A procedural quest generator for Conan Exiles

António Santos Ferreira Machado

Thesis to obtain the Master of Science Degree in

Computer Science Engineering

Supervisors: Prof. João Miguel De Sousa de Assis Dias
Prof. Pedro Alexandre Simões dos Santos

Examination Committee
Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. João Miguel De Sousa de Assis Dias
Member of the Committee: Samuel Mascarenhas Ph.D
To my Insecurity.
Acknowledgments

I would like to start by thanking Funcom, first and foremost, for having believed in a project envisioned by a student with a passion for games, for the challenge they have presented me with and, in particular, for having shared with me the tools and guidance that allowed the vision of this student to become a reality.

To my advisors João Dias and Pedro Santos, who have accompanied me and advised me throughout this long journey I wish to extend my warmest appreciation for all their time and effort dedicated to this project.

My friends and colleagues, I would like to thank you for having borne with me as this project evolved, for the ideas and fears we’ve exchanged, for the laughs and the jokes we’ve shared, and for the honest and valuable support you’ve freely given me.

To my family… Thank you for keeping me alive.

These acknowledgments would not be complete without paying special attention to the ‘Instituto Superior Técnico’ – to the place that became a home away from home, for long days and longer nights, I thank you.

To all those who will venture reading or perusing through the pages that follow, an everlasting note of thanks.
Resumo

A dissertação desenvolve um modelo de geração procedimental de missões para jogos de computador. Este modelo ambiciona ser capaz de gerar histórias iterativas para Role Playing Games (RPGs) suficientemente credíveis para rivalizar com aquelas produzidas por autores humanos. O sistema proposto utiliza uma gramática derivada da análise estrutural das principais missões do aclamado jogo, "The Witcher 3 - The Wild Hunt". Do ponto de vista da teoria, o nosso modelo estende o trabalho de outros autores em missões MMORPG por forma a acomodar gramáticas capazes de representar missões mais complexas.

O modelo proposto foi implementado num sistema de missões para o jogo Conan Exiles, que presentemente carece desse sistema. Finalmente, o sistema foi testado com jogadores reais. Os resultados dos testes foram muito promissores pois, não obstante a natureza experimental da implementação, o introdução do sistema de missões não piorou a satisfação global dos jogadores de Conan Exiles. Acreditamos que a capacidade de gerar histórias coerentes e credíveis sem autoria humana tem um valor comercial importante, pois reduz o tempo e os recursos necessários ao desenvolvimento de conteúdos.

Palavras-chave: geração procedimental, narrativa interativa, rpg, missão, história, conan exiles
Abstract

The dissertation proposes a procedural quest generation model as a means to generate an interactive story for Role Playing Games that can rival human-authored ones. The system uses a grammar derived from a structural analysis of the main quests in the acclaimed RPG game, “The Witcher3- The Wild Hunt”. From a theory vantage point, our system extends previous work on MMORPG quests by other academics. The modifications introduced enable grammars capable of representing more complex quests.

The model is implemented in a quest system for the survival sandbox game Conan Exiles, which currently lacks a quest system. We also conducted test with human players. The results were very reassuring in the sense that, in spite of its experimental nature, the introduction of the quest system did not deteriorate the players overall enjoyment of Conan Exiles. We believe that being able to generate coherent and believable stories without human authorship has significant value for the game industry as it reduces costs in terms of time and financial resources necessary for content development.

Keywords: procedural generation, interactive storytelling, rpg, quests, story, conan exiles
# Contents

Acknowledgments ........................................................................................................... v
Resumo ............................................................................................................................. vi
Abstract ......................................................................................................................... vii
List of Tables .................................................................................................................. xiii
List of Figures ................................................................................................................... xvi
Nomenclature .................................................................................................................. 1
Glossary ........................................................................................................................... 1

1 Introduction .................................................................................................................. 1
   1.1 Interactive Storytelling ............................................................................................. 2
   1.2 Problem & Hypothesis ............................................................................................. 2
   1.3 Contribution ............................................................................................................. 3
   1.4 Outline ..................................................................................................................... 4

2 Related Work ................................................................................................................ 5
   2.1 Centralized Architecture Examples ........................................................................ 6
      2.1.1 IPOCL .................................................................................................................. 7
      2.1.2 Glaive ................................................................................................................ 9
   2.2 Distributed Architecture Examples ......................................................................... 10
      2.2.1 Continual Multi-Agent Planning ..................................................................... 11
      2.2.2 CONAN .............................................................................................................. 12
      2.2.3 Prototype Quest Generator ............................................................................. 14
   2.3 Mixed Architecture Examples ............................................................................... 16
      2.3.1 IDA ................................................................................................................... 17
      2.3.2 IMPRATical .................................................................................................... 18
   2.4 Plotline Adaptation ............................................................................................... 19
      2.4.1 GAME FORGE ................................................................................................. 20
      2.4.2 PaSSAGE ......................................................................................................... 21
   2.5 Discussion ............................................................................................................... 22
   2.6 A Quest Generator for a Commercial Game ....................................................... 23
      2.6.1 Making a Personal Game vs. Modding a Commercial Game ....................... 23
List of Tables

2.1 Example of character preferences. Actions preceded by a ‘–’ have a lower cost and those
    with a + sign have a higher cost. ................................................................. 13
2.2 Motivations taken from [8]. ................................................................. 15
2.3 Distribution of quests observed based on motivation, taken from [8]. ......... 15
2.4 Example of an NPC with a tier with minor rank. ................................. 16

3.1 Strategies for each NPC’s motivation. .................................................. 35
3.2 Action rules in BNF (Backus Normal Form). ........................................ 36
3.3 Atomic actions. ................................................................. 37
3.4 Initial set of weights that would be used by the generator, if it had the complete grammar. 41

5.1 Mean scores of the variables for each experience. .............................. 65
5.2 Results of the Wilcoxon Signed Test for "Enjoyment". ...................... 67
5.3 Results of the Wilcoxon Signed Test for "Flow". .............................. 68
List of Figures

1.1 A screenshot from a dungeon in Diablo 3[9]. ................................................. 2
2.1 Action syntax taken from [26]. ................................................................. 7
2.2 Glaive’s input taken from [35]. ................................................................. 9
2.3 Glaive’s plan graph example taken from [35]. ......................................... 10
2.4 Example of a characters plan for reaching its goals. Between two statements there is always a player action[4]. ................................................................. 14
2.5 Simplistic plot from IDA [21]. ................................................................. 18
2.6 Determining relevance for a single agent’s actions. Solid lines represent causal links between effect and fact or between fact and precondition relations. Fadded lines indicate a no-op. In the backwards step, two actions have their preconditions met in the current state. One of them is found to be relevant to the actor’s intents[30]. .................. 19
2.7 Original plotline from [17]. ................................................................. 20
2.8 Adapted plotline from [17]. ................................................................. 20
2.9 “My little pony” mod for Skyrim [28]. ..................................................... 24
2.10 Conan Exiles game-play [10]. ............................................................... 25
2.11 A screenshot of the DevKit. ................................................................. 25
2.12 The hierarchy of character classes. ......................................................... 28
3.1 Proposed System. .............................................................................. 31
3.2 The Witcher 3 game-play[24]. ............................................................... 32
4.1 Mapping of the model to Unreal Engine/Conan Exiles. .......................... 45
4.2 Event for getting recipes directly from the player. ................................. 48
4.3 Event for getting the npcs closest to the QuestGiver. ............................. 48
4.4 W_NPCDialogue with two additional buttons. ....................................... 50
4.5 W_QuestGiving UI for presenting the quest to the player. ..................... 51
4.6 W_Compass with an active target point (center), and W_QuestSteps showing the action “Go to Shaleback” (right corner). ................................. 52
4.7 Implementation of the event, corresponding to action “Goto”. ................. 53
4.8 Implementation of the event, corresponding to action “Kill”. .................. 54
4.9 Implementation of the event, corresponding to action “Listen”. ............... 54
4.10 Implementation of the event, corresponding to action "Report". 55
4.11 Implementation of the event, corresponding to action "Gather". 55
4.12 Implementation of the event, corresponding to action "Build". 56
4.13 Implementation of the event, corresponding to action "Give". 56
4.14 The heightmap chosen for both test scenarios. Highlights indicate the locations of the BP_QuestGivers. 58
4.15 Necessary settings for BP_ManualSpawnPoint. 59
4.16 Necessary settings for BP_CampOwner. 60
4.17 Necessary settings for NPCTerritorySpawner. 60
4.18 Merla Lilibeth, one of the quest givers added in the test scenario. 61

5.1 Boxplot related to the answers of the "Enjoyment" section of the questionnaire. The first entry for each item refers to answers to the Conan Exiles experience. 66
5.2 Boxplot related to the answers of the "Flow" section of the questionnaire. The first entry for each item refers to answers to the Conan Exiles experience. 66

A.1 Key to all other Figures. 75
A.2 Initial strategy of example quest. 76
A.3 Expansions of <goto> (rule number 11) and <prepare> (rule number 17), both newly added. 76
A.4 Expansions 5 to 12, use of newly added action "examine". 77
A.5 Expansions 22 to 29. Note expansion of <goto> after <learn> is closed and the use of newly added action "follow". 77
A.6 The Beast of White Orchard quest using the new set of rules. The order in which actions performed by the player should be read depth first then left to right. 79

B.1 Conan Exiles’ basic controls. 80
B.2 Player’s inventory UI. Activated by pressing "I" key. 81
B.3 Player’s crafting station. 82
B.4 Player’s level up menu. 83
B.5 Player’s recipes UI. 83
Chapter 1

Introduction

Computer Role Playing Games (CRPGs), commonly referred to as simply Role Playing Games (RPGs) are a video game genre where a player embodies a story world character (or several, commonly referred to as party) and must overcome a series of linked challenges, ultimately achieving some overarching goal or the conclusion of a central storyline. Furthermore, the player can develop his/her character(s) through consequential decisions.

RPGs are known for being content-heavy games, possessing vast worlds with various non-playable characters (NPCs), as well as intricate story-lines and side-quests [13]. The process of this content creation takes a considerable amount of time and financial resources[17]. However, it’s consumption is much faster. After a player completes every main quest and side-quest, the game’s replay value drops off considerably.

Consequently, RPGs are perfect candidates for the application of Procedural Content Generation (PCG), which is the use of computer algorithms for creating content that meets a set of evaluation criteria [13][32]. This becomes quite useful, when trying to produce content for an industry, that is becoming increasingly demanding [15]. According to Hartsook [13] there are two broad uses of PCG in games: content creation and adaptation of gameplay.

Through the automatic generation of content one could offload the task of content creation, thereby reducing the amount of work done by the humans and making development costs cheaper. Also, by learning players’ information (something that can’t be known during design-time) related to their preferences, desires and abilities, one could adapt game content. Having a game with personalized story and world could maximize player pleasure and minimize frustration and boredom.

PCG has become quite popular in recent years and is currently being used in a variety of different sub-fields. Examples of generated content include: dungeons and maps (Binding of Isaac: Rebirth [23], Diablo series [9]); enemies (Left 4 Dead [33]), animation of character behaviour (Spore [1]), weapons (Borderlands 2 [11]) and even whole universes (No Man’s Sky [12]).
1.1 Interactive Storytelling

Interactive storytelling is a narrative-based experience, where a sequence of events unfolds through the actions and interactions of world characters with the story world, allowing for a narrative to emerge [31][6].

A system based on interactive storytelling must be capable of generating interactive narratives in a coherent and believable fashion. This means that the sequence of events that constitute the overarching story must be causally and temporally coherent and characters that partake in these events must act in a believable manner [5]. In storytelling systems, characters are perceived as being believable when the actions they execute are motivated by their desires and intentions and these are consistent with the knowledge they possess about the current story world [26].

To be able to achieve all of this, an interactive storytelling system must somehow manage all possible actions that constitute an event and are executed by story world characters, as to maintain the logical progression of the story. This management is usually done by a planner (an algorithm that reasons and selects which actions are best fitting for achieving a specific goal). These goals can be global or individual, and are typically pre-defined by a human author.

Planners can be implemented in a centralized manner, where a single planner is in charge of doing all the decision making or, alternatively, be distributed for each character, that will then reason about their own or even other characters actions themselves. Given an initial state and a set of actions, the planner would then be responsible of fabricating a plan, which will fulfil a goal. Plans are used for representing plots, due to their structural similarity, in terms of temporal and causal coherence.

1.2 Problem & Hypothesis

In the context of RPGs, the sequence of events that constitute the overarching story, can also be called quests. Quests are tasks given by NPCs to a player in the form of requests, that ask the player
to complete goals often in return for some reward. If quests are seen as the mere movement from one location to another to achieve a goal, which involves character actions and dialogue, one can see that a sequence of these quests could shape a story.

In games, story is the progression of the player through space [2] [13]. The set of quests that are necessary to complete the game form up the main story. Additionally, side-quests are often offered to the player to extend the gameplay [17]. Most RPGs possess hundreds of quests, which require a lot of human authoring. Having a system that procedurally generates these quests can potentially increase the variability and replayability of games.

So the general problem we are going to tackle, is the use of PCG in the generation of interactive stories, that measure up to those generated by developers. Thus reducing their authoring effort.

We propose a system that procedurally generates these quests, as means to generate an interactive story. This dissertation presents a centralized quest system, that can generate quests and monitor their completion by a human player. The system uses a quest structure obtained from a structural analysis of the main story quests from "The Witcher 3 - The Wild Hunt" (Witcher), an acclaimed single player RPG game. This analysis departs from and further extends the work of Doran and Parberry [8] that consists of a structural analysis of several MMORPG quests.

We consider a quest's structure to be all actions performed by the player, since the moment the quest is given to him/her, until the quest's goal is achieved. Using this structure, the system can procedurally generate a variety of intricate quests. This system was implemented and tested with human players on the survival sandbox game "Conan Exiles", which is currently lacking a quest system.

1.3 Contribution

The contributions of this dissertation are four-fold:

1. We extend the grammar defined in the prototype quest generator of Doran and Parberry [8], which resulted in a published paper [20]. This extension was done in order to be able to represent bigger and more complex quests;

2. We have defined and validated a quest generation model in a paradigm directly applicable to games, that uses the extended grammar;

3. We have implemented this quest generation model and integrated it into a quest system developed for a specific game;

4. We have conducted tests with human players, in order to ascertain their overall enjoyment with the quest system.
1.4 Outline

The remainder of this document is divided as follows. In Chapter 2, Related Work, we review several interactive storytelling systems researched by academic groups. Here we want to find the best approach in order to implement it in a commercial game. In Chapter 3, Quest Generation Model, we present an analysis on the main story quests of a single player RPG, and a theoretical model of a quest generator adapted to be used in a game engine. In Chapter 4, Implementation, we show how the theoretical model was implemented in a quest system for the survival game Conan Exiles. In Chapter 5, Evaluation, we present the results of our the tests in the quest system with human players. Finally, in Chapter 6 we conclude this dissertation with a summary of the work presented and future work.
Chapter 2

Related Work

When creating a narrative generation system, one must take into account two universal attributes of narratives, plot coherence and character believability [26]. This means that there must be a casual and temporal sequencing of plot events [5], and each character must be treated as an autonomous agent with its own beliefs and intents [30]. The player must perceive their actions as intentional, without breaking the suspension of disbelief [26]. Depending on the attribute one is focusing on, a system can either be described as plot-centred or character-centred [6][18].

A plot-centred system has a more restrained environment in terms of authoring, enhancing the control over plot development [18]. The plot is usually pre-authored and has a set of predetermined actions and goals. The system must then follow it, to guide the characters through the narrative [6], thus making an interesting story that the player can take part in. Having pre-determined plots can be very restrictive in terms of player agency (the amount of freedom the player has), since it makes the player follow a pre-determined path as opposed to giving him a perceived authoring of the story.

A character-centred system is driven by a goal-based behaviour of characters [6]. It is motivated by the interactions between world characters and the impact these can have on the story progression, reducing the cost of authoring [18]. Here characters have sets of goals in which they act upon in order to achieve them, thus allowing for believable behaviour to emerge. This approach has the problem of being hard to manage and implement if one is looking to achieve global author goals (descriptions of the world that the author would like the system to achieve).

Riedl and Young [26] classify storytelling systems as using one of two approaches:; simulation-based or deliberative approach.

A simulation-based approach, also viewed as emergent system, focuses on modelling the story world characters behaviours and decision-making processes. To do so, it has first to define a set of world characters, which are often autonomous agents, and a world context. During run-time it has to determine all actions that each character should take at every step. The story will then emerge through their interactions with the world.

In a deliberative system one often considers the creation of a narrative based on the perspective of a human author, the artist's vision [36]. The author has control over the narrative structure and defines
certain goals and constraints that the story most hold by the time it ends. The system is then responsible for solving the problem of choosing a sequence of actions for all world characters that have a predefined narrative structure in mind [26].

In a way, this two approaches can be viewed, as either plot-centred and character-centred. Simulation-based uses in-story characters’ interaction to generate its narrative, while deliberative systems try to follow a pre-authored story, and generates a character’s actions accordingly.

An interactive storytelling system, to create stories that are believable, has to effectively simulate a story world with a coherent plot and guide or control the story world characters believably for an audience. Therefore it requires methods from several AI sub-fields [5], being planning the sub-field that receives more focus in narrative generation. Recent systems delegate to a planner the task of reasoning about selecting, ordering and presenting a set of actions for the characters and events in the story world in the form of a plan. The plan would then constitute a narrative that is both coherent and believable.

Plans and narrative plots are structurally similar, they both present a sequence of events that are temporally and causally related. The plot of a narrative is an enumeration of all the events that occur in the story world, together they describe all the changes in it over time. These are caused by actions, either intentionally performed by a character in the story world or unintentionally.

A plan is able to change a world from one state to another. It consists of a set of ordered operators that if seen as events that occur in a story world, then a narrative plot can be modelled through a plan. Riedl and Young [26] define the narrative planning problem as constructing a plan which achieve’s author’s goals out of steps that are clearly motivated and goal-oriented towards individual character goals.

Broekema [6] has done a compilation of various works from the last two decades, and chose to distinguish several systems based on the architecture of the planner. The planner can be classified using one of these three architectures: centralized, distributed and mixed. The works presented in this chapter are divided using these distinctions.

2.1 Centralized Architecture Examples

In a centralized architecture, according to [6], there is a central entity that has full control over the virtual world. It reasons about all action selection and chooses them for all characters and objects. All except for the player of course. Because of a player’s unpredictability, it has to make sure that all characters, including player character, follow the story. To do this it has to intervene, by blocking and delaying actions or giving the player some kind of hint. This of course does not allow for character agents and the player to influence the narrative, which precludes it’s variation. This type of manager is typically used in a deliberative, plot-driven system [26].
2.1.1 IPOCL

Riedl and Young [26] present a refinement search planner IPOCL, an algorithm used to solve the narrative planning problem. It is based on the POCL planner [26], but with the modification to support character intentionality independent of authorial intent. This is accomplished by extending the plan structure to include intentions of individual agents. IPOCL simultaneously searches the space of plans and the space of character intentions.

