PROCEDURAL GENERATION OF CONTENT FOR
ONLINE ROLE PLAYING GAMES

Jonathon Doran

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

August 2014

APPROVED:

Ian Parberry, Major Professor
Armin Mikler, Committee Member
Robert Renka, Committee Member
Robert Akl, Committee Member
Barrett Bryant, Chair of the Department
of Computer Science and
Engineering
Costas Tsatsoulis, Dean of the College of
Engineering
Mark Wardell, Dean of the Toulouse
Graduate School

Video game players demand a volume of content far in excess of the ability of game designers to create it. For example, a single quest might take a week to develop and test, which means that companies such as Blizzard are spending millions of dollars each month on new content for their games. As a result, both players and developers are frustrated with the inability to meet the demand for new content.

By generating content on-demand, it is possible to create custom content for each player based on player preferences. It is also possible to make use of the current world state during generation, something which cannot be done with current techniques. Using developers to create rules and assets for a content generator instead of creating content directly will lower development costs as well as reduce the development time for new game content to seconds rather than days.

This work is part of the field of computational creativity, and involves the use of computers to create aesthetically pleasing game content, such as terrain, characters, and quests. I demonstrate agent-based terrain generation, and economic modeling of game spaces. I also demonstrate the autonomous generation of quests for online role playing games, and the ability to play these quests using an emulated Everquest server.
ACKNOWLEDGMENTS

I would like to express my thanks to my thesis advisor, Professor Ian Parberry, for his support and guidance through the entire graduate process. In addition to Dr. Parberry, I would also like to thank my other committee members: Dr. Robert Akl, Professor Armin Mikler, and Professor Robert Renka, for trusting me to complete what must have seemed to be an overambitious project.

And of course, my parrot Sasha, who probably won’t read this, but deserves thanks for her patience during some rather long days.
# TABLE OF CONTENTS

| ACKNOWLEDGMENTS                                                | iii |
| LIST OF TABLES                                                 | viii|
| LIST OF FIGURES                                                 | ix  |
| CHAPTER 1 INTRODUCTION                                          | 1   |
| 1.1. History of Wargames                                        | 1   |
| 1.2. Tabletop RPGs                                              | 3   |
| 1.3. Computer RPGs                                              | 4   |
| 1.4. Terminology                                                | 5   |
| CHAPTER 2 GAME CONTENT OVERVIEW                                 | 9   |
| 2.1. Introduction                                               | 9   |
| 2.2. NPCs – Characters and Creatures                            | 10  |
| 2.3. Terrain                                                    | 11  |
| 2.3.1. Natural Geometry                                         | 12  |
| 2.3.2. Cultural Objects                                          | 13  |
| 2.4. Storyline                                                  | 13  |
| 2.5. On-Line Artificial Intelligence                            | 13  |
| 2.6. Items                                                      | 14  |
| CHAPTER 3 PROCEDURAL CONTENT GENERATION                         | 15  |
| 3.1. Overview                                                   | 15  |
| 3.2. Potential Advantages                                       | 16  |
| 3.3. Potential Drawbacks                                        | 17  |
| 3.4. Use of Procedurally Generated Content                      | 17  |
| CHAPTER 4 TERRAIN                                               | 19  |
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 7.1</td>
<td>NPC Motivations</td>
<td>81</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Strategies for gaining knowledge</td>
<td>82</td>
</tr>
<tr>
<td>Table 7.3</td>
<td>Strategies for increasing comfort</td>
<td>82</td>
</tr>
<tr>
<td>Table 7.4</td>
<td>Strategies for increasing reputation</td>
<td>83</td>
</tr>
<tr>
<td>Table 7.5</td>
<td>Strategies for increasing Serenity</td>
<td>84</td>
</tr>
<tr>
<td>Table 7.6</td>
<td>Strategies for increasing Protection</td>
<td>85</td>
</tr>
<tr>
<td>Table 7.7</td>
<td>Strategies for increasing Conquest</td>
<td>86</td>
</tr>
<tr>
<td>Table 7.8</td>
<td>Strategies for increasing Wealth</td>
<td>86</td>
</tr>
<tr>
<td>Table 7.9</td>
<td>Strategies for increasing Ability</td>
<td>86</td>
</tr>
<tr>
<td>Table 7.10</td>
<td>Strategies for increasing Equipment</td>
<td>86</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>Figure 1.1</td>
<td>The board game Senet, pictured in a fresco found in Merknera’s tomb in Egypt. (Image courtesy of the Canadian Museum of History, 2009.71.904.1 a-b)</td>
<td></td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Senet board game and playing pieces. (Photograph by Keith Schengili-Roberts, Copyright 2004-2014 the Brooklyn Museum)</td>
<td></td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>An overview of RPG content</td>
<td></td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Coastlines produced by coastline agents with small (top left), medium (top right) and large (bottom) action sizes</td>
<td></td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Beaches produced by beach agents with (left to right) small, medium, and large beach width.</td>
<td></td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Mountains with narrower features (left), and with wider features (right).</td>
<td></td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Screen shot of a prototype designer interface.</td>
<td></td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Hill agents produce hills, similar to the way mountain agents produce mountains.</td>
<td></td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>River agents generated a dry river bed (left), and three rivers that meet at the ocean (right).</td>
<td></td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Responsibilities of an economic system.</td>
<td></td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>The system has stable long-term behavior, moving in and out of equilibrium.</td>
<td></td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Detail of the previous figure, showing similar behaviors among commodities.</td>
<td></td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Long-term supply/demand behavior, showing the system in a variety of regimes.</td>
<td></td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>An interruption in wood supply and subsequent recovery.</td>
<td></td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Agent profitability by type for the forest-fire scenario, demonstrating an orbit about the equilibrium.</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5.7. The distribution of agent roles over time. 68
Figure 5.8. Two random economies 69
Figure 6.1. Cindl’s Polar Bear Collection (from Everquest) 78
Figure 9.1. Point scaling with level 91
Figure 9.2. Waves of level 10 opponents vs level 30-37 characters 92
Figure 10.1. Prolog rules describing an alligator egg 94
Figure 10.2. Prolog rules describing a merchant NPC 95
Figure 11.1. Checking dynamic entities for edible properties 97
Figure 11.2. Obtaining the zone-id of an NPC 98
Figure 11.3. Random selection of items from a list 98
Figure 11.4. Generating positive random numbers in a Cauchy distribution 100
Figure 11.5. Selecting random NPCs with a given role. 100
Figure 11.6. Determining the proper object pronoun. 101
Figure 11.7. Division of points among subquests. 101
Figure 11.8. An example assembly definition, and code to select an assembly 102
Figure 11.9. One of the rules used to assemble lists of templates for potential encounters. 103
Figure 11.10. One of the rules used to create static humanoids. 104
Figure 12.1. An overview of quest generation and playing 106
CHAPTER 1

