A Server-Side Framework for the Execution of Procedurally Generated Quests in an MMORPG

Jonathon Doran
Ian Parberry

Technical Report LARC-2015-01

Laboratory for Recreational Computing
Department of Computer Science & Engineering
University of North Texas
Denton, Texas, USA

February, 2015

A Server-Side Framework for the Execution of Procedurally Generated Quests in an MMORPG

Jonathon Doran
Dept. of Computer Science
and Information Systems
Bradley University
jhdoran@bradley.edu

Ian Parberry
Dept. of Computer Science
and Engineering
University of North Texas
http://larc.unt.edu/ian

ABSTRACT

We describe a framework for executing procedurally generated quests implemented in the MMORPG Everquest using the Open Source EQEmu Everquest server. Quests play out at run-time using a collection of *triggers*, which consist of a testable game state condition and a script that is to be run when the condition is satisfied. We describe the interface between the quest generator and the server which enables the seamless integration of the procedurally generated quests within the existing server architecture. To demonstrate how this process takes place in real time, we analyze a nontrivial procedurally generated quest and describe the key server-controlled actions that derive from it.

1. INTRODUCTION

Massively Multiplayer Online Role Playing Games (commonly abbreviated MMORPG) pose a significant challenge for procedural content generation. Of the content which might be procedurally generated, the *quest* is perhaps the most difficult. Quests are tasks assigned to players in the game, combining narrative elements and problem solving with combat and survival in a hostile world. Creating these quests requires the creation of in-game agents, items, and dialog. A quest generator must inform the game engine about which tasks need to be performed by both the player and the engine, what the criteria for success is, and when events should occur.

Procedural quest generators place a number of requirements on a game server. These can arise from the need to have the generator run with little or no human intervention, the need to introduce new quests into a world without breaking any existing functionality, and the need to remove quests without negative consequences. We demonstrate the design and implementation of a procedural quest generator for the MMORPG Everquest using an Open Source emulated server and an Everquest client released by Sony Online Entertainment via Steam in 2010 (See Figure 1). We believe that our framework is general enough to be adaptable to other server architectures provided certain requirements are met.

The remainder of this paper is divided into five sections. In Section 2 we consider related work. In Section 3 we briefly describe the procedural quest generator. In Section 4 we discuss the specific requirements that the Everquest server imposes on the execution of procedurally generated quests at run-time. In Section 5 we describe the interface between the quest generator and the Everquest server. In Section 6 we show how we were able to meet the server's requirements and control quest execution. In Section 7 we analyze a sample procedurally generated quest and describe how the server is able to make the quest play out in real time.

2. RELATED WORK

Massively Multiplayer Online Role Playing Games (MMORPGs) are persistent interactive worlds shared by many players. Players face many challenges in these games, among them are structured activities known as quests. Quests consist of objectives, tasks, and success or failure conditions (see Ashmore and Nitsche [2], and Doran and Parberry [7]). Dickey [5] notes that players strategize, collaborate, and plan their solutions to these challenges as a major form of gameplay. Rewards from quests are often the primary motivation for players to engage in gameplay, and in these cases a compelling story is not required. Quests can play a significant role in content delivery by providing narrative and guiding player involvement in the world (see Grey and Bryson [8], Joslin, Brown, and Drennan [9], Smith et al. [13], and Tomai, Salazar, and Salinas [15]). Quests can be used to relate epic stories (see, for example, Bateman and Boon [3]). One of our long-term goals is to determine if improved storylines can result in players focusing as much on the storyline as the reward.

There is no intrinsic meaning for quests, only potential meaning (Tronstad [17]), which means that discovery of this meaning is a task for the player. Tronstad also notes that as quests are a search for meaning, once solved they cannot be performed again, since we only have one opportunity to experience a quest for the first time. To maintain a body of novel content, one must constantly introduce new content into the game. Procedural quest generation is a logical solution, if the generators are capable of providing the meaning of which Tronstad speaks (see Ashmore [2], Kybartas [10], Reed et al. [11], Sullivan, Mateas, and Wardrip-Fruin [14], Tomai, Salazar, and Salinas [16], and Zook et al. [19]).

