22). In general we see that the mid-difficulty has longer confidence intervals (higher variance) than the immediate difficulty, but except for the difference in mid-difficulty between DUCT-1 and DUCT-50, the confidence intervals do not contain zero, indicating significant difference. Further, for all the differences between DUCT-50 and the default AI, the confidence interval is relatively far from zero indicating high confidence that the difference is real and the effect size is large. The differences between DUCT-1 and the default AI are smaller but still noticeable even with the limited sample size of the study. Overall we can conclude that the algorithms are ordered by difficulty as expected: default AI is easiest while DUCT-50 is the most difficult.

Initially, we feared that the high difficulty of the AI will be frustrating to the subjects and perceived fun will be negatively influenced by increasing difficulty. This however did not happen – from the experiments performed, there are no significant differences in the fun variable (lower bound of all CIs is lower than -0.2 and the upper bound is always higher than 0.7). Although it is possible that significant differences could be shown with larger sample size, it is unlikely that the effect is large. See Figure 23 for details.

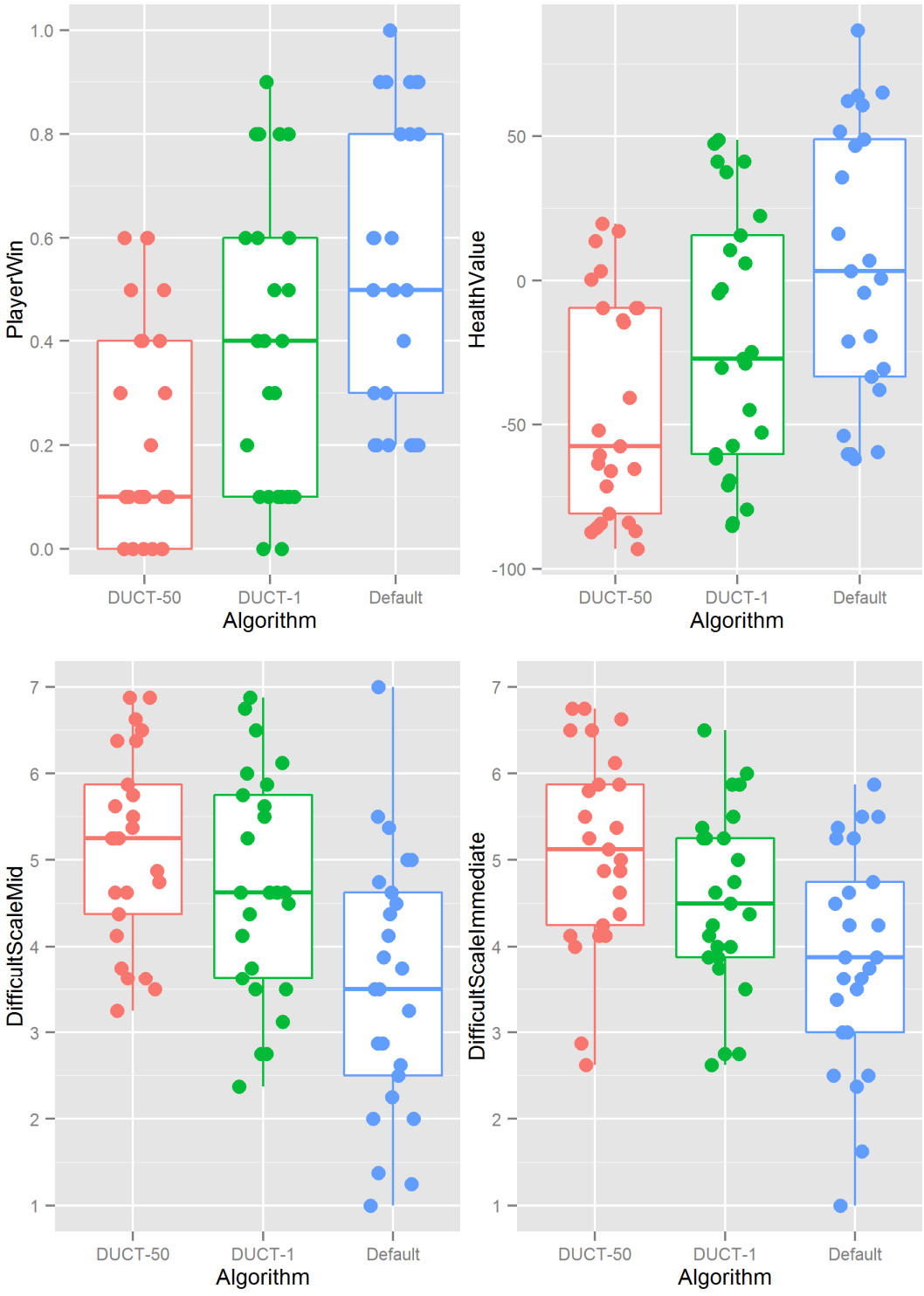


Figure 19: Box plots of main dependent variables per algorithm.
The box spans from 1st to 3rd quartile with a band at median; the whiskers extend to 1.5 times the inter-quartile range or the maximum/minimum of the data, whichever is smaller. Each dot corresponds to a single participant-algorithm combination.

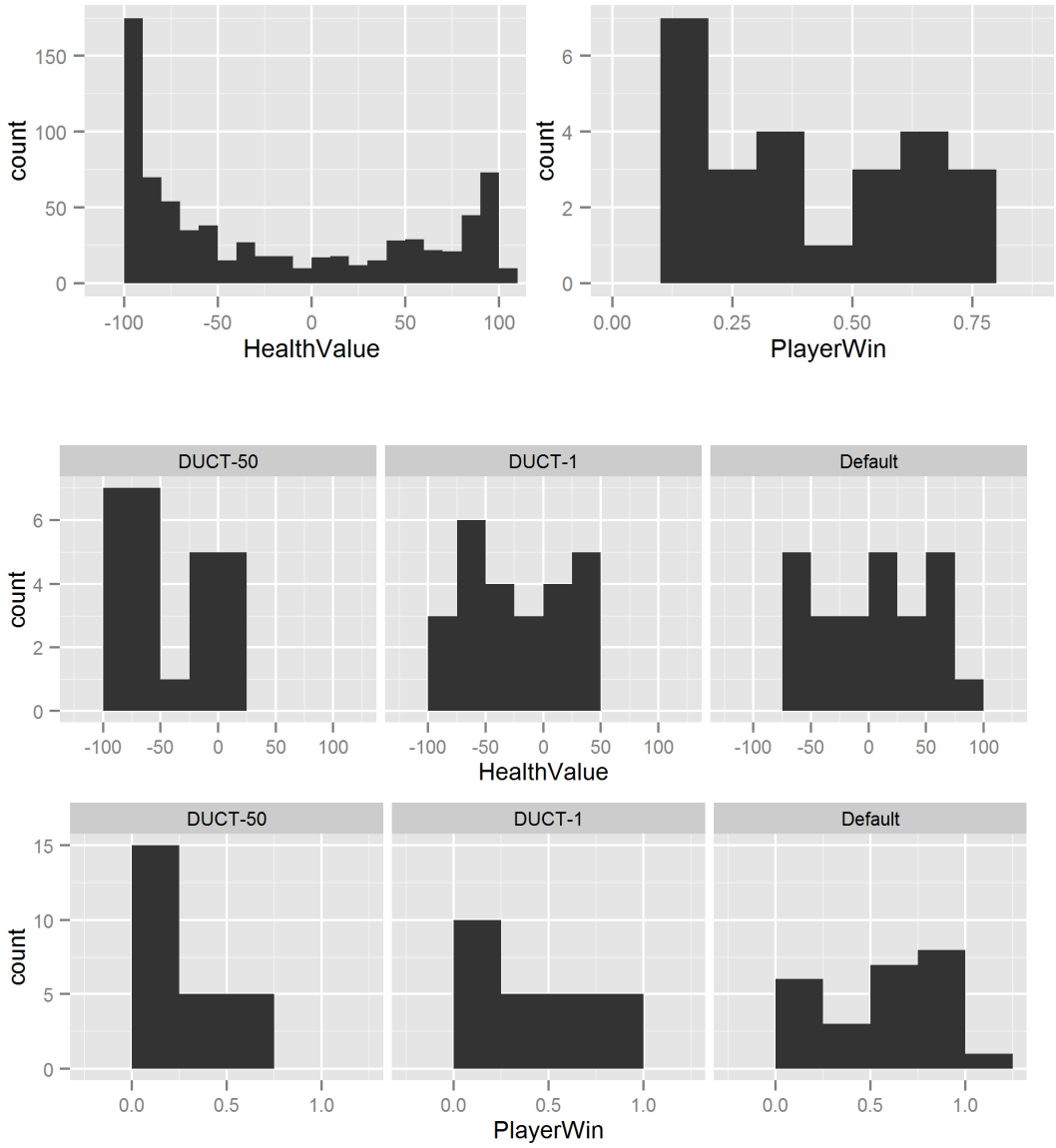


Figure 20: Histograms of the objective difficulty variables.
Top-left: overall histogram for health value (each duel counted once),
Top-right: histogram of fraction of wins per player (each subject counted once),
Middle: mean health value per subject aggregated by algorithm (each subject counted once),
Bottom: fraction of wins per subject aggregated by algorithm (each subject counted once).
Note that the variables cannot be considered to be normally distributed.