INTRODUCTION

*The beginning is the most important part of the work.*

Plato

Computer role-playing games are a recent invention, but the concept has been around for a very long time. Around 3100 BC, the board game Senet was played in Egypt [60]. Games such as Senet were believed to be used to teach “would-be rulers and nobles the ideas and tenets of combat” [85]. Figure 1.1 shows players playing Senet, and Figure 1.2 shows a Senet board with pieces. These wargames, like later role playing games were simulations of an alternate reality.

1.1. History of Wargames

In 1664, Christopher Weikmann publishes Koenigspiel, a chess variant used for developing combat strategies [58].

In 1780, Johann Christian Ludwig Hellwig publishes rules for a chess variant War Chess where squares correspond to a specific type of terrain. This is the forerunner of

Figure 1.1. The board game Senet, pictured in a fresco found in Merknera’s tomb in Egypt. (Image courtesy of the Canadian Museum of History, 2009.71.904.1 a-b)
miniature-based wargames [27, 84].

In 1811, Georg Leopold von Reiswitz introduces Kriegsspiel, a wargame played on a sand table [40, 19]. Sand tables had previously been used as military planning tools [58].

1913 H.G.Wells publishes rules for Little Wars, the first miniature-based game for the public [92].

1952, Charles Roberts creates the game Tactics, later published by the Avalon Game Company [40]. This was the first commercially successful board wargame.

1954-1955: J.R.R. Tolkien publishes *The Lord of the Rings*, which would later serve as the inspiration for fantasy-based role playing games.

1968: Dave Wesely works on non-zero sum multiplayer games, adopting Strategos by Charles Totten for entertainment rather than training. Wesely keeps Totten’s idea of an impartial referee. This would later become the game master (GM) or dungeon master (DM). Wesely produced a Napoleonic era wargame. [86].

1970: Dave Arneson combines medieval fantasy with wargaming, creating the Blackmoor world. This was a forerunner to Dungeons & Dragons (D&D). The tremendous popularity of ‘The Lord of the Rings’ combined with wargaming at its peak. Players were no longer limited to the historical world, but were free to enter a Tolkienesque world.
1971: Gary Gygax and Jeff Perren release Chainmail [34]. This was rules for a wargame in a medieval setting.

1974: Gary Gygax and Dave Arneson release Dungeons & Dragons [25]. This is the first modern role-playing game. Combines wargames, medieval fantasy, and the children’s game make believe.

In the mid 1970s we see the genre split into two: The traditional tabletop games, and computer role-playing games. Tabletop games were the most expressive of the games, but computer role-playing games became the more accessible.

1.2. Tabletop RPGs

The tabletop (or pen and paper) role-playing game is the gold standard by which computer RPGs are judged. These games allow open-ended play, where the players can attempt any action they can describe. In contrast, the computer RPG is limited by its programming.

All action in tabletop games is coordinated by a participant acting as a game master (GM), or dungeon master (DM) in D&D. “The gamemaster is akin to a play’s director, a novel’s author, a film’s editor, a legend’s storyteller, a performance’s actor, and a sporting event’s referee.” [40]. The GM creates the world and all of its inhabitants, then presents the players with descriptions of the world. The players, acting out the roles of characters in the world, describe their actions and the GM interprets this information and describes the results. Together these participants create a simulated world where almost anything could happen.

Each type of RPG has its own set of rules, which serve as rough guidelines for the GM when interpreting player actions. It is generally understood that the GM may override any rule when they believe it will help the story. Tabletop games usually make use of polyhedron dice as a source of randomness, where combinations of die rolls are used to create random values from desired probability distributions.

Sid Meier described a game as “a series of interesting choices” [61], meaning that at each point in time multiple, non-trivial choices must be available to players. Players will
ideally ask “what if I had made a different choice,” which encourages replay and experimentation. The advantage of a human moderated tabletop game is that there are no fixed limits on the number of choices, nor the complexity of these choices.

One of the benefits of a large set of choices is that it allows players to engage in behaviors they would not otherwise perform. “Implicit in the intention to engross is the idea of emancipation – that players, through identification with their characters, are freed from their social bonds.” [40]. In addition to experimenting with actions, player attitudes and motivations can also be the subject of role playing. “The gaming escape involves two related components: release from the constraints of self and release from restrictions on behavior” [19]. The novel and unexpected action is often what the player seeks, and the tabletop games can permit this form of experimentation.

