A procedural quest generator for Conan Exiles: Extended Summary

António Santos Ferreira Machado
Universidade de Lisboa, Instituto Superior Técnico
Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
antonio.machado@tecnico.ulisboa.pt

ABSTRACT

The dissertation proposes a procedural quest generation model as a means to generate interactive stories 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 Witcher 3 - 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 with 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.

Author Keywords

procedural generation; interactive storytelling; RPG; quests; story; NPC; player.

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 [3]. The process of this content creation takes a considerable amount of time and financial resources[5]. 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 [3][7]. This becomes quite useful when trying to produce content for an industry that is becoming increasingly demanding [4]. According to Hartsook[3] there are two broad uses of PCG in games: content creation and adaptation of game-play.

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.

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

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 [1] that consists of a structural analysis of several MMORPG quests.

The contributions of this dissertation are four-fold: we extend the grammar defined in the prototype quest generator of Doran and Parberry[1], which resulted in a published paper[6]. This extension was done in order to be able to represent bigger and more complex quests; we have defined and validated a quest generation model that uses the extended grammar in a paradigm directly applicable to games; we have implemented this quest generation model and integrated it into a quest system developed for a specific game; we have conducted tests

VJ2016 ISBN X-XXXXX-XX-X/XX/XX.
DOI:

with human players, in order to ascertain their overall enjoyment with the quest system.

The remainder of this document is divided as follows. In Chapter 2, Related Work, we review procedural generation approaches researched by other academics. 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 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 ideas for future work.

RELATED WORK

To solve the problem of plot-line adaptation, Li and Riedl [5] present an off-line 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, 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. 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 met. Once complete, the resulting story structure is converted and sent to GAME FORGE system[1][3] that renders a world that supports the story and executes the game. Indeed, although Li and Riedl [5] 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.

[1]Components of GAME FORGE, were presented in different papers, one presenting an off-line planning algorithm[5], and one describing the whole GAME FORGE system[3].

Doran and Parberry [1] did a structural analysis of almost 3000 human-authored quests from several Massive Multiplayer 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. 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-Normal Form (BNF). Terminal symbols are atomic actions and non-terminal symbols are action rules, that extend to further actions or action 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:

<Knowledge> ::= <Deliver item for study>

<Deliver item for study> ::= <get> <goto> give

Where: <Deliver item for study> is the strategy; <get> <goto> give" is the associated sequence of actions; get> and <goto> are non-terminal actions; "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 an NPC motivation the generator 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" [1].

As a rough generalization, single player RPGs tend to focus more on the journey(story), which plays a central role in these type of games. At the end the player either explores the open world or simply restarts the game in a new save file. In a contrast, in MMORPGs this journey feels more like a grind that the player wants to complete as quickly as possible, in order to get to the endgame content, where the true game begins. Quests generated using the structure previously described, which was used by Doran and Parberry on their prototype generator [1], are solely based on MMORPGs. We believe that the quests used in this approach are too simple and might not offer very compelling stories to the player.

According to Doran and Parberry's structural analysis, human authored quests have a shared structure. Having this in mind, we tried to analyse quests from single player RPGs, which tend to have a strong focus on the story component of the game, using the rules defined by Doran and Parberry [1]. We chose The Witcher 3 - The Wild Hunt, since it was received with critical acclaim and was a financial success, having also won several awards for Game of the Year from multiple publications. However, only the main story quests were analysed, which are in a total of 58, since side-quests are more similar to MMORPG quests.

This analysis resulted in a published paper[6]. Here we present our analysis of Witcher main story quests. The result of this analysis was 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, and show that the new grammar is capable of representing more complex quests.

Conan Exiles

Conan Exiles[2] 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 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.

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, allowed 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 quest to the player.

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. After a quest is created it will be added to a set of available quests for the player to choose from (see Figure 1).

Here we present our model of a quest generator that uses the extended grammar to generate quest. 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.

Quest Generator

In this section we are going to take a look at our adaptation of the prototype generator model to the Unreal Engine, which will call Quest System. We will describe the primary component of the Quest System, which the quest generator. The quest generator is a component that is attached to every quest giver NPC. The generator uses the grammar previously defined to produce a quest filled with game world content, that is motivated by the quest giver's intentions.

Initial Input

Every time a quest is generated, the algorithm must receive as initial input a grammar and a knowledge base. The grammar contains: A set of *Nodes*[2]; A set of *Rules*; 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 12. 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 : $head \rightarrow body$. 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 : $head \rightarrow body$. 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

[2]Remember the prototype generator created quests in the form of trees, where nodes represent possible actions.