Aarseth's [1] analysis of the quests in Everquest has suggested that the player's task is finding "one acceptable path", which is evidence that player agency is limited (see also Wardrip-Fruin et al. [18]). We can posit that the introduction of additional paths would be of great benefit to playability.

The reduction of resource requirements is important for architectures which need to scale, such as those found in

MMORPGs. In the study of computer networks we have seen how explicit notification of relevant events can reduce unwanted traffic, leading to less resources needed to process this traffic (see Smed, Kaukoranta, and Hakonen [12]). The reduction of traffic in networks is an appropriate model for reducing event handling in a game. In both situations we consider solicited versus unsolicited event notification and handling. This publisher/subscriber pattern was also discussed in terms of client/server communications by Caltagirone et al. [4]. We have adopted this technique by requiring quests to subscribe to certain types of events at appropriate points during their execution.

3. QUEST GENERATION

We generate quests using the technique previously described by the authors in Doran and Parberry [7], starting with an NPC and selecting a strategy template appropriate for its motivation. These templates are part of a plan library, and can to be expanded to the desired level of complexity. Complications and follow-on quests can be added to a template, causing more strategies to be used. These new strategies can also be expanded as needed.

The generator was modified slightly to assign a number of *points* to strategies representing their difficulty, and randomly dividing these points among new strategies during the expansion process. If we view the set of strategies used as a graph, we see a tree structure growing from the original root goal. Points over a minimum value are allocated randomly to leaves, which then add child nodes that become new leaves. Points are consumed by each of the strategy templates, so the initial point total limits the size of the generated tree.

This differs from planning algorithms, as we start with a viable solution in the form of a trivial goal, and add additional subgoals while preserving the overall strategy; thereby by adding obstacles to the path the player must take to satisfy the original goal. This might be done by making a needed asset hard to obtain, relying on knowledge that is not obvious or commonly known. This process will continue as long as necessary to consume the points allocated to the new branches. Planning algorithms, on the other hand, start with an initial state and a goal and attempt to generate a graph that connects the initial state to the goal state. There is no guarantee that a viable solution exists, and computationally expensive search techniques must be employed to find any solution. Unlike planning solutions, our approach builds solutions in constant time.

If quest generation fails, which might be due to requiring more points for a branch than are available or exhausting available world assets, the quest generator employs backtracking to attempt an alternate solution. In practice we found that the limiting factor in creating large quests is the size of the world knowledge base. For example, a small, finite set of world locations is inadequate unless the generator can reuse locations, but it is preferable to avoid this reuse to help keep quests believable while preserving variety.

Novelty in procedurally generated content is a very important quality, as it creates the variety of content which players desire (Doran and Parberry [6]). This variety is obtained by

Figure 1: A screen shot of the Everquest client.

changing the subquests (or nodes) created, and the details of each node, such as assets and dialog. By changing the distribution of points among quest nodes, we permit different tree topologies to be created and change the difficulty of the subquests generated with each node. Asset selection (such as NPCs and items) can also have a significant impact on the structure and appearance of a quest. NPCs in particular have motivations which limit the types of subquests possible from the node referencing the NPC. In general, the number of unique combinations of assets and nodes increases significantly with the number of points allocated.

4. SERVER REQUIREMENTS

Our preliminary work on quest generation (Doran and Parberry [7]) dealt with the generation of quests in isolation. Interfacing a quest generator to a large, commercial quality MMORPG game engine is a complex task that taxes our abstractions to the limit. We selected the game Everquest for this purpose because the client could easily be purchased, and there is an available open-source server emulator EQEmu[1].

An EQEmu quest is implemented as a set of Perl or Lua scripts associated with the corresponding game assets. Each script may declare a handler for any of the supported events and, by interacting with server objects, change the state of the world. To simplify quest management, we enforced the requirement that each quest be implemented with a single script that can be added or removed from the server without impacting any other quests.

The run-time support for the quests created by our generator makes use of information stored in a database. Each quest is represented by a single file containing the information needed to create the script and perform the necessary database updates. We chose to implement this file as an XML document. A quest can therefore be shared with another server by sharing the XML file created by our generator.