To generate a narrative, the system must first receive a problem. The IPOCL planning problem consists of:

- an initial state, containing all facts in the story world;
- a set of objects, depicting characters, locations and items;
- a set of author goals;
- a set of possible actions.

Like in POCL actions are represented as STRIP-like constructs consisting of the operator name, a precondition and an effect, these last two consist of zero or more first order logic literals. Riedl and Young distinguish between two types of actions: the ones that are performed intentionally by a character, which they call non-happenings; and happenings, which are unintentional actions (see Figure 2.1).

![Figure 2.1: Action syntax taken from [26].](image)

An action in a IPOCL plan, to be considered as being performed intentionally must belong to a frame of commitment. A frame of commitment is used to register a character’s individual goals and the actions necessary to achieve them. It consists of:

- a subset of steps from a plan;
- a reference to the characters that has to execute the steps;
- the goal that is trying to be achieved by the character;
- the final step of the subset that contains the goal as an effect.

Each frame of commitment must be associated to a condition that must be established in the world by what they call a motivating step that has an effect of the form "intends". This is a special predicate
used to record character’s intentions. This predicate must temporally precede all steps in the frame. For example a warrior can’t have a goal of cleaning a wound without one being inflicted to him. The IPOCL extends the data structure of the POCL plan to include said frames. So, an IPOCL plan consists of:

- a set of steps;
- a set of binding constraints on the parameters of steps;
- a set of ordering constraints on the steps;
- a set of frames of commitment;
- a set of causal links.

A causal link is the connection between two steps that represents a causal relationship between them, made via some condition that needs to be satisfied. For a plan to be complete, all steps that are not unintentional must belong to a frame of commitment. This means that all preconditions of all plan steps are established and that there are no causal threats in the plan (an effect that undoes the preconditions of another step). A plan is not complete, if it contains flaws. IPCOL, as stated before is a refinement search planner, so it uses an iterative, least-commitment process of identifying them in a plan and correct them. The plan refinement part of the IPOCL algorithm is divided in three parts.

1. A causal planning part, which identifies a precondition in a step that is not satisfied by any causal link. It chooses non-deterministically an existing step that resolves this. If there is none, it instantiates a new action. If a new step was instantiated and it corresponds to a final step of an undiscovered intention, then a new frame of commitment is constructed. Because a step can be performed as part of more than one intention, it has to search the plan for more frames of commitment the step can be part of. For each frame of commitment found this way it registers an intent flaw in the plan. After this, it has to mend the newly introduced flaws.

2. A motivation planning part, in charge of making certain that story world characters have a motivation associated to steps in the plan. The process looks for frames, resolves threats and establishes a motivation link between a motivating step and a frame of commitment.

3. An intent planning part, which resolves intent flaws by including a step in the interval of a frame of commitment, non-deterministically in one of two ways. It either makes the step part of the frame and refines the plan or does not include it and removes the flaw from the plan. When a step is added, an entire sequence of establishing steps may follow, since the newly added step may have causally linked steps that precede it. Each step can be included in the frame if and only if:

(a) It is to be executed by the same character of said frame;
(b) It is not yet part of the character’s frame of commitment;
(c) It does not yet have an intent flaw associated to it.

If all conditions hold, an intent flaw is created.
All three parts after completion recursively call The IPOCL algorithm that then non-deterministically selects one of the three processes again. Some limitations can be found in this approach. First POCL based algorithms are often too slow which prevents their use in an interactive experience during run-time [35]. And second, as Riedl and Young[26] point out, it limits the types of narratives that can be produced, since it is unable to consider characters failing to achieve their goals in the produced narrative structure. Conflict can arise between two characters with contradictory goals, which is something natural in most stories.

2.1.2 Glaive

Glaive is a state-space heuristic search planning algorithm that solves the narrative planning problem defined by Riedl an Young [26]. Additionally it includes solutions with failed character plans. It reasons directly about character intentionality and about how characters cooperate and conflict. This is facilitated by alternating world representations, in addition to tracking author’s goals. Glaive attempts to combine the speed of Fast-Forward algorithm with the causal reasoning capabilities of IPOCL and CPOCL [35].

It receives as input: a domain with a set of parametrized actions that can occur as events in a story; and a problem containing the initial state and the author’s goals (see Figure 2.2). Both written in Planning Domain Definition Language. The solution is a valid intentional plan, which is a sequence of steps that are causally linked (every step is explained in some possible world) and the author’s goals are true after all steps are taken. The plan also includes what Ware [35] defines as intentional paths, similar to frames of commitment defined by Riedl and Young [26], that represent individual characters’ plans described in terms of character goals and causal structures (see Figure 2.3).

Beginning at the initial state, the Glaive algorithm non-deterministically selects a potentially motivated step whose preconditions are satisfied and adds it to the plan. It then applies the effects of the previously chosen step to the current world state. For each effect in the form character "intends" goal, defined like Riedl and Young [26], it adds a new goal to the tracked set of character goals. If any character consents to the applied step, the step is added to the set of unexplained steps, this implies
a commitment to explain why characters execute it. A step to become explained must belong to the intentional path of each consenting character and all steps in the path must also be explained.

Figure 2.3: Glaive’s plan graph example taken from [35].

The algorithm continues by verifying if each step belonging to an intentional path of a goal intended by a character is explained. If so the step is removed from the set of unexplained steps of every state, where the instance of that step is unexplained [35]. This way it can be included in the valid plan of the individual characters’ failed plans. Glaive then returns a solution if it finds a state that achieves the author’s goals and all steps in the plan are explained, otherwise it keeps searching. The Glaive algorithm uses an heuristic that is responsible for its speed. The returned value is the maximum of two estimates: one derived by reasoning backwards from each character goal, using goal graphs; and another by reasoning forward from the current state to the author’s goals [35], using graph plans.

Both previously described approaches can be seen as using a centralized drama manager, a single planner has control over the story world and its characters. Besides achieving pre-defined author goals, they manage to reason about character intentionality and achieve their individual goals, giving the feeling of believability to an audience.

2.2 Distributed Architecture Examples

In a distributed architecture on the other hand does not have a central entity that controls world characters. Instead it uses autonomous agents that control their own character in the story. Every character agent reasons about their actions autonomously and select them. This permits the characters to create narrative with their actions allowing for story variation and the possibility for player autonomy [6]. It is implemented as a multi-agent system and usually used in a simulation-based system [26], or emergent system.
2.2.1 Continual Multi-Agent Planning

Brenner [5] has developed MAPSIM a simulation-based system, that generates multi-agent simulations. In MAPSIM character agents are implemented using the distributed algorithm Continual Multi-agent Planning (CMP). CMP models plans made by individual characters. This plans can be executed individually by multiple characters, allowing the generation of plots that are both plot-centred and character-centred.

This system uses Multi-agent Planning Language (MAPL) ¹ not only to model the story world environment, but also to represent character agents’ physical actions, means of communicating and sensing, (shared) beliefs and (joint) goals. Actions are extended with two additional types of preconditions when compared to others previously described on this document: knowledge preconditions, since an agent can only execute an action if it has the necessary information to do so; commitment preconditions, because every MAPL action has an autonomous controlling agent responsible for its execution. This enforces characters to agree to pursue it.

Thanks to the variables used by MAPL, it is possible to express incomplete knowledge and alter unknown values with information gathering actions. MAPL variables also help to model agent’s beliefs, which can be affected either via: sensing the current perceived value of a variable, or via communication ². Additionally, mutual beliefs are affected via co-presence. If two characters are in the same place then they are able to perceive the same things, besides each other.

A CMP agent is able to generate plausible behaviour by deliberatively switching between (re)planning, executing a plan, monitoring its effects and communicating with other characters. The plan, is partially ordered and represents a sequence of actions for various agents necessary to achieve the planning agent’s goal, that are causally and temporally linked.

If an agent has sensed an external event or its goals have changed, the agent enters the planning phase, where it will try to correct the parts of the plan that make it invalid. Characters can adhere to temporary goals, or gather information necessary to make the main goal achievable. A character may also have a sub-goal related to another character’s goal. The planning agent must then make sure that other characters involved in its plan are committed to their sub-goals. This can be done either by negotiating or persuading the character. MAPL allows for the activation and deactivation of these sub-goals, similarly to BDI models turning committed desires into intentions, by using the keyword “current”, which also refers to the current state.

Once it has found a valid plan, an agent begins the execution phase. It non-deterministically chooses an action on the first level of the plan. The action will be executed by said agent or the action’s controlling agent. Finally, the agent monitors the effects of the executed action and updates its knowledge of the world state with the perceptions made. CMP is then able to: generate believable character behaviour by interleaving physical action, sensing and communication; and also generate dynamic plots in which beliefs, motivations and character traits may change [5].

¹ multi-agent variant of PDDL that uses multi-valued state variables as an alternative to the typical propositions
² declarative statements, questions, requests and acknowledgements, all modelled like wh-questions
2.2.2 CONAN

In distributed systems stories emerge from the interaction between characters, that are moved by their own intentions. In order to maintain those stories coherent, Breault [4] suggests, that if a human player can act as a "director agent", a coherent story would emerge from the interactions between the player and characters. These interactions are depicted as the assignment of a quest to the player by the characters who make them. This will produce interesting emergent narratives, since the human player would have the capability of choosing which quests given by the characters he/she can undertake.

Breault [4] proposes a distributed quest generation system capable of producing quests, that are coherent and solve goals, relevant to character preferences. This means, given an initial state, and the domain files with all possible actions, the NPC's will produce plans to achieve a goal in accordance with their predefined preferences. These plans are then given to player as quests in the form of requests.

The initial state of the world consists of:

- A set of objects: these can either be characters, monsters, items, locations or information;
- A set of facts: these are in the form of statements, which are a combination of predicates and objects belonging to the previously presented set. Here are a few examples:
  - (character king);
  - (location castle);
  - (item sword);
  - (has king sword);
  - (at king castle);
  - (adjacent castle village).

In his work, he uses two initial world states with different sizes. This way he can measure the effects on generated quests, when comparing the results of the smaller world with the ones from the more complex world. He uses the modified Aladdin world (small), which is frequently used in literature [26][30], and a larger world that he defined, with several more characters, items, locations and predicates.

The domain files contain the set of possible actions, necessary to achieve character goals. Actions are implemented using PDDL and are the same as defined by Doran and Parberry [8], in their structural analysis of quests. These actions represent all actions a player can perform. Each action contains:

- A name;
- A list of parameters. These can be of the same type as the objects previously mentioned;
- A list of preconditions. These have to be met for the action to be performed;
- A list of effects. Definition of the world state after the action is performed.

For example the action "move" is defined like this:
(:action move
 :parameters (?player ?toloc ?fromloc)
 :effect (and (at ?p ?to) (not (at ?p ?from)) (increase (total-cost) ?x)))

Each character possesses a set of preferences, represented as values weighting all actions. This mapping is specifically implemented in every character, in the form of an array. Each action thus has a weight, in accordance to what the character likes and dislikes. These weights are then added to the total cost of a plan. Actions that are in-line with the character's preferences will have a lower cost, while actions that go against them have a higher cost (see Table 2.1). This mapping of preferred actions guarantees that characters remain coherent in terms of choice of action, and offers the sense of believability in characters.

<table>
<thead>
<tr>
<th>Character</th>
<th>Preferences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aladdin</td>
<td>[&quot;+kill&quot;, &quot;-exchange&quot;, &quot;-use&quot;, &quot;+escort&quot;]</td>
</tr>
<tr>
<td>Dragon</td>
<td>[&quot;-damage&quot;, &quot;-take&quot;, &quot;-report&quot;, &quot;+escort&quot;, &quot;-defend&quot;]</td>
</tr>
<tr>
<td>Genie</td>
<td>[&quot;-kill&quot;, &quot;-exchange&quot;, &quot;-defend&quot;, &quot;-read&quot;]</td>
</tr>
<tr>
<td>Jasmine</td>
<td>[&quot;+kill&quot;, &quot;-spy&quot;, &quot;-take&quot;, &quot;-stealth&quot;]</td>
</tr>
<tr>
<td>Jafar</td>
<td>[&quot;-kill&quot;, &quot;-spy&quot;, &quot;-take&quot;, &quot;-stealth&quot;, &quot;move&quot;]</td>
</tr>
</tbody>
</table>

Table 2.1: Example of character preferences. Actions preceded by a ‘–’ have a lower cost and those with a + sign have a higher cost.

Goals are represented as sets of statements, in the same form as described above. However, instead of already being combined with objects, they have variables restricted to the type of object. For example:

- (has ?c ?i), where ?c is a character and ?i is information;
- (has ?p ?o), where ?p is the player and ?o is an item;
- (dead ?m), where ?m is a monster;
- (explored ?l), where ?l is a location.

The algorithm responsible for generating the quest, begins by creating 4 random goal states. To do this, it selects 4 random predicates as defined above and assigns a legal object to replace the variable. It chooses only 4, because more would be computationally taxing. After creating the goals, the algorithm will find the least expensive plan to reach each of them, which includes actions to be performed by the player.

It calculates the average cost of the actions included in each plan. Remember that the cost of an action is based on the weights defined in the character's preferences. So if a preferred action has a cost of 1, the optimal plan would have an average cost of 1 as well. The plan with the lowest cost will be given out as a quest to the player. This process is done individually by a planning agent. The player, in
Figure 2.4: Example of a characters plan for reaching its goals. Between two statements there is always a player action[4].

its turn, will try to solve the proposed problem, by taking the steps described in the plan (see Figure 2.4 for an example of a plan).

CONAN uses the Fast Downward planning system [14], a classical planning system based on heuristic search encoded in PDDL2.2. It supports many heuristics, but the CONAN planner was implemented using A*.

2.2.3 Prototype Quest Generator

Doran and Parberry [8] did a structural analysis of almost 3000 human-authored quests from several Massive Multi-player Online Role Playing Games (MMORPG). The analysis showed a common structure shared by human-authored quests, "changing only details such as settings, but preserving the relationship between actions". They observed structural patterns in quests, which occurred in predictable situations, each with its own implicit preconditions and effects.

They first observed that quests can be categorized into 9 distinct NPC motivations: Knowledge, Comfort, Reputation, Serenity, Protection, Conquest, Wealth, Ability and Equipment (see Tables 2.2 and 2.3). They believe the use of motivations to be essential for ensuring intentionality in the generation quests. Quests are thus intended to represent a NPC’s prime concern.

Each of these motivations contains 2-7 motivation-specific strategies. In turn, each of these strategies is composed of a sequence of 1-6 actions, that the player must perform. Each action is further defined as either an atomic action performed by the player, or a recursive sequence of other actions or action variants.

The quest structure is presented in the form of a grammar in Backus-Naur Form. Terminal symbols are atomic actions and non-terminal symbols are action rules, that extend to further actions or action
<table>
<thead>
<tr>
<th>Motivation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge</td>
<td>Information known to a character</td>
</tr>
<tr>
<td>Comfort</td>
<td>Physical comfort</td>
</tr>
<tr>
<td>Reputation</td>
<td>How others perceive a character</td>
</tr>
<tr>
<td>Serenity</td>
<td>Peace of mind</td>
</tr>
<tr>
<td>Protection</td>
<td>Security against threats</td>
</tr>
<tr>
<td>Conquest</td>
<td>Desire to prevail over enemies</td>
</tr>
<tr>
<td>Wealth</td>
<td>Economic power</td>
</tr>
<tr>
<td>Ability</td>
<td>Characters skills</td>
</tr>
<tr>
<td>Equipment</td>
<td>Usable assets</td>
</tr>
</tbody>
</table>

Table 2.2: Motivations taken from [8].

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Quests</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge</td>
<td>138</td>
<td>18.3%</td>
</tr>
<tr>
<td>Comfort</td>
<td>12</td>
<td>1.6%</td>
</tr>
<tr>
<td>Reputation</td>
<td>49</td>
<td>6.5%</td>
</tr>
<tr>
<td>Serenity</td>
<td>103</td>
<td>13.7%</td>
</tr>
<tr>
<td>Protection</td>
<td>137</td>
<td>18.2%</td>
</tr>
<tr>
<td>Conquest</td>
<td>152</td>
<td>20.2%</td>
</tr>
<tr>
<td>Wealth</td>
<td>15</td>
<td>2.0%</td>
</tr>
<tr>
<td>Ability</td>
<td>8</td>
<td>1.1%</td>
</tr>
<tr>
<td>Equipment</td>
<td>139</td>
<td>18.5%</td>
</tr>
</tbody>
</table>

Table 2.3: Distribution of quests observed based on motivation, taken from [8].

rules. Here, atomic actions are viewed as concrete actions performed by the player during the game. The sequence of actions, that the player is required to perform to complete the quest, can be viewed as the leaves in a tree, with the root representing the entire quest. Actions can also be replaced by sub-quests, that use the same structure.

A quest with the Knowledge motivation, could be for example described in the following way:

\[
\text{<Knowledge>} := \text{<Deliver item for study>}
\]
\[
\text{<Deliver item for study>} := \text{<get>} \text{<goto>} \text{give}
\]

Where:

- \text{<Deliver item for study>} is the strategy;
- \text{"<get>} \text{<goto>} \text{give}" is the associated sequence of actions;
- \text{<get>} and \text{<goto>} are non-terminal actions;
- \text{"give"} is a terminal action.

With this structure made from the extracted rules and the commonalities of the analysed quests, the authors are able to demonstrate a prototype system that procedurally generates quests, which in their view are appropriate for use in RPGs. The generator starts with a NPC motivation, then it consults the list of specific strategies, selects one and creates a quest that addresses the motivation. The generator was written in Prolog, due to its “ability to backtrack and try alternative solutions” [8].
We understand that there might be some confusion as to why this work is presented in this section. The prototype generator creates quests devised by a single entity, that includes interactions between different characters. This means that the prototype can be described as using a more centralized architecture. However, we believe that: if the subquests included in a quest are generated by another character, with its own motivations and grammar, their work could be described as using a distributed architecture.

2.2.3.1 The Quality System

Bruss et al.[7] developed a progressive tier system called Quality System in order to increase cohesiveness and connectivity between quests generated by the quest generator defined by Doran and Parberry [8]. They introduce 2 concepts:

- **NPC profession**, which influences the type of quests that are generated;
- **NPC tier** which helps rank the quest in terms of influence and level of the NPC.