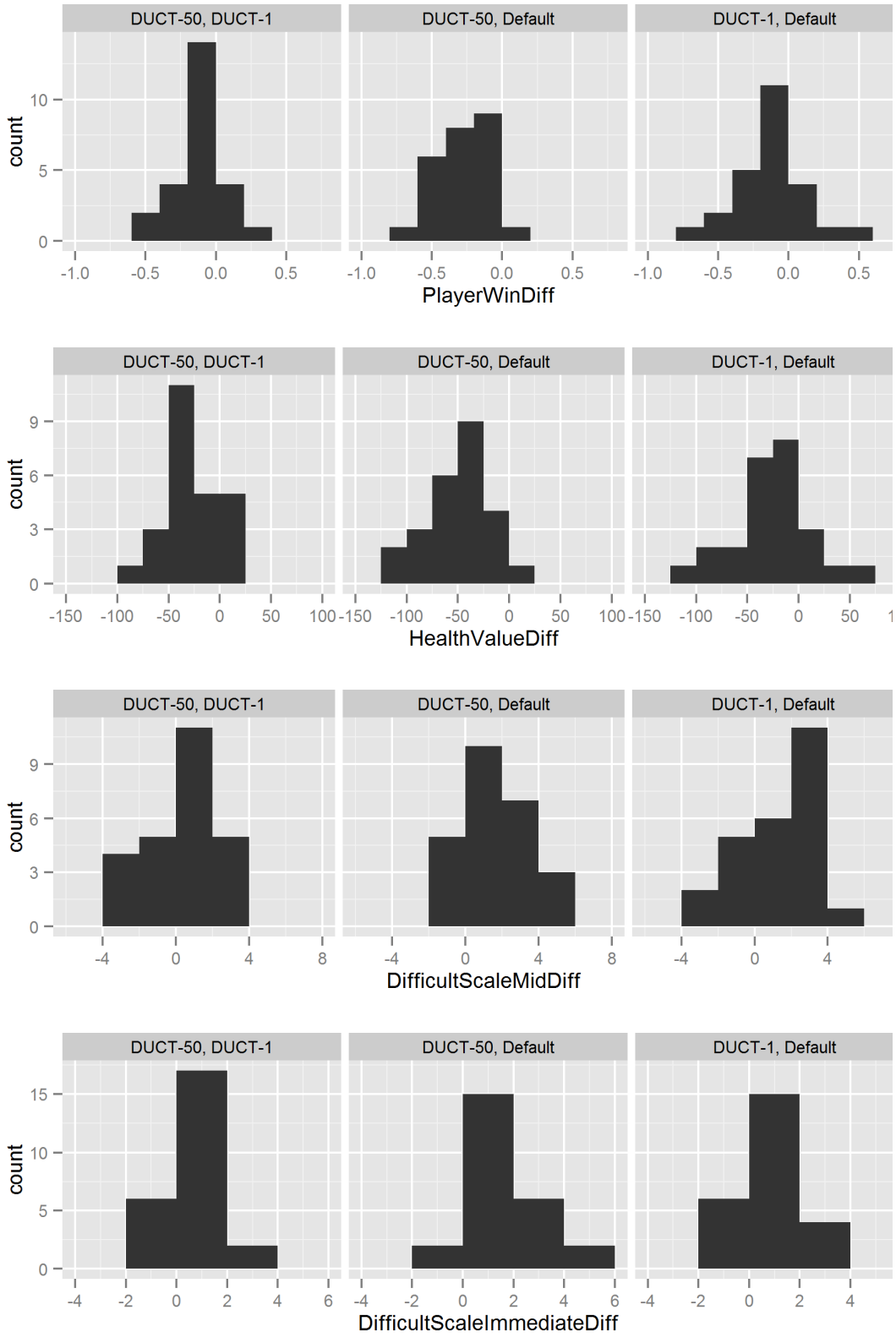


Figure 21: Histograms of the within-subject differences of the main dependent variables for each algorithm pair (each player counted once per algorithm pair).

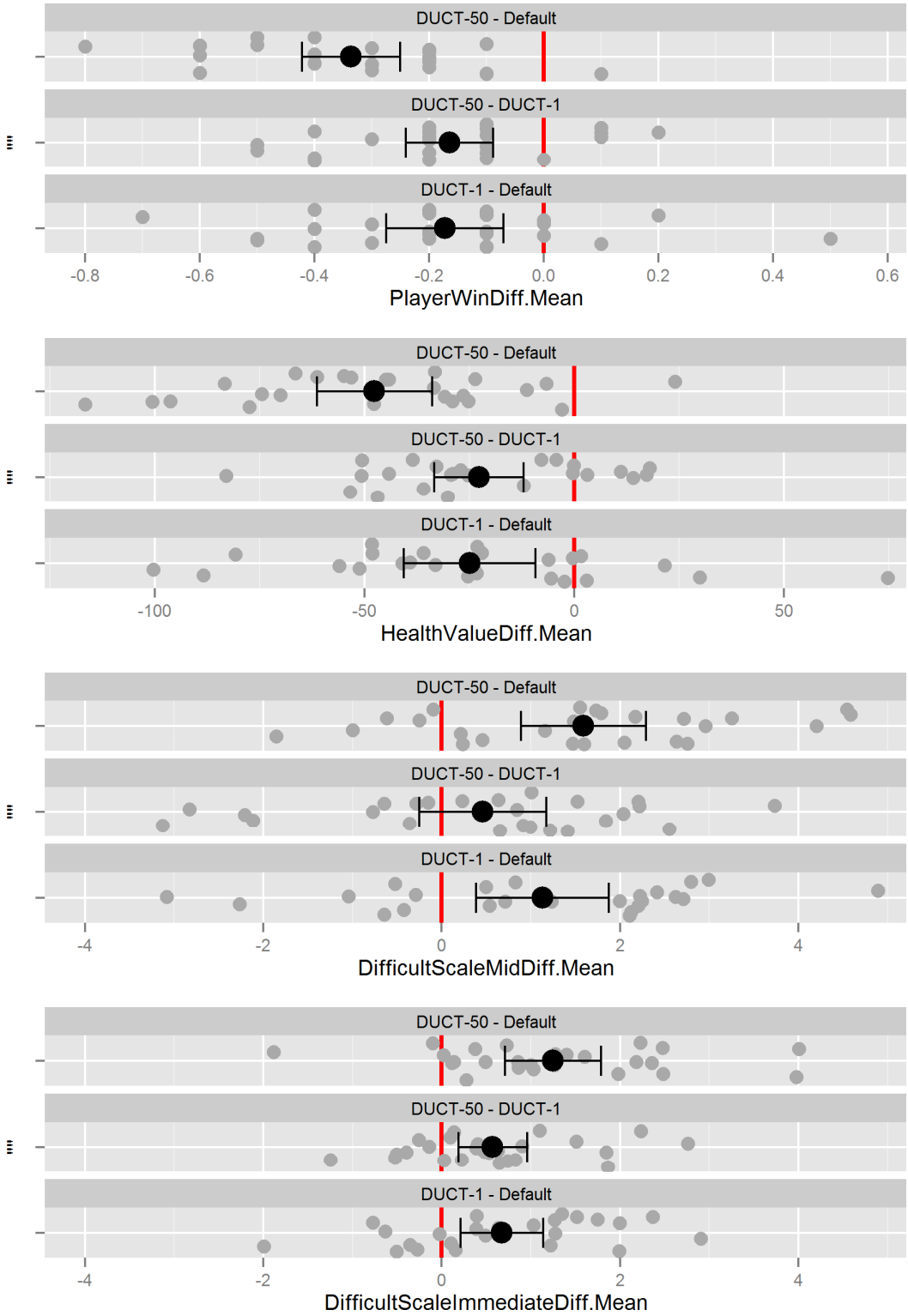


Figure 22: Confidence intervals (CIs) and the overall distribution of differences in the main dependent variables.
Each gray dot represents the difference of one subject, the black dots represent the mean values and the whiskers represent 95% CIs. Red line marks zero.

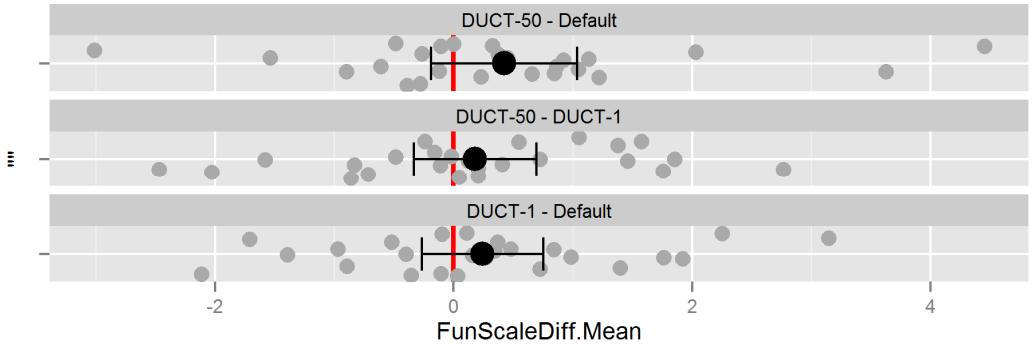


Figure 23: Confidence intervals (CIs) and the overall distribution of differences in the fun dependent variable.
Each gray dot represents the difference of one subject, the black dots represent the mean values and the whiskers represent 95% CIs. Red line marks zero.

	DUCT-50 – Default		DUCT-50 – DUCT-1		DUCT-1 – Default	
	Mean	**CI**	**Mean**	**CI**	**Mean**	**CI**
Player Win	-0.34	-0.42 ; -0.25	-0.16	-0.24 ; -0.09	-0.17	-0.27 ; -0.07
Health value	-47	-61 ; -34	-23	-33 ; -12	-25	-40 ; -09
Mid difficulty	1.59	0.9 ; 2.3	0.46	-0.3 ; 1.2	1.13	0.4 ; 1.9
Immediate difficulty	0.42	0.7 ; 1.8	0.57	0.2 ; 1.0	0.68	0.2 ; 1.1
Fun	0.43	-0.2 ; 1.0	0.18	-0.3 ; 0.7	0.25	-0.3 ; 0.8

Table 29: Mean and 95% confidence intervals (CI) for differences of the main dependent variables between algorithms.
Note that except for the difference in mid difficulty between DUCT-50 and DUCT-1, and the Fun variable, no CI contains zero.

7.7 Discussion

The combat model presented in this chapter proved successful in controlling NPCs in one-on-one combat and was more difficult for human subjects to conquer than the default combat AI of the alpha version of KC:D without showing any detrimental effects on fun. While this suggests, that search-based adversarial approaches are in general a feasible alternative to both reactive and non-adversarial deliberative methods for AI components with low size complexity but large rule complexity, there are numerous caveats. The biggest issue is that combat in KC:D proved to not be a particularly challenging scenario from the perspective of intelligence – performance in combat is dominated by skill and the performance of the search algorithms quickly leveled out with increased time available. Further, the best parameters we found for the UCT and DUCT algorithms (Section 7.5.6) suggest that a very greedy approach is preferred. Moreover, as discussed in Section 7.6.1, the differences among algorithms were minimal. A possible interpretation is that all algorithms quickly differentiate the generally good decisions from the generally bad ones and it is difficult to significantly improve over this coarse classification.

One of the biggest drawbacks of the search-based approaches in general is that due to heavy optimizations and abstraction, the model for search cannot simply reuse

code with the main game mechanics, so the mechanics need to be kept in sync between the game and the model, reducing the manageability of the codebase. Further, there is a mutual dependency between the search model and the scripts that execute the decisions in the game: optimizing the model often requires a modification to the model's action set, requiring new or modified scripts, while executing the scripts in the full game environment may reveal problems that need to be addressed in the model (as for the AttackNow and AttackOnApproach actions described in Section 7.5.2.1). To an extent, the model and the scripts need to be iteratively developed together.

The example of AttackNow and AttackOnApproach actions also demonstrates one further issue with the search-based approach – a very small change in the model (adding an action) can result in a big change in the behavior of the NPCs. A related problem is that while most changes to game mechanics can be mirrored in the model with ease (e.g., a different formula for calculating damage), others may break the very assumptions on which the model is made and force a huge restructuring of the model (e.g., making maneuvering in 2D space a fundamental mechanic) or, in the worst case, make search-based approach unfeasible. In this sense, the model can be very fragile.