The run-time support for quests will require information on game events as they occur. The exact events, and the data associated with them, will of course be different from game to game. In general an event will correspond to some state

[1]`http://www.eqemulator.org`

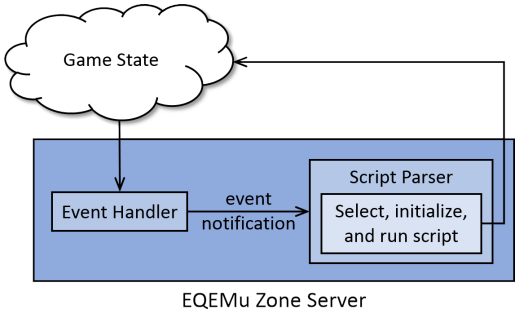

Figure 2: The EQEmu zone server responds to game events by having an event handler send a notification to a script handler, which selects, initializes, and runs a script.

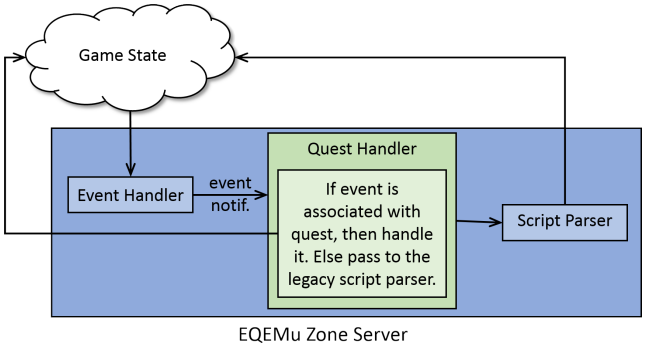

Figure 3: We modify the EQEmu zone server so that our quest handler has the first opportunity to respond to game events.

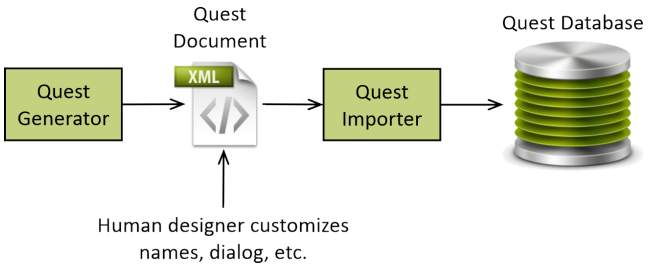

Figure 4: Getting the quest from the quest generator to the quest database.

change in the world, for example, this could be a non-player character (NPC) entering or leaving the world, an item being acquired, or the player visiting some location. The run-time support for quests created by our generator requires custom event handling. The previous EQEmu server generated events by sending an enumeration and several parameters to script parsers, which then created an appropriate initial state for a event handler script and then called one of the scripts associated with an asset (see Figure 2).

This sequence of actions requires that the code that notifies the system of an event must know the type of asset that will handle the event; additionally, any optional parameters must be passed to the script parsers. The resulting event handling code is spread over several classes, and knowledge of how a given event is to be be handled is required of the code that raises the event. Our approach is similar, but stores all information associated with an event into an Event object which is passed to the Trigger Manager. This simplifies the script interface, and provides a general event interface which can easily be extended if new events are desired.

Events are represented in EQEmu by discrete objects that are wholly self-contained. Each event is passed into common event handling code which determines the proper event handler and makes the necessary calls to process it. The system can support as many custom event handlers as necessary. Our run-time is given the first chance to handle each event, and in the case of failure, the event will be passed back to the legacy asset scripts (see Figure 3). Multiple quest systems can coexist at run-time without risk of interference.

5. THE SERVER INTERFACE

As described above, our quest generator produces a single XML document for each quest, which document must then be loaded onto the game server. We implemented a Java quest importer (see Figure 4) that processes these XML documents and, based on the information inside, either adds or removes a quest from the server. This application creates the run-time scripts based on the XML elements and updates the game server database. These operations are obviously game-dependent, but the principle is common to all MMORPG engines: The quest generator must communicate quests to the server using some combination of flat files or databases. Our XML format can in principle be easily extended to other forms of data storage used by a game. Notice that although the generator supplies the structure of the quest, some input is required from a human designer to customize things such as NPC dialog and the names of NPCs and items.