Instead of randomly generating an initial motivation, like in Doran and Parberry[8], the Quality System generates a random NPC with a random profession. This NPC will then be responsible for generating a quest best suiting his profession. For example a farmer might want the player to kill some wolves that are a harassing his sheep. To provide the player with a sense of progression, professions are assigned into tiers with ascending qualities (minor, major and epic). Each tier has categories containing professions motivations and items, fitting the quality of the tier (see Table 2.4).

<table>
<thead>
<tr>
<th>Profession</th>
<th>Motivations</th>
<th>Items</th>
<th>Enemies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Farmer</td>
<td>Knowledge 3,4, Comfort 2, Serenity 4, 5, Protection 3, Equipment 1</td>
<td>Seeds, Farmingtools, Food/harvest</td>
<td>Wolves, Rats</td>
</tr>
</tbody>
</table>

Table 2.4: Example of an NPC with a tier with minor rank.

In their system, the first quest is always of minor quality. Once a quest is completed, the system performs a calculation to determine if it should advance to next tier, which would have a different NPC providing the quest. This change would be highlighted using a connection text. The tier of the next quest is determined randomly with an increasing chance of progress dependent on the number of quests completed in the current tier.

2.3 Mixed Architecture Examples

According to Broekema [6], the emergent approach yields to what he calls a Narrative Paradox, since a plot is needed in order to have a structured narrative. If this is the case then characters don’t possess
the freedom to create one with their interactions. Broekema [6] argues that it is impossible to have a predetermined plot that forms a strong story and strong autonomous characters free to do whatever they want.

It is possible though to have both in a weakened form, a mixed architecture. A mixed planner follows a global plot-line and also has autonomous agents, representing story world characters with their own goals that reason about their actions. This architecture then uses a central manager to handle plot events and direct the partially autonomous characters so they don’t stray from plot, thus maintaining the story’s normal progression.

Having systems with autonomous character agents that reason about their own actions and intents might be good for the emergence of believable character interaction, but there is no guarantee that they will follow an authors intentions or generate interesting stories. In contrast, single planners tend to struggle on making characters believable, but are able to follow pre-authored plots to achieve an authors intent.

2.3.1 IDA

IDA (Interactive Drama Architecture) [21] is an interactive storytelling system that creates experiences driven by pre-authored plots, but at the same time offers the player a considerable amount of freedom. This is possible through an omniscient story Director, which takes the initial state of the world and a plot description as inputs. The director has 3 major roles to ensure narrative integrity:

1. Direct the intelligent agents that populate the world. The character agents are capable of processing basic world knowledge, use goal-based behaviours and have predefined characteristics. They maintain their own structure of goals and memory elements, which are created when they interact or observe the story world, or can even be given by the director, if it deems the plot content relevant to the agent. The character agents are capable of behaving autonomously, but are designed to be directed by the director agent.

2. The director must monitor player behaviour. Sometimes the player might execute actions that go against the plot’s final goals. In that case the director must dynamically change the world/plot in order to incorporate the player’s or character’s action.

3. The director has to instantiate all content in the plot, as the player goes through it.

They represent their plot through a state-based model. The plot is a directed graph of plot-points, that need to be reached. Transitions between states ensure causal and temporal coherence between events. They contain time constraints, to guarantee that the plot does not take too long to reach a specific state, as well as agent actions. Each plot point possesses logical clauses and constraints, describing how the world should look, and any direct interaction with player (see Figure 2.5).

When monitoring the player, the director agent uses error recognition and movement prediction, to avoid any action that compromises the plot. The director agent will execute a director action to make sure that the story world ends up in the right state. To do so the director has multiple methods to influence
the story world. It can give characters new goals, information about the world or specific actions that they must perform directly. These will change the memory of the agents. Furthermore, the director can create, remove or manipulate objects. For example, instead of waiting for the characters to come to the same room when the plot requires this, the director can direct one of the characters to perform the required action, and move it to the room where the other character is.

2.3.2 IMPRACTical

Teutenberg [30] proposes a mixed approach to story generation, where we have character agents responsible for reasoning about their individual intentions and checking for relevant actions regarding those intentions; and also a director agent, which is a single planner, in charge of generating stories that satisfy relevant actions constraints, acting as an external guidance. Teutenberg[30] argues that decomposing a system this way produces stories more efficiently and more expressive, when compared to single planners.

The IMPRACTical (Intentional Multi-agent Planning with Relevant ACTions) system is able to model both intent and action causality in the narrative, performing global heuristic searches and delegating intent to multiple planning agents. For this approach they require that every narrative action be causally linked to an individual character's intent, and that the intent is achieved prior to the action's execution, because actions need to make sense according to story world knowledge.

They identify three types of reasoning on characters' intentions. Sometimes characters, while reasoning in isolation, can't fulfill their intentions by themselves. However, if another character shares the same intent, and can't also achieve it by himself, both characters can cooperate in order to fulfill the shared intent. Another possibility, would be predicting other characters' actions and include said actions in its own plan, as a mean of justification. Furthermore, a character can also give out commands to other characters to help it achieve its intent. In turn, the character, which was issued a command can further issue commands in order to satisfy the first character's will, thus creating a chain of commands.

The concept of IMPRACTical is then to make a group of agents that represent story world characters, reason about intent, and given a current world state as well as domain information, to provide the single planner, in charge of generating the narrative, with all the relevant actions. The narrative generator will then explore the many potential branches without making an early commitment to a specific plan.
Figure 2.6: Determining relevance for a single agent’s actions. Solid lines represent causal links between effect and fact or between fact and precondition relations. Faded lines indicate a no-op. In the backwards step, two actions have their preconditions met in the current state. One of them is found to be relevant to the actor’s intents[30].

for an action to belong to.

Besides having access to the world state containing all true facts and domain information, characters are also aware of all other characters in the world and all intents associated, as well as their reasoning process to fulfil them. So a character is then able to know which characters share the same intent, and plan to make use of other agents’ future actions. Command issues are integrated by considering them as an typical action to be included in the planning agent’s plan.

All this is implemented in a single function, which relies on the concept of a relax domain, like in Glaive [35], where an action’s effects can’t delete the necessary preconditions of another action. To solve a problem in the relaxed domain, it must first construct a relaxed planning graph, by finding all reachable actions and applying them to found facts, moving forward from the current state. A solution is then extracted by searching back through the graph, which according to Teutenberg[30] guarantees causal relationship between all actions. IMPRACTical successfully combines global search and multi-agent planning as a solution for generating believable stories (see Figure 2.6).

2.4 Plotline Adaptation

As stated before one of the ways we can use PCG in games is to adapt the manner in which content is presented to the player. The game content can be altered in order to meet a player’s preferences or capabilities. Since one of the themes of this dissertation is storytelling, we believe it to be beneficial to give an insight on works related to plot line adaptation.
2.4.1 GAME FORGE

To solve the problem of plot-line adaptation, Li and Riedl [17] present an offline algorithm that, given a main plot-line (consisting of a sequence of quests), a library of quests, and a set of player requirements, produces “a sound, coherent variation” of the original plot-line, while preserving the human authors’ intent and meeting player requirements.

The complete plot-line is represented by a partially ordered, hierarchical plan composed of events that will unfold in a virtual world. These events occur within and outside quests, and are represented by actions performed by the player and non-player characters in the virtual world. Events possess preconditions that must be satisfied, and effects that become true. Causal relationship between two events is established through links via some condition that needs to be satisfied. They allow abstraction hierarchies, through decomposition of abstract events into less abstract ones.

In their approach quests are represented as top-level abstractions. Quests have only one effect, the acknowledgement of its completion, and they may or may not have preconditions. Quests are then decomposed into two abstract events: a task and a reward, which are further decomposed into basic actions (see Figure 2.7).

A quest that requires the player to hunt down a witch, can be decomposed in the following way: the player would first have to get a water bucket to pour on the witch, ultimately killing her (the task event), in order to acquire the trust of the king (the reward event). In between, events like the witch dropping her shoes, the player picking them up and showing them to the king as proof of the achievement, would complete the plotline. Narrative soundness and coherence are then guaranteed, through the satisfaction of all preconditions of an event by connecting each event through causal links, and thus creating a path that leads to a significant outcome.

The game plot adaptation algorithm takes the partial-order plan described, as well as the set of player preferences. The search is conducted by adding and removing events until success criteria are

---

Figure 2.7: Original plotline from [17].

Figure 2.8: Adapted plotline from [17].
met (see adapted plot-line in Figure 2.8). Once complete, the resulting story structure is converted and sent to GAME FORGE system\(^3\) [13] that renders a world that supports the story and executes the game. Indeed, although Li and Riedl [17] are capable of producing quests with a certain amount of control, based on players’ requirements, their approach is still dependent on human authoring for the sequence of quests that constitute the plot line. Furthermore, the quests are customized and generated at the start of the game, as opposed to being generated while the player is in play.

### 2.4.2 PaSSAGE

PaSSAGE (Player Specific Stories via Automatically Generated Events)[31] is an interactive storytelling system that makes its decisions, in terms of which events are included in the story, based on a model of a player’s preferred style of play. It uses this gathered knowledge to dynamically adapt the content offered to the player. Hoping to improve a player’s enjoyment of the game directly.

The player model in PaSSAGE is based on the player types defined by Robin Laws[16]. He defines 6 types of players:

- Power Gamer: a player who prefers improving his/her character and gaining reaches;
- Butt Kicker: a player who prefers combat;
- Tactician: a player who prefers to overcome challenging yet logical obstacles;
- Specialist: a player who prefers to always play with the same character in every setting;
- Method Actor: a player who prefers to take dramatic actions;
- Storyteller: a player who prefers complex plots.

PaSSAGE’s player model only considers five types. These are represented in an array as weights. The higher the weight, the more probable it is for the player to belong to that type. The initial model is set through a series of questions with pre-authored answers that the player must respond at the start of their game experience.

PaSSAGE’s decision making process is divided in three phases:

1. **Event Selection**: Story events are drawn from a pre-authored library with information concerning which player type each one suits. This means each event has one or more branches, which reflect potential courses for the player. PaSSAGE examines each event’s set of branches and chooses the best fitting to the model’s current values.

2. **Event Specification**: Story events are scripted generically, making them independent of time, place and character identity. These details are only determined at run-time. This proves useful for games that offer large quantities of side-quests, and force the player to find the ones he/she likes. In this approach, an event is chosen based on the model and activated near the player’s

\(^3\)Components of GAME FORGE, were presented in different papers, one presenting an offline planning algorithm[17], and one describing the whole GAME FORGE system[13].
location. However, event activation can only take place once all details have been specified. This is managed via triggers, functions that once activated, search for suitable actors from a subset of the game environment to fill the roles.

3. **Event Refinement**: After selecting and specifying an event and its branch, there is no guarantee that the player finds said branch as a viable action to perform. This approach uses hinting, performed by actors during run-time, in an attempt to direct the player towards the chosen branch. Thus refining the occurrences in the event.

PaSSAGE was implemented using Aurora Neverwinter Tool-set, and was given a library containing a set of events inspired by a fairy tale. They authored a story and divided it into three stages. For each stage two events were added, each with one or two branches for specific player types. They conducted a user study consisting of 90 participants. Each participant played through one of three stories: two fixed and one adaptive. Afterwards each rated their experience. Players found adaptive versions more entertaining than fixed stories, and also felt higher agency in adaptive versions.

### 2.5 Discussion

After looking at the storytelling systems’ architectures and analysing the way stories are created in them, we will now go through what we believe to be some of the most relevant characteristics for our project. A storytelling system has to be capable of presenting and guiding the player through a coherent plot.

The process of creating a plot is delegated to an algorithm, usually called planner, that reasons about which events, characters and objects will take part in the presented plot. To accomplish this, we must define and give to the algorithm the following sets:

- **World Facts**;
- **World Objects**;
- **Plot Goals**;
- **Character Actions**.

**World Facts**, is a set of statements that represent the way the world looks. For example: where are certain characters located, or what items does a character possess.

**World objects**, is basically a set with elements living in the game world, such as: characters (king, guard, witch), items (sword, potion, book), monsters (dragon, griffin, troll) and locations (castle, village, forest).

**Plot Goals**, is the set of statements that need to become true at the end of the story. These can be defined by a human author or created by the game characters themselves, based on the happenings in the world.
Character Actions, is the set of all actions that characters (including the player) can perform in the world. They are defined by their name, a set of parameters referencing World Objects, a set of conditions and a set of effects. In order for an action to be executed, its conditions must be satisfied by World Facts. Once executed the effects become World Facts.

Some approaches have considered the plot as a quest. A quest is task commonly offered to a player in RPGs by some character. Typically, in this type of game, the main story is comprised of several of these quests. So naturally, we also have a particular interest for approaches that create stories, based on the goals of world characters, since we want to develop a system to be used in an actual game.

Today’s games possess huge worlds, with loads of characters. It would be interesting to see these characters reason about goals that they want to achieve and about the necessary actions to do so. We want to be able to offer as much variability as possible in generated quests, since we are looking to reduce the authoring effort that is needed to write stories for these types of worlds. Using a system based on quests that achieve characters’ goals could help to accomplish that as well as increase player influence in the story.

Before deciding on the most appropriate approach, we will need to look for a released game, where we can implement a storytelling system on.

### 2.6 A Quest Generator for a Commercial Game

In this section we will discuss what game we should use and what is the most appropriate system to implement on that game. We need a game that gives us the freedom to add any number of structures and allows us to alter NPCs’ behaviour using scripts.

#### 2.6.1 Making a Personal Game vs. Modding a Commercial Game

There are two courses we can take, in terms of where our desired system will be implemented. We can develop our own game with a small world and minor game mechanics, using an open-source game engine, like Unity, Game Maker or Unreal Engine 4. This would certainly give us the ability to do anything we want, but it would also be cumbersome, since we would have to implement all game mechanics and necessary user interface.

Alternatively, we can look for a game which has a development tool made available by the game’s developers, that allows us to modify (modding) it. Choosing one with a large and active community, could help evaluate the success of our implementation more properly, when compared to testing a game of our own.

Mods help extend the replay value and interest of a game. They have become increasingly important in the commercial success of some games. Most popular and recent games that allow modding are FallOut4 [29], Dragon Age: Inquisition [3] or The Elder Scrolls V: Skyrim [28]. However, most of the available mods for games are purely aesthetic (see Figure 2.9). Standard tool kits don’t give a lot of freedom to the modder to implement its own scripts and structures.
Nonetheless, we found that Funcom, the creators of Age of Conan, had recently released a game with an additional development kit, where you can "freely" add your own scripted content to the game. This game is a survival sandbox called Conan Exiles, which we have chosen for our project. One of the main reasons being that Conan Exiles does not possess any type of quest system.

2.6.2 Conan Exiles

Conan Exiles [10] is an open-world survival game developed by Funcom and released as early access on the PC on January 2017. The player is an Exile in the brutal lands of Conan The Barbarian, one of thousands cast out to fend for themselves in a wasteland swept by terrible sandstorms and besieged on every side by enemies. Here the players must fight to survive by building tools and structures and dominating their foes.

The player possesses personal statistics like hunger, thirst, temperature, which he/she must pay attention to. The player's first battle is against the harsh environment. The game allows the player to grow crops, hunt animals for food or harvest resources to build tools and shelters. Players can explore the vast world alone, or form guilds together with other players and build a stronghold to withstand fierce invasions from other guilds. The game offers to modes: Player vs. Player and Player vs Environment (this one both multi-player and single player).

The game was developed using Unreal Engine 4. Additionally, Funcom opened the game to Mods and provided a development kit (DevKit) to do so. The DevKit is available through Epic Games Launcher under the "Modding" tab. The DevKit is a modified version of the Unreal Engine editor and will let you do "almost" everything the developers can do except C++ code changes. Nonetheless, it is still a great tool for implementing a system with quality graphics and good game-play mechanics, since it allows us to define our own functions and structures which we will go through further on. Community members have set up a discord chat room for exiles modding.
2.6.3 Unreal Engine & Conan Exiles DevKit

Devkit is the name of Conan Exile’s modding kit that was made available by Funcom to facilitate the development of mods by the community. Conan Exiles was developed using Unreal Engine 4 which uses C++ and Blueprint scripting. The DevKit is a version of their Unreal project, that includes the tools needed to create custom content and also uses their game files.

The DevKit is a very powerful tool as it is practically the same tool that Funcom itself used to create Conan Exiles. However, the Devkit does not allow for the modder to use C++ scripting. So we can only edit assets made directly in the Engine with Blueprint scripting. It is not possible to rename, move or delete existing game assets. New assets can only be added in the specific mod folder. We are also allowed to test the mod directly in the current viewport. The version that is available for the general user is free and anyone can download it via Epic Game Launcher. There are not many guides or tutorials for modding in Conan Exiles.
There is an official channel in the Discord application, where modders can share their ideas and questions. It is also possible to chat with some developers there. The Devkit is updated constantly, since the game is still in development, therefore we recommended that users always keep a back up of their own mods.

2.6.3.1 A Visual Scripting Language

Blueprints are used to define object-oriented (OO) classes or objects in the engine. Blueprints work by using graphs of Nodes for various purposes:

- managing sub-object instancing for classes;
- storing and modifying default properties;
- object construction;
- individual functions;
- general game-play events.

All specific to each instance of the Blueprint in order to implement behaviour and other functionality.

By connecting Nodes, Events, Functions, and Variables with Wires, it is possible to create complex game play elements. Blueprints can be one of several types that each have their own specific use from: creating new types, to scripting level events, to defining interfaces or macros to be used by other Blueprints.

Events are nodes that are called from game play code to begin execution of an individual network within the EventGraph. The EventGraph of a Blueprint is a standard graph that uses events and function calls to perform actions in response to game play events associated with the Blueprint and add functionality that is common to all instances of that Blueprint. These events can be accessed within Blueprints in order to implement new functionality or override or augment the default functionality. Any number of Events can be used within a single EventGraph.

Events work in similar way as functions but with some minor exceptions. An Event cannot have any output, while a function can return values. An Event can use Delay nodes or time-lines, the function can’t. Functions can use local variables. Also, Events can be replicated, functions can’t.

2.6.3.2 Conan Characters

An Actor is any object that can be placed into a level. Actors are a generic Class that supports 3D transformations such as translation, rotation, and scale. Actors can be created (spawned) and destroyed through game play code (C++ or Blueprints). They may contain a collection of components, which can be used to control how actors move, how they are rendered, etc. The other main function of an Actor is the replication of properties and function calls across the network during play.

ActorComponent is the base class for components, which are a special type of Object designed to be used as sub-objects within Actors. These are generally used to define reusable behaviour that can be
added to different types of Actors or to easily have swappable parts in order to change the behaviour or functionality of some particular aspect of the owning Actor. Components created as sub-objects within an Actor are instanced, meaning each Actor instance of a particular class gets its own unique instances of the Components.