For the reasons above, the model proposed in this chapter will likely not be adopted for AI in KC:D, at least not in the short term. Should the model by used in the actual game, some further developments would be needed, especially combat with multiple NPCs would have to be supported. A possible approach to this is to represent additional combatants on either side as nature. Since combatants can directly interact with only one opponent, additional opponent could be reasonably modeled as a deterministic machine that attacks in regular intervals and whose attacks cannot be blocked.

8 Discussion

To conclude the present work, we first summarize the contributions of this thesis and then discuss directions for future work.

8.1 Summary of the Main Contributions

In this work we have presented three novel techniques to handle complexity of AI in OWGs. First have described a general way for structuring AI code in a manner similar to OOP – the BOs (Subgoal 1, Chapter 5), second we have shown how constraint programming can be employed to specify behaviors from a global viewpoint in complement to the local view of BOs (Subgoal 2, Chapter 6). Last, we have presented the use of adversarial search to mitigate the need for complex and parameter-sensitive hand-crafted scripts (Subgoal 3, Chapter 7). The overall goal of introducing and evaluating new techniques that extend the capabilities of game development teams to build AI in complex scenarios has thus been fulfilled. We believe that the techniques presented in this work have the potential to become "AI design patterns" for future games.

Since we accompanied all of those techniques with a prototype implementation fully integrated in a complete OWG, we can be certain that no major obstacles for use in practice have been swept under the rug. We also did not restrict ourselves to technical evaluation and also evaluated the developer-facing techniques (BOs and global specification) with developers and the more player-facing adversarial search with human players.

We consider BOs a mature technology that is ready to be applied in many game contexts and was heavily tested in actual use. Using constraint programming for global specification of behaviors is, in our view, less mature as it was not yet used in production environment. We are however confident that the method is useful in its present state, which is backed by its evaluation in test scenarios. The situation system is also scheduled to be used in a production build of KC:D in the near future. The use of adversarial search for controlling NPCs has proved promising, but is definitely the least production-ready of the contributions of this thesis.

The main problem we encountered during our case study with adversarial search was that the reduction of complexity in the high-level AI code came at cost of duplicated implementation of some of the game mechanics and increased fragility of the code at the lower level. At the same time, some of the benefits of adversarial search are also provided by planning approaches, which have similar implementation cost, but are computationally cheaper. We therefore conclude that adversarial search will likely be useful only in a small subset of games where the advantages of a goal based approach that explicitly models the opponent offset both development and computational cost.

In a broader perspective, designing and validating AI techniques with respect to their ability to mitigate complexity has shown to be relevant for practical applications. The analysis of complexity also moved attention from individual NPCs to whole AI components which, in our opinion, better reflects the considerations for a practical implementation. Often the effort required to implement the behavior of a single NPC is not proportional (for the better or for the worse) to the effort required to populate the world with many different NPCs. We have proposed a framework for

classifying the complexity of AI components which should be useful to explicitly consider the target type of complexity when developing future AI techniques. The complexity classification further has the potential to make claims about usefulness of a given technique more specific by emphasizing the type of complexity it aims to deal with.

While we focused primarily on OWGs, all of the techniques and analyzes described in this thesis should translate easily to many other game genres that feature a rich environment and where AI is an important part of the game.

8.2 Future Work

Handling complexity is a general task of OWG AI development – it is highly unlikely that any single approach would be the silver bullet that solves all problems and thus many areas for future work emerge.

Further development beyond BOs may be possible by taking inspiration from other programming paradigms that have matured enough to attain widespread adoption, in particular from functional programming.

It may also be possible to avoid developing behaviors of low rule complexity by hand completely and rely instead either on learning approaches or on automated programming by example.

The global view for behavior specification using CSP should be further validated in different use cases (e.g., generating quests, monitoring overall game state). It may also be possible to employ more powerful CSP variants (e.g., constraint optimization) or completely different, but declarative approaches (e.g., answer set programming).

Since the need to develop and maintain an abstract model is an important disadvantage of search-based methods, techniques to extract such models directly from the game would be an important step forward. Recent advances in both machine learning and program analysis could make automatic model extraction feasible and practical. An orthogonal approach would be to devise languages for description of game mechanics that could be interpreted as both a continuous model that can drive the evolution of the game world and as a discrete abstract model amenable to search.

While we have shown how to manage large size complexity and large rule complexity individually, it is still unclear how to handle cases with both large rule and large size complexity. Promising approaches in that direction include combining search with machine learning to learn both useful representations for search and good heuristics or pruning functions.

List of Author's Publications

A) Publications whose results were used in this thesis

1. **Černý, M**., Plch, T.,Marko, M., Gemrot, J., Ondracek, P., Brom, C.: Using Behavior Objects to Manage Complexity in Virtual Worlds. In *IEEE Transactions on Computational Intelligence and AI in Games*, early view. doi: 10.1109/TCIAIG.2016.2528499
2. **Černý, M.**, Plch, T, Brom, C.: Beyond Smart Objects: Behavior-Oriented Programming for NPCs in Large Open Worlds. In Lengyel, Eric (eds.) *Game Engine Gems 3*. USA: CRC Press, 2016. pp. 267-280. ISBN 978-1-4987-5565-8.
3. **Černý, M.**, Brom, C., Barták, R., Antoš, M.: Spice it up! Enriching Open World NPC Simulation Using Constraint Satisfaction In: *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2014)*, pp. 16 - 22, (2014). **Best paper award.**
4. Plch, T., Marko, M., Ondracek, P., **Černý, M.**, Gemrot, J., Brom, C.: An AI System for Large Open Virtual World In: *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2014)*, pp. 44-51, (2014)
5. Plch, T., Marko, M., Ondracek, P., **Černý, M.**, Gemrot, J., Brom, C.: Modular Behavior Trees: Language for Fast AI in Open-World Video Games In: *Proceedings of 21st European Conference on Artificial Intelligence (ECAI 2014)*, pp. 1209-1211 , (2014)
6. **Černý, M.**, Plch, T., Marko, M., Ondracek, P., Brom, C.: Smart Areas: A Modular Approach to Simulation of Daily Life in an Open World Video Game. In: *Proceedings of 6th International Conference on Agents and Artificial Intelligence (ICAART 2014)*, Volume 1, pp. 703-708, (2014)
7. **Černý, M.**, Gemrot, J.: HTN or State Space - Who Should Do Planning in Your Game? In: *Proceedings of GAMEON'2013*, pp 59--65. (2013) **Best paper award.**
8. Bartak, R., Brom, C., **Černý, M.**, Gemrot, J.: Planning and Reactive Agents in Dynamic Game Environments: An Experimental Study. In Joaquim Filipe, Ana Fred (eds.): *Proceedings of 5th International Conference on Agents and Artificial Intelligence (ICAART 2013)*, Volume 1, pp. 234-240, (2013)

B) Other publications

9. **Černý, M.**, Barták, R., Brom, C., Gemrot, J.: To Plan or to Simply React? An Experimental Study of Action Planning in a Game Environment In: *Computational Intelligence* (Early View) DOI: 10.1111/coin.12079
10. **Černý, M.**: Sarah and Sally: Creating a Likeable and Competent AI Sidekick for a Videogame In: *Experimental AI in Games: Papers from the AIIDE 2015 Workshop (EXAG 2015)*, pp 2 – 8, (2015).

11. Cook, M., Eiserloh, S., Robertson, J., Young, M., Thompson, T., Churchill, D., Č**erný, M.**, Hernandez, S. P., Bulitko, V.:Playable Experiences at AIIDE 2015. In: *Proceedings of Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (AIIDE 2015), pp. 227 - 231, (2015).
12. Hromada, T., **Černý, M.**, Bída, M., Brom, C.: Generating Side Quests from Building Blocks In: *ICIDS 2015*. LNCS, vol. 9445, pp 235 – 242, Springer (2015)
13. **Černý, M.**, Moens, M.-F.: aMUSE: Translating Text to Point and Click Games. In: *Proceedings of the 41st Convention of the AISB*, pp. 42-43 (2015)
14. Bída, M., **Černý, M**., Brom, C.: Follow-up on Automatic Story Clustering for Interactive Narrative Authoring. In: *Proceedings of the 41st Convention of the AISB*, pp. 37 - 41 (2015)
15. Holaň, D., Gemrot, J., **Černý, M.**, Brom, C.: EmohawkVille: Virtual City for Everyone. In: *Proceedings of the 41st Convention of the AISB*, pp. 23 - 24 (2015)
16. Gemrot, J., Brom, C., **Černý, M.**: Teaching Intelligent Virtual Agents Programming through Simulated Children Games In: *Proceedings of GAMEON'2014*, pp. 43 - 49 (2014)
17. Gemrot, J., **Černý, M.**, Brom, C.: Why you should empirically evaluate your AI tool: From SPOSH to yaPOSH. In: *Proceedings of 6th International Conference on Agents and Artificial Intelligence (ICAART 2014)*, Volume 1, pp. 461-468, (2014)
18. Gemrot, J., **Černý, M.**, Brom, C.: Teaching How to Engineer Behaviors for Videogame Characters. In: *SplashE*, Portland, USA, 2014. Oral presentation.
19. Bída, M., **Černý, M.**, Brom, C.: Towards Automatic Story Clustering for Interactive Narrative Authoring. In: Koenitz, H., Sezen, T.I., Ferri, G., Haahr, M., Sezen, D., Çatak, G. (eds.) *ICIDS 2013*. LNCS, vol. 8230, pp. 95--106. Springer, Heidelberg (2013)
20. Bída, M., **Černý, M.**, Brom, C.: SimDate3D - Level Two. In: Koenitz, H., Sezen, T.I., Ferri, G., Haahr, M., Sezen, D., Çatak, G. (eds.) *ICIDS 2013*. LNCS, vol. 8230, pp. 128--131. Springer, Heidelberg (2013)
21. Holaň, D., Gemrot, J., **Černý, M.**: EmohawkVille: Towards Complex Dynamic Virtual Worlds. In: *Proceedings of GAMEON'2013*, pp. 52—58 (2013).
22. Bída, M., **Černý, M.**, Gemrot, J., Brom, C.: Evolution of GameBots project. In: Herrlich, M., Malaka, R., Masuch, M. (eds.) *ICEC 2012*. LNCS, vol. 7522, pp. 397--400. Springer, Heidelberg. (2012)
23. Kadlec, R., Toth, C., **Černý, M.**, Bartak, R., Brom, C.: Planning Is the Game: Action Planning as a Design Tool and Game Mechanism. In: *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2012)*, pp. 160-166, AAAI Press, (2012)

Bibliography

ALI, M. L. and GOODWIN, S. D., 2008. Applications of CSP solving in camera control. *5th IEEE Consumer Communications and Networking Conference*. 2008. p. 1040–1044.