1.3. Computer RPGs

It can be inconvenient to routinely play with other people, as is required of tabletop games. Schedules change, other players move away, leaving a player with an inconsistent and unreliable experience. With the availability of the microcomputer in the mid-1970’s, it was natural for people to consider the possibility of allowing the computer to replace some of the human elements. In particular, players began to look to the computer as a replacement for the human GM. The possible advantages are attractive: Computers are always available, and they never have any interests which conflict with the players. Their disadvantage is that it is currently impossible to provide the open-ended game experience which a human GM can create. Matt Barton explains this as the inability to deal with such an abstract problem: “At best, the computerized versions could simulate the mathematics of D&D combat and to some extent the strategy and exploration components, but the inherent abstractness and aloofness of the medium seemed to stop true role-playing at the gate” [8]. However recent games suggest that computers are becoming much better that handling these abstract problems.

Rusty Rutherford’s pedit5 for PLATO (1975) appears to be the first computer adaptation of the fantasy role playing game. This game featured dungeon exploration, magic,
and other fantasy elements [66]. PEDIT5 introduced the concept of the NPC to computer
games. This was a graphical game, providing a top-down view using the PLATO panel dis-
plays. Written at the University of Illinois. PLATO allowed one to create their own fonts,
so Rutherford was able to create icons for creatures and items in the game.

In 1975, Will Crowther created the Colossal Cave Adventure on a DEC PDP-10. This was based on his exploration of the Flint Mammoth Cave system. This was a notable
achievement due to the limited capabilities of the PDP-10.

In 1977, Zork was released. This was inspired by the Colossal Cave Adventure and
Dungeons & Dragons. Zork was the first game to feature an NPC opponent with some
sophistication.

In 1979, Space for the Apple II introduced a sci-fi RPG. Richard Garriott released
Akalabeth for the Apple II. Both of these games began to establish the microcomputer as a
serious gaming platform. Prior to this, mainframe computers were the most common gaming
platform.

1980 gave us ‘Rogue’, where ASCII graphics allowed the games to move away from
pure text into a limited graphic realm.

In each case, the simulations are a shadow of the gameplay one could expect with
human players. But by providing enough choices and high availability, they began to be seen
as credible substitutes. If you could not find a convenient gaming group, then one of these
computer RPGs was better than nothing. The limitations of the computer RPG became
challenges to overcome.

1.4. Terminology

Discussions of role-playing games involve the use of a lot of specialized terminology.
This is understandable, since one well-chosen term can replace a paragraph of explanation.
I provide the following definitions to assist the reader in understanding terms which will be
used in subsequent chapters.

• AC: armor class. A measure of the ability of a character to withstand physical
damage.
• Character: A person in a game world, who may either a player (PC), or a non-player character (NPC). A member of a game’s *dramatis personae*.

• Character class: Similar to a profession. A description of the abilities and roles that a character may take on.

• Content: The parts of a game which are not executable code. For example, the terrain, characters, items, dialog, story, and quests found in a world.

• Drop: (noun) An item which is obtained from (or dropped by) a defeated enemy in an RPG. See loot.
  
  Example: Bone chips are drops obtained from skeletons.

  (verb) The act of introducing an item into the game from a defeated enemy.
  
  Example: Polar bear cubs can drop regular bear pelts, or rarely polar bear skins.

• Dungeon: A maze-like environment, typically underground, found in fantasy role playing games. This is a setting where adventures occur.

• Faction: (1) A group of associated non-player characters. (2) The reputation one has with such a faction.

• GM: (see game master)

• Game master: Also known as a GM, dungeon master, or DM. A facilitator of action within a table-top role playing game. Typically this involves creating a story, managing the non-player characters, and determining the outcome of player actions. Multiplayer online roleplaying games also have game masters, but their role tends to be limited to managing game servers and customer service.

• Group: Two or more players who cooperate to achieve an in-game goal. By grouping, these characters share experience, and usually the rewards which come from successfully completing goals.

• KOS: Kill on sight. An extremely negative faction value causing characters in that faction to immediately attack.

• Level: A measure of power or difficulty of game content. Characters advance in level through successful encounters within the game. Opponents are selected based
on levels appropriate for the players they are intended to face.

- **Loot:**
  (noun) a treasure found within the game.
  (verb) the act of acquiring treasure.

- **Meta-content:** Content that describes content. For example, the rules in the knowledge base which govern a quest generator’s behavior.

- **Mob:** A shortened term describing mobile objects. A mob is a character or creature found in the game world.

- **Motivation:** A need or desire of an NPC that causes them to offer a player a quest. Motivation places some constraints on the actions that may be part of a quest. An example of one possible motivation would be the desire of an NPC to learn more about an enemy.

- **MMORPG:** Massively multiplayer online role-playing game. A computer RPG which can have a large number of players in the world simultaneously. These typically have one or more servers which players connect to over a network.

- **NPC:** (see non-player character)

- **Nodrop:** An item restriction preventing the player from dropping or trading an item to another character. The only way these items may be removed from inventory is to either destroy the item or use them in some stage of a quest.

- **Norent:** An item restriction preventing the player from saving the item in the player’s inventory across game sessions. This forces the player to acquire and use the item in the same session, which increases the difficulty of the quest.