For convenience we define a structure called a *trigger*, which consists of a test to be applied against the game state and a script to be executed if the test succeeds (see Figure 6, bottom left). For example, the test could specify player arrival at a certain location, and the script could result in an NPC at that location giving an important object to the player. While in principle the firing condition may be an arbitrary Boolean formula involving any number of game state variables, certain triggers are more common (see Table 1). The null trigger, which fires immediately upon creation, is a useful way to compose a sequence of actions.

Figure 5 shows the structure of the XML file produced by

Type	Firing Condition
null	immediately
item	item is created
converse	player conversation matches regular expression
give	player gives an item to an NPC
proximity	player enters a certain area
acquire	item enters player's inventory
subquest	subquest completes

Table 1: Some common trigger types and their firing conditions.

```
<?xml version="1.0" encoding="utf-8"?>
<quest>
  <title lang="en">graph_0</title>
  <id>5a729d34-30c3-11e4-a10d-001d7d0a5e7c</id>

  <node>
    <name>Root</name>
    <task>gather parts for a Simple Pauldron</task>

    <assets>
      <item>
        <id>\$item_3</id>
        ...
      </item>
      ...
    </assets>

    <triggers>
      <match>
        <id>hail_1</id>
        <zone>394</zone>
        <sequence>0</sequence>
        <regex lang="en">\bhail\b</regex>
        <Perl> ... </Perl>
        <task>speak with Blacksmith Jones</task>
        <repeatable>
      </match>
      ...
    </triggers>
    ...
  </node>
  ...
</quest>
```

Figure 5: Structure of a quest file

the generator. Each quest is given a title, which is used by the importer to display a list of quests. A GUID is assigned by the generator, so that quest names and quest ids do not need to be globally unique. When a quest is loaded onto a server, it is assigned a locally unique identifier (such as a counter incremented for each unique quest loaded), and the GUID and title are associated with this identifier. All database modifications are logged with the corresponding quest identifier, allowing the importer to later remove the quest.

Our generator represents quests as graphs, which structure can be seen within the XML document. The Quest Graph consists of a set of nodes with triggers that correspond to graph edges (see Figure 6, right). Each node is assigned a name and an optional task text. The task text can be

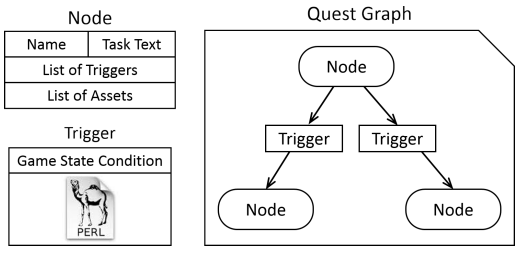

Figure 6: A Quest Graph node (top left), a trigger (bottom left), and a quest graph (right).

Figure 7: A screen shot of the Everquest journal.

used within a game to identify quest steps, as we do with Everquest's quest journal (see Figure 7). Each node contains a set of assets that need to be created at runtime, and a set of triggers (see Figure 6, top left). The first use of an asset causes an asset record to be written, and further use of the asset can be performed by reference to the asset id. Game-specific properties are included in this asset record, but we assume that in general any game will assign some set of properties to any object in the world. Our importer creates database entries for quest-specific assets, introducing them into the game and allowing characters to interact with them. If a quest is later removed, the database entries for intermediate (non-reward) items are also removed and these items disappear from the world.

We can consider each quest to be a finite state machine in which the triggers are the events that can change its state. For example, trigger hail_1 has the type "match" which requires player speech match a regular expression given in the trigger. In Figure 5 we see a trigger which requires the word "hail" to be spoken by the player before the quest will advance. This is a typical trigger word used in Everquest. The Perl element contains functions written by the generator that will be executed if the regular expression is matched, possibly providing a spoken response or other NPC action. This combination of an arbitrary set of triggers and a very capable scripting language can allow any event which might occur in a game to be paired with any server-side responses that might be required. All game-specific logic is contained in the meta-rules in the quest generator, which are separate from the general rules which might apply to any game.