ANDRUSZKIEWICZ, Piotr, 2015. Optimizing MCTS Performance for Tactical Coordination in TOTAL WAR: ATILLA. *nucl.ai Conference* [online]. 2015. [Accessed 20 May 2016]. Available from: http://archives.nucl.ai/recording/optimizing-mcts-performance-for-tactical-coordination-in-total-war-atilla/

AUER, Peter, CESA-BIANCHI, Nicolo and FISCHER, Paul, 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*. 2002. Vol. 47, p. 235–256.

AUER, Peter, CESA-BIANCHI, Nicolo, FREUND, Yoav and SCHAPIRE, Robert E., 2002. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*. 2002. Vol. 32, p. 48–77.

BALLA, Radha Krishna and FERN, Alan, 2009. UCT for tactical assault planning in real-time strategy games. In: *International Joint Conference on Artificial Intelligence*. 2009. p. 40–45. ISBN 9781577354260.

BALLARD, Bruce W., 1983. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*. 1983. Vol. 21, no. 3, p. 327–350.

BARRIGA, Nicolas A, STANESCU, Marius and BURO, Michael, 2015. Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games. In: *The Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2015. p. 9–15.

BARTÁK, Roman, 2015. Programování s omezujícími podmínkami / Constraint Programming. [online]. 2015. [Accessed 3 June 2016]. Available from: http://kti.mff.cuni.cz/~bartak/podminky/

BERG, Jur, GUY, Stephen J, LIN, Ming and MANOCHA, Dinesh, 2011. Reciprocal n-Body Collision Avoidance. *Springer Tracts in Advanced Robotics*. 2011. Vol. 70, p. 3–19.

BETHESDA GAME STUDIOS, 2011, Elder Scrolls V: Skyrim. [online]. 2011. [Accessed 22 December 2015]. Available from: http://www.elderscrolls.com/

BJARNASON, Ronald, FERN, Alan and TADEPALLI, Prasad, 2009. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. *Proceedigns of the 19th International Conference on Automated Planning and Scheduling*. 2009. P. 26–33.

BOŠANSKÝ, Branislav, 2014. *Iterative Algorithms for Solving Finite Sequential Zero-Sum Games*. PhD Thesis. Czech Technical University in Prague.

BOURNE, Owen, SATTAR, Abdul and GOODWIN, Scott, 2008. A constraint-based autonomous 3D camera system. *Constraints*. 2008. Vol. 13, no. 1-2, p. 180–205.

BRATMAN, Michal E., 1987. *Intentions, Plans, and Practical Reason*. Cambridge: Harvard University Press. ISBN 0-674-45818-4.

BROM, Cyril, LUKAVSKÝ, Jiří, ŠERÝ, Ondřej, POCH, Tomáš and ŠAFRATA, Pavel, 2006. Affordances and level-of-detail AI for virtual humans. In: *Proceedings of the Game Set and Match*. 2006. p. 134–145.

BROOKS, Rodney A, 1986. *A Robust Layered Control System For a Mobile Robot*. Technical Report. Massechusetts Institution of Technology.

BROOKS, Rodney a, 1990. Elephants Don't Play Chess. *Robotics and Autonomous Systems*. 1990. Vol. 6, p. 3–15.

BROOKS, Rodney A., 1991. Intelligence without representation. *Artificial Intelligence*. 1991. Vol. 47, no. 1-3, p. 139–159.

BROWNE, Cameron B., POWLEY, Edward, WHITEHOUSE, Daniel, LUCAS, Simon M., COWLING, Peter I., ROHLFSHAGEN, Philipp, TAVENER, Stephen, PEREZ, Diego, SAMOTHRAKIS, Spyridon and COLTON, Simon, 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012. Vol. 4, no. 1, p. 1–43.

BRYSON, Joanna Joy, 2001. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD Thesis. Massachusetts Institute of Technology.

BURO, Michael and ONTAÑÓN, Santiago, 2015. Adversarial Hierarchical-Task Network Planning for Complex Real-Time Games. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015. p. 1652–1658.

CANARY, Aaron and CHAMPANDARD, Alex J., 2014. Scaling and Coordinating AI for Open Worlds in Saints Row IV. *AIGameDev.com* [online]. 2014. Available from: http://aigamedev.com/premium/interview/saints-row/

CARR, Peter, EISERLOH, Squirell and INGEBRETSON, Peter, 2014. Invited Talks. In: *Proceedings of the Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014. p. xvii.

CD PROJEKT RED, 2015. The Witcher 3: Wild Hunt. [online]. 2015. [Accessed 22 December 2015]. Available from: http://thewitcher.com/witcher3

ČERNÝ·, Martin, BROM, Cyril, BARTÁK, Roman and ANTOŠ, Martin, 2014. Spice It Up! Enriching Open World NPC Simulation Using Constraint Satisfaction. In: *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press. 2014. p. 16–22.

ČERNÝ, Martin, BARTÁK, Roman, BROM, Cyril and GEMROT, Jakub, 2015. To Plan or to Simply React? An Experimental Study of Action Planning in a Game Environment. *Computational Intelligence [accepted]*. 2015.

ČERNÝ, Martin and GEMROT, Jakub, 2013. HTN or State Space-Who Should Do Planning in Your Game? In: *Proceedings of GAMEON'2013*. 2013. p. 59–65.

ČERNÝ, Martin, PLCH, Tomáš, MARKO, Matěj, GEMROT, Jakub, ONDRÁČEK, Petr and BROM, Cyril, 2016. Using Behavior Objects to Manage Complexity in Virtual Worlds. *IEEE Transactions on Computational Intelligence and AI in Games [accepted]*. 2016.

ČERNÝ, Martin, PLCH, Tomáš, MARKO, Matěj, ONDRÁČEK, Petr and BROM, Cyril, 2014. Smart Areas: A Modular Approach to Simulation of Daily Life in an Open World Video Game. *6th International Conference on Agents and Artificial Intelligence (ICAART 2014)*. 2014. P. 703–708.

CHAMPANDARD, Alex J., 2004. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. Indianapolis: New Riders. ISBN 1-5927-3004-3.

CHAMPANDARD, Alex J., 2007a. Teaming Up with Halo's AI: 42 Tricks to Assist Your Game. *AIGameDev.com* [online]. 2007. [Accessed 22 May 2015] Available from: http://aigamedev.com/open/review/halo-ai/

CHAMPANDARD, Alex J., 2007b. Understanding Behavior Trees. *AIGameDev.com* [online]. 2007. [Accessed 22 May 2015] Available from: http://aigamedev.com/open/article/bt-overview/

CHAMPANDARD, Alex J., 2007c. Using Decorators to Improve Behaviors. *AIGameDev.com* [online]. 2007. [Accessed 22 May 2015] Available from: http://aigamedev.com/open/article/decorator/

CHAMPANDARD, Alex J., 2007d. Enabling Concurrency in Your Behavior Hierarchy. *AIGameDev.com* [online]. 2007. [Accessed 22 May 2015] Available from: http://aigamedev.com/open/article/parallel/

CHAMPANDARD, Alex J., 2011. A Guide to Creating Rich and Varied Behaviors (F.E.A.R. 2 AI Analysis). *AIGameDev.com* [online]. 2011. [Accessed 22 May 2015] Available from: http://aigamedev.com/premium/tutorial/fear2-analysis/

CHAMPANDARD, Alex J., 2013. Planning in Games: An Overview and Lessons Learned. *AIGameDev.com* [online]. 2013. [Accessed 22 May 2015] Available from: http://aigamedev.com/open/review/planning-in-games/

CHAMPANDARD, Alex, VERWEIJ, Tim and STRAATMAN, Remco, 2009. Killzone 2 Multiplayer Bots. *Game AI Conference* [online]. 2009. Available from: http://aigamedev.com/open/coverage/killzone2/

CHURCHILL, David and BURO, Michael, 2013. Portfolio greedy search and simulation for large-scale combat in Starcraft. In: *IEEE Conference on Computational Inteligence in Games*. 2013. p. 1–8. ISBN 978-1-4673-5311-3.

CHURCHILL, David, SAFFIDINE, Abdallah and BURO, Michael, 2012. Fast Heuristic Search for RTS Game Combat Scenarios. In: *The Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2012. p. 112–117.

COULOM, Rémi, 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: *Proceedings of the 5th International Conference on Computers and Games*. p. 72–83.

CRONBACH, Lee J., 1951. Coefficient alpha and the internal structure of tests. *Psychometrika*. September 1951. Vol. 16, no. 3, p. 297–334.

CRYTEK, 2013. Smart Object System. *CRYENGINE Manual* [online]. 2013. [Accessed 22 May 2015] Available from: http://docs.cryengine.com/display/SDKDOC2/Smart+Object+System

CUMMING, G., 2014. The new statistics: Why and how. *Psychological Science*. 2014. Vol. 25, no. 1, p. 7–29.

DE SEVIN, Etienne, CHOPINAUD, Caroline and MARS, Clodéric, 2015. Smart Zones To Create the Ambiance of Life. In: *Game AI Pro 2*. CRC Press. p. 89–100. ISBN 978-1482254792.

DECHTER, Rina, 2003. *Constraint Processing*. Morgan Kaufmann Publishers. ISBN 1-55860-890-7.

DENISOVA, Alena and CAIRNS, Paul, 2015. The Placebo Effect in Digital Games. *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*. 2015. p. 23–33.

ENGESER, Stefan and RHEINBERG, Falko, 2008. Flow, performance and moderators of challenge-skill balance. *Motivation and Emotion*. 2008. Vol. 32, no. 3, p. 158–172.

EPIC INC., 2016. Unreal Engine. [online]. 2016. [Accessed 5 June 2016]. Available from: https://www.unrealengine.com/

ESPARCIA-ALCAZAR, Anna I., MARTINEZ-GARCIA, Anais, MORA, Antonio, MERELO, J. J. and GARCIA-SANCHEZ, Pablo, 2010. Controlling bots in a First Person Shooter game using genetic algorithms. In: *IEEE Congress on Evolutionary Computation*. 2010. p. 1–8. ISBN 978-1-4244-6909-3.

FIKES, Richard E. and NILSSON, Nils J., 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*. 1971. Vol. 2, no. 3-4, p. 189–208.

FU, D and HOULETTE, R, 2004. The ultimate guide to FSMs in games. In: *AI game programming Wisdom*. Charles River Media. p. 283–302.

GAMMA, Erich, JOHNSON, Ralph, HELM, Richard and VLISSIDES, John, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley. ISBN 978-0201633610.

GEMROT, Jakub, ČERNÝ, Martin and BROM, Cyril, 2014. Why you should empirically evaluate your AI tool: From SPOSH to yaPOSH. In: *Proceedings of 6th International Conference on Agents and Artificial Intelligence*. 2014. p. 461–468.

GENESERETH, Michael and BJÖRNSSON, Yngvi, 2013. The International General Game Playing Competition. *AI Magazine*. 2013. Vol. 34, no. 2, p. 107–111.

GHALLAB, Malik, NAU, Dana and TRAVERSO, Paolo, 2004. *Automated Planning: Theory and Practice*. San Francisco: Morgan Kaufmann Publishers. ISBN 1-55860-856-7.