Every in game character inherits and extends the UE base class AActor. Conan Exiles’ characters are defined through the class AConanCharacter, which is the base class that one uses whenever one wishes to refer to a game character. This class is defined in C++, and is extended by the Blueprint class BaseBPChar. In Unreal, in order to be able to add something to the game world, it has to be done with Blueprints. So in order to add stuff to the game world, but still be able to write with code, we must first define a class in C++, and then extend it using Blueprints (note that it is still possible to refer to content defined in C++ in Blueprints, so there is no major downside to it).
BaseBPChar is then further extended by the blueprint class BaseBPCombat. This is where some of the gameplay mechanics, like movement, combat, character death, etc are implemented. This was one of the blueprints, where we have added some of our own code, which we will get into more detail further down in this dissertation. BaseBPCombat again is further extended by two different blueprint classes: SurvivalChar and BaseNpc. It is at this level of the hierarchy, that a distinction between player character and non-playable character is made. SurvivalChar is extended by another class BasePlayerChar, which is what is primarily used for referencing the actual player.

The QuestGiver, which was defined in a blueprint class BP_QuestGiver, extends the class BP_HumanoidDialogueNpc. The difference being the QuestGenerator component that will be presented...
in section 4.1. A full view of the hierarchy of character classes can be seen in Figure 2.12.

2.6.3.3 DevKit Problems

There is one big problem while working with the DevKit: its steep learning curve. Users with no experience with Unreal Engine and its visual scripting language can become quite lost. The game and the DevKit are both very recent, there aren’t many tutorials or information available, so a user might not know where to start. There is channel on Discord dedicated to the DekKit, but most modders there are only interested in making aesthetic changes. So most of them can’t help a user wanting to experiment with game-play mechanics. Having an Unreal background helps a lot.

Also, the game is still being developed. There are still some visible bugs and bug fixes. And there are constant updates almost every week. If the user happens to be editing an asset that was recently updated, he/she will lose all its changes. Especially with DataTables. Also the DevKit is extremely heavy in terms of memory usage and disk space and requires a good amount of processing. Users with average computers might have trouble running the DevKit.

The DevKit being restricted to visual scripting language only, makes implementing complex functions extremely complicating. Since every thing is a node, some functions become to big and confusing, making it extremely difficult to manage a user script. Additionally, the user is stuck with the language limitations, for example in the types of structures that are available.

Luckily, we have signed an agreement with Funcom, and they have shared an early version of their project, called ConanSandbox. This allows us to use Unreal Engine to its full extent, by also giving us access to C++ scripting. There were also some problems that came with using this project, but these will be discussed later in this dissertation.

2.7 Conclusion

Having analysed Conan Exiles and its DevKit and despite the limitations, we can conclude that Conan Exiles is the most suitable game to implement our system on. Having access to both C++ scripting and Blueprint scripting, allows us to implement a system without much restrictions. Instead of building a completely new world with new characters, items and having to implement game mechanics from scratch, we simply have to add a component to a character. This component contains the implementation of an algorithm that is capable of offering a plot-line to the player. We must now choose an architecture of those studied. It is fundamental that the architecture used can be easily adapted to the game.

Since our focus is strongly directed to stories in video games, we have chosen an approach that is based on telling a story through a series of generated quests. So works like those of Doran and Parberry [8], CONAN [4] and GAME FORGE [17] are more in-line with what we are looking to develop. We want to focus in particular on the prototype quest generator from Doran and Parberry, since here stories, or rather quests aren’t pre-authored and instead emerge from characters motivations.

However, we believe that the quests used in this approach are too simple and don’t offer very compelling stories to the player. We have made some extensions to their grammar, that will be discussed
in the next chapter. Additionally, their grammar was implemented using Prolog, we will now have to think of a way to reproduce this using C++ and Blueprints.

As we have stated before, Doran and Parberry's might be viewed as using a centralized architecture, since they only describe a single generator that creates complete quests by choosing every action and every goal and every object included in it. However, we believe that their generator has the potential to be used both in a distributed architecture and a mixed one.

Based on the analysis made to the works here detailed, we believe that in order for the system to be considered distributed: the algorithm must be shared by several NPCs (Quest Givers) and the generation of the sub-quests included in a quest, should be generated by another NPC with its own motivations. In this architecture each Quest Giver has access to its own grammar.

In order for the system to be considered a mixed interactive storytelling system, the generator must also be distributed through specific characters in the game world and each NPC also has its own grammar. The difference here would be that, the NPC that is generating the current quest must share the depth of the tree with the NPC in charge of generating the sub-quest.
Chapter 3

Quest Generation Model

Our goal is to implement a system that procedurally generates interactive stories as a series of quests in Conan Exiles. With this we hope to increase variability of content and reduce authoring effort. In our model the task of generating quests will be distributed between a set of NPCs that will have a generator as a component. Each NPC will generate a quest that satisfies a motivation strategy. During quest generation, details from the actions will be filled with characters, items, and locations using the Conan Exiles’s world database, until a concrete quest is generated. NPCs must be capable of communicating with other NPCs, in order to generate sub-quests with their own motivations (see Figure 3.1). After a quest is created it will be added to a set of available quests for the player to choose from.

In this chapter we will present our extended version of the grammar defined by Doran and Parberry, as well as our model of a quest generator that uses this extended grammar to generate quest. We will also present a model for the architecture defined in the previous chapter. It was adapted for Unreal Engine in order to be used in the sandbox game Conan Exiles. A whole new model had to be defined,
since the generator defined by Doran and Parberry was written in Prolog and Unreal Engine uses C++ and Blueprints.

3.1 Quest Structure

Quests generated using the structure described by Doran and Parberry on their prototype generator, are solely based on MMORPGs. We believe that these quests are too simple and might not offer very compelling stories, since we see MMORPG quests as being very similar to side-quests. According to their structural analysis, human authored quests possess a shared structure. Having this in mind, we tried to analyse quests from a single player RPG. Single player RPGs tend to have a strong focus on the story component of the game. Using the rules defined by Doran and Parberry, we chose to analyse quests from The Witcher 3 - The Wild Hunt, since it is a contemporary game with a complex structure of quests and was received with critical acclaim.

3.1.1 The Witcher 3 Synopsis

The Witcher 3: The Wild Hunt [24] is a 2015 single player action role-playing video game developed by CD Projekt RED and published by CD Projekt. The game received critical acclaim upon release, with praise directed primarily toward its gameplay, narrative, world design and visuals. It became the most awarded game of 2015, receiving numerous Game of the Year awards from gaming publications, critics, and award events, and is regarded by many to be one of the greatest games of all time.

In the game, players control the main character Geralt of Rivia, who is a Witcher. Witchers are monster hunters trained since childhood in combat, tracking, and magic, and transformed by mutagens to be stronger, faster, and resistant to toxins. Geralt is aided by several characters: the powerful sorceresses Yennefer of Vengerberg, his lover; a former love interest Triss Merigold; the bard Dandelion; the dwarven warrior Zoltan Chivay; and Geralt's Witcher mentor Vesemir. Geralt is spurred into action by the reappearance of his and Yennefer's adopted daughter Ciri.

Figure 3.2: The Witcher 3 game-play[24].
Ciri is someone born with innate and potentially vast magical abilities. Following the apparent death of her parents, she was trained as a Witcher by Geralt and guided in magic by Yennefer. She is unaware that the Emperor is her biological father, having sired her under an assumed name while subject to a curse. Ciri had disappeared years ago to escape the Wild Hunt, a group of spectral warriors led by the King of the Wild Hunt, who hail from a parallel dimension. They are determined to capture her and use her powers.

Geralt seeks to find Ciri and help her against the Wild Hunt. Throughout the game, players battle against the world’s many dangers using weapons and magic. They can interact with various non-player characters, and complete main story quests and side quests to acquire experience points and gold used to increase Geralt’s various abilities and gear. The game’s central story features multiple endings that are determined by Geralt’s choices made by the player during certain points of the story [37].

3.1.2 Structural Analysis

For the structural analysis, 58 main story quests from “The Witcher 3 - The Wild Hunt” (Witcher) were examined, to determine whether they also shared the structure extracted by Doran and Parberry from their own analysis. Quest descriptions were obtained from written and video walkthroughs. Quest descriptions consisted of the sequence of actions the player had to perform, NPC dialogue and causal and temporal relationships between the different quests.

Using the descriptions obtained and the resulting quest structure from the analysis described by Doran and Parberry, we undertook an attempt to represent Witcher quests. Alas, representing these main quests proved to be more difficult than expected. Indeed, the NPCs that gave out the quests, shared the same or similar motivations (previously described), but since the action rules Doran and Parberry defined had some limitations, it was impossible to fully describe Witcher quests.

In order to come up with the necessary changes to counter these limitations (both described in the upcoming paragraphs), each quest was analysed in the following way: first, it was necessary to record every player action in the respective quest; second, a tree was built for each quest, in the manner described in the related work section, using the rules defined by Doran and Parberry, while adding the changes required to represent each quest; third, the changes made were extracted and added to the new set of rules, the results of which can be seen in Tables 3.1, 3.2 and 3.3. Changes made are highlighted in bold. The sequence of actions and rules are written in Backus Naur Form, a notation technique for grammars.

When comparing with the structure of Doran and Parberry, the first change worthy of noting, regards the last action of a quest’s strategy, specifically “give” and “report” actions. During our analysis, we observed that some Witcher quests required the player to give or report something, as a last step before completing a quest. But some of the strategies that they defined don’t allow this. The opposite also happened, where strategies defined in their work, have a last action give/report, but in the game the player isn’t required to do any of those.

To give a few examples: in the short main story quest “Disturbance”, the player must explore the
castle, to find and remove an object ("repair"), that is messing with one of Yennefer’s spells. After its removal, the player is required to report back to Yennefer. This sequence isn’t fully represented in their structure, because it is missing the report action. A second example would be the main story quest "The Sunstone", which requires the player to find and gather the lost item Sunstone. Using Doran and Parberry’s structure, the player would be obliged to give the Sunstone after it was gathered. However, this quest finishes as soon as the Sunstone is acquired.

This was a recurrent problem, so to counter this obstacle, we decided to put in almost every strategy, one of two action rules, namely a <give> or a <report>. This way, it is now possible to decide during generation, whether a quest's final action should require the player to either report a quest's completion, to give an item back, or neither.

The second change resides in the <goto> set of rules (see Table 3.2). According to Doran and Parberry’s rules, it wasn’t possible to be given a sub-quest without having to learn something (see Table 3.2, rule 9 and rules 12-14), which is something that happened often in The Witcher 3. So, it was necessary to add a <prepare> rule, that offered this possibility (see Table 3.2, rules 10 and 16). Also, the rule that required learning (rule 10 from Table 3.2) no longer has a mandatory "goto" action, which limited the order in which certain events could occur. Now, the new <goto> action rule can be expanded to offer more possibilities to the quest. It was also added the action rule <rescue> (see Table 3.2, rules 33-36), to allow the player to simply free a character, and not having to escort it every single time.

The action rule <defeat> (see Table 3.2, rules 27 and 28) was added, so it could be possible to decide during the quest generation whether a strategy would require the player to either kill or merely damage an enemy. Before, strategies had one of these actions imposed, which conflicted with the representation of some of the Witcher quests, that had one of those strategies.

In some Witcher quests, where it was only required for the player to damage someone or something, the rules defined by Doran and Parberry would instead have the player kill someone or something, and vice-versa. An example of this is the quest "Bald Mountain". After the death of their mentor Vesemir, Geralt (the player's character) and Ciri bloody for vengeance, track down Vesemir's killer Imlerith and ultimately kill him. Using Doran and Parberry’s set, it wouldn't be possible to represent this. In their set, the strategy "Revenge, Justice" from the "Serenity" motivation has a mandatory "damage" atomic action, while what we seek is a "kill" atomic action.

Finally, for the actions (see Table 3.3), we added four new atomic actions that can be performed by the player. During the analysis, we encountered some actions that were not represented in their set of actions. The most significant ones being "examine" and "follow". Doran and Parberry’s rules only considered learning either through listening to a character, after doing a sub-quest for said character, or by reading a book.
### Table 3.1: Strategies for each NPC’s motivation.

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Strategy</th>
<th>Sequence of Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge</td>
<td>Deliver item for study&lt;br&gt;Spy&lt;br&gt;Interview NPC&lt;br&gt;Use item on field</td>
<td>&lt;get&gt; &lt;give&gt; &lt;goto&gt; spy &lt;report&gt; &lt;goto&gt; listen &lt;report&gt; &lt;get&gt; &lt;goto&gt; use &lt;give&gt;</td>
</tr>
<tr>
<td>Comfort</td>
<td>Obtain luxuries&lt;br&gt;Kill pests</td>
<td>&lt;get&gt; &lt;give&gt; &lt;goto&gt; &lt;defeat&gt; &lt;report&gt;</td>
</tr>
<tr>
<td>Reputation</td>
<td>Obtain rare items&lt;br&gt;Kill enemies&lt;br&gt;Visit dangerous place</td>
<td>&lt;get&gt; &lt;give&gt; &lt;goto&gt; &lt;defeat&gt; &lt;report&gt; &lt;goto&gt; &lt;report&gt;</td>
</tr>
<tr>
<td>Serenity</td>
<td>Revenge, Justice&lt;br&gt;Capture Criminal&lt;br&gt;Check on NPC (1)&lt;br&gt;Check on NPC (2)&lt;br&gt;Recover lost/stolen item&lt;br&gt;Rescue NPC</td>
<td>&lt;goto&gt; &lt;defeat&gt; &lt;report&gt; &lt;goto&gt; &lt;capture&gt; &lt;report&gt; &lt;goto&gt; listen &lt;report&gt; &lt;goto&gt; take &lt;give&gt; &lt;get&gt; &lt;give&gt; &lt;goto&gt; &lt;rescue&gt; &lt;report&gt;</td>
</tr>
<tr>
<td>Protection</td>
<td>Attack threatening entities&lt;br&gt;Capture Criminal&lt;br&gt;Treat or Repair (1)&lt;br&gt;Treat or Repair (2)&lt;br&gt;Create Diversion (1)&lt;br&gt;Create Diversion (2)&lt;br&gt;Assemble fortification&lt;br&gt;Guard entity&lt;br&gt;Recruit</td>
<td>&lt;goto&gt; &lt;defeat&gt; &lt;report&gt; &lt;goto&gt; &lt;capture&gt; &lt;report&gt; &lt;goto&gt; &lt;report&gt; &lt;get&gt; &lt;goto&gt; repair &lt;report&gt; &lt;get&gt; &lt;goto&gt; use &lt;report&gt; &lt;goto&gt; repair &lt;report&gt; &lt;get&gt; &lt;goto&gt; use &lt;report&gt; &lt;goto&gt; damage &lt;report&gt; &lt;goto&gt; repair &lt;report&gt; &lt;goto&gt; defend &lt;report&gt; &lt;goto&gt; listen &lt;report&gt;</td>
</tr>
<tr>
<td>Conquest</td>
<td>Attack enemy&lt;br&gt;Steal stuff&lt;br&gt;Recruit</td>
<td>&lt;goto&gt; &lt;defeat&gt; &lt;report&gt; &lt;goto&gt; &lt;steal&gt; &lt;give&gt; &lt;goto&gt; listen &lt;report&gt;</td>
</tr>
<tr>
<td>Wealth</td>
<td>Gather raw materials&lt;br&gt;Steal valuables for resale&lt;br&gt;Make valuables for resale</td>
<td>&lt;goto&gt; &lt;get&gt; &lt;report&gt; &lt;goto&gt; &lt;steal&gt; &lt;give&gt; &lt;goto&gt; repair &lt;give&gt;</td>
</tr>
<tr>
<td>Ability</td>
<td>Assemble tool for new skill&lt;br&gt;Obtain training materials&lt;br&gt;Use existing tools&lt;br&gt;Train combat&lt;br&gt;Practice skill&lt;br&gt;Research skill (1)&lt;br&gt;Research skill (2)</td>
<td>&lt;goto&gt; repair use &lt;get&gt; use &lt;goto&gt; use &lt;goto&gt; damage &lt;goto&gt; use &lt;get&gt; use &lt;report&gt; &lt;get&gt; experiment &lt;report&gt;</td>
</tr>
<tr>
<td>Equipment</td>
<td>Assemble&lt;br&gt;Deliver supplies&lt;br&gt;Steal supplies&lt;br&gt;Trade for supplies</td>
<td>&lt;goto&gt; repair &lt;give&gt; &lt;get&gt; &lt;give&gt; &lt;steal&gt; &lt;give&gt; &lt;get&gt; &lt;goto&gt; exchange</td>
</tr>
</tbody>
</table>

After analysing Witcher quests it was clear that one can also learn by examining clues or objects. One example is the quest “Novigrad Dreaming”, where a ghost leaves drawings that, after being examined, show the player what he/she must do next. The action “follow” also appears a lot, practically in every quest. While following a character, the player can learn a bit of the character’s back-story and of the possible relationship with the main character Geralt.