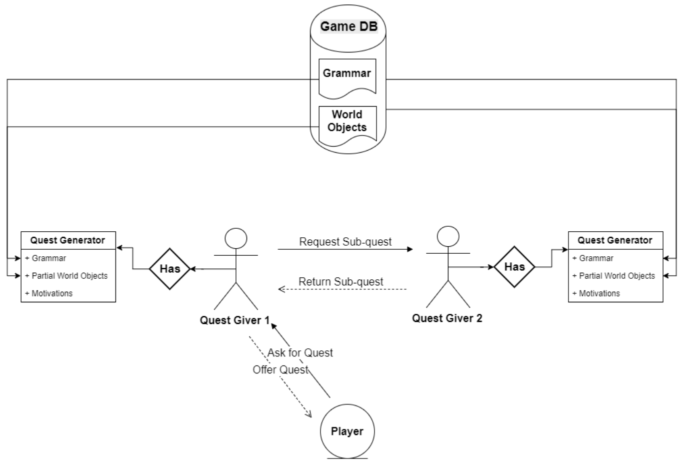

Figure 1. Quest Generation Model.

create a quest with actions like: *kill(Hyena)*, *goto(Cave)* and *learn(Recipe)*.

Algorithm

The algorithm, presented here, is capable of understanding the grammar previously described and use it to create different quests. Additionally, it will 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 the algorithm should be like, keeping in mind that the grammar was originally conceived to be used with Prolog.

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

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, by the game environment.

After selecting the rule/strategy, a node will be 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 subquests that take part in the quest being created, must be generated by other NPCs. These must have their own grammar, knowledge base and motivations. During node expansion, if the *NonTerminal* being evaluated is of type *Subquest* (see Table **??**), we must select another NPC that has a quest generator component and call the algorithm *CreateQuest* on it. The subquest generated should then be added to the main quest.

The grammar described in section "Related Work" was further extended to include weights related to the each sequence of actions in a *NonTerminal* node's set. Having weights gives control over the variety of actions sequences within the quest.

The algorithm *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

Algorithm 2 Expand Node

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

of actions from the set of rules of the *NonTerminal* node being expanded. To obtain the index number, first a random number *rand* (from 0.0 to 1.0) must be generated and then compared to a set of ranges. The sum of all weights related to the set of rules of a *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.

Algorithm 3 Weighted Choice

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

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 wieght being divided by the value of a converge function. The convergence function can be defined in several ways depending on how big one wishes the quest to be. For example it could be either: $\sqrt{depth}$, for slower convergence rate to 0; $2^{depth-1}$, for a moderate convergence rate; or $depth^2$, 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 chosing rules that add more *NonTerminal* nodes to the quest structure. That is 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 **??** 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 ε action(empty). 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 UPDATEWEIGHTS($rules, weights, index, depth$)
    $weights[index] = weights[index]/$CONVERGENCE(depth);
    $portion = (1 - weights[index])/NumberOfUpdates$;
    for all $weight \in weights$ do
        if $rules[weight.index]$.CONTAINSTERMINALS then
            $weight += portion$;
        end if
    end for
    return $weights$;
end function
```

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 *Expand* algorithm on each of them. The third and final algorithm is *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