GIBSON, James Jerome, 1986. *The Ecological Approach to Visual Perception*. London: Routledge. ISBN 978-0898599596.

GOSLING, Tim and ANDRUSZKIEWICZ, Piotr, 2014. Divide and Conquer, The Campaign AI of Total War: ROME II. *Game/AI Conference Vienna* [online]. 2014. [Accessed 20 May 2016]. Available from: http://archives.nucl.ai/recording/divide-and-conquer-the-campaign-ai-of-total-war-rome-ii/

GRAHAM, David, 2011. AI Development Postmortems: Inside The Sims: Medieval. In: *Game Developers Conference 2011*. 2011.

GRAVES, A.R. and CZARNECKI, C., 2000. Design patterns for behavior-based robotics. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*. 2000. Vol. 30, no. 1, p. 36–41.

GREENBLATT, Richard, EASTLAKE, Donald E. and CROCKER, Stephen D., 1967. The Greenblatt Chess Program. In: *Proceedings of the 1967, fall joint computer conference*. 1967. p. 801–810.

HARDINGHAM, Ian, 2012. Search-Based Adversarial AI for Bots in FROZEN SYNAPSE and its Procedural Levels. *AIGameDev.com* [online]. 2012. [Accessed 20 May 2016]. Available from: http://aigamedev.com/premium/interview/frozen-synapse/

HART, Sergiu and MAS-COLELL, Andreu, 2000. A Simple Adaptive Procedure Leading to Correlated Equilibrium. *Econometrica*. 2000. Vol. 68, no. 5, p. 1127–1150.

HE, Ji, CHEN, Jianshu, HE, Xiaodong, GAO, Jianfeng, LI, Lihong, DENG, Li and OSTENDORF, Mari, 2015. Deep Reinforcement Learning with an Action Space Defined by Natural Language. *ArXiV* [online]. 2015. Available from: http://arxiv.org/abs/1511.04636

HEFNY, AS, HATEM, AA, SHALABY, MM and ATIYA, AF, 2008. Cerberus: Applying supervised and reinforcement learning techniques to capture the flag games. *Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2008. p. 179–184.

HORSWILL, Ian and FOGED, Leif, 2012. Fast Procedural Level Population with Playability Constraints. *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2012. p. 20–25.

IASSENEV, D., 2008. A-Life, Emergent AI and S.T.A.L.K.E.R. *AIGameDev.com* [online]. 2008. [Accessed 22 May 2015] Available from: http://aigamedev.com/open/interviews/stalker-alife/

INGEBRETSON, Peter and REBUSCHATIS, Max, 2014. Concurrent Interactions in The Sims 4. In: *Game Developers Conference*. 2014.

ISLA, Damian, 2005. Handling Complexity in the Halo 2 AI. In: *GDC 2005 Proceeding*. 2005.

ISLA, Damian, 2014. Far Cry's AI: A Manifesto for Systemic Gameplay. *Game/AI Conference Vienna* [online]. 2014. [Accessed 20 May 2016]. Available from: http://gameaiconf.com/recording/far-cry-4/

JUSTESEN, Niels, TILLMAN, Balint, TOGELIUS, Julian and RISI, Sebastian, 2014. Script- and cluster-based UCT for StarCraft. *IEEE Conference on Computatonal Intelligence and Games*. 2014. p. 1-8.

KALLMANN, Marcelo and THALMANN, Daniel, 2002. Modeling Behaviors of Interactive Objects for Real-Time Virtual Environments. *Journal of Visual Languages & Computing*. 2002. Vol. 13, no. 2, p. 177–195.

KENDALL, Elizabeth, KRISHNA, P. V. Murali, PATHAK, Chirag V. and SURESH, C. B., 1998. Patterns of intelligent and mobile agents. In: *Proceedings of the second international conference on Autonomous agents*. 1998. p. 92–99.

KICZALES, Gregor, HILSDALE, Erik, HUGUNIN, Jim, KERSTEN, Mik, PALM, Jeffrey and GRISWOLD, William G, 2001. An Overview of AspectJ. *Object-Oriented Programming: 15th European Conference*. 2001. p. 327–354.

KICZALES, Gregor, LAMPING, John, MENDHEKAR, Anurag, MAEDA, Chris, LOPES, Cristina, LOINGTIER, Jean-Marc and IRWIN, John, 1997. Aspect-oriented programming. In: *ECOOP'97 — Object-Oriented Programming*. p. 220–242. ISBN 978-3-540-63089-0.

KOCSIS, Levente, SZEPESVÁRI, Csaba and WILLEMSON, Jan, 2006. *Improved Monte-Carlo Search*. Technical report. University of Tartu, Estonia.

KOVARSKY, Alexander and BURO, Michael, 2005. Heuristic search applied to abstract combat games. *Advances in Artificial Intelligence*. 2005. Vol. 3501, p. 66–78.

KUHLMANN, Gregory and STONE, Peter, 2006. Automatic heuristic construction in a complete general game player. *Proceedings of the Twenty-First National Conference on Artificial Intelligence*. 2006. p. 1457–62.

LANCTOT ·, Marc, LISÝ, Viliam and WINANDS, Mark H. M., 2013. Monte Carlo Tree Search in Simultaneous Move Games with Applications to Goofspiel. *Proceedings of IJCAI 2013 Workshop on Computer Games*. 2013. p. 28–43.

LANCTOT, Marc, SAFFIDINE, Abdallah, VENESS, Joel, ARCHIBALD, Christopher and WINANDS, Mark H. M., 2013, Monte Carlo *-Minimax Search. *ArXiV* [online]. 2013. Available from: http://arxiv.org/abs/1304.6057

LI, Weizi and ALLBECK, Jan, 2011. Populations with Purpose. *Motion in Games*. 2011. p. 132–143.

LISÝ, Viliam, KOVAŘÍK, Vojtěch, LANCTOT, Marc and BOŠANSKÝ, Branislav, 2013. Convergence of Monte Carlo Tree Search in Simultaneous Move Games. *Advances in Neural Information Processing Systems 26*. 2013. p. 2112–2120.

LOYALL, Aaron Bryan, 1997. *Believable Agents: Building Interactive Personalities*. PhD Thesis. Carnegie Mellon University.

LUDWIG, Jeremy and FARLEY, Arthur, 2009. Examining Extended Dynamic Scripting in a Tactical Game Framework. *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*. 2009. p. 76–81.

LUDWIG, Jeremy R and FARLEY, Arthur, 2008. Using hierarchical dynamic scripting to create adaptive adversaries. *Proceedings of the 2008 Behavior Representation in Modeling and Simulation (BRIMS) Conference*. 2008.

MAROUENE, Kefi, PAUL, Richard and VINCENT, Barichard, 2011. Use of virtual reality and constraint programming techniques in interactive 3D objects layout. In: *The 19th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. Pilsen, Czech Republic. 2011. p. 51–54. ISBN 978-80-86943-81-7.

MARSLAND, T. A., 1986. A review of game-tree pruning. *ICCA journal*. 1986. Vol. 9, no. 1, p. 3–19.

MCAULEY, Edward, DUNCAN, Terry and TAMMEN, Vance V., 1989. Psychometric Properties of the Intrinsic Motivation Inventory in a Competitive Sport Setting: A Confirmatory Factor Analysis. *Research Quarterly for Exercise and Sport*. March 1989. Vol. 60, no. 1, p. 48–58.

MCNAUGHTON, Matthew, CUTUMISU, Maria, SZAFRON, Duane, SCHAEFFER, Jonathan, REDFORD, James and PARKER, Dominique, 2004. ScriptEase: Generative design patterns for computer role-playing games. In: *Proceedings of the 19th IEEE international conference on Automated software engineering*. 2004. p. 88–99.

MNIH, Volodymyr, BADIA, Adrià Puigdomènech, MIRZA, Mehdi, GRAVES, Alex, LILLICRAP, Timothy P., HARLEY, Tim, SILVER, David and KAVUKCUOGLU, Koray, 2016. Asynchronous Methods for Deep Reinforcement Learning. *ArXiV* [online]. 2016. Available from: http://arxiv.org/abs/1602.01783

MNIH, Volodymyr, KAVUKCUOGLU, Koray, SILVER, David, RUSU, Andrei A., VENESS, Joel, BELLEMARE, Marc G., GRAVES, Alex, RIEDMILLER, Martin, FIDJELAND, Andreas K., OSTROVSKI, Georg, PETERSEN, Stig, BEATTIE, Charles, SADIK, Amir, ANTONOGLOU, Ioannis, KING, Helen, KUMARAN, Dharshan, WIERSTRA, Daan, LEGG, Shane and HASSABIS, Demis, 2015. Human-level control through deep reinforcement learning. *Nature*. 2015. Vol. 518, no. 7540, p. 529–533.

MOUNTAIN, Gwaredd, 2015. Tactical Planning and Real-time MCTS in Fable Legends. *nucl.ai Conference* [online]. 2015. [Accessed 20 May 2016]. Available from: http://archives.nucl.ai/recording/tactical-planning-and-real-time-mcts-in-fable-legends/

MUELLER, Jan and CHAMPANDARD, Alex J., 2015. Sunset Overdrive from Boss Behaviors to Open-World AI Optimization. *AIGameDev.com* [online]. 2015. [Accessed 20 May 2016]. Available from: http://aigamedev.com/premium/interview/sunset-overdrive/

MUÑOZ-AVILA, Héctor, BAUCKHAGE, Christian, BIDA, Michal, CONGDON, Clare and KENDALL, Graham, 2013. Learning and Game AI. In: *Artificial and Computational Intelligence in Games*. 2013. p. 33–43. ISBN 978-3-939-89762-0.

MUSSER, David R. and STEPANOV, Alexander A., 1989. Generic programming. In: *Symbolic and Algebraic Computation: International symposium*. LNCS 358. p. 13–25. ISBN 978-3-540-51084-0.

NILSSON, N, 1982. Artificial Intelligence: Engineering, Science, or Slogan. *The AI Magazine*. 1982. Vol. 3, no. 1, p. 2–9.

ORKIN, Jeff, 2006. Three states and a plan: the AI of FEAR. *Game Developers Conference*. 2006. Vol. 2006, no. 1, p. 1–18.

PARERA, Joan, 2013, Combat AI and Animations in CASTLEVANIA: Lord of Shadows. *AIGameDev.com* [online]. 2013. [Accessed 20 May 2016] Available from: http://aigamedev.com/premium/interview/castlevania-lord-shadows/

PARNAS, D. L., 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. 1972. Vol. 15, no. 12, p. 1053–1058.

PEDICA, Claudio and VILHJÁLMSSON, Hannes Högni, 2009. Spontaneous avatar behavior for human territoriality. In: I*ntelligent Virtual Agents*. p. 344–357. ISBN 3642043798.