Actions “wait” and “free” were also added, the first rarely appearing, while the second is an alternative to simply escorting a character, after being rescued. This action appeared more often than the “escort” action in the Witcher’s main story quests. A good example for the use of the action “free”, is the
<table>
<thead>
<tr>
<th>#</th>
<th>Rules</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td><code>&lt;Quest&gt;</code> ::= <code>&lt;Knowledge&gt;</code></td>
<td>This is the root of a quest, which expands into one of the 9 motivations. Which will eventually be expanded into one of the strategies, specific to said motivation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td><code>&lt;subquest&gt;</code> ::= ε</td>
<td>There is nothing to do.</td>
</tr>
<tr>
<td>2.</td>
<td><code>&lt;subquest&gt;</code> ::= <code>&lt;Quest&gt;</code> <code>&lt;goto&gt;</code></td>
<td>Go perform a quest an return.</td>
</tr>
<tr>
<td>3.</td>
<td><code>&lt;goto&gt;</code> ::= ε</td>
<td>You are already there.</td>
</tr>
<tr>
<td>4.</td>
<td><code>&lt;goto&gt;</code> ::= <code>goto</code></td>
<td>Go to a known location.</td>
</tr>
<tr>
<td>5.</td>
<td><code>&lt;goto&gt;</code> ::= <code>wait</code></td>
<td>Wait at a location for someone or something.</td>
</tr>
<tr>
<td>6.</td>
<td><code>&lt;goto&gt;</code> ::= <code>explore</code></td>
<td>Just wander around and look.</td>
</tr>
<tr>
<td>7.</td>
<td><code>&lt;goto&gt;</code> ::= <code>&lt;learn&gt;</code> <code>&lt;goto&gt;</code></td>
<td>Follow somebody or something.</td>
</tr>
<tr>
<td>8.</td>
<td><code>&lt;goto&gt;</code> ::= <code>&lt;stealth&gt;</code></td>
<td>Sneak by someone.</td>
</tr>
<tr>
<td>9.</td>
<td><code>&lt;goto&gt;</code> ::= <code>&lt;prepare&gt;</code> <code>&lt;goto&gt;</code></td>
<td>Find out where to go and go there.</td>
</tr>
<tr>
<td>10.</td>
<td><code>&lt;goto&gt;</code> ::= <code>&lt;learn&gt;</code> <code>&lt;goto&gt;</code></td>
<td>Prepare before going somewhere.</td>
</tr>
<tr>
<td>11.</td>
<td><code>&lt;learn&gt;</code> ::= ε</td>
<td>You already know it.</td>
</tr>
<tr>
<td>12.</td>
<td><code>&lt;learn&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;subquest&gt;</code> <code>listen</code></td>
<td>Go someplace, perform a subquest, get info from NPC.</td>
</tr>
<tr>
<td>13.</td>
<td><code>&lt;learn&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;get&gt;</code> <code>read</code></td>
<td>Go someplace, get something and read it.</td>
</tr>
<tr>
<td>14.</td>
<td><code>&lt;learn&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;subquest&gt;</code> <code>examine</code></td>
<td>Go someplace, perform a subquest, examine something.</td>
</tr>
<tr>
<td>15.</td>
<td><code>&lt;prepare&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;subquest&gt;</code></td>
<td>Go someplace and perform a subquest.</td>
</tr>
<tr>
<td>16.</td>
<td><code>&lt;get&gt;</code> ::= ε</td>
<td>You already have it.</td>
</tr>
<tr>
<td>17.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;steal&gt;</code></td>
<td>Steal it from somebody.</td>
</tr>
<tr>
<td>18.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;goto&gt;</code> <code>gather</code></td>
<td>Go someplace and pick something up that’s lying around.</td>
</tr>
<tr>
<td>19.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;goto&gt;</code> <code>take</code></td>
<td>Go someplace and take something.</td>
</tr>
<tr>
<td>20.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;get&gt;</code> <code>&lt;goto&gt;</code> <code>exchange</code></td>
<td>Get something, go to someonee and exchange.</td>
</tr>
<tr>
<td>21.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;get&gt;</code> <code>&lt;subquest&gt;</code></td>
<td>Get something, perform a subquest.</td>
</tr>
<tr>
<td>22.</td>
<td><code>&lt;steal&gt;</code> ::= <code>&lt;goto&gt;</code> <code>stealth</code> <code>take</code></td>
<td>Go someplace, sneak on somebody and take something.</td>
</tr>
<tr>
<td>23.</td>
<td><code>&lt;steal&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;defeat&gt;</code> <code>take</code></td>
<td>Go someplace, defeat somebody and take something.</td>
</tr>
<tr>
<td>24.</td>
<td><code>&lt;capture&gt;</code> ::= <code>&lt;goto&gt;</code> <code>use capture</code></td>
<td>Go someplace, use something to capture somebody.</td>
</tr>
<tr>
<td>25.</td>
<td><code>&lt;capture&gt;</code> ::= <code>&lt;goto&gt;</code> <code>damage capture</code></td>
<td>Go someplace, damage to capture somebody.</td>
</tr>
<tr>
<td>26.</td>
<td><code>&lt;capture&gt;</code> ::= <code>&lt;goto&gt;</code> <code>capture</code></td>
<td>Go someplace and capture somebody.</td>
</tr>
<tr>
<td>27.</td>
<td><code>&lt;defeat&gt;</code> ::= <code>&lt;goto&gt;</code> <code>damage</code></td>
<td>Go someplace and damage somebody.</td>
</tr>
<tr>
<td>28.</td>
<td><code>&lt;defeat&gt;</code> ::= <code>&lt;goto&gt;</code> <code>kill</code></td>
<td>Go someplace and kill somebody.</td>
</tr>
<tr>
<td>29.</td>
<td><code>&lt;report&gt;</code> ::= ε</td>
<td>There is nothing to report.</td>
</tr>
<tr>
<td>30.</td>
<td><code>&lt;report&gt;</code> ::= <code>&lt;goto&gt;</code> <code>report</code></td>
<td>Go someplace and report to somebody.</td>
</tr>
<tr>
<td>31.</td>
<td><code>&lt;give&gt;</code> ::= ε</td>
<td>There is nothing to give.</td>
</tr>
<tr>
<td>32.</td>
<td><code>&lt;give&gt;</code> ::= <code>&lt;goto&gt;</code> <code>give</code></td>
<td>Go someplace and give something to somebody.</td>
</tr>
<tr>
<td>33.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>&lt;free&gt;</code></td>
<td>Free somebody from imprisonment.</td>
</tr>
<tr>
<td>34.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>&lt;defeat&gt;</code> <code>free</code></td>
<td>Defeat somebody and free somebody from imprisonment.</td>
</tr>
<tr>
<td>35.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>&lt;escort&gt;</code></td>
<td>Escort somebody to someplace.</td>
</tr>
<tr>
<td>36.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>&lt;defeat&gt;</code> <code>escort</code></td>
<td>Defeat somebody and escort a different somebody to someplace.</td>
</tr>
</tbody>
</table>

Table 3.2: Action rules in BNF (Backus Normal Form).

quest "A Poet Under Pressure". Here the player must rescue the Halfling Dandelion. After following a Witch Hunter that fled from a failed ambush, the player enters the house where Dandelion is being held captive. After you defeat the Witch Hunter, the player can free Dandelion and report to Irina. This quest is also a good example for the addition of the action rule `<defeat>`. In Doran and Parberry’s strategies, the "Rescue NPC" strategy (see Table 3.1, motivation Serenity) requires the player to "damage" the captor, whilst in this quest the player is required to "kill" the captor.

In order to validate the changes made to the grammar defined by Doran and Parberry, we have
<table>
<thead>
<tr>
<th>#</th>
<th>Action</th>
<th>Pre-condition</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ε</td>
<td>None.</td>
<td>None.</td>
</tr>
<tr>
<td>2</td>
<td>capture</td>
<td>Somebody is there.</td>
<td>They are your prisoner.</td>
</tr>
<tr>
<td>3</td>
<td>damage</td>
<td>Somebody or something is there.</td>
<td>It is more damaged.</td>
</tr>
<tr>
<td>4</td>
<td>defend</td>
<td>Somebody or something is there.</td>
<td>Attempts to damage it have failed.</td>
</tr>
<tr>
<td>5</td>
<td>escort</td>
<td>Somebody is there.</td>
<td>They will now accompany you.</td>
</tr>
<tr>
<td>6</td>
<td>examine</td>
<td>Somebody or something is there.</td>
<td>You have information about it.</td>
</tr>
<tr>
<td>7</td>
<td>exchange</td>
<td>Somebody is there, they and you have some-</td>
<td>You have theirs, they have yours.</td>
</tr>
<tr>
<td>8</td>
<td>experiment</td>
<td>Something is there.</td>
<td>Perhaps you have learned what it is for.</td>
</tr>
<tr>
<td>9</td>
<td>explore</td>
<td>None.</td>
<td>Wander around at random.</td>
</tr>
<tr>
<td>10</td>
<td>follow</td>
<td>Somebody or something is there.</td>
<td>You will now accompany them.</td>
</tr>
<tr>
<td>11</td>
<td>free</td>
<td>Somebody is there.</td>
<td>They are no longer prisoner.</td>
</tr>
<tr>
<td>12</td>
<td>gather</td>
<td>Something is there.</td>
<td>You have it.</td>
</tr>
<tr>
<td>13</td>
<td>give</td>
<td>Somebody is there, you have something.</td>
<td>They have it, you don’t.</td>
</tr>
<tr>
<td>14</td>
<td>goto</td>
<td>You know where to go and how to get there.</td>
<td>You are there.</td>
</tr>
<tr>
<td>15</td>
<td>kill</td>
<td>Somebody is there.</td>
<td>They are dead.</td>
</tr>
<tr>
<td>16</td>
<td>listen</td>
<td>Somebody is there.</td>
<td>You have some of their information.</td>
</tr>
<tr>
<td>17</td>
<td>read</td>
<td>Something is there.</td>
<td>You have information from it.</td>
</tr>
<tr>
<td>18</td>
<td>repair</td>
<td>Something is there.</td>
<td>It is fixed, built or resolved.</td>
</tr>
<tr>
<td>19</td>
<td>report</td>
<td>Somebody is there.</td>
<td>They have information you have.</td>
</tr>
<tr>
<td>20</td>
<td>spy</td>
<td>Somebody or something is there.</td>
<td>You have information from it.</td>
</tr>
<tr>
<td>21</td>
<td>stealth</td>
<td>Somebody is there.</td>
<td>Sneak up on them.</td>
</tr>
<tr>
<td>22</td>
<td>take</td>
<td>Somebody is there, they have something.</td>
<td>You have it, they don’t.</td>
</tr>
<tr>
<td>23</td>
<td>use</td>
<td>Somebody or something is there.</td>
<td>It has affected them.</td>
</tr>
<tr>
<td>24</td>
<td>wait</td>
<td>None.</td>
<td>Wait for something to happen.</td>
</tr>
</tbody>
</table>

Table 3.3: Atomic actions.

...
- A set of Strategies.

Nodes are divided into two smaller sets:

- A set of Motivation nodes, these represent the root of each quest, which reflect a NPC’s intentions. There are 9 in total;

- A set of Action nodes.

Actions are further split into two other sets:

- A set of NonTerminal nodes, which have a total of 12;

- A set of Terminal nodes, which have a total of 24. These represent actions the player performs in the game.

Additionally, every node in the Action set contains a set of parameters. These parameters reference actual game objects that take part in the action.

Each rule in Rules is represented as: head → body. The head of the rule must specifically be a node in NonTerminal. The body of the rule, is a sequence of nodes from Action. The body is added to the quest structure, every time the rule is selected.

Each strategy in Strategies is also represented as: head → body. But here the head of the rule must be a Motivation node. The body of the rule, is again a sequence of nodes from Action.

The knowledge base contains everything an NPC knows and everything that is relevant to the NPC. The knowledge base can be defined as having 4 different sets:

- A set of Characters;

- A set of Items;

- A set of Enemies;

- A set of Locations.

These sets are used to fill the parameters of the actions defined in the grammar. Using the grammar and the information in the knowledge base, the quest giver will be able to create a quest with actions like: kill(Hyena), goto(Cave) and learn(Recipe).

3.2.2 Algorithm

The algorithm, present here, is capable of understanding the grammar previously described and use it to create different quests. Additionally, it must have access to a sub-set of world objects, in order to fill parameters associated with the actions defined in the grammar. In this sub-section, we explain our reasoning about what we believe such an algorithm should be like, having in mind that the grammar was originally conceived to be used with Prolog.
3.2.2.1 Creating a Quest

Given the initial input, the quest giver will call CreateQuest the algorithm responsible for starting the generation process. This algorithm is responsible for creating an instance of a quest and assigning a root node to it. This root node is the quest giver’s motivation. This motivation can be randomly generated, can be previously predefined at game start, or can even be given by the game environment. For example: A Conan Exiles NPC, might get attacked by a group of hostile exiles. So he will want to seek the player’s protection and ask the player to defeat them. After assigning the root, its time to begin the expansion phase (see Algorithm 1).

Algorithm 1 Create Quest

```
function CREATEQUEST(Grammar, KB)
    quest ← NEWOBJECT<QUEST>();
    quest.Root ← QuestGiver.Motivation;
    EXPAND(quest.Root);
    return quest;
end function
```

3.2.2.2 Expanding Nodes

The second algorithm is called Expand(Node). It is a recursive algorithm, that will be called for every node that is generated and consequently added to the quest structure. Depending on the type of the selected node (NonTerminal, Terminal or Motivation), the algorithm will act in the following manner:

**Terminal** If the current node is a terminal, this means we have reached a leaf of the tree. Terminal nodes can not be expanded. Instead, they will be added to a list of leaf nodes in the quest, since this represent the actual actions the player has to perform. These are sorted by the order in which they are selected for expansion.

**NonTerminal** If it is a NonTerminal node, its children will be selected from Rules Table. The name of the node serve as index value, to select one of the bodies associated with the matching head. The body that is to be generated, must be selected stochastically, as a mean to assure a certain variety in the quests generated. For example: we can give each rule’s body a weight ranging from $[0, 1]$, and then generate a random value to be compared against these weights.

**Motivation** If it is a Motivation node, its children will be selected from Strategies Table. The strategy can be selected in the same way as the initial motivation, either by randomly generated, by having it predefined or given by the game environment.

After selecting the rule/strategy, a node is generated for every action in it, and their parameters are bound to world objects (see Algorithm 2).

Since we are trying to model a distributed system, the sub-quests that take part in the quest being created, must be generated by other NPCs. These must have their own grammar, knowledge base
Algorithm 2 Expand Node

function \textsc{Expand}(node)
    if node \in T then
        quest.Steps.ADD(node);
    else
        if node \in \textit{NonTerminal} then
            if node.isSubquest then
                \textsc{KB.GETRANDOMQUESTGIVER.CREATEQUEST};
            else
                rules \leftarrow \textsc{Rules}(node);
                index = \textsc{WEIGHTEDCHOICE}(weights);
                sequence = \textsc{R}(node, index);
                weights = \textsc{UPDATEWEIGHTS}(rules, weights, index, depth);
            end if
        else if node \in \textit{Motivation} then
            strategies \leftarrow \textsc{Strategies}(node);
            index = \textsc{SELECTSTRATEGY}(strategies);
            sequence = \textsc{Strategies}(node, index);
        end if
    end if
    for all action \in sequence do
        node.Children.ADD(action);
    end for
    \textsc{BINDPARAMETERS}(node);
    for all child \in node.Children do
        \textsc{EXPAND}(child);
    end for
end function

and motivations. During node expansion, if the \textit{NonTerminal} being evaluated is of type \textit{Subquest} (see Table 3.2), we must select another NPC that has a quest generator component and call the algorithm \textit{CreateQuest} on it. The subquest generated should then be added to the main quest.

3.2.2.3 Weighted Choice

The grammar described previously in this chapter was further extended to include weights related to the each sequence of actions in a \textit{NonTerminal} node’s set. Having weights gives control over the variety of actions sequences within the quest.

The algorithm \textit{WeightedChoice} (see Algorithm 3) was developed for the sole purpose of generating an index number based on a range of weights. This index is used to select a sequence of actions from the set of rules of the \textit{NonTerminal} node being expanded. To obtain the index number, first a random number \textit{rand} (from 0.0 to 1.0) must be generated and then compared to a set of ranges. This set of ranges are based on the weights shown in Table 3.4. The sum of all weights related to the set of rules of a \textit{NonTerminal} node must always add up to 1. The range in which the randomly generated number falls in to, will determine the index corresponding to a sequence of actions in the node’s set of rules.

The values for the weights presented in Table 3.4, were determined based on the frequency that these sequences appeared, obtained by mere observation of the drawn Witcher quest. Perhaps, a future
Table 3.4: Initial set of weights that would be used by the generator, if it had the complete grammar.

<table>
<thead>
<tr>
<th>#</th>
<th>Rules</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>&lt;subquest&gt;</code> ::= <code>ϵ</code></td>
<td>0.3</td>
</tr>
<tr>
<td>2.</td>
<td><code>&lt;subquest&gt;</code> ::= <code>&lt;Quest&gt;</code> <code>&lt;goto&gt;</code></td>
<td>0.7</td>
</tr>
<tr>
<td>3.</td>
<td><code>&lt;goto&gt;</code> ::= <code>ϵ</code></td>
<td>0.0</td>
</tr>
<tr>
<td>4.</td>
<td><code>&lt;goto&gt;</code> ::= <code>goto</code></td>
<td>0.15</td>
</tr>
<tr>
<td>5.</td>
<td><code>&lt;goto&gt;</code> ::= <code>wait</code></td>
<td>-</td>
</tr>
<tr>
<td>6.</td>
<td><code>&lt;goto&gt;</code> ::= <code>explore</code></td>
<td>0.1</td>
</tr>
<tr>
<td>7.</td>
<td><code>&lt;goto&gt;</code> ::= <code>follow</code></td>
<td>0.1</td>
</tr>
<tr>
<td>8.</td>
<td><code>&lt;goto&gt;</code> ::= <code>stealth</code></td>
<td>-</td>
</tr>
<tr>
<td>9.</td>
<td><code>&lt;goto&gt;</code> ::= <code>&lt;learn&gt;</code> <code>&lt;goto&gt;</code></td>
<td>0.3</td>
</tr>
<tr>
<td>10.</td>
<td><code>&lt;goto&gt;</code> ::= <code>&lt;prepare&gt;</code> <code>&lt;goto&gt;</code></td>
<td>0.35</td>
</tr>
<tr>
<td>11.</td>
<td><code>&lt;learn&gt;</code> ::= <code>ϵ</code></td>
<td>0.0</td>
</tr>
<tr>
<td>12.</td>
<td><code>&lt;learn&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;subquest&gt;</code> <code>listen</code></td>
<td>0.4</td>
</tr>
<tr>
<td>13.</td>
<td><code>&lt;learn&gt;</code> ::= <code>&lt;get&gt;</code> <code>read</code></td>
<td>0.3</td>
</tr>
<tr>
<td>14.</td>
<td><code>&lt;learn&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;subquest&gt;</code> <code>examine</code></td>
<td>0.3</td>
</tr>
<tr>
<td>15.</td>
<td><code>&lt;prepare&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;subquest&gt;</code></td>
<td>1.0</td>
</tr>
<tr>
<td>16.</td>
<td><code>&lt;get&gt;</code> ::= <code>ϵ</code></td>
<td>0.0</td>
</tr>
<tr>
<td>17.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;steal&gt;</code></td>
<td>0.2</td>
</tr>
<tr>
<td>18.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;goto&gt;</code> <code>gather</code></td>
<td>0.35</td>
</tr>
<tr>
<td>19.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;goto&gt;</code> <code>take</code></td>
<td>0.35</td>
</tr>
<tr>
<td>20.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;get&gt;</code> <code>&lt;goto&gt;</code> <code>exchange</code></td>
<td>0.05</td>
</tr>
<tr>
<td>21.</td>
<td><code>&lt;get&gt;</code> ::= <code>&lt;get&gt;</code> <code>&lt;subquest&gt;</code></td>
<td>0.05</td>
</tr>
<tr>
<td>22.</td>
<td><code>&lt;steal&gt;</code> ::= <code>&lt;goto&gt;</code> <code>stealth</code> <code>take</code></td>
<td>0.6</td>
</tr>
<tr>
<td>23.</td>
<td><code>&lt;steal&gt;</code> ::= <code>&lt;goto&gt;</code> <code>&lt;defeat&gt;</code> <code>take</code></td>
<td>0.4</td>
</tr>
<tr>
<td>24.</td>
<td><code>&lt;capture&gt;</code> ::= <code>&lt;goto&gt;</code> <code>use</code> <code>capture</code></td>
<td>0.3</td>
</tr>
<tr>
<td>25.</td>
<td><code>&lt;capture&gt;</code> ::= <code>&lt;goto&gt;</code> <code>damage</code> <code>capture</code></td>
<td>0.3</td>
</tr>
<tr>
<td>26.</td>
<td><code>&lt;capture&gt;</code> ::= <code>&lt;goto&gt;</code> <code>capture</code></td>
<td>0.4</td>
</tr>
<tr>
<td>27.</td>
<td><code>&lt;defeat&gt;</code> ::= <code>&lt;goto&gt;</code> <code>damage</code></td>
<td>0.3</td>
</tr>
<tr>
<td>28.</td>
<td><code>&lt;defeat&gt;</code> ::= <code>&lt;goto&gt;</code> <code>kill</code></td>
<td>0.7</td>
</tr>
<tr>
<td>29.</td>
<td><code>&lt;report&gt;</code> ::= <code>ϵ</code></td>
<td>0.1</td>
</tr>
<tr>
<td>30.</td>
<td><code>&lt;report&gt;</code> ::= <code>&lt;goto&gt;</code> <code>report</code></td>
<td>0.9</td>
</tr>
<tr>
<td>31.</td>
<td><code>&lt;give&gt;</code> ::= <code>ϵ</code></td>
<td>0.1</td>
</tr>
<tr>
<td>32.</td>
<td><code>&lt;give&gt;</code> ::= <code>&lt;goto&gt;</code> <code>give</code></td>
<td>0.9</td>
</tr>
<tr>
<td>33.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>free</code></td>
<td>0.15</td>
</tr>
<tr>
<td>34.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>&lt;defeat&gt;</code> <code>free</code></td>
<td>0.4</td>
</tr>
<tr>
<td>35.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>escort</code></td>
<td>0.15</td>
</tr>
<tr>
<td>36.</td>
<td><code>&lt;rescue&gt;</code> ::= <code>&lt;defeat&gt;</code> <code>escort</code></td>
<td>0.3</td>
</tr>
</tbody>
</table>