- **Non-player character:** Also known as an NPC. A character in an RPG which is not controlled by any player. This type of character can be controlled by a game master (for tabletop play) or by a computer (for computer play). These characters tend to be able to interact with players, and therefore play a major role in providing game content to a player.
• Player: A human participant in a game, as opposed to a game master or non-player character.
• PC: (see player character)
• Player character: A player controlled character within a game, as opposed to a non-player character who is controlled either by a game master or a computer.
• Procedural content generation: Computer generated content, as opposed to human generated content. Generating this content is a creative act, and requires both random and structured behavior by a computer algorithm.
• Quest: A challenge for players which is overcome by a series of player actions. A quest usually benefits an NPC or faction in the game, and is motivated by the needs and desires of this NPC or faction. Games typically reward players for completing a quest.
• Raid: In the game Everquest, a raid is a collection of up to 72 characters who cooperate to complete some goal. Raid encounters are the most difficult content found within the game, for example slaying dragons.
• RPG (see role-playing game)
• Role-playing game: A type of collaborative storytelling game where one or more players take on roles of characters within a simulated or fictional world. Gameplay consists of having adventures in this world, exploration, combat, and character development. These types of games can be either a tabletop role playing game, a computer role playing game, or a live action role playing game.
• Strategy: In the quest generation language, a set of actions which accomplish some goal.
• Tradeskill: A character skill which allows items to be created (or crafted) within the game. Examples would include blacksmithing, baking, and tailoring.
CHAPTER 2

GAME CONTENT OVERVIEW

If you see me on Friday, you'll see different material on Saturday night.
Kathy Griffin

2.1. Introduction

Role-playing games (RPGs) make use of a lot of game content. Some of this content, such as items and non-player characters (NPCs) may already exist in the world, and some may need to be created to support other new content.

Content may be created for a new area of the world, or to augment an existing area. When approaching the topic of procedural generation of game content, one needs to have a clear idea of what content is, and what makes content acceptable for a given game. I divide content of a new world area into several types:

- NPCs: The characters and creatures created for an area of the world.
- Terrain: The terrain geometry and cultural features of the map.
- Storyline: The background and current stories involving an area of the world.
- Economics: Item values for an area of the world.
- Quests: Player tasks involving an area of the world.
- Items: The inanimate objects used in quests.
- Dialog: NPC dialog, usually in support of quests.

Figure 2.1 shows the influence relationship between different types of content. For example, the NPCs present in an area of the world depend on the terrain. Certain types of NPCs may require an aquatic environment, others might require a mountainous environment. Once the terrain has been determined, it will be possible to determine which types of NPCs may be placed in an area.

NPCs in turn help to determine the economics of an area, since I assume that they supply and demand resources. NPCs also help determine the available quests, as quests are assumed to exist for the benefit of an NPC.
Storyline decisions may require additional NPCs and dialog for these NPCs. Stories generally rely on characters, and at any time there is a finite number in the world.

Economic decisions influence storyline generation, since any storylines must be consistent with the availability of resources. For example, if metal is expensive or rare, then a game cannot have storylines which assume abundant metal. Changes in economics can influence the pool of NPCs, since it may not be feasible to support certain NPCs in an area. For example, a scarcity of metal would limit the number of blacksmiths in an area. We should expect that blacksmiths would disappear from the region, and there might be an increase in other NPC types such as woodworkers. The state of the economy also influences quests which can be created due to the changing value of quest rewards, intermediate items, and indirectly through a change in distribution of NPC types.

Quests may require additional NPCs and their dialog, as well as asset allocations. Again, a finite population can be exhausted. The ability to add new NPCs and items can prevent the overuse of individual assets.

2.2. NPCs – Characters and Creatures

The characters and creatures (mobs) found in a world provide opportunities for characters to interact with them. The most interactive characters will be those controlled by
other players. NPCs usually can have limited conversations with players, and can exchange
items or provide other services such as healing. The creatures found within the world are
the least interactive, and usually are limited to combat.

When creating characters, a generator must determine the location and physical attributes associated with the new character. For example, the characters appearance, health, and equipment carried by the character. The characters role in the world, motivations and goals are important to determine how it will interact with players. A quest generator will need to determine the types of information available to a character for dialog. Much of this information is influenced by the economic state of the region where the character is to be placed.

If a game engine is unable to dynamically generate dialog on the fly, the quest generator will need to create all dialog for the character, or defer the task to a human developer. In most cases it is possible to at least generate boilerplate dialog which a developer could later refine.

Non-interactive characters, such as creatures, are created as opponents for players as part of a planned encounter. When constructing an encounter, care must be taken that these opponents are neither too difficult, nor too easy for the player to overcome. In addition to the aesthetic properties of these creatures (which model to use), attributes that determine character performance must be defined. These include special combat abilities, health, and damage resistance. Some of this information is influenced by the local terrain. Techniques for determining these abilities will be discussed later in Chapter 9.

2.3. Terrain

Terrain is the natural world geometry, such as oceans, lakes, and dry land. This is the setting in which all game activity occurs. In the simplest form, terrain can be a heightmap giving the elevation of each discrete point on the map. A drawback with this technique is the inability to have locations that differ only in elevation, such as overhangs or caverns. Such constructs would permit a player to stand at the same X-Y coordinate either at an underground location, or at an above ground location, and therefore violate the 1:1 relation
of coordinates to elevation. More advanced terrain engines represent the terrain as voxels, or a large 3-D model built out of polygons. This allows the construction and representation of arbitrary shapes, but requires more computing power.

2.3.1. Natural Geometry

A world’s terrain will imply obstacles, as certain types of terrain can be impossible to navigate over. Cliffs are good examples of a natural obstacle, which characters must avoid when traveling.

The AI code which controls non-player characters needs to have some knowledge of the terrain in order to direct character movement. Pathfinding algorithms, such as A*, can treat the world as a graph of nodes representing points on the map with edges representing reachability. The engine can determine the closest terrain node within line of sight of a character, and direct the character to move to that node. From that point on, node-to-node movement can be directed based on edges connecting nodes until the character is within line of sight of its destination (assuming no other obstacles are in the way).