PLCH, Tomáš, MARKO, Matěj, ONDRÁČEK, Petr, ČERNÝ, Martin, GEMROT, Jakub and BROM, Cyril, 2014. An AI System for Large Open Virtual World. In: *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press. 2014. p. 44–51.

POLICARPO, D, URBANO, P and LOUREIRO, T, 2010. Dynamic scripting applied to a First-Person Shooter. In: *Information Systems and Technologies*. 2010. p. 1–6. ISBN 9781424472277.

R CORE TEAM, 2015. *R: A Language and Environment for Statistical Computing*. Vienna, Austria.

REED, Christopher and GEISLER, Benjamin, 2004. Jumping, Climbing, and Tactical Reasoning: How to Get More Out of a Navigation System. In: *AI Game Programming Wisdom II*. Charles River Media. p. 141–150. ISBN 978-1584502890.

REVELLE, William, 2015. *psych: Procedures for Psychological, Psychometric, and Personality Research*. Technical report. Northwestern University. Evanston, Illinois.

ROCKSTAR GAMES, 2010. Red Dead Redemption. [online]. 2010. [Accessed 22 December 2015]. Available from: http://www.rockstargames.com/reddeadredemption/

RUSSEL, Stuart J. and NORVIG, Peter, 2010. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River: Prentice Hall. ISBN 978-0-13-604259-4.

RUSSELL, Adam, 2008. Situationist Game AI. In: *AI Game Programming Wisdom 4*. p. 3–16. ISBN 978-1-58450-523-5.

RYAN, Richard M., 1982. Control and information in the intrapersonal sphere: An extension of cognitive evaluation theory. *Journal of Personality and Social Psychology*. 1982. Vol. 43, no. 3, p. 450–461.

SAFFIDINE, Abdallah, FINNSSON, Hilmar and BURO, Michael, 2012. Alpha-Beta Pruning for Games with Simultaneous Moves. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. 2012. p. 22–26.

SCHIEX, Thomas, RÉGIN, Jean-Charles, GASPIN, Christine and VERFAILLIE, Gérard, 1996. Lazy Arc Consistency. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. 1996. P. 216–221.

SCHIFFEL, Stephan and THIELSCHER, Michael, 2007. Fluxplayer : A Successful General Game Player. *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*. 2007. Vol. 22, no. 2, p. 1191–1196.

SCHRUM, Jacob, KARPOV, Igor V. and MIIKKULAINEN, Risto, 2012. Human-like combat behaviour via multiobjective neuroevolution. In: *Believable Bots: Can Computers Play Like People?* p. 119–150. ISBN 978-3-6423-2323-2.

SCHRUM, Jacob and MIIKKULAINEN, Risto, 2010. Evolving Agent Behavior in Multiobjective Domains Using Fitness-Based Shaping. *Proceedings of the 12th annual conference on Genetic and Evolutionary Computation*. 2010. p. 439–446.

SCHUYTEMA, Paul and MANYEN, Mark, 2005. *Game Development with Lua*. Charles River Media. ISBN 978-1-5845-0404-7.

SCULLEY, D, HOLT, Gary, GOLOVIN, Daniel, DAVYDOV, Eugene, PHILLIPS, Todd, EBNER, Dietmar, CHAUDHARY, Vinay and YOUNG, Michael, 2014. Machine Learning: The High Interest Credit Card of Technical Debt. *Software Engineering for Machine Learning (NIPS 2014 Workshop)Software Engineering for Machine Learning (NIPS 2014 Workshop)*. 2014. p. 1–9.

SHAFIEI, Mohammad, STURTEVANT, Nathan and SCHAEFFER, Jonathan, 2009. Comparing UCT versus CFR in simultaneous games. In: *Proceeding of the IJCAI Workshop on General Game Playing*. 2009. p. 75–82.

SHAO, Wei and TERZOPOULOS, Demetri, 2005. Autonomous pedestrians. In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2005. p. 19–28.

SHNEIDERMAN, Ben and PLAISANT, Catherine, 2005. *Designing the user interface: Strategies for effective human-computer interaction*. Pearson Education.

SHOULSON, Alexander, GILBERT, Max L., KAPADIA, Mubbasir and BADLER, Norman I., 2013. An Event-Centric Planning Approach for Dynamic Real-Time Narrative. *Proceedings of Motion on Games*. 2013. P. 121–130.

SILVER, David, HUANG, Aja, MADDISON, Chris J., GUEZ, Arthur, SIFRE, Laurent, VAN DEN DRIESSCHE, George, SCHRITTWIESER, Julian, ANTONOGLOU, Ioannis, PANNEERSHELVAM, Veda, LANCTOT, Marc, DIELEMAN, Sander, GREWE, Dominik, NHAM, John, KALCHBRENNER, Nal, SUTSKEVER, Ilya, LILLICRAP, Timothy, LEACH, Madeleine, KAVUKCUOGLU, Koray, GRAEPEL, Thore and HASSABIS, Demis, 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016. Vol. 529, no. 7587, p. 484–489.

SKUBCH, Hendrik, 2015, Not Just Planning: STRIPs for Ambient NPC Interactions in Final Fantasy XV. *nucl.ai Conference* [online]. 2015. [Accessed 20 May 2016]. Available from: http://archives.nucl.ai/recording/not-just-planning-strips-for-ambient-npc-interactions-in-final-fantasy-xv/

SMALL, Ryan and CONGDON, Clare Bates, 2009. Agent Smith: Towards an evolutionary rule-based agent for interactive dynamic games. In: *IEEE Congress on Evolutionary Computation*. 2009. p. 660–666. ISBN 978-1-4244-2958-5.

SMITH, Adam M., ANDERSEN, Erik, MATEAS, Michael and POPOVIĆ, Zoran, 2012. A case study of expressively constrainable level design automation tools for a puzzle game. *Foundations of Digital Games*. 2012. p. 156–164.

SMITH, Adam M. and MATEAS, Michael, 2010. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. *IEEE Conference on Computational Intelligence and Games*. 2010. p. 273–280.

SMITH, Brian Cantwell, 1982. *Procedural Reflection in Programming Languages*. Phd Thesis. Massachusetts Institute of Technology.

SMITH, Gillian, WHITEHEAD, Jim and MATEAS, Michael, 2011. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*. 2011. Vol. 3, no. 3, p. 201–215.

SPRONCK, Pieter, PONSEN, Marc, SPRINKHUIZEN-KUYPER, Ida and POSTMA, Eric, 2006. Adaptive game AI with dynamic scripting. *Machine Learning*. 2006. Vol. 63, no. 3, p. 217–248.

STOCKER, Catherine, SUN, Libo, HUANG, Pengfei, QIN, Wenhu, ALLBECK, Jan M. and BADLER, Norman I., 2010. Smart Events and Primed Agents. In: *Intelligent Virtual Agents*. LNCS 6356. Springer. p. 15–27. ISBN 978-3-642-15891-9.

STRAATMAN, Remco, VERWEIJ, Tim, CHAMPANDARD, Alex, MORCUS, Robert and HYLKE, Kleve, 2013. Hierarchical AI for Multiplayer Bots in Killzone 3. In: *Game AI Pro*. CRC Press. p. 377–390. ISBN 978-1466565968.

STURTEVANT, Natharn R., ORKIN, Jeff, ZUBEK, Robert, COOK, Michael, WARE, Stephen G., STITH, Christian, YOUNG, R. Michael, WRIGHT, Phillip, EISERLOH, Squirrel, RAMIREZ-SANABRIA, Alejandro, BULITKO, Vadim and LORD, Kieran, 2014. Playable Experiences at AIIDE 2014. In: *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2014. p. 227–231. ISBN 978-1-5773-5681-3.

SUNG, Mankyu, GLEICHER, Michael and CHENNEY, Stephen, 2004. Scalable behaviors for crowd simulation. *Computer Graphics Forum*. 2004. Vol. 23, no. 3, p. 519–528.

TAK, Mandy J W, LANCTOT, Marc and WINANDS, Mark H M, 2014. Monte Carlo Tree Search variants for simultaneous move games. In: *IEEE Conference on Computational Intelligence and Games*. IEEE. 2014. p. 1–8. ISBN 978-1-4799-3547-5.

TECCHIA, Franco, LOSCOS, Céline, CONROY, Ruth and CHRYSANTHOU, Yiorgos, 2001. Agent Behaviour Simulator (ABS): A Platform for Urban Behaviour Development. *The First International Game Technology Conference and Idea Expo*. 2001.

TOUBMAN, Armon, ROESSINGH, Jan Joris, SPRONCK, Pieter, PLAAT, Aske and VAN DEN HERIK, Jaap, 2014. Dynamic Scripting with Team Coordination in Air Combat Simulation. In: *Modern Advances in Applied Intelligence*. p. 440–449. ISBN 978-3-319-07455-9; 978-3-319-07454-2.

UNITY TECHNOLOGIES, 2016, Unity. [online]. 2016. [Accessed 5 June 2016]. Available from: https://unity3d.com/

VAN HOORN, Niels, TOGELIUS, Julian and SCHMIDHUBER, Jürgen, 2009. Hierarchical controller learning in a first-person shooter. *IEEE Symposium on Computational Intelligence and Games*. 2009. p. 294–301.

VEHKALA, Mika, 2012. Crowds in Hitman: Absolution. *AIGameDev.com* [online]. 2012. [Accessed 5 June 2016] Available from: http://aigamedev.com/ultimate/video/hitmancrowds/

VENESS, Joel and BLAIR, Alan, 2007. Effective use of transposition tables in stochastic game tree search. In: *IEEE Symposium on Computational Intelligence and Games*. 2007. p. 112–116. ISBN 1424407095.

VIGNA, Sebastiano, 2014. Further scramblings of Marsaglia's xorshift generators. *ArXiV* [online]. 2014. Available from: http://arxiv.org/abs/1404.0390

WALSH, Toby, 2003. Constraint Patterns. In: *Principles and Practice of Constraint Programming*. p. 53–64.

WANG, Di and TAN, Ah-Hwee, 2015. Creating Autonomous Adaptive Agents in a Real-Time First-Person Shooter Computer Game. *IEEE Transactions on Computational Intelligence and AI in Games*. 2015. Vol. 7, no. 2, p. 123–138.

WEBER, Ben G., MAWHORTER, Peter, MATEAS, Michael and JHALA, Arnav, 2010. Reactive planning idioms for multi-scale game AI. *IEEE Conference on Computational Intelligence and Games*. 2010. p. 115–122.

WICKHAM, Hadley, 2009. *ggplot2: elegant graphics for data analysis*. New York: Springer. ISBN 978-0-387-98140-6.

WOOLDRIDGE, Michael and JENNINGS, Nicholas R., 1998. Pitfalls of agent-oriented development. In: *Proceedings of the second international conference on Autonomous agents - AGENTS '98*. New York, New York, USA : ACM Press. 1998. p. 385–391. ISBN 0897919831.