Each trigger is assigned a sequence number that indicates the order in which triggers are required to fire. It is possible to have several triggers with the same sequence number, and therefore able to be performed at the same point in time. That is, triggers are partially ordered and allow the player to choose which parts of the quest they will work on next. In Section 6 we will show how these sequence numbers are used at run-time.

Triggers may be marked as repeatable and/or optional, allowing different combinations of trigger firings to be specified by the generator. Repeatable triggers are needed at points where the player may restart the quest following a failure

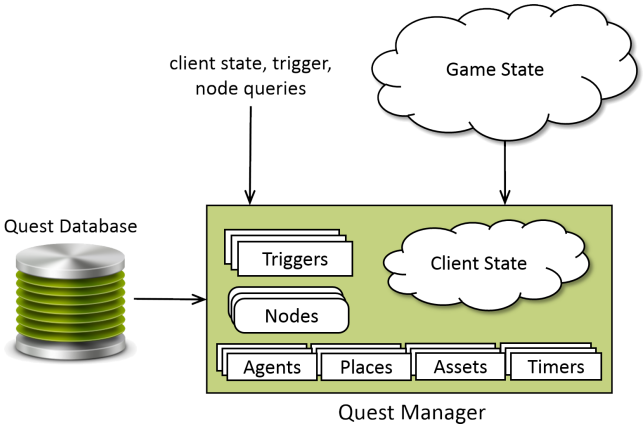

Figure 8: The quest manager manages all of the things associated with a quest, including client state, triggers, graph nodes, timers, and world entities such as places, agents, and assets.

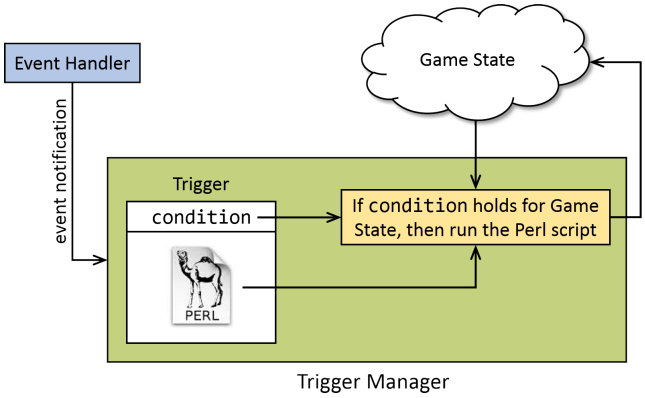

Figure 9: When notified of an event by the EQEmu event handler, the trigger manager determines which triggers should fire by testing them against the game state, then executes their scripts.

to complete a later quest stage. For example, a repeatable trigger can be used to permit the initial conversation with an NPC to be repeated if the player fails to advance beyond the first checkpoint. Without checkpoints (and the corresponding rollback logic), the player is committed to either complete the next quest stage or fail the quest. Optional triggers are for events that might occur and necessitate a response, but which are not required to successfully complete the quest. Node, asset, and trigger data are stored in a database for our Everquest server to access. The PERL functions are collected into a single PERL script, which exists in a quest-specific namespace. This means that function names only need to be quest-unique, simplifying the process of working with multiple generated quests.

6. THE QUEST & TRIGGER MANAGERS

Although the emulated Everquest server was initially capable of processing events and performing quests, it was not able to work with quests produced by our generator. Modifications were made to the server to allow it to execute the procedurally generated quests loaded by the process described in section 5. We created a *quest manager* which provides an interface to scripts, and manages client state, timers, and registrations of world entities that need to be notified when global events (not associated with any client) occur (see Figure 8). For example, NPCs that move along a route of waypoints need an event to be generated when a waypoint is reached. This event triggers the manager to assign the NPC the next waypoint in the route.