Algorithm 3 Weighted Choice

```plaintext
function WEIGHTEDCHOICE(weights)
    rand = RANDOM(0, 1);
    limit1 = 0;
    limit2 = 0;
    for all weight ∈ weights do
        limit2 = limit2 + weight;
        if limit1 =< rand <= limit2 then
            return weight.index;
        else
            limit1 = limit2;
        end if
    end for
end function
```

statistical study will present more precise values.
3.2.2.4 Update Weights

Once the index for selecting the sequence of actions is returned, the set of weights of the current NonTerminal being evaluated will be updated (see Algorithm 4). The algorithm UpdateWeights is given three parameters:

- The sub-set of rules, related to node being expanded;
- The sub-set of weights, associated with the rules;
- The index of the chosen sequence;
- The current depth of the tree.

This algorithm will first reduce the weight of the selected sequence. The new updated weight will be the result of the current weight being divided by the value of a converge function. The convergence function can be defined in several ways, it mostly depends on how big one wishes the quest to be. For example it could be either: $\sqrt{\text{depth}}$, for a slower convergence rate to 0; $2^{\text{depth}-1}$, for a moderate convergence rate; or $\text{depth}^2$, for a faster convergence rate. This function is the one that essentially controls the size of the quest.

But now with the weight of the chosen sequence updated, the some of all the weights isn’t equal to 1. So we want to update the rest of weights, so the sum of all amounts to 1 again. The missing portion will be divided between every other rule in the sub-set, that contains Terminal actions.

Using weights and updating them is important, because it gives us control over the rules that are selected, and hence the size of the quest. When the algorithm reaches a certain depth, we want it to avoid choosing rules that add more NonTerminal nodes to the quest structure. This is why we only update the weight if the sequence of actions isn’t mainly composed of NonTerminal actions. The most important case is when we are expanding <goto> nodes, because rules 9 and 10 in Table 3.4 are essentially gateways to add sub quests to the quest structure, consequently adding more actions to it. At a deeper depth, we would like to avoid this. In cases like <learn> and <subquest> we have the $\epsilon$ action, which its weight will always be incremented. This is will serve as a safety measure, in the case of a <learn> being added in a greater depth of the quest tree.

Algorithm 4 Update Weights

```
function UPDATE_WEIGHTS(rules, weights, index, depth)
    weights[index] = weights[index]/CONVERGENCE(depth);
    portion = (1 - weights[index])/NumberOfUpdates;
    for all weight ∈ weights do
        if rules[weight.index].CONTAINS TERMINALS then
            weight+ = portion;
        end if
    end for
    return weights;
end function
```
3.2.2.5 Binding Parameters

Having completed the process of creating nodes for each action in the chosen sequence of actions, we are left with binding the parameters of each of these nodes, before we can add them to the quest structure and call the \textit{Expand} algorithm on each of them. The third and final algorithm is \textit{BindParameters()}. This algorithm is in charge of assigning a world object to every action node recently generated.

This step is needed to ensure logical coherence between sequential actions. So once again, we have to extend the quest structure to include parameter restrictions on every action in a sequence of actions. Parameters can either be restricted by the parent node, the sibling node or not restricted at all, which in this case we use a object that hasn’t been referenced yet. This restrictions are encoded as flags 0 for parent, 1 for new and 2 for sibling restriction.

Algorithm 5 Bind Parameters

\begin{verbatim}
function BindParameters(node)
    for all child ∈ Children do
        for all param1 ∈ child.Parameters do
            if param1.Restriction == 0 then
                for all param2 ∈ node.Parameters do
                    if param1.Type == param2.Type then
                        param1.Target ← param2.Target;
                    end if
                end for
            end if
            else if parameters.Restriction == 1 then
                param1.Target ← NEWPARAMETER(param1.Type, K);
            end if
            else
                for all param2 ∈ node.Children[param1.SiblingIndex].Parameters do
                    if param1.Type == param2.Type then
                        param1.Target ← param2.Target;
                    end if
                end for
            end if
        end for
    end for
end function
\end{verbatim}

For example imagine that during expansion: the rule \textit{get → get goto exchange} was selected, the nodes corresponding to the sequence of actions in the rule’s body were generated and \textit{BindParameters} was called: the parameters of the action sequence should have the following relationships \textit{get(item1) → get(item2) goto(location1, npc1) exchange(item2, item1, npc1)}. We want to avoid getting sequences such as this \textit{get(item1) → get(item2) goto(location1, npc1) exchange(item3, item4, npc2)}, where there is no relation between actions.

So these correspondences between parameters must be established while every node in the quest is expanded. Algorithm 5 shows the pseudo-code for the whole binding parameters process. Each action parameter associated to an action should have the following properties predefined at the start of the game:

- The parameter type (NPC, Item, Location);
- The restriction flag (0, 1, 2);
The sibling reference, in case flag is equal to 2.

The algorithm will first verify the way the parameter is restricted. In case of the flag value being 1, it will have to get a reference to an object that is stored in the knowledge base. In the cases where it isn’t needed to bind the parameter to a new object (this means the parameter is restricted by either the parent or the sibling), BindParameters simply compares the parameters to see if the types match, being that the case it copies the values that are stored in the parent/sibling to the action parameter currently being evaluated. In case of the sibling it will use the stored sibling index, to get the values from the appropriate sibling action.

Once every leaf node has been reached and there are no more nodes to expand, the function CreateQuest will terminate and return a quest with: a root node; a structure containing all generated nodes; and the quest’s steps containing all player actions as well as the participating game objects. The quest giver must then make the quest available for the player to accept it. When a player accepts a quest, the system will need to monitor player development within the quest.

3.3 Overview

In this chapter we began by presenting our analysis of Witcher main story quests. This analysis resulted in the extension of the grammar defined by Doran and Parberry. The major differences are the addition of 4 player actions, and 5 NonTerminal actions with their respective set of rules. This changes were validated with a quest example, that shows the new grammar’s capabilities of representing more complex quests. We complete this chapter by defining a model in an imperative paradigm adapted to the direct use in games. In the next chapter, we will go into further detail on how we implemented and also integrated both the quest generator and the quest monitor in the survival game Conan Exiles.
Chapter 4

Implementation

In this chapter, we present the implementation and integration of the Quest Generation Model in Conan Exiles. Our Quest System is able to generate different quests for a player, and also monitor its development (the state of the quest the player is in). Conan Exiles is still in early access, and has no NPCs offering quests to the player. It is being developed in Unreal Engine, using both C++ and Blueprints (game-play scripting node-based interface within Unreal Editor). Consequently the whole Quest Generation System (Quest Generator + Quest Monitor), described in this chapter, was developed with Unreal Engine (UE) so it could be possible to test it in the game (see Figure 4.1). The quest generator uses simplified version of the grammar and is attached to a specific set of NPCs, as a component. The quest monitor is a component that is attached to the player character, that verifies if each action present in the generated quest is being executed correctly.

Figure 4.1: Mapping of the model to Unreal Engine/Conan Exiles.

Given the complexity of Conan Exiles project and UE itself, we have decided to simplify the model in two ways. Firstly, instead of developing a distributed system, we have implemented it in a central manner. This means a single NPC is in charge of generating the whole quest. This was done because
NPCs are only spawned based on proximity to the player. If two NPCs are too far apart, and the player is only close to one of them, the closer one won’t be able to detect the other NPC (it is not spawned). So a central model facilitated quest generation. Secondly, the grammar defined in Chapter 3 was also simplified, Conan Exiles is still in early access and does not possess enough mechanics to implement all 25 actions.

4.1 Quest Generator

As stated above, the quest system is divided into two components. The first component is the UQuestGenerator, implemented as a UE ActorComponent class, that is attached to a class called BP_QuestGiver. The algorithm presented in the previous chapter (see Algorithms 1, 2, 3, 4 and 5) is implemented in our UQuestGenerator. So every instance of a BP_QuestGiver will be capable of generating a quest through its UQuestGenerator component.

The UQuestGenerator will also have access to a reduced version of the grammar defined in Chapter 3, as well as a knowledge base containing: a set of BaseNpcs, ConanExiles’ Blueprint class for non-playable characters; a set of player-known Recipes\(^1\); and a set of Gameltems, a set of other BP_QuestGivers.

4.1.1 Input

4.1.1.1 Grammar

The grammar was mapped using UE DataTables. The data fields in these tables can be any valid UObject (the base class for all UE classes) property, including asset references. This helped mitigate the amount of work and complexity involved, and provided the ability to visualize and parameterize quests creation as well as facilitate the tweaking and balancing of data, based on feedback received.

The grammar is divided into 22 datatables: one for each Motivation (9 in total), containing its associated strategies and a description for each one; one for each NonTerminal (11 in total), containing its associated sequences of actions (see Table 3.2), weights related to each sequence, and a set of restrictions on the parameters of each action in that sequence; one containing in each row a Motivation and a reference to its associated datatable; and one containing in each row a NonTerminal and a reference to its associated datatable.

On BeginPlay (event launched by each UObject instance at start of game) the QuestGenerator will copy the content off of these datatables, to 2 different arrays:

- an array of type FQSMotivation structure, containing an enumerator MotivationNode and array of type FQSStrategyData structure, containing a string array with the action sequence and an array of type structure representing restrictions on parameters one for each action in the action sequence;

\(^1\)In Conan Exiles the player needs to learn Recipes in order to be able to craft new items. This will be used for the build action. Recipes extend the base item class Gameltem
• an array of type FQSRules structure, containing an enumerator of type NonTerminalNode and an array of type FQSNonTerminalData structure, containing a string array with the action sequence, a float for the weight of the action sequence, and an array of type structure representing restrictions on parameters one for each action in the action sequence.

These will eventually be copied to a quest on its creation.

The actual grammar that was used by the algorithm, was reduced to include only 7 terminal actions or tree leaves. These are: Kill, Goto, Gather, Build, Give, Report and Listen. Consequently rules and strategies that might include terminal actions not implemented in the grammar, were removed from it.

4.1.1.2 Knowledge Base

The knowledge base is comprised of 4 different arrays: one for quest givers, one for other NPCs, one for gatherable items, and one for recipes. Every time a quest is generated, all of the arrays are refilled, because during generation every time a reference is selected from an array it is removed. This was done so there wouldn’t be a reference to the same object twice in a quest. The arrays were populated in the following manners:

Although quest givers are NPCs, there was a necessity to distinguish them from other NPCs. This is because practically every other NPC in the game is hostile, even human exiles. There are only 5 NPCs within the game, that are friendly (non-hostile). But these only offer a few conversation lines. The array for the quest givers, is an array of type "FQuestGiverEntry" structure, that we have defined. It contains the name of the BP_QuestGiver and it location (a vector). These were defined in a UDatatable. The specific BP_QuestGiver reference is then obtained within the execution of an event associated with an action. We had to do it this way, because, as stated before, NPCs are spawned based on proximity to the player.

The player can craft items based on the recipes it has learned. So it was only natural, to develop an event (called "GetPlayerRecipes") that gets the learned recipes from the player. However, for testing purposes, it was limited to recipes that didn’t require a specific crafting station. For example: a "glass bottle mold", can only be crafted if the player has already built a blacksmith station. This means only items that could be crafted on the player were considered for the knowledge base (see Figure 4.2).

The array for hostile NPCs is set by an event called "GetCloseNpcs" (see Figure 4.3). Again for testing purposes, we have limited the NPCs based on distance to the BP_QuestGiver that is generating the quest. Since the play test map was limited to single quadrant or heightmap, it was necessary to limit the distance to a BP_QuestGiver, that would not include NPCs from other heightmaps.

The array for gatherable items, was also limited to a certain amount, again due to the play test map. So naturally, only items that could be gathered in the test map were selected. The amount of each item that the player needs to gather, will be 1/5 of the maximum stack size in the players inventory (which varies from item to item). This is the list of gatherable items: Stone; Bark; Wood; Branch; Bone; Plant Fiber; Hide; Feral Flesh; Savoury Flesh; Shaleback Egg; Handful of Insects; Seeds; Aloe Leaves; Orange Phykos.

47
4.1.2 Output

The output of the quest generator will be a quest, with a tree like structure, where all vertices of the tree are viewed as quest nodes. Leaf vertices are called terminal nodes, while non-leaf vertices are called non-terminal nodes. The root of each quest or subquest is designated as motivation node.

A node is defined in a C++ class called UQuestNode. A UQuestNode has a reference to its parent UQuestNode, in case of the root node it will be null. The UQuest also has an array of UQuestNodes representing its children, which in the case of terminal nodes will be of size 0.

A UQuestNode is distinguished through its name, which is a string translated from a type enumerator. Three types of enumerator were defined, one for each type of node:

- EMotivationNode, has 9 entries;
- ENonTerminalNode, has 11 entries;
- ETerminalNode, has 25 entries.

Enumerators can be used either as a string or an integer. This allowed us to use them as indexes for accessing the arrays related to the grammar, or even use them as names to execute functions (further details will be given later in this chapter).
The \textit{UQuestNode} also contains an array of type \textit{FNodeDetails} structure, representing action parameters. The structure is defined as follows:

- a \textit{TargetType}, of type enumerator called \textit{EParameterType} (NPC, Item, etc);
- a \textit{TargetName} of type string (the name of the actual object);
- a \textit{TargetReference} of type \textit{AConanCharacter} (this will only be used for NPC type parameter, otherwise it will be null);
- \textit{Quantity} of type integer, amount needed to complete an action;
- an \textit{ItemTemplateID}, used for identifying items;
- an integer \textit{Restriction}, that works as a flag for determining to which action the parameter is restricted;
- an integer \textit{SiblingReference}, which basically gives the position of the sibling to which the parameter is restricted. If that is the case.

The quest is defined in a C++ class called \textit{UQuestStructure}, that directly extends \textit{UObject}. It contains an array of type \textit{UQuestNode}, where the node at index 0 is the root of the quest. The following properties were declared in \textit{UQuestStructure} for easing the task of the quest monitor component: an array of type \textit{UQuestNode}, containing all \textit{TerminalNodes} generated by the algorithm; and an integer called \textit{CurrentAction}, starting at 0.

\textit{A UQuestStructure} also has 3 fields, that are only used by the \textit{UQuestGenerator}, that represent the whole grammar: an array of type \textit{FQSStrategies} structure; an array of type \textit{FQSRules} structure. These represent the same as the ones in \textit{UQuestGenerator}. Every time a quest is created, a copy from the related arrays in \textit{UQuestGenerator} will be assigned to the quest's respective ones. This is done because the copied structure containing the weights is altered several times during generation (because of the "UpdateWeights" algorithm). This way we have a clean structure of weights every time we want a new quest.

The algorithm then creates a quest in the form of a tree, by creating, expanding and adding nodes to the quest structure, until all leaf nodes have been reached and setting the necessary world objects that will partake in the quest.

\section*{4.2 Player UI}

It wouldn’t be possible for the player to accept and perform a quest, without some type of user interface for: presenting quests, interacting with a NPC, and guiding the player. Here we present the UI elements called \textit{Widgets}, which are a series of pre-made functions that can be used to construct an interface (things like buttons, check-boxes, sliders, progress bars, etc.), that were specifically developed for the quest system and used within the game.
4.2.1 Npc Dialogue

When the player interacts with a BP_QuestGiver NPC, a NPC’s dialogue window pops up, showing several buttons for the player to select. The W_NPCDialogue is a widget, that was already implemented in Conan Exiles. It allows for the interaction between the player and a friendly npc. This widget contains several buttons: like "Talk", "Goodbye", and buttons for learning recipes, emotes and feats. We have extended this widget, to give the player a few more possible interactions with the npcs. In total, 4 additional buttons were added:

- A button for asking the BP_QuestGiver to give a task to perform (highlighted in Figure 4.4), which is "Got any tasks to complete?". This initiates the process of generating a quest previously described. It was done this way, because in the game, characters are spawned by their distance to the player and with variant times. And the algorithm needs to have knowledge of them, prior to generating the quest. Nonetheless, quests are generated practically instantly.

- A button for viewing and accepting the generated quest (also highlighted in Figure 4.4), which is the "Available Quests" button. Whenever a BP_QuestGiver has a quest available, this UI button will be enabled, and once it is pressed, it will show another UI window with details of the quest. There needs to be some kind of visual feedback of the quest to the player, so it was only natural to add this possibility in our quest system, and in particular in this widget.