ZHAO, Richard and SZAFRON, Duane, 2014. Using Cyclic Scheduling to Generate Believable Behavior in Games. In: *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2014. p. 94–101. ISBN 9781577356813.

List of Tables

Table 1: Two basic dimensions of OWG AI complexity.....11
Table 2: Complexity classification for the techniques in Chapter 5.....21
Table 3: Questions in the first round of interviews. 44
Table 4: Questions in second round of interviews. 46
Table 5: Complexity classification for the techniques in Chapter 6.....50
Table 6: Number of successful and failed searches.....61
Table 7: Average search times and standard deviations for individual situations and algorithms in Scenario 1.....62
Table 8: Maximum search times for individual situations and algorithms in Scenario 1.....62
Table 9: Average search times and standard deviations for individual situations and algorithms in Scenario 2.....62
Table 10: Maximum search times for individual situations and algorithms in Scenario 2.....63
Table 11: Complexity classification for the techniques in Chapter 7.....65
Table 12: Summary of parameters of the search algorithms.....92
Table 13: Best evolved parameters for Alpha-Beta.....93
Table 14: Best evolved parameters for plain UCT. 93
Table 15: Best evolved parameters for decoupled UCT. 93
Table 16: Results of tournament among all time variants of the Alpha-Beta algorithm and the random baseline.....94
Table 17: Results of tournament among all time variants of the UCT algorithm and the random baseline. 94
Table 18: Results of tournament among all time variants of the DUCT algorithm and the random baseline. 94
Table 19: Results of tournament among all algorithm variants with 1ms computation time. 95
Table 20: Results of tournament among all algorithm variants with 20ms computation time. 95
Table 21: Results of tournament among all algorithm variants with 50ms computation time. 95
Table 22: Results of tournament among all algorithm variants with 100ms computation time. 95
Table 23: Results of in-game tournament among all time variants of the Alpha-Beta algorithm and the default AI. 96
Table 24: Results of in-game tournament among all time variants of the UCT algorithm and the default AI. 96
Table 25: Results of tournament among all time variants of the DUCT algorithm and the default AI.....96

Table 26: Results of in-game tournament among all algorithm variants with 1ms computation time. 97
Table 27: Results of in-game tournament among all algorithm variants with 20ms computation time. 97
Table 28: Results of in-game tournament among all algorithm variants with 50ms computation time. 97
Table 29: Mean and 95% confidence intervals (CI) for differences of the main dependent variables between algorithms. 108
Table 30: Candidate parameter combinations for UCT-1ms. 135
Table 31:Results of tournament between the UCT-1ms candidates.....135
Table 32:Candidate parameter combinations for UCT-20ms. 135
Table 33:Results of tournament between the UCT-20ms candidates.....136
Table 34:Candidate parameter combinations for UCT-50ms. 136
Table 35:Results of tournament between the UCT-50ms candidates.....136
Table 36: Candidate parameter combinations for UCT-100ms. 136
Table 37:Results of tournament between the UCT-100ms candidates.....137
Table 38: Candidate parameter combinations for DUCT-1ms 137
Table 39: Results of tournament between the DUCT-1ms candidates.. 137
Table 40: Candidate parameter combinations for DUCT-20ms 137
Table 41: Results of tournament between the DUCT-20ms candidates.. 137
Table 42: Candidate parameter combinations for DUCT-50ms 138
Table 43: Results of tournament between the DUCT-50ms candidates.. 138
Table 44: Candidate parameter combinations for DUCT-100ms.....138
Table 45:Results of tournament between the DUCT-100ms candidates.. 138
Table 46: Candidate parameter combinations for AB-1ms.....139
Table 47:Results of tournament between the AB-1ms candidates. 139
Table 48: Candidate parameter combinations for AB-20ms.....139
Table 49:Results of tournament between the AB-20ms candidates. 139
Table 50: Candidate parameter combinations for AB-50ms.....140
Table 51:Results of tournament between the AB-50ms candidates. 140
Table 52: Candidate parameter combinations for AB-100ms.....140
Table 53:Results of tournament between the AB-100ms candidates. 140

List of Algorithms

Algorithm 1: Plain backtracking.....56
Algorithm 2: Bactracking with lazy node consistency. 57
Algorithm 3: Backtracking with forward checking. 58

List of Figures

Figure 1: Diagrams of script selection functions AI of components with different types of complexity. 10
Figure 2: Example behavior tree representing a simple guard behavior for a shooter game. 18
Figure 3: Script cleanup example. 19
Figure 4: An example usage of behavior objects: a pub with multiple chairs. 28
Figure 5: Injecting script into an NPC's main script. 30
Figure 6: Possible data access between a BO instance and a holder. 31
Figure 7: A simple class diagram of various types of behavior objects and their basic properties. 32
Figure 8: Handling specific behaviors at a high-level smart area to control movement within the area. 38
Figure 9: Using smart object to "decorate" a script in a smart area. 39
Figure 10: Domain contents while using forward checking with lazy node consistency. 57
Figure 11: Distance constraint structure for the "Difficult" situation. 60
Figure 12: A screenshot from the quantitative evaluation setup. 61
Figure 13: A transition diagram of the node types in the abstract combat model. 85
Figure 14: Data gathering scenario for the abstract combat model. 86
Figure 15: Search tree visualization in level editor. 88
Figure 16: Visualizing the abstract model state by overlaying it with the game world. 88
Figure 17: The arena used for experiments with human subjects. 100
Figure 18: Histograms of reported psychometric values. 102
Figure 19: Box plots of main dependent variables per algorithm. 104
Figure 20: Histograms of the objective difficulty variables. 105
Figure 21: Histograms of the within-subject differences of the main dependent variables for each algorithm pair (each player counted once per algorithm pair). 106
Figure 22: Confidence intervals (CIs) and the overall distribution of differences in the main dependent variables. 107
Figure 23: Confidence intervals (CIs) and the overall distribution of differences in the fun dependent variable. 108

List of Abbreviations

AB	Alpha-Beta algorithm.
BO	Behavior object (see Chapter 5)
BT	Behavior tree, as in (Champandard 2007b)
CPU	Central processing unit
CSP	Constraint satisfaction problem
DUCT	Decoupled upper confidence bounds on trees – a variant of MCTS.
HTN	Hierarchical task networks, as in (Ghallab et al. 2004)
KC:D	Kingdom Come: Deliverance
MCTS	Monte Carlo Tree Search
NPC	Non-player character
OOP	Object-oriented programming
OWG	Open-world game
RTS	Real-time strategy
SE	Smart entity (see Section 5.5.2)
UCT	Upper confidence bounds on trees – a variant of MCTS.

Appendix A – Digital Attachment Contents

- `Videos` – Videos of all of the systems implemented in this thesis working in the complete virtual environment of KC:D. Due to size constraints, some of the videos are not present in the attachment downloadable from the thesis repository and are stored only on the DVD disc accompanying the physical copy of the thesis.
- `Videos/videos.txt` – Description of the individual videos.
- `GlobalSpec/Source` – Source code for the CSP solvers used for evaluation of the situation system (see Section 6.5.1). Only the code for the solvers is included, not the whole virtual environment.
- `GlobalSpec/solversDocumentation.pdf` – Documentation for the CSP solvers code.
- `GlobalSpec/Results` – Full dataset from the quantitative tests of the situation system (see Section 6.6.1)
- `Search/Source` – Code for the search-based AI and the standalone abstract model of combat discussed in Section 7.5. Only the code for the abstract model and search algorithms is included, not the whole virtual environment.
- `Search/searchDocumentation.pdf` – Documentation for the search-based AI code.
- `Search/Questionnaires` – PDFs of the questionnaires used in the human evaluation of search-based combat AI, both Czech originals and English translations (see section 7.6.3).
- `Search/preregistration.pdf` – PDF of the AsPredicted preregistration form for the human evaluation of search-based combat AI (see Section 7.6.3).
- `Search/Results` – Full dataset from the evaluation of the search-based AI and the R code to analyze it (see Section 7.6).
- `thesis.pdf` – Text of this thesis.
- `readme.txt` – Description of the contents of the digital attachment.

Appendix B – Evolution Results for Adversarial Search Algorithms

Here, we show the candidate parameter combinations for the individual algorithm variants and the results of tournaments that selected the best. See Section 7.5.6 for details.

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
U1-a	0.19	8	0.90	0.08
U1-b	0.19	8	0.90	0.04
U1-c	0.19	8	0.90	0.13
U1-d	0.16	6	0.77	0.05
U1-e	0.13	6	0.77	0.03
U1-f	0.13	6	0.77	0.04

Table 30: Candidate parameter combinations for UCT-1ms.

	U1-a	U1-b	U1-c	U1-d	U1-e	U1-f	#Wins
U1-a		0.495	0.506	0.508	0.472	0.466	**2**
U1-b	0.505		0.524	0.511	0.479	0.488	**3**
U1-c	0.494	0.476		0.455	0.491	0.487	**0**
U1-d	0.492	0.489	0.545		0.481	0.474	**1**
U1-e	**0.528**	**0.521**	**0.509**	**0.519**		**0.505**	**5**
U1-f	0.534	0.512	0.513	0.526	0.495		**4**

Table 31:Results of tournament between the UCT-1ms candidates.

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
U20-a	0.31	15	0.88	0.11
U20-b	0.26	09	0.88	0.11
U20-c	0.31	09	0.88	0.09
U20-d	0.27	14	0.68	0.18
U20-e	0.27	14	0.81	0.18
U20-f	0.27	14	0.67	0.18

Table 32:Candidate parameter combinations for UCT-20ms.

	U20-a	U20-b	U20-c	U20-d	U20-e	U20-f	#Wins
U20-a		**0.483**	**0.502**	**0.529**	**0.545**	**0.535**	**4**
U20-b	0.517		0.509	0.494	0.496	0.533	**3**
U20-c	0.498	0.491		0.520	0.498	0.553	**2**
U20-d	0.471	0.506	0.480		0.490	0.472	**1**
U20-e	0.455	0.504	0.502	0.510		0.499	**3**
U20-f	0.465	0.467	0.447	0.528	0.501		**2**

Table 33:Results of tournament between the UCT-20ms candidates.

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
U50-a	0.44	17	0.78	0.27
U50-b	0.44	16	0.82	0.27
U50-c	0.77	16	0.78	0.30
U50-d	0.29	18	0.60	0.20
U50-e	0.16	18	0.45	0.09
U50-f	0.16	14	0.70	0.09

Table 34:Candidate parameter combinations for UCT-50ms.

	U50-a	U50-b	U50-c	U50-d	U50-e	U50-f	#Wins
U50-a		0.502	0.543	0.490	0.470	0.450	**2**
U50-b	0.498		0.552	0.483	0.460	0.446	**1**
U50-c	0.457	0.448		0.501	0.444	0.430	**1**
U50-d	0.510	0.517	0.499		0.474	0.482	**2**
U50-e	0.530	0.540	0.556	0.526		0.486	**4**
U50-f	**0.550**	**0.554**	**0.570**	**0.518**	**0.514**		**5**

Table 35:Results of tournament between the UCT-50ms candidates.