The quest manager also has the ability to checkpoint and rollback quest state in the event that part of a quest needs to be repeated. For example, if a player obtains an item needed to complete a quest and then manages to somehow lose it, the quest state is rolled back to the point prior to the player obtaining the item. The quest manager keeps track of modifications to the local world state, so that these can be removed when the quest advances or the player leaves the zone. One of the local modifications supported is selective visibility, which makes assets only visible to players associated with the quest. Association means that the player has the quest, and is at the proper point in the quest to see the asset, or that the player is grouped with someone meeting those requirements. This allows groups of players to cooperate on quests without affecting other players. The quest manager was implemented as a singleton pattern, and therefore exists in its own globally accessible namespace.

Cache managers were created for nodes and triggers that are active in the world. An active node or trigger is one with at least one player at the corresponding graph node, or waiting to complete the trigger. This optimization allows events to be screened against active objects rather than all objects associated with a quest.

The *trigger manager* handles all events generated by the game server, and attempts to match them with active triggers (see Figure 9). The match occurs when the event meets all of the firing requirements, and there is at least one player with the trigger active. Upon detecting a match, the trigger is said to fire and the trigger-specific script is executed. The trigger may or may not complete as a result of this firing. Some triggers require multiple firings before they complete, such as one might find when a player is asked to collect several objects in a set. Each collection advances the state of the trigger, until the terminal state is reached and the trigger completes. Upon completion the node associated with the trigger is notified, and the trigger is deactivated. If there are no outstanding triggers required by this node, the node can then complete and advance the quest state to the node at the next sequence. When the terminal node in the graph completes, the quest ends. It is assumed that the terminal node takes care of any rewards associated with the quest.

7. AN EXAMPLE QUEST

The capabilities of this system can be seen by viewing a sample quest generated by our generator and playable in-game in Everquest. The overall structure of the quest is shown in Figure 10, where the quest starts at the Root node, and the player is required to complete subquests represented by other graph nodes either as prerequisites or postrequisites. In this example, all nodes represent follow-on quests to be completed as part of, or after the preceding node. The quest

starts with the character Blacksmith Jones asking the player to gather materials and make a piece of armor. This requires the player to obtain metal panels and a venom sac from a poisonous snake. These ingredients are determined randomly, and the recipe is only usable by the player performing the quest. The player is directed to see Councilmember Ithakis for the metal panels, and the Councilmember offers to give the player the panels in return for a favor. The Councilmember wants the players to locate a lucerne leaf (another ingredient which could be used to make armor), and suggests that the player ask Farmer Jones for help. The farmer is happy to give the player a leaf, which is then turned over to the Councilmember. The players are then asked to deliver a message to a character named Nech Ilya, saying that Councilmember Ithakis has the lucerne he needs. After this is completed, the player receives the metal plates.

Blacksmith Jones suggests greenscale vipers might be a good source of venom sacs, and the players must find some of these snakes in the world and kill them until they find one with a rare intact venom sac. With this item, they are able to create the custom armor piece for the Blacksmith, and earn their reward.

Generation of this quest requires the selection of appropriate tasks each NPC would like performed, and the creation of custom items for the quest. Special metal plates and venom sacs are only available to the player running the quest, or any character in the same group or raiding party as the player running the quest. Custom character dialog is created for each participating character, as well as control records which bring the items and characters into the world at the appropriate time.

The quest plays out as follows:

1. When the player enters the Crescent Reach zone, a converse trigger is created, requiring the player to hail Blacksmith Jones.
2. The player hails Blackmith Jones, activating the converse trigger.
3. Blacksmith Jones asks the player to help gather materials for a piece of armor that he is making.
4. Several subquest triggers are activated, causing null triggers at the start of each subquest to fire.
   (a) The null triggers deliver instructions for each subquest.
5. Blacksmith Jones suggests that the player ask a Councilmember for help getting metal panels.
   (a) An item trigger is created and activated requiring 6 metal panels.
6. Councilmember Ithakis demands that the player run an errand (perform a subquest) in return for the metal.
   (a) A subquest trigger is created and activated.
7. Councilmember Ithakis asks the player to bring him a lucerne leaf, which is used as an armor temper.