Some aspects of the terrain can be used to determine placement of other assets. Aquatic creatures should be placed in areas covered by water. Other creatures might be restricted to mountainous areas.

My quest generator makes use of pre-generated terrain, and therefore does not create any new terrain. If new terrain is desired, my earlier work on agent-based terrain generation is capable of producing high quality terrain automatically [18].

I chose to hand-annotate the terrain, by creating rules in the knowledge-base which described key points. This did not take too long, and only needed to be done once per map. The result is consistent usage of terrain. I also hand crafted a pathing network, described above, to aid pathfinding. While there are tools which can automatically generate pathing information from terrain geometry, I found that I could get better performance by manually placing the terrain nodes. This too only needed to be done once per map. I expect future terrain generators will be able to provide this annotation autonomously.
2.3.2. Cultural Objects

In addition to natural geometry, we can expect to see immobile objects which were presumably placed in the world by characters or creatures. These are referred to as cultural objects, to differentiate them from natural objects. Examples include buildings, roads, and lairs. These locations are frequently used as interaction points, and a generator will need to create additional objects from time to time to support new quests. To prevent the world from becoming cluttered with these objects, I have implemented selective visibility logic in the game server, which will be discussed in detail later. This logic only allows characters performing a quest, or those assisting them, to see cultural objects generated specifically for the quest.

When a generator needs to create new cultural objects, it must determine either the objects appropriate for an area, or the areas appropriate for an object. Indoor or underground locations are unlikely to be good locations for buildings. Similarly objects such as furniture are more appropriate indoors than out of doors.

2.4. Storyline

I have not worked on storyline as part of this research, but this is a significant part of my future research plans. Story generation has been studied for decades, with limited results. I believe that the techniques explored suffered due to too large of a subject domain, and that game content (in particular quests) offer a more suitable domain. Of particular interest is the use of drama managers. This is an example of off-line AI, where significant time can be spent without affecting game operation.

2.5. On-Line Artificial Intelligence

In contrast to the off-line AI, there is a need to provide information usable by a runtime AI engine. A prime example of this type of information is the economic state of the world. An economic simulation may be used to update prices, so as to reflect the current supply and demand of various commodities.
2.6. Items

Smaller items, capable of being carried and transferred from character to character are also needed by quests. In their most basic form, these can be thought of as Hitchcock’s MacGuffins, an item of no intrinsic value but which motivates characters [87]. Players are commonly asked to collect or deliver these items to other characters. Some quests use these items as evidence of the player having completed a particular step in a quest, as the only way to acquire the item is through those quest stages. In this later case, the object should be made nodrop, to prevent people from bypassing game content by having another player acquire the item for them.

When a quest generator needs specialty items, it can create them by selecting properties such as an appropriate name, graphic model, and inventory icon. Items may have restrictions on usage within the game, for example the player may be unable to trade the item with others (nodrop), the item may only exist temporarily and will disappear at the end of the current game session (norent). These restrictions prevent players from bypassing quest mechanics, and can introduce time constraints which make the quest more difficult.
CHAPTER 3

PROCEDURAL CONTENT GENERATION

The creation of the world did not take place once and for all time, but takes place every day.

Samuel Beckett

3.1. Overview

Procedural content generation is the process of using computers to create content, especially game content. I have previously defined content to mean all of the data outside of the game engine. The content is the fun stuff, and is usually created by humans. Creation is by definition a creative act, and we normally do not consider computers to be especially creative. Computers have pseudo-random number generators which allow them to take unpredictable actions, but randomness is not normally a feature we associate with good content. We may consider game content to be a type of art, and art is not considered to be random. So on the surface it may seem that it would be hopeless to attempt creative acts using a computer, however humans have a role which allows the process to occur. Humans can place constraints on the random actions, and if done well these constraints will reduce the chaos one associates with random output.

Procedural generation of quests is therefore the act of creating random, but not quite random quests.

Procedurally generated content should ideally have these properties [18]:

- Novelty: Contains an element of randomness and unpredictability.
- Structure: Is not merely random noise, but contains larger structures which make sense.
- Interest: Has a combination of randomness and structure that players find engaging.
- Speed: Can be generated quickly enough for the application.
- Controllability: Can be generated according to a set of natural designer-centric parameters.
3.2. Potential Advantages

Game content has traditionally been created by humans, and represents a significant investment of time and money [69], and is one of the greatest costs associated with game production. Content such as terrain, items, characters, dialog, story and quests can be generated automatically [42], but games are just starting to take advantage of this. Currently, most game content is hand crafted by designers. Procedural content generation creates the possibility of producing quality content at a fraction of the cost of hand generated content, by replacing human designers with automated processes which are much faster. There is no requirement that all content for a game be procedurally generated, in fact there are advantages to a hybrid design where a large quantity of content is procedurally generated and a smaller team of human designers augments this content. For example, Bethesda Softwork’s Skyrim features content generated by Bethesda’s Radiant AI engine, as well as human generated content. As a result of using procedural generation, smaller companies can generate content in quantities previously seen only in games with large development teams. These techniques also permit developers to take more risks than they would otherwise, since the cost of failure is much lower.

A game can benefit greatly from procedural generation due to increased replayability provided by novel content. A single game executable can provide endless entertainment if the game content is continuously updated, which in turn increases the games longevity. For example, the Civilization series of games procedurally generates maps which result in a significantly different game play experience for each map. Some games like the SSI gold box games used a common engine, and differed only in the content provided with this engine. By replacing the content, SSI was able to release multiple games which used the same engine. This culminated with the release of Dungeon Hack, which coupled the gold-box run-time with procedural dungeon generation, allowing players to generate random games at will. By presenting novel content to a player, the player is always experiencing the world for the first time.