```
function BINDPARAMETERS($node$)
    for all $child \in Children$ do
        for all $param1 \in child.Parameters$ do
            if $param1.Restriction == 0$ then                ▷ Parent Case
                for all $param2 \in node.Parameters$ do
                    if $param1.Type == param2.Type$ then
                        $param1.Target \leftarrow param2.Target$;
                    end if
                end for
            else if $parameters.Restriction == 1$ then        ▷ New Case
                $param1.Target \leftarrow$ NEWPARAMETER(param1.Type, K);
            else                                              ▷ Sibling Case
                for all $param2 \in node.Children[param1.SiblingIndex].Parameters$ do
                    if $param1.Type == param2.Type$ then
                        $param1.Target \leftarrow param2.Target$;
                    end if
                end for
            end if
        end for
    end for
end function
```

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

So these correspondences between parameters must be established while every node in the quest is expanded. Function 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 an 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.

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 2). 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 defined in the generated quest is being executed correctly.

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 to 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 **??** was also simplified, Conan Exiles is still in early access and does not possess enough mechanics to implement all 25 actions.

Quest Generator

As stated above, the quest system is divided in 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 it's UQuestGenerator component.

The UQuestGenerator will also have access to a reduced version of the grammar defined in Chapter **??**, as well as a knowledge base containing: a set of BaseNpcs, ConanExiles' Blueprint class for non-playable characters; a set of player-known Recipes[3]; and a set of GameItems.

[3]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 GameItem

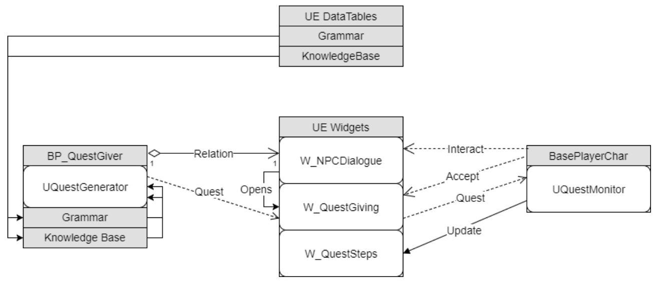

Figure 2. Mapping of the model to Unreal Engine/Conan Exiles.

Input

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 **??**), 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 3 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;

- an array of type *FQSRules* structure , containing an enumerator *NonTerminalNode* and 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 quest's structure, were removed from the grammar.

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 conversations 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 me crafted on the player were considered for the knowledge base.

The array for hostile NPCs is set by an event called "GetCloseNpcs". 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 one, 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.

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 *UQuestNode* also has a array of *UQuestNode*s 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 *UQuestNode* also contains an array of type *FNodeDetails* structure, representing action parameters. The structure is defined as follows:

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

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

A *UQuestStructure* also has 3 fields, that are only used by the *UQuestGenerator*, that represent the whole grammar: an array of type *FQSStrategies* structure; an array of type *FQSRules* structure . These represent the same as the ones in *UQuestGenerator*. Every time a quest is created, a copy from the related arrays in *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.

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 QuestMonitor then simply verifies if the current action has been completed. If it has been completed, the QuestMonitor notifies the UI W_QuestSteps, to show the next action in the quest.

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. 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 Event Dispatcher. These work in a similar way as C# delegates or C++ function pointers.

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 defined in our grammar.

To give a few examples: first, for the action kill, every time the player kills a character (this check is done in BaseBPCombat, which handles combat mechanics for every AConanCharatcer), it will call the event dispatcher, "SignalPlayerKilledCharacter". 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 (this process also happens with actions that require multiples of the same UObject, like gather, take, etc.); second, for the goto action), every time the player moves (also handled in BaseBPCombat), 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 the is within a certain range, then event considers the player is in the same location and ends; last, for the action give, every time it is required for the player to deliver an item to an NPC (which in this and several other cases it is an NPC of type BP_QuestGiver), it will add a button to the UI window W_NPCDialogue (already mentioned in this chapter). The process of adding a button happens in several other actions as well, like listen, report, exchange, etc. Actions that require direct interaction with an Npc. Once the UI window 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.

Having confirmed that the current action was correctly performed, the UQuestMonitor will proceed to the next action in the quest, which 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. 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.

EVALUATION

The tests aim to evaluate two aspects of the player's experience in terms of player enjoyment and game-play flow, thus, validating the work presented in this document[4]. 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 play 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 quest, and complete it, 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]"Enjoyment" was gauged by 10 items such as: "I had fun playing the game" or "I felt bored playing the game (R)". "Flow" was also gauged by 10 items such as: "The gameplay ran fluidly and smoothly" or "I didn't notice time passing".

	Conan Exiles		**Quest System**	
	Enjoyment	Flow	Enjoyment	Flow
Average Score	5,252	5,229	5,178	4,891

Table 1. Mean scores of the variables for each experience.

The order effects of repeated measurements 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. Each player was required to play for about an hour and ten minutes, distributed through three sessions: a brief tutorial, and both scenarios previously described, using repeated measure with counter balance. The first session lasts about 10 minutes and the latter two last 30 minutes each.

We had 22 participants 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. 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.

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

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 Likert scale. 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. 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 **??** and **??**. The overall conclusion of this tests suggests that the inclusion of our quest system did not worsen the overall experience for the players. We believe that these results are very reassuring, especially if take 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.

CONCLUSION

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[1] 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. NPCs create a quest

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 a more content variety. Concomitantly we think they may reduce the time and financial costs associated to that development.

FUTURE WORK

There are several ways of continuing the work executed throughout this project, but we are just going to point out the most important ones. 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. Also, the system currently works in a centralized fashion. One NPC generates the whole quest, which includes other NPCs' 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.

REFERENCES

1. Jonathon Doran and Ian Parberry. 2011. A prototype quest generator based on a structural analysis of quests from four MMORPGs. *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games - PCGames '11* (2011), 1–8.

2. Funcom. 2017. Conan Exiles. (2017).

3. Ken Hartsook, Alexander Zook, Sauvik Das, and Mark O. Riedl. 2011. Toward supporting stories with procedurally generated game worlds. *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011* (2011), 297–304.

4. Mark Hendrikx, Sebastiaan Meijer, Joeri V A N D E R Velden, and Alexandru Iosup. 2011. Procedural Content Generation for Games : A Survey. 1, February (2011), 1–24.

5. Boyang Li and Mo Riedl. 2010. An Offline Planning Approach to Game Plotline Adaptation. *Association for the Advancement of Artificial Intelligence-Aiide* (2010), 45–50.

6. A. Machado, P. Santos, and J. Dias. 2016. Towards a Procedurally Generated Experience: A Structural Analysis of Quests. *VideojogosâĂŹ16, CovilhÃč, Portugal* (November 2016).

7. Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. 2011. Search-based Procedural Content Generation : A Taxonomy and Survey. (2011), 1–15.