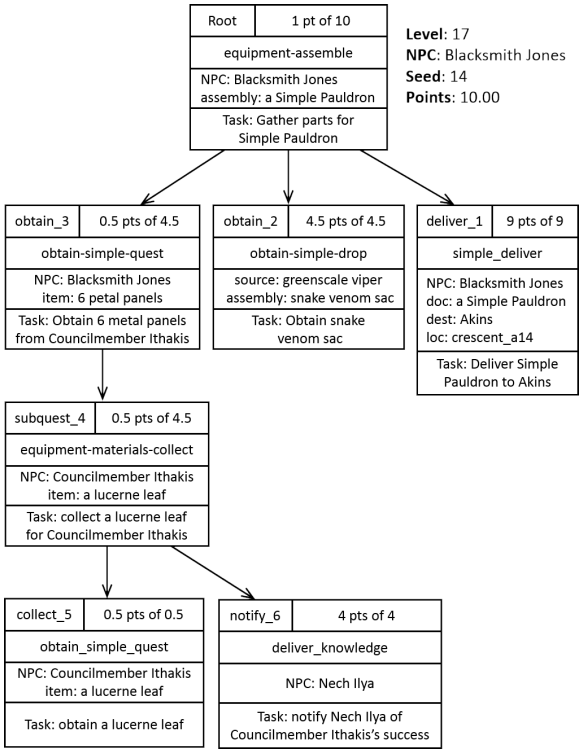

Figure 10: The structure of a sample quest.

   (a) A null trigger activates and fires, causing the council member to ask for a lucerne leaf.
8. Councilmember Ithakis directs the player to Farmer Joen, who gives the player the leaf.
   (a) A collection subquest is activated, which activates another null trigger which in turn directs the player to Farmer Joen.
   (b) A converse trigger is activated, looking for the player to mention "need" to Farmer Joen.
   (c) When the player says the magic word, the converse trigger fires causing Farmer Joen to give the player a leaf.
   (d) A give trigger activates, requiring the player to deliver the leaf to Councilmember Ithakis.
9. Councilmember Ithakis asks the player to tell an NPC named Nech Ilya about finding lucerne leaves.
   (a) Councilmember Ithakis demands that the player inform Nech Ilya that he (Councilmember Ithakis) now has a lucerne leaf.
   (b) A subquest trigger activates, which in turn activates a proximity trigger around the area where Nech Ilya will appear.
10. The player must find Nech Ilya, and speak with him.
   (a) When the player enters the area covered by the proximity trigger, it fires
   (b) A signal is scheduled which will spawn Nech Ilya.

(c) Another signal is scheduled, which will periodically print a random tracking message and then reschedule itself.

(d) Eventually Nech Ilya spawns, and all of the signals are canceled.

(e) A converse trigger is activated.

(f) When the player hails Nech Ilya, the converse trigger fires. Nech Ilya thanks the player, and the subquest completes.

11. The next time the player meets him, Councilmember Ithakis gives them the metal panels.

    (a) A null trigger fires, causing Councilmember Ithakis to deliver the message.

12. Blacksmith Jones suggests that the player hunt greenscale vipers to obtain a venom sac.

    (a) A null trigger fires, causing Blacksmith Jones to deliver the message.

    (b) An acquire trigger is activated, waiting for 1 snake venom sack to enter the player's inventory.

13. The player must locate these snakes and begin killing them. The snakes will rarely have an intact venom sac once killed.

    (a) When a venom sac enters the inventory, the acquire trigger fires and the subquest completes.

14. The player delivers the materials to Blacksmith Jones, who makes the armor piece.

15. Blacksmith Jones gives the new armor piece to the player, and asks that they deliver it to an NPC named Akins.

    (a) A give trigger activates, waiting for the player to give Akins the armor piece.

16. The player finds Akins and gives him the armor.

    (a) When the player gives Akins the armor, the give trigger completes, and the subquest completes.