Another benefit of procedurally generated content is the ability to generate content
in real-time while the game is being played. A game might track a player’s actions within
the game and estimate the players preferences. This information could then be used later to
create content specifically for this player [69]. There is no practical method for humans to
generate per-player custom content.

Procedural generation can result in new types of content not previously seen, as a
result of emergent properties in the generation process, which further increases the novelty.
Novelty is an important element, since with human generated content there is rarely an
opportunity for a player to discover something not previously seen by other players. With
procedural content generation, every player has the potential for discovering something new,
not just the first few players of the game. This is partially the result of defeating the spoiler
sites and walkthroughs which are seen soon after a new game is released. There is no point
in sharing information on one-time quests with other players.

3.3. Potential Drawbacks

There are potential drawbacks of procedural generation, which must be considered
by developers interested in the process. First, poorly designed generation rules can result in
obvious repetition, exposing the sameness of content. The game now becomes predictable,
even though the content is randomly generated. Careful creation of generation rules can
prevent or minimize this property. It is also helpful to have a large knowledge-base or
ontology to draw upon.

3.4. Use of Procedurally Generated Content

One of the earliest uses of PCG was seen in the game Rogue, developed in 1980. This
game randomly generated dungeon maps each time it is played. In much the same way as
Civilization, the random maps provided an incredible amount of replayability, since tactical
situations emerged from areas easily reached on one map.

In order for content to be effectively generated, it must have an explicit representation
(or model) within the generator. A generator would then make random selections within
the domain, subject to constraints, or apply fitness functions to evaluate the acceptability
of the solution. A large set of choices is essential, as a solution space grows exponentially with respect to the number of choices. A solution space of this type is usually seen as problematic, and is referred to as the curse of dimensionality. However, when generating content the richness of the space is an asset, and I choose to refer to this space as the blessing of dimensionality. Note that the goal of a generator is not to generate a single solution (or an optimal solution), but a randomly selected interesting solution.

Planning is often used for more complex content (such as quests), but a grammar-based approach is also viable, if not preferred. Remember that we are not looking for a particular (or optimal) solution, just an interesting one. This lack of constraint also increases the amount of potential content, since there is usually only one optimal solution a planner might find. So the use of planning algorithms would seem to add an unnecessary complexity. Planning is PSPACE-complete at best, as it can be viewed as ”finding a path in an implicitly represented graph, which is exponentially larger than its representation” [7]. In contrast, a grammar-based approach using LR(k) grammars is O(N) on the size of the grammar [37].
CHAPTER 4

TERRAIN

This land may be profitable to those that will adventure it.
Henry Hudson

Terrain is a type of content which forms the setting in which game events occur. Frequently, terrain is implemented as a heightmap (a mapping of X-Y location onto world height), although this technique does not easily support caves or overhangs as these create multiple elevations for a given location. It is possible to create holes in terrain, and warp players from these points to auxiliary height-based terrain as a means of implementing caves. Voxel engines implement a non-heightmap terrain which easily supports overhangs and caves, but which requires additional processing power. As hardware becomes more advanced, it is reasonable to believe that voxel techniques will become more common. There are also polygonal solutions, such as that used by Everquest. The terrain is represented as a set of triangles, which permit any shape to be constructed. Locations now have X-Y-Z coordinates that are unique.

Terrain data, like other forms of procedurally generated content, requires constrained randomness. Randomness is useful to ensure novel terrain, but constraints are needed to ensure internal structure. When creating heightmaps it is useful to observe that generation is a highly localized process. A point’s elevation depends far more on the elevation of nearby points than it does on elevation of remote points. This observation led to the implementation of an agent-based terrain generation system featured in prior work [18]. This implementation was significant in that it was interesting, controllable, and could easily scale by adding additional agents.

4.1. Previous Work

Previous work on generating terrain has mostly been based on the use of fractal techniques. Most of the existing systems have a very small number of designer-selectable parameters, and as a result provide the designer with limited opportunities to control what
type of terrain is generated. One of my goals was to allow the designer more (and more natural) control over the generated terrain without sacrificing too much of the other four desirable attributes of novelty, structure, interest, and speed discussed in Chapter 3.

Olsen [49] discusses erosion algorithms that use fractal Brownian motion (1/f noise), and perturbed Voronoi diagrams. He provides some metrics for evaluating terrain (such as low average height and a high standard deviation for slope) which are used to compute a game suitability score used to evaluate generated terrains. Musgrave, Kolb, and Mace [46] use noise synthesis to create eroded fractal terrain. Control of this algorithm appears to be limited to modulation of the noise frequency, in particular scaling and translation of values. Control does not appear to be feature-based, that is, designers do not express their desires in terms of terrain features. Evaluation of their technique was limited to the efficiency of generation, not the quality or ease of use.

Szeliski and Terzopoulos [82] address fractal terrain generation in one section of their paper. Their approach uses real digital elevation data which they perturb using splines. Fractals are used to add in detail to the resulting heightmaps. The character of the output can be controlled by the initial elevation data, the functions used to determine the spline shapes, and the amount of noise used by the fractal generation. Van Pabst and Jense [88] generate terrain by creating a multifractal field based on four parameters. A separate utility is used to analyze existing digital elevation data and extract these four parameters. Belhadj and Audibert [9] create ridge and river networks and then use fractals to transform this data into a complete heightmap. The ridge and river network are created by randomly depositing particles and allowing them to interact with each other and the terrain. As with most fractal-based approaches, their algorithm does not appear to be controllable. Pi et al. [59] create fractal landscapes using Perlin noise. They do not indicate any method of controlling the quality of the generated terrain other than a scaling term applied to the noise.