- Two buttons that represent the actions "Report" and "Give" (additional information about these will be given later in this chapter, as well as a visual component). These were added mainly to allow the quest performance.

![Figure 4.4: W_NPCDialogue with two additional buttons.](image)
4.2.2 Quest Giving

Once the quest is generated and the player presses the "Available Quest" button, another window pops up showing the details of the generated quest. The top of the window, displays the BP_QuestGiver's name and the number of the quest (for example if it was the first, second, third, etc generated quest by the BP_QuestGiver). The details are just the chosen strategy and the sequence of actions related to that strategy, that the player needs to perform in order to complete it. Two additional buttons were added to this widget, which is called W_QuestGiving. These are an "Accept" and "Decline" button (see Figure 4.5).

Selecting the "Accept" button, will copy the quest to the UQuestMonitor and will immediately start the quest. Accepting the quest will disable the buttons "Got any task to complete?" and "Available Quest" in every W_NPCDialogue widget of a quest giving Npc. We did not implement a quest log UI. So this was done so it wouldn’t be possible for the player to generate and accept another quest while performing another.

![Figure 4.5: W_QuestGiving UI for presenting the quest to the player.](image)

4.2.3 Quest Steps & Compass

Conan Exiles game world is extremely big and dangerous. Exploring the world and completing quest without some sort of feedback, would be a very difficult task. The following widgets were therefore developed as a mean to facilitate the players game-play experience (see Figure 4.6).

First we have the widget W_QuestSteps. This widget's purpose is to keep track of the development of the accepted quest. It displays two text boxes: one with the BP_QuestGiver’s name and the quests number; another with the action's name and specific targets. W_QuestSteps is enabled, as soon as the player accepts a quest, and starts by showing the first action in it. Once the player performs the current
action, the next action in the quest will be shown, and so on until the quest is completed. This way the player always knows what it has to do.

Also for the purpose of offering some guidance to the player while completing a quest, we have created a rotational compass widget, called W_Compass. Besides giving the player knowledge of north’s direction, it will show the target point’s location of the current action. Like stated before every action has a target object which it references. Unfortunately, due to the way gatherable items are spawned in the world, it was extremely difficult if not impossible to get their location, since items like rocks, trees and branches are "painted" as UE foliage, so the compass will only show an objective for NPCs.

A solution for this would be to create a datatable with the locations of where you can find big amounts of one specific item. This would imply creating a structure for this specific purpose, containing: an item uniqueid, an item name, an array of locations.

Figure 4.6: W_Compass with an active target point (center), and W_QuestSteps showing the action "Go to Shaleback" (right corner).

### 4.3 Quest Monitor

The second component of our quest system is called the UQuestMonitor. Again, it is a UActorComponent that is attached to the player class, which is BasePlayerChar (this class like BaseNPC, also extends several other classes including ConanCharacter). The UQuestMonitor is responsible for tracking the player, while he/she is performing the current quest. As stated before, we have introduced a UI that shows the player the current action he/she needs to execute, and a reference to the action's target UObject. The UQuestMonitor then simply verifies if the current action has been completed. If it has been completed, the UQuestMonitor notifies the UI W_QuestSteps, to show the next action in the quest.
4.3.1 Monitoring Actions

Once the player has accepted a quest, this quest is passed on to the UQuestMonitor (as well as to the UI W_QuestSteps). In its Tick event (called in every frame), the UQuestMonitor will check if the current action has changed. If it has, it will call an Event with the name corresponding to the new current action, using Reflection\(^2\). We have created a function that searches all the available functions of our UQuestMonitor and if it manages to find a function whose name is the same as the current action, it executes it. This is the most common case of using reflection. These events however, do not track the action execution specifically. Instead, for each corresponding action, they will bind another event to an EventDispatcher\(^3\). These work in a similar way as C# delegates or C++ function pointers\(^4\).

Again, for each action we have an event, that possesses the same name as the action, that binds another event to an event dispatcher. Each of these bound events, will then check if the current action has been completed. These bound events will be called from different classes in ConanSandbox project, depending on the type of action. Our quest system currently only supports 7 of the 25 possible player actions shown in Chapter 3. The following functions are the ones that were found more important, for giving a basic feeling of quest performance. These are:

**Goto** Every time the player moves (handled in BaseBPCombat, like previously mentioned), the event dispatcher ”SignalPlayerAtLocation” will be called. The event bound by this dispatcher will compare the current position of the player with the current position of the referenced character. If the distance between them is within a certain range, then the event considers the player is in the same location and ends (see Figure 4.7).

![Figure 4.7: Implementation of the event, corresponding to action "Goto".](image)

**Kill** Every time the player kills a character (this check is also done in BaseBPCombat), it will call the

---

\(^2\)UE4 has a built-in reflection system that can be used to perform changes in a game at runtime. Reflection is the ability of a program to inspect its own code (or other associated code) and change at runtime.

\(^3\)Event dispatchers are the only way to use communication where you dont care about the target object. When an Event Dispatcher is called, any Blueprints that implement and have Events bound to the Event Dispatcher will also execute when the Event Dispatcher is called. In other words, whenever the Event Dispatcher is called, any Events that are bound to it are also called allowing you to execute multiple Events at once from a single source.

\(^4\)A delegate (function pointer) is a class that wraps a pointer or reference to a member method (with a particular parameter list and return type) of an object’s class to be called on an object instance, and also providing a method to trigger that call. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance. It is mostly used in conjunction with events. Event handlers are nothing more than methods that are invoked through delegates.
event dispatcher, "SignalPlayerKilledCharacter" (see Figure 4.8). The event bound to this dispatcher, will then verify if it is the same character referenced by the kill action. In case the action requires multiple targets, the bound event will decrement the remaining count until it reaches zero.

![Figure 4.8: Implementation of the event, corresponding to action "Kill".](image)

**Listen** When the player is required to learn the location of its target, the player will have to obtain this information from a NPC of type BP_QuestGiver. The player will have to interact and press the button talk of that NPC’s W_Dialogue widget. This will call the event dispatcher "SignalPlayerTalkingToNpc" (see Figure 4.9). The event bound to it will simply check if the target reference in the action’s parameter is the same as the NPC the player is talking to.

![Figure 4.9: Implementation of the event, corresponding to action "Listen".](image)

**Report** Every time it is required for the player to report back to the Npc that gave out the quest, a button will be added to its W_NPCDialogue widget. Once the widget is open, selecting the button will call the event dispatcher "SignalPlayerReportingToNpc" (see Figure 4.10). The event bound to this
dispatcher will check if the target reference in the action's parameter is the same as the NPC the player is reporting back to.

**Figure 4.10: Implementation of the event, corresponding to action "Report".**

**Gather** Once called it will bind two events each to its own event dispatcher: "SignalItemAdded" and "SignalItemIntStatChanged" (both defined in InventoryItem.h, a class from Conan Exiles). Then every time the player gathers a certain amount of items, a call is made to either one of these dispatchers (see Figure 4.11). If it is the first time that type of item is added to the player, the event bound to "SignalItemAdded" will be executed. If it is simply a statistic change it will execute the one bound to "SignalItemIntStatChanged". Both events however, perform the same check. They will first verify if it is the same item referenced by the gather action. Since it is always required to gather more than one, they will further check if the right amount was collected.

**Build** This action also takes advantage of the same dispatchers mentioned in the action "Gather". But instead of just verifying the count in the Backpack Inventory, it will have to do a sum check of the total amount in the backpack with the total amount in the shortcutbar. This is done because when an item is crafted, it will be first added to the shortcutbar if it has available boxes (see Figure 4.12).

**Give** Every time it is required for the player to deliver an item to an NPC, again a NPC of type BP_QuestGiver, it will add a button to the widget W_NPCDialogue. Once the widget is open, selecting the button
will call the event dispatcher "SignalPlayerGiveItemToNpc", that checks the player's inventories (in Conan Exiles there are 4 different types of inventories, for this action only the ShortCutBar and Backpack inventories are relevant) for the required amount of the referenced item. If there are enough items in both inventories, the items are removed and the action ends, otherwise it will warn the player that there aren't enough (see Figure 4.13).

4.3.2 Moving to next Action

Having confirmed that the current action was correctly performed, we must immediately unbind the event from the event dispatcher. This must be done, because otherwise we would keep getting calls from other parts of the code even though the action was already done. After unbinding the event, we delete the node from the quest and move to the next available action. We must delete the node, because most of the times we store a reference to a UObject, that sooner or later will be handled by the UE Garbage Collector. If we don’t handle this, it might cause a crash in the game.

Moving to the next available action is implemented in an event called "CurrentActionCompleted".
This event first, increments the current action counter, and then checks if the quest is complete. If there are still actions to perform it will call an event dispatcher defined in the UI W_QuestSteps, called "ShowNextAction", to change the action being displayed.

4.3.3 Completing Quests

If there are no more actions to perform, then the player has completed the quest, and the function "FinalizeQuest" is called. This function simply awards the player experience points (XP) for its completion and deletes any reference to the current quest, for the same reason as the nodes. Awarding the player XP requires the use of the Progression System already done in Conan Exiles.

In their BP_ProgressionSystem we define an event called GiveQuestXP, which further calls a function with the same name, that receives a quest as input. This function awards XP, based on the number of actions the quest has. It multiplies the number of actions by 30 points (merely serves as a symbolic value), which then will be multiplied by Conan Exiles XP rate multiplier, and finally added to the player’s current XP points.

4.4 Play Test Scenario

In order to test the quality of the implemented quest system in a real game experience, two play test scenarios were conceived:

- **Conan Exiles Scenario**: here it is expected for the player to play the game as it was intended. The player should try to survive in Conan Exiles’ hostile environment. This will require the gathering of resources, to craft tools (weapons, harvest tools, camp-fires, beds, etc) and structures (foundations, walls, ceilings, shrines, etc) from recipes, acquired from spending points earned for levelling up.

- **Quest System Scenario**: here the player should explore the scenario and look for the quest givers stationed all over the map. The player should approach either of them, ask for a quest, and complete it. Then keep repeating this process, while still performing similar tasks as the ones from the previous scenario. Basically trying to survive the hostile environment, by crafting the necessary items to do so.

4.4.1 Map Location

Both play test scenarios use the same portion of the map from Conan Exiles’ world, here referred to as heightmap. The heightmap was further reduced by some barriers, parallel to the heightmap’s boundaries. The chosen heightmap is part of the first area the player finds, when starting a new game. It was important to give the player the same feeling in both experiences, by avoiding extreme differences. Besides having the same resources and enemy NPCs, this portion contains one friendly NPC, called
"Shaman Nunu". The only difference between the two is the quest givers introduced in the quest system scenario. Figure 4.14 shows a top view of the heightmap in the second scenario.

![Figure 4.14: The heightmap chosen for both test scenarios. Highlights indicate the locations of the BP_QuestGivers.](image)

### 4.4.2 Quest Givers

For the second scenario 8 quest givers were added, 4 male and 4 female. We have given to each sex a set of dialogues with voice overs. A total of 42 dialogue lines: 2 for each strategy used (a total of 17); 2 for each action deemed fit to require a line of dialogue, like report, give, etc. In order to give dialogues to NPCs, we had to add additional blueprint classes and datatables. We have extend the BP_DialogueParams blueprint with our own version for quest givers. This class basically holds the value of the dialogue datatable entry relevant to the NPC. We altered it to include entries of our own datatables, and extended the class with two others: one for male characters (BP_DialogueParams_QuestGiverMale) and one for female characters (BP_DialogueParams_QuestGiverFemale). Two datatables were created one containing the strategies’ dialogues and another containing the actions’ dialogues.

In order to spawn the quest givers, we based on the way friendly characters already available within the game are spawned. To spawn a character one must take the following steps:

1. Edit the datatables SpawnDatatable and WeightedSpawnTableRow, by adding an entry for an npc in each one. The WeightedSpawnTableRow contains: an identifier for the table, an identifier for the SpawnDatatable, and a weight, with the value 1.0 for friendly npcs. The SpawnDatatable contains the details of the npc: the name; templates for physical aspect; templates for various stats; a directories for the corresponding blueprint class, the associated behaviour tree, the AI controler
and the dialogue parameters blueprint discussed earlier.

2. Edit the ConanSandbox map or Gameplay_Npcs map by: dragging to either one of these levels 3 specific actors for each quest giver and changing some of their details. These are:

(a) A BP_ManualSpawnPoint: this corresponds to the actual location of the quest giver we want to spawn. Here we must edit the "Default" section of the blueprint details, by adding an element to the "Human Options" list. Then edit the field "SpawnTable" of the added element, with the name of the row in SpawnDatatable. A visual aid is shown in Figure 4.15;

![Figure 4.15: Necessary settings for BP_ManualSpawnPoint.](image)

(b) A BP_CampOwner: we must edit the details of the actor component BP_CampComponent by adding an element to the "Camp Actors" list, in the "Camp" section, corresponding to the added BP_ManualSpawnPoint. A visual aid is shown in Figure 4.16;
(c) A NPCTerritorySpawner: this actor is frequently used to spawn most of the hostile npcs (mostly wildlife). Here we must edit the scale in the "Transform" section (we used the same values for friendly NPCs), and add a reference to the related BP_CampOwner in the "Camp" property of the "Static Navigation Override" section. A visual aid is shown in Figure 4.17;

The quest givers have to be easily identifiable. Hostile human NPCs are usually identifiable by a lit fireplace, where several group up. So it was decided to add a tent near their spawn location. As it can be seen in Figure 4.18.
Figure 4.18: Merla Lilibeth, one of the quest givers added in the test scenario.
Chapter 5

Evaluation

Having finished the implementation of the quest system presented in the previous chapter, it is time to evaluate it. In this chapter we will describe how the tests were conducted and detail the results obtained through questionnaires.

5.1 Player Test

There are three aspects that we can evaluated: quest quality, player enjoyment, and game-play flow. With the player test we hope to measure the game experience through these and validate the work presented in this document.

A way to measure the generated quest's quality would be by comparison with other quests. These could be either from RPG games (human authored) or other generated ones (remember Conan Exiles doesn’t offer quests). Another option to validate quest quality would be to compare quests textually. Our algorithm would record the quest in a textual document, and then participants would compare them to quests transcribed from other games. We would rather have our quests tested by being played. Player enjoyment and flow can be measured through surveys imposed on the players after each test. We will cover this in subsection 5.1.2.

We decided to have the participants play Conan Exiles in two different experiences: the original game (Version A) and the game with our quest system (Version B) like described in the scenarios in section 4.4. So now we have to decide how to allocate participating players. Will we have all participants play both scenarios (repeated measure), or will we equally distribute them between the two scenarios (independent measure), by allocating them randomly to either one.

We have decided to allocate players using repeated measure. We fill the contras are more easily managed, and overall it will be less time consuming. The order affects can be avoided, by counter balancing the scenarios. This means splitting the experience into two groups: one where participants perform the scenarios in the order version A first and Version B second; and one where participants perform the scenarios in the opposite order [22].

So in the end, each player will be required to play for about an hour and ten minutes, distributed
through three sessions: a brief tutorial discussed in subsection 5.1.1, and both scenarios described in section 4.4, using repeated measure with counter balance. The first session lasts about 10 minutes and the latter two last 30 minutes each.

5.1.1 Tutorial

In order, to allow every player to have an equal start of the test, a tutorial session was conceived. This way, when the player actually gets to the real test, it will be comfortable enough with all mechanics and UIs. We wanted the player to have knowledge more or less equivalent to the first couple of hours of game-play. For this a Players Guide was written for the players to read (with additional tips) (see Appendix B), and three tasks were given to the player to complete in under 10 minutes:

- Craft a Stone Pick;
- Craft 3 pieces of clothing;
- Kill a gazelle or a hyena.

The first two tasks force the player to learn how to gather resources efficiently. It will also help the player get acquainted with the Inventory menu, which tends to occupy much of the game-play time. The last task, will make the player explore the world (which is the same as in the two scenarios), since they are spawned in specific areas of the world map.

5.1.2 Questionnaire

If we are looking to evaluate the player’s game experience, we should address in detail how he/she perceives the game on a more affective and enjoyment level. In other words, the player’s game experience can be understood as the end player’s perceptions and responses resulting from the gameplay in terms of enjoyment or engagement to the game.

The following variables are considered a priority: enjoyment, usefulness and flow. Additionally, social believability (in terms of assets) can be considered as a relevant aspect, however we found that this should not be evaluated in our experience. We decided to focus solely on player enjoyment and flow. In order to capture these variables, we conceived a questionnaire for players to answer at the end of each scenario.

The overall enjoyment of the player serves as a general subjective assessment of the game. For a short measure of enjoyment we combined the sub-scales related to his variable from two different questionnaires: GUESS [19] and IMI [27]. Amounting to a total of 10 questions. This offers a manageable approach for gathering a valid assessment with a low evaluation load for participants.

Flow is an indicator for the fun and immersion of a game. Flow itself is a concept that integrates different aspects, and can thus be characterised by the following features: challenge, concentration, action fluency, time transformation and awareness [34]. To capture this variable we based our questions in the Flow Short Scale [25][34]. It consists of 10 questions to be answered on a 7 point rating scale. It is
divided into two factors: smoothness of action and immersion in task. This questionnaire has a practical length and takes into account all characteristics of flow.

Players will have to answer these questions twice, one for each of the scenarios. Additionally, we have added a few questions on a more personal level, which only need to be answered once. Like sex and age of the player, gaming habits and if the player had already played the game before. The full questionnaire can be viewed in the Appendix C.

5.2 Results

We had 22 people performing our tests. Around 90% of participants were male. Participants were mainly students, with ages ranging from 18 to 30 years old. Only one participant had previously played Conan Exiles. On average, 52.4% of the participants play games for more than 12 hours a week, which means that more that 50% can be considered as hardcore gamers. More than 90% of participants had played an RPG before, and more than 70% had played a survival game before.

The players were given half an hour to play Conan Exiles and instructed to play as it felt natural to them. They spent their time gathering resources, killing NPCs and crafting items. We have recorded some of the players actions: 90% crafted a fireplace and cooked in it; 82% crafted a bed; 90% crafted one of the first learnable weapons(sword or club); 36% built a house; 13% crafted a bow; 18% crafted a shield, 9% crafted a water-pouch; and only 18% found and talked to the friendly NPC Nunu. The average level reached in this experience was 5.13.

In their half hour of playing Conan Exiles with our Quest System, players have completed an average of 2.86 quests (per session). The highest number of quests completed was 6, and the lowest was 0. The average level reached was 4. The difference of levels between the two experiences can be explained by the fact that in our Quest System players spent less time killing NPCs. Killing NPCs is what gives the most experience points for leveling up. Alternatively to completing quests, players also spent time crafting objects like fireplace, bed, stone pick/hatchet, and clothes. In some cases, players focused on leveling up in order to learn how to craft weapons. They would then use the weapon to kill an NPC referenced in the quest.