17. Upon returning to Blacksmith Jones, the quest completes and the Blacksmith rewards the player.

8. CONCLUSION AND FUTURE WORK

We have demonstrated how procedurally generated quests can be implemented in Everquest. In its current state, the framework requires some input from a human designer in the form of character names and dialog, which suggests that we explore dialog generation techniques. At the same time we observe that while a narrative can be written to explain the quest graph, we suspect that players might prefer a more traditional story arc.

References

[1] E. Aarseth. Quest Games As Post-Narrative Discourse. *Narrative Across Media: The Languages of Storytelling*, pages 361–376, 2004.

[2] C. Ashmore and M. Nitsche. The Quest in a Generated World. In *Proc. 2007 Digital Games Research Assoc.(DiGRA) Conference: Situated Play*, pages 503–509, 2007.

[3] C. Bateman and R. Boon. *21st Century Game Design (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA, 2005.

[4] S. Caltagirone, M. Keys, B. Schlief, and M. J. Willshire. Architecture for a Massively Multiplayer Online Role Playing Game Engine. *Journal of Computing Sciences in Colleges*, 18(2):105–116, 2002.

[5] M. D. Dickey. Game Design and Learning: A Conjectural Analysis of How Massively Multiple Online Role-playing Games (MMORPGs) Foster Intrinsic Motivation. *Educational Technology Research and Development*, 55(3):253–273, 2007.

[6] J. Doran and I. Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, 2010.

[7] J. Doran and I. Parberry. A prototype quest generator based on a structural analysis of quests from four MMORPGs. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, pages 1–8. ACM, 2011.

[8] J. Grey and J. J. Bryson. Procedural Quests: A Focus for Agent Interaction in Role-Playing Games. In *Proceedings of the AISB 2011 Symposium: AI & Games*, pages 3–10. University of Bath, 2011.

[9] S. Joslin, R. Brown, and P. Drennan. Modelling Quest Data for Game Designers. In *Proceedings of the 2006 International Conference on Game Research and Development*, pages 184–190. Murdoch University, 2006.

[10] B. Kybartas. *Design and Analysis of ReGEN*. PhD thesis, McGill University, 2013.

[11] A. A. Reed, B. Samuel, A. Sullivan, R. Grant, A. Grow, J. Lazaro, J. Mahal, S. Kurniawan, M. A. Walker, and N. Wardrip-Fruin. A Step Towards the Future of Role-Playing Games: The SpyFeet Mobile RPG Project. In *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[12] J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of Networking in Multiplayer Computer Games. *Electronic Library, The*, 20(2):87–97, 2002.

[13] G. Smith, R. Anderson, B. Kopleck, Z. Lindblad, L. Scott, A. Wardell, J. Whitehead, and M. Mateas. Situating Quests: Design Patterns for Quest and Level Design in Role-Playing Games. In *Interactive Storytelling*, pages 326–329. Springer, 2011.

[14] A. Sullivan, M. Mateas, and N. Wardrip-Fruin. Making Quests Playable: Choices, CRPGs, and the Grail Framework. *Leonardo Electronic Almanac*, 2011.

[15] E. Tomai, R. Salazar, and D. Salinas. A MMORPG Prototype for Investigating Adaptive Quest Narratives and Player Behavior. In *Proceedings of the International Conference on the Foundations of Digital Games*. ACM, 2012.

[16] E. Tomai, R. Salazar, and D. Salinas. Adaptive Quests for Dynamic World Change in MMORPGs. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 286–287. ACM, 2012.

[17] R. Tronstad. A matter of insignificance: The MUD puzzle quest as seductive discourse. *CyberText Yearbook*, 2002–2003.

[18] N. Wardrip-Fruin, M. Mateas, S. Dow, and S. Sali. Agency Reconsidered. *Breaking New Ground: Innovation in Games, Play, Practice and Theory. Proceedings of DiGRA 2009*, 2009.

[19] A. Zook, S. Lee-Urban, M. O. Riedl, H. K. Holden, R. A. Sottilare, and K. W. Brawner. Automated Scenario Generation: Toward Tailored and Optimized Military Training in Virtual Environments. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 164–171. ACM, 2012.