Ong et al. [50] apply genetic algorithms to terrain generation, and also focus on controllability. Their approach uses a sketch of the boundary which could be either designer-provided or machine generated. A database of representative heightmap samples appears to
be the main form of control for their algorithm. Frade et al. [21] present another genetic technique which they claim is more controllable than fractal techniques.

Li et al. [38] use machine learning to model example heightmaps and then later use these models to synthesize new terrain. This requires that a suitable set of training examples be created for each desired output archetype. One issue with machine learning is that it is often not clear what is being learned, that is to say that one does not necessarily have control over what features in the training data are important.

Kamal and Uddin [30] present a technique that is minimally feature controllable. This technique repeatedly creates random polygons using a series of randomly placed lines and performs random walks that raise the points in these polygons. We believe this technique can be equated to Brownian motion, with the main difference being the order in which the individual points are changed. This technique offers limited control, as one is able to create multiple mountains and mountain ranges.

Lechner et al. [35] use an agent-based approach to determine urban land use in a growing city. They share with us a focus on producing a tool for the non-technical designer, in this case artists and geographers. Their solution was controllable, but the parameter choices seemed rather abstract and suggested that designers would need to learn the effects through experimentation rather than intuition.

The agent-based implementation differs from these papers in that the input to the generator is a set of parameters that describe the quantity and quality of familiar terrain features, such as mountains and rivers. Once such a set of parameters is defined, the generator can create variations on that theme without further input from the designer. Only one of the above papers (Kamal and Uddin [30]) has any type of feature-level control, and some of the previous work (including [50, 38, 88]) was example based, requiring the designer to provide examples of the types of output desired.

4.2. Software Agents

While there currently is no consensus on what a software agent is (see, for example, Rudowsky [64]), I start with Russell and Norvig’s definition of an agent that perceives
its environment through sensors and acts on it through effectors[65]. All agents are given autonomy, other than some constraints on what the agent is to produce. Using Russell and Norvig’s list of agent properties, my terrain agents:

- Work in an accessible environment
- Are deterministic in the results of their effectors
- Are non-episodic
- Work in a dynamic environment
- Work in a discrete environment

The developing terrain is fully observable by each agent, as agents may query the height of any nearby point. While the actions chosen by an agent are non-deterministic, the results of these actions are deterministic. Agents do not remember previous actions, nor do they plan ahead, making them non-episodic. The environment is dynamic, as the result of other agents concurrently working on the world. And the environment is discrete, since heights are represented as discrete integers.

The agents ran in three distinct phases, which can be loosely summarized as coastline, landform, and erosion. In the first or coastline phase, a large number of agents work to generate the outline of a landmass, possibly surrounded by water. In the second or landform phase, a larger assortment of agents work to define the features of the map (raising mountains, shaping the lowlands, creating beaches). In the third or erosion phase, rivers are added by eroding parts of the terrain. Agents run concurrently and asynchronously within each phase.

Agents are allowed to see the current elevation of any point on the map, and are allowed to modify these points at will. The presence of other agents will likely cause the environment to change around them without warning. Observing the height (and derived values such as being on a coastline) and modifying heights are the extent of an agent’s actions and percepts. These observations are analogous to sensory perceptions, and the modification is analogous to the use of effectors.

Each agent when it is created receives a number of tokens that it will consume as discrete actions are taken. This limits the lifetime of an agent, and is one way in which
the terrain generation can be influenced by the designer. Macro features of the map may be controlled by the designer by specifying the number of each agent type to run and the number of tokens each agent receives. For example, since each river agent produces one river, the number of rivers is altered by raising or lowering the number of river agents. The terrain generator provides a default set of agents and tokens based on the map size. This default set can be overridden by the designer to create different types of terrain.

As a proof-of-concept I used five different types of agents, although it should be kept in mind that many more agent types are possible. The five agent types are as follows:

1. Coastline agents create a landmass, possibly surrounded by water.
2. Smoothing agents perform random walks, averaging the height of nearby points.
3. Beach agents produce flat areas on parts of the coastline.
4. Mountain agents raise mountain chains.
5. River agents erode terrain producing rivers that run from mountains to the ocean.

Each of these agent types is treated in a separate subsection below.

4.2.1. Coastline Agents

Coastline agents create the outline of the landmass before any heightmap data is calculated. Coastline generation starts with a single agent that is responsible for the entire landmass. This agent subdivides the task by creating multiple subordinate coastline agents and assigning them part of the landmass to work on. These in turn subdivide, and the process repeats until each agent is working with a small number of vertices (on the order of 1000). Each of these agents work independently of other agents, but all agents have access to the environment. In this case, the environment consists of the landmass in whatever state of development it may be in when the agent is running. Agents modify the environment by raising points above sea level (initially the entire map is below sea level).

Each agent starts with a single seed point on the edge of the landmass, a preferred direction, and the number of points the agent is expected to generate. If for some reason the seed point is surrounded by land by the time the agent begins executing, the agent
begins searching in its preferred direction for a coastline point. Once the agent places itself on the coastline it creates two points at random: one is an attractor point, the other is a repulsor. These must be in different directions. When the agent evaluates candidate points for elevation above sea level, points closer to the attractor point are scored higher by the agent, and points closer to the repulsor are scored lower.