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
U100-a	0.42	11	0.69	0.01
U100-b	0.25	11	0.64	0.01
U100-c	0.25	11	0.77	0.01
U100-d	0.21	25	0.97	0.16
U100-e	0.21	25	1.00	0.16
U100-f	0.17	13	1.00	0.10

Table 36: Candidate parameter combinations for UCT-100ms.

	U100-a	U100-b	U100-c	U100-d	U100-e	U100-f	#Wins
U100-a		0.478	0.484	0.548	0.513	0.509	**3**
U100-b	0.522		0.502	0.500	0.526	0.497	**3.5**
U100-c	**0.516**	**0.498**		**0.526**	**0.506**	**0.503**	**4**
U100-d	0.452	0.500	0.474		0.480	0.471	**0.5**
U100-e	0.487	0.474	0.494	0.520		0.510	**2**
U100-f	0.491	0.503	0.497	0.529	0.490		**2**

Table 37:Results of tournament between the UCT-100ms candidates.

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
D1-a	0.25	11	0.80	0.04
D1-b	0.25	11	0.88	0.04
D1-c	0.07	06	0.91	0.11
D1-d	0.07	05	0.91	0.11
D1-e	0.07	11	0.91	0.06

Table 38: Candidate parameter combinations for DUCT-1ms

	D1-a	D1-b	D1-c	D1-d	D1-e	#Wins
D1-a		0.487	0.456	0.466	0.493	**0**
D1-b	0.513		0.484	0.467	0.485	**1**
D1-c	**0.544**	**0.516**		**0.514**	**0.537**	**4**
D1-d	0.534	0.533	0.486		0.551	**3**
D1-e	0.507	0.515	0.463	0.449		**2**

Table 39: Results of tournament between the DUCT-1ms candidates..

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
D20-a	0.15	12	0.85	0.12
D20-b	0.15	12	0.95	0.12
D20-c	0.10	12	0.75	0.09
D20-d	0.10	08	0.82	0.08
D20-e	0.10	08	0.82	0.09

Table 40: Candidate parameter combinations for DUCT-20ms

	D20-a	D20-b	D20-c	D20-d	D20-e	#Wins
D20-a		0.516	0.500	0.504	0.498	**2.5**
D20-b	0.484		0.500	0.504	0.502	**2.5**
D20-c	0.500	0.500		0.514	0.487	**2**
D20-d	0.496	0.496	0.486		0.484	**0**
D20-e	**0.502**	**0.498**	**0.513**	**0.516**		**3**

Table 41: Results of tournament between the DUCT-20ms candidates..

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
D50-a	0.30	21	0.84	0.10
D50-b	0.29	09	0.93	0.01
D50-c	0.42	21	0.93	0.06
D50-d	0.24	27	0.67	0.27
D50-e	0.19	21	0.81	0.39
D50-f	0.24	40	0.70	0.39

Table 42: Candidate parameter combinations for DUCT-50ms

	D50-a	D50-b	D50-c	D50-d	D50-e	D50-f	#Wins
D50-a		**0.502**	**0.518**	**0.501**	**0.515**	**0.514**	**5**
D50-b	0.498		0.520	0.510	0.512	0.518	**4**
D50-c	0.482	0.480		0.509	0.498	0.497	**1**
D50-d	0.499	0.490	0.491		0.507	0.521	**2**
D50-e	0.485	0.488	0.502	0.493		0.493	**1**
D50-f	0.486	0.482	0.503	0.479	0.507		**1**

Table 43: Results of tournament between the DUCT-50ms candidates..

Designation	Exploration Factor	Playout Steps	Health Weight	Stamina Weight
D100-a	0.13	25	0.76	0.13
D100-b	0.11	25	0.76	0.07
D100-c	0.11	25	0.76	0.13
D100-d	0.47	22	0.83	0.08
D100-e	0.47	18	0.83	0.07
D100-f	0.70	18	0.87	0.08

Table 44: Candidate parameter combinations for DUCT-100ms.

	D100-a	D100-b	D100-c	D100-d	D100-e	D100-f	#Wins
D100-a		0.509	0.489	0.492	0.511	0.535	**3**
D100-b	0.491		0.500	0.496	0.523	0.495	**1.5**
D100-c	**0.511**	**0.500**		**0.521**	**0.488**	**0.522**	**3.5**
D100-d	0.508	0.504	0.479		0.487	0.501	**3**
D100-e	0.489	0.477	0.512	0.513		0.515	**3**
D100-f	0.465	0.505	0.478	0.499	0.485		**1**

Table 45:Results of tournament between the DUCT-100ms candidates..

Designation	Health Weight	Stamina Weight	Transposition Table Size
A1-a	0.89	0.05	8192
A1-b	0.92	0.03	32768
A1-c	0.92	0.03	8192
A1-d	0.75	0.07	65536
A1-e	0.71	0.03	131072
A1-f	0.80	0.03	131072

Table 46: Candidate parameter combinations for AB-1ms.

	A1-a	A1-b	A1-c	A1-d	A1-e	A1-f	#Wins
A1-a		0.498	0.486	0.505	0.500	0.493	**1.5**
A1-b	0.502		0.483	0.491	0.500	0.494	**1.5**
A1-c	0.514	0.517		0.487	0.504	0.500	**3.5**
A1-d	0.495	0.509	0.513		0.497	0.492	**2**
A1-e	0.500	0.500	0.496	0.503		0.486	**1.5**
A1-f	**0.507**	**0.506**	**0.500**	**0.508**	**0.514**		**4.5**

Table 47:Results of tournament between the AB-1ms candidates.

Designation	Health Weight	Stamina Weight	Transposition Table Size
A20-a	0.88	0.01	2048
A20-b	0.93	0.00	8192
A20-c	0.36	0.00	4096
A20-d	0.77	0.02	32
A20-e	0.77	0.02	0
A20-f	0.68	0.02	0

Table 48: Candidate parameter combinations for AB-20ms.

	A20-a	A20-b	A20-c	A20-d	A20-e	A20-f	#Wins
A20-a		0.535	0.526	0.514	0.491	0.487	**3**
A20-b	0.465		0.507	0.476	0.486	0.487	**1**
A20-c	0.474	0.493		0.469	0.454	0.482	**0**
A20-d	0.486	0.524	0.531		0.479	0.498	**2**
A20-e	**0.509**	**0.514**	**0.546**	**0.521**		**0.512**	**5**
A20-f	0.513	0.513	0.518	0.502	0.488		**4**

Table 49:Results of tournament between the AB-20ms candidates.

Designation	Health Weight	Stamina Weight	Transposition Table Size
A50-a	0.56	0.01	8092
A50-b	0.54	0.01	8092
A50-c	0.62	0.01	8092
A50-d	0.73	0.03	32
A50-e	0.73	0.03	2

Table 50: Candidate parameter combinations for AB-50ms.

	A50-a	A50-b	A50-c	A50-d	A50-e	#Wins
A50-a		**0.508**	**0.505**	**0.519**	**0.504**	**4**
A50-b	0.492		0.514	0.517	0.512	**3**
A50-c	0.495	0.486		0.508	0.514	**2**
A50-d	0.481	0.483	0.492		0.487	**0**
A50-e	0.496	0.488	0.486	0.513		**1**

Table 51:Results of tournament between the AB-50ms candidates.

Designation	Health Weight	Stamina Weight	Transposition Table Size
A100-a	0.90	0.02	4096
A100-b	0.61	0.02	8
A100-c	0.92	0.02	8
A100-d	0.72	0.01	2048
A100-e	0.68	0.01	4096
A100-f	0.68	0.02	4096

Table 52: Candidate parameter combinations for AB-100ms.

	A100-a	A100-b	A100-c	A100-d	A100-e	A100-f	#Wins
A100-a		0.491	0.500	0.512	0.507	0.507	**3.5**
A100-b	0.509		0.497	0.505	0.494	0.492	**2**
A100-c	0.500	0.503		0.502	0.493	0.498	**2.5**
A100-d	0.488	0.495	0.498		0.515	0.506	**2**
A100-e	**0.493**	**0.506**	**0.507**	**0.485**		**0.501**	**4**
A100-f	0.493	0.508	0.502	0.494	0.499		**2**

Table 53:Results of tournament between the AB-100ms candidates.

Appendix C – Preregistration of the Human Evaluation

Combat AI - Warhorse, Spring 2016 (#518)

Created: 03/14/2016
Made public: 03/27/2016

Author(s)
Martin Černý (Charles University in Prague) – cerny@gamedev.cuni.cz

1) What's the main question being asked or hypothesis being tested in this study?

An intelligent algorithm using UCT for controlling opponent in a computer game will be more difficult to beat than a simple default algorithm already implemented in the game. Further, adding computing time to the algorithm will improve its performance.

2) Describe the key dependent variable(s) specifying how they will be measured.

Proportion of fights won by the human player, average of (player's health - opponent's health) at the end of each fight. Player-assesed difficulty of the opponents through a questionnaire (4 item, 7 point likert scale) at the end of each phase. Player assesed difficulty of the opponents through quick questionnaires after 40% of the fights (2 item, 7 point likert scale).

3) How many and which conditions will participants be assigned to?

6 within-subject conditions:

2 different equipments for the computer controlled opponent (light and heavy armor)

3 different algorithms (the Default AI, UCT with 1 ms for thinking, UCT with 50 ms for thinking).

For each equipment, players will play 5 fights against each algorithm (30 fights per subject in total).

4) Specify exactly which analyses you will conduct to examine the main question/hypothesis.

We will compare 95% confidence intervals for the within-subject differences between the variables for all pairs of algorithms, following the methodology of new statistics (Cummings 2011). Parametric methods will be used, if the data will be normally distributed (assessed by histogram shape) - in doubt, both parametric and non-parametric analysis will be reported. For the main analysis, we will combine results for both equipment conditions.

5) Any secondary analyses?

Analysis of differences between the equipment conditions (exploratory). Further we will assess self-reported enjoyment of fighting against the individual algorithms.

Participants will also directly compare all pairs of the algorithms (collected for control and exploratory purposes).

6) How many observations will be collected or what will determine sample size? No need to justify decision, but be precise about exactly how the number will be determined.

The experiment will be offered to students of game programming at our university (some courses require participation in any of a set of studies for course completion) and through social networks. We will collect data until there are 25 participants or until the end of April 2016.

7) Anything else you would like to pre-register? (e.g., data exclusions, variables collected for exploratory purposes, unusual analyses planned?)

We will collect intrinsic motivation inventory and flow scale to check whether participants were engaged throughout the study. We will also collect self-reported scale to check whether the players actually try to beat the opponents or are just acting randomly and whether they know what they are doing. Participants with average over 6 in those questions will be excluded.

8) Have any data been collected for this study already?

No, no data have been collected for this study yet.