As stated before, the tests were conducted using repeated measurements. All players played both of the scenarios in an order that was randomly assigned. After playing each scenario, the players had to answer a questionnaire, in order to assess the overall experience based on enjoyment and flow.

Below we present the overall scores of the players’ experience regarding enjoyment and gameplay flow, for both Conan Exiles and our Quest System. The average scores for each of the measured variables are shown in Table 5.1, on a scale from 1 to 7.

<table>
<thead>
<tr>
<th></th>
<th>Conan Exiles</th>
<th>Quest System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enjoyment</td>
<td>5.252</td>
<td>5.178</td>
</tr>
<tr>
<td>Flow</td>
<td>5.229</td>
<td>4.891</td>
</tr>
</tbody>
</table>

Table 5.1: Mean scores of the variables for each experience.
To get a fuller vision of the distribution of the answers we present below box plots for “Enjoyment” and “Flow” (see Figures 5.1 and 5.2).

Figure 5.1: Boxplot related to the answers of the “Enjoyment” section of the questionnaire. The first entry for each item refers to answers to the Conan Exiles experience.

Figure 5.2: Boxplot related to the answers of the “Flow” section of the questionnaire. The first entry for each item refers to answers to the Conan Exiles experience.

As can be observed, the scores for the Conan Exiles experience were generally higher for both
variables. However the difference is less pronounced regarding "Enjoyment". The discrepancy between the outcomes for the "Flow" variable, may be explained by several factors namely: the noticeable difference in terms of performance between the two games. The Quest System version was lagging more than the Conan Exiles' (something that players reported); The Quest System was implemented on an older version of the Conan Exiles which is less optimized; Finally, each scenario was tested on computers with different specifications. An inspection of the box plots reveals that the distribution of the answers exhibits a considerable heterogeneity in terms of spread, skewness and also number of outliers.

The ultimate goal of the analysis of the responses is to determine whether the observed mean differences are statistically significant. To determine the appropriate tests, we started with some exploratory analysis of the responses, in order to determine whether they could be reasonably assumed as generated by a normal population. If the answers were affirmative the appropriate procedure would be a dependent t-test. Otherwise we would have to resort to a non parametric Wilcoxon rank paired test.

As part of the exploratory analysis we have also checked the reliability of the studied measures. For this we performed a Cronbach's Alpha. The coefficient of reliability ranged from 70% to 90%, meaning that the internal consistency of the scale is acceptable.

The assumption of normality was clearly rejected for 31 out of 40 variables. The inspection of the box plots corroborates this conclusion. Taking also into consideration the small sample size, we decided to use the non parametric test.

For most items, we can not reject the null hypothesis that the experience of playing both games was equally satisfying. The exception is the item: "The game-play ran fluidly and smoothly." (with mean responses CE = 5.55; QS=3.64), for which the experience of playing Conan Exiles was significantly better. We have already alluded to some of the factors that may explain this difference. The detailed results for this test can be viewed in Tables 5.2 and 5.3.

<table>
<thead>
<tr>
<th>I enjoyed playing the game.</th>
<th>0.099</th>
</tr>
</thead>
<tbody>
<tr>
<td>I had fun playing the game.</td>
<td>0.123</td>
</tr>
<tr>
<td>I felt bored while playing the game.</td>
<td>0.507</td>
</tr>
<tr>
<td>Playing the game didn’t hold my attention at all.</td>
<td>0.43</td>
</tr>
<tr>
<td>While I was playing the game, I was thinking how much I enjoyed it.</td>
<td>0.858</td>
</tr>
<tr>
<td>I found the game experience pleasurable.</td>
<td>0.623</td>
</tr>
<tr>
<td>If given the chance, I want to play this game again.</td>
<td>0.434</td>
</tr>
<tr>
<td>I am likely to recommend this game to others.</td>
<td>0.943</td>
</tr>
<tr>
<td>I felt motivated while playing the game.</td>
<td>0.483</td>
</tr>
<tr>
<td>I thought the game was a fun experience, rather than a task I was just doing.</td>
<td>0.841</td>
</tr>
</tbody>
</table>

Table 5.2: Results of the Wilcoxon Signed Test for "Enjoyment".
I felt just the right amount of challenge. | 0.18
---|---
**The game-play ran fluidly and smoothly.** | 0.001
I didn't notice time passing. | 0.68
I had no difficulty concentrating. | 0.544
My mind was completely clear. | 0.868
---|---
I was totally absorbed while playing the game. | 0.857
The right thoughts and movements occurred of their own accord. | 0.643
I knew what I had to do each step of the way. | 0.458
I felt that I had everything under control. | 0.315
I was completely lost in thought. | 0.776

Table 5.3: Results of the Wilcoxon Signed Test for "Flow".

into consideration that our quest system, given its experimental nature, did not present to the players a typical and familiar interface and, as such, may have biased negatively their appraisal.
Chapter 6

Conclusions

The overarching purpose of this dissertation was to explore ways to create story related content in video games with minimal human intervention, while measuring up to human-authored alternatives.

In order to achieve this, we have introduced adjustments to the quest structure, as it has been defined by Dorian and Parberry, in their analysis of MMORPG quests. These adjustments were based on our study of the 58 main quests from the prize-winning single player RPG game “The Witcher 3 – The Wild Hunt”. In their paper, Doran and Parberry[8] themselves concluded that further work was necessary to prove the capabilities of their generator in producing quests equal in quality as human authored ones.

We believe that our adjustments to their grammar, make it more expressive and able to generate bigger and more complex. It should be noted, however, that our analysis revealed that “Witcher” quests and MMORPG’s differ in significant ways from one another. This suggests that our grammar can be improved through the analysis of other single player RPGs.

Our goal was to create a system that uses a procedural approach to story generation. We defined a quest generation model, that uses this extended grammar, and adapted it to be used in the context of video games. We have implemented this model in the survival sandbox game Conan Exiles, currently in early access and lacking a quest system. The developed quest system contains: a component, capable of generating quests that are motivated by a NPC’s own intentions; a component capable of monitoring the execution of the actions contained in the quest; and some UI elements that extend the player-NPC interaction for the purpose of giving and presenting quests.

The system was tested with human players. The results were very reassuring in the sense that, in spite of its experimental nature, the introduction of the quest system did not deteriorate the players overall enjoyment of Conan Exiles.

We believe that our proposal is a step forward in enabling the development of games with improved replayability and enjoyment by offering players more content variety. Simultaneously we think they may reduce the time and financial costs associated to that development.
6.1 Future Work

There are several ways of continuing the work executed throughout this project. The first and most obvious is to finish implementing the monitoring of all actions defined in our grammar. Without them we can never fully test the capabilities of the quest system. Some of the actions (few examples are take, exchange and repair) could be implemented somewhat the same way as give or build. It would require adding a few more dispatchers and UI buttons.

The system currently works in a centralized fashion. One NPC generates the whole quest, which includes other NPC’s sub-quests. Our goal however, was to implement our model in a distributed fashion. This means that when an NPC requires a sub-quest, each should call another NPC’s quest generator to create one. The generator that receives the call would then return an array containing all actions of the sub-quest fully parameterized with objects from its own knowledge base. Our suggestion would be to try to use event dispatchers. Some NPC’s could bind one to an event “CreateSubquest” that when called would return a sub-quest.

NPC motivations and strategies as well as the objects that partake in the quest are currently selected randomly. This offers a quick solution for generating quests, but gives the feeling of a disconnected story, which of course isn’t good for a storytelling system. We see two possible solutions for this. One solution would be to follow the work of Bruss et al.[7] and add a progressive system. Here the motivations and the NPCs knowledge would require some human authoring. Another solution would be to add some kind of environmental awareness and social behaviour components to quest givers. This way changes in the world perceived by the NPC as well as interactions with other characters would alter its inner state and consequently the motivation/strategy chosen for quest generation.

In our system the player is capable of choosing which quests to undertake, it might be possible that the player does not want to search the whole world for his/her preferred quests. This might lead to a state of frustration for the player. We could add a player model to our system, that learns the player’s preferences, and adapts the generated content. A solution could be to use Robin Laws[16] player types, and generate sub-quests that suit those styles of play.

Currently the player can only perform one quest at a time. Allowing the player to accept more than one quest, would require to implement a quest manager UI for logging quests that are either accepted, completed or available. For those that are available the player would be required to move to the location in order to accept the quest. Such an UI was actually requested by some of the participants. Additionally, we could improve the way that the dialogues intrinsic to the quest are implemented. We suggest the development of a dialogue system.
Bibliography


[34] R. Vollmeyer and F. Rheinberg. Motivational effects on self-regulated learning with different tasks. 


Appendix A

Quest Example

In this appendix section, we analyse the quest "The Beast of White Orchard", one of "The Witcher 3" main story quests, using the new set of rules (see Figure A.6 for full quest). The quest is given out by a "Nilfgaardian" Commander with the promise of information about the witch Yennefer, which Geralt (the player) is currently looking for. The Nilfgaardian army is having trouble with a Griffin, that has been randomly attacking its soldiers, and the Commander wants Geralt to kill it. Looks simple, but this quest requires the player several preparatory steps. First the player must find information by talking to a hunter about the griffin. Information like where its current location is or why it is on a rampage. He/she also has to gather a plant with a strong scent that attracts it, by talking with a herbalist. In a way, the player needs to prepare before the encounter with the griffin. This quest can be seen as having the motivation "Comfort", using the strategy "Kill Pests", which starts with the sequence of actions "<goto> <defeat> <report>" (see Figure A.2 and Table 3.1).

The first action rule <goto>, implies that the player must go to the griffin's location. In Doran and Parberry's rules, the player would either have to learn the location, in case it was unknown, or explore. In this quest, instead, the player was required to prepare before facing the griffin, namely to learn about the griffin and gather something to lure it. So, in this case we use rule number 10 (from Table 3.2) "<prepare> <goto>". The action <prepare> will be expanded to "<goto> <subquest>" (rule number 16)

Figure A.1: Key to all other Figures.
Figure A.2: Initial strategy of example quest.

Figure A.3: Expansions of <goto> (rule number 11) and <prepare> (rule number 17), both newly added.

(see Figure A.3).

The next <goto> (4th expanded action), requires the player to go to the site where Nilfgaardian soldiers were attacked, but first he/she must talk to the hunter, for guidance. So, the <goto> is expanded to "<learn> <goto>" using rule number 0. The <learn> is then expanded using rule number 12 "<goto> <subquest> listen". The player must first go to the Hunter’s house to find that he isn’t there, which requires the player to explore and examine clues that direct him/her to the Hunter’s location (see Figure A.4). Here is the first instance where the use of Doran and Parberry’s rules, is unable to represent the quest. It wouldn’t be possible for the player to find the hunter, without the atomic action “examine” and the newly added rule number 15, which isn’t represented in their set of rules.

After finally reaching the Hunter (12th expansion), the <QUEST> action rule is then expanded to Comfort motivation strategy "Kill Pests". Here the hunter asks the player to kill some wild dogs that are troubling him (13th-20th expanded actions). After reporting to the hunter, the player must follow him to the site where the soldiers were attacked. Again, this atomic action isn’t present in the rules defined in [8]. Once at the site, the player has again to examine some clues, that lead to tracks that must be followed. Finally leading to the griffin’s nest, where the player learns the final details about the griffin (21st-29th expanded actions) (see Figure A.5). Notice that if we were using the set of rules defined by Doran and Parberry, this sequence of actions wouldn’t be possible. Instead we would have an atomic action “goto”, that would make the player go directly to the nest, alternatively to finding his/her way over there.
Figure A.4: Expansions 5 to 12, use of newly added action "examine".

Figure A.5: Expansions 22 to 29. Note expansion of <goto> after <learn> is closed and the use of newly added action "follow".
Thus, closes the <goto> action rule expanded after <prepare> (3rd expanded action) (see Figure A.3). Now the <subquest> is expanded using rule number 2 (from Table 3.2), "<goto> <QUEST>". Before facing the griffin, the player must still gather buckhorn to be used as lure, but first he/she must learn its location. The player must first talk to a herbalist to get this information. This is all represented through expanded actions 32-36 (see Figure A.3). Although it wasn’t given by any specific NPC, the gathering of the buckhorn is represented through the expansion of <QUEST> using the strategy "Gather Raw Materials" from motivation Wealth, which can be resumed to a simple atomic action "gather", since the player is already at the location (see Table 3.1).

Having gathered every bit of information and the necessary lure, it is now time to move to the griffin’s location and ultimately defeat it. The <goto> from the original strategy ends with an atomic "goto" (see Figure A.3), continuing with the expansion of the action rule <defeat> (see Figure A.2). As stated before, the action rule <defeat> was added, so it would be possible for a generator to decide whether a strategy would have the player kill or damage an enemy. In the end, both actions can be summarized as defeating an opponent. The choice of which action to perform, could then be given either to the player or to the NPC giving out the quest. In this case, the <defeat> rule is expanded using rule 28 "<goto> kill" (see Table 3.2).

The next step for the player, is to use the previously gathered buckhorn to lure the griffin out. Since it isn’t required to learn anything, we expand the <goto> action using rule 10, <prepare> <goto>. The <prepare> action rule is then expanded using rule 16. Now we have "<goto> <QUEST> <goto>", both <goto> are empty since the player is already where he/she needs to be, and <QUEST> is expanded using the strategy "Use existing tools" from Ability motivation, to an atomic action "use"(see Table 3.1 and Figure A.6). Since the player isn’t required to move, we are left only with the decisive action of killing the griffin (51st expanded action). Finishing with the <report> action rule, which is expanded using rule 30, "<goto> report"(see Table 3.2). The player most now return to the Nilfgaardian camp and report to the Commander (52nd-54th expanded action).

As can be observed, in figure A.6, all required actions were successfully represented using this new set of rules. The addition of 4 new atomic actions, as well as the addition of action rules <report> and <give> to the strategies, was simply to help represent certain sequences of actions executed by the player in "The Witcher 3". The biggest difference, and probably the most influential, was the removal of the atomic actions "goto", after the <learn> and <QUEST> actions rules (see rules 2 and 9 from Table 3.2). These atomic "goto"s were restricting greatly the order in which actions could be performed. Their removal, allows for more variability, but we lose control over the size of the generated quest. Nonetheless, it is possible to restrict the expansion of these action rules. By means of limiting the depth of the tree, or simply by giving a probability for expanding a rule, making some rules less or more likely in certain depths.
Figure A.6: The Beast of White Orchard quest using the new set of rules. The order in which actions performed by the player should be read depth first then left to right.
Appendix B

Player Guide

B.1 Controls

![Conan Exiles' basic controls.](image)

B.2 Inventories

There are 3 player inventories in Conan Exiles: Backpack, Shortcut Bar and Equipment. Each has its own specific purpose, but in general they serve to store items you have collected or crafted in game. To pass items between inventories, you simply click a drag the item to a box of the inventory of your choosing.

**Backpack**  this is the default inventory for all the items you collect or craft. It can only be accessed
through the Inventory menu (I key);

**Shortcut Bar** this is where you should put items, that you need to have quick access to. Like harvesting tools (Pick, Hatchet, etc), weapons, water bottles or even food. You can access items in this inventory, outside of the inventory menu. You can simply click on the keys 1-8 and the item set in that position will be selected;

**Equipment** this is where you equip your armor/ clothing. Each box is specific to a part of your body. Helmets only go to the head, shoes only go to your feet, etc. It can only be accessed through the Inventory menu (I key).

![Figure B.2: Player’s inventory UI. Activated by pressing "I" key.](image)

**B.3 Harvesting**

Harvesting is the act of collecting/ gathering resources, like stone, wood, fiber, etc. There are two ways to do harvesting. You can either pick them up from the ground (small stones, branches, plants, eggs, etc), or you can use a tool you have crafted (pick or hatchet), some tools give you more of a type of resource that others. Tools can be used almost in everything (trees, rocks, dead bodies). Items you harvest will go directly to your backpack inventory. Here is list of items that will be harvestable and where you can find them:
B.4 Crafting

Crafting is the act of creating items. To craft items, you will first need to gather specific items. Each craftable item will require different quantities of one or more gatherable items. To craft an item you will have to open the Inventory menu (I key). The crafting station will be on your right and looks like the following:

![Crafting Station Image]

Figure B.3: Player’s crafting station.

B.5 Leveling Up

Every time you reach a certain level of experience points (Xp), you will gain a new level. With every new level you will get Attribute Points and Knowledge Points. To spend these points, you have to open the inventory menu (I key), and click on the “Level up” button:
Here you can choose, which attributes to spend your points in. To spend knowledge points you will have to click the “Learn Recipes” button, which will open the following UI:
Appendix C

Questionnaires

Questionnaire that players had to answer after test session. C.1 is answered only once. C.2 and C.3 are answered twice, one time for each scenario played.

C.1 Personal

1. Sex (multiple choice): Male or Female.
3. In average, every week I play games for (multiple choice): 0 hours, 1-6 hours, 6-12 hours, 12-20 hours, 20-30 hours, >30 hours.
4. I have played RPGs before (multiple choice): Yes or No.
5. The last RPG I played was (written answer): name of the game or n/a.
6. I have played survival games before (multiple choice): Yes or No.
7. The last Survival game I played was (written answer): name of the game or n/a.
8. I have played Conan Exiles before (multiple choice): Yes or No.

C.2 Enjoyment

Items are answered on a seven-point Likert scale with anchors at every rating point (e.g., 1 = Strongly Disagree, 5 = Somewhat Agree, and 7 = Strongly Agree).

1. I enjoyed playing the game.
2. I had fun playing the game.
3. I felt bored while playing the game.
4. Playing the game didn’t hold my attention at all.

5. While I was playing the game, I was thinking how much I enjoyed it.

6. I found the game experience pleasurable.

7. If given the chance, I want to play this game again.

8. I am likely to recommend this game to others.

9. I felt motivated while playing the game.

10. I thought the game was a fun experience, rather than a task I was just doing.

C.3 Flow

Items are answered on a seven-point Likert scale with anchors at every rating point (e.g., 1 = Strongly Disagree, 5 = Somewhat Agree, and 7 = Strongly Agree).

1. I felt just the right amount of challenge.

2. The gameplay ran fluidly and smoothly.

3. I didn’t notice time passing.

4. I had no difficulty concentrating.

5. My mind was completely clear.

6. I was totally absorbed while playing the game.

7. The right thoughts and movements occurred of their own accord.

8. I knew what I had to do each step of the way.

9. I felt that I had everything under control.

10. I was completely lost in thought.