The score for a point \( p \) is given by
\[
d_r(p) - d_a(p) + 3d_e(p),
\]
where \( d_a(p) \) is the square of the distance from \( p \) to the attractor, \( d_r(p) \) is the square of the distance from \( p \) to the repulsor, and \( d_e(p) \) is the square of the closest distance from \( p \) to the edge of the map. These terms encourage an agent to move towards the attractor, move away from the repulsor, and to avoid the map edges as much as possible. Since each agent randomly selected its attractor and repulsor points, each agent would score a given point differently.

The agent expands the landmass by adding points to the edges of the mass. It calculates a score for all surrounding points that are not part of the landmass, and moves to the highest scoring point. This point is then elevated above sea level, becoming part of the coastline.

Note that multiple agents are running concurrently, and that each agent will have unique goals and scoring biases. Agents will elevate one point above sea level, and then warp to another part of the coastline they are working on. This causes agents to operate at random, but in a local area of the map. The goal is to give map features some localized consistency, rather than adding points at random to the coastline. This effect is seen by the presence of capes that have smaller capes. The higher level agent will bias its children, who will in turn bias their children to create smaller features.

Each coastline agent has two configurable parameters that collectively define the range of vertices where an agent will be elevating points rather than subdividing its task. Figure 4.1 shows the shape of three landmasses created by varying these parameters. I used the same seed for the random number generator in each of these three runs so that variations in coastline shape are due only to changes in agent actions.
Figure 4.1. Coastlines produced by coastline agents with small (top left), medium (top right) and large (bottom) action sizes
I requested 150-220 vertex actions for the landmass on the left, 500-550 vertex actions for the landmass in the middle, and 950-1000 vertex actions for the landmass on the right. In general, a lower vertex range will result in more agents operating on smaller regions, and as a result the coastline will have more fine detail than a landmass that had a larger vertex range. A landmass created with a high vertex range will have larger, wider features and will tend to be more circular, as is the case with the landmass on the right of Figure 4.1.

4.2.2. Smoothing Agents

Smoothing agents make random walks around the map adjusting the height of an arbitrary point \( p \) to be the average of points in an extended von Neumann neighborhood of \( p \) consisting of the four orthogonal map points surrounding \( p \) on the elevation grid and the four points beyond these [33]. A weighted average height is calculated, with the center point given 3 times the weight of the other points. Therefore nine points with a total weight of eleven are used. This provides some inertia to prevent elevations from rapidly changing. I believe that this extended neighborhood is responsible for the emergence of interesting curved features on the map. Use of an 8-cell Moore neighborhood resulted in less interesting results.

Each smoothing agent returns to its point of origin periodically. This encourages smoothing agents to operate in a local area, which is useful when certain features of the map need more smoothing than others.

The only configurable parameter for smoothing agents is the number of times that the agent will return to its start point. Setting this number to a large value causes the agent to spend most of its time near the start point. This provides a great deal of smoothing for that area, rather than less smoothing spread over a larger area in the case where the agent is allowed to wander further away.

4.2.3. Beach Agents

Beach agents create flat sandy areas next to the main coastline after the coastline agents have finished. Before they begin, points on the main coastline are identified using
breadth-first search. Beach agents then use these points to place themselves on the coastline. They then perform random walks flattening areas of beach, following the shoreline. Beach agents adjust the height of the beach to allow random fluctuations in elevation so the beach is not a uniform flat space. After moving to a spot on the coastline, the agent will lower the nearby points and jump inland to perform a random walk. This creates variable sized sandy areas that can extend a short distance from the water. After the random walk is complete, the agent returns to the coastline and continues to walk along the shore. If an agent becomes stuck (for example running into a mountain range) and is unable to continue its walk, it moves to another randomly chosen point on the main coastline and continues. Beach agents avoid high areas, so any mountains that are next to the ocean are left alone.

One of the more important parameters for the beach agent is the altitude limit, above which the agent abandons an area and moves elsewhere. When the altitude limit is low, the raised area near the middle of the beach is allowed to remain. When the altitude limit is high, the agent is able to continue its work in this area and flattens the mound.

Beach agents set the height of the beach to random values within a specified range specified by the designer. When this range is narrow, flat beaches are created. When it is raised a bit we see more bumps. The designer can also control the width of a beach by indicating how far inland the beach agents should begin flattening, and how long their random walk should be. Figure 4.2 shows the effects of varying the width of a beach.
4.2.4. Mountain Agents

Mountain agents raise mountain ranges. Each starts at a random point on land and selects a preferred direction of travel. As a mountain agent moves in this direction it raises an inverted V shaped wedge of points with the center line becoming the ridge line. The agent will move along this ridge and will periodically decide to change direction within a 90 degree cone from its original direction. The effect is that the agent zig-zags but heads generally in the same direction. If an agent runs into the ocean or the map edge, it changes direction to avoid this obstacle.

The width of the V-shaped wedge determines the general width of the mountains, and to a large degree the slope of the mountain sides. The rate at which the slope drops in elevation is randomly determined for each wedge (within a designer-specified range), which produces some interesting features on the sides of the mountains. Mountain agents also periodically create foothills running perpendicular to the mountain range axis. Smoothing is performed on the mountain after the wedge is raised, blending the heights and leaving gentler transitions between nearby points.

Prior terrain generators have used other techniques for creating mountains, such as fault generation [36, 70], fractal midpoint displacement [71], and point deposition [2, 49, 72]. While we make no explicit attempt to simulate faults, our mountain agent’s terrain elevation is similar, with the major difference being that the mountain agent determines its path as it operates, avoiding obstacles in its way, whereas fault simulators determine the fault’s position prior to modifying the landscape.
The mountain agent’s simplistic wedge raising produces acceptable results, mainly due to the interaction of the smoothing agents that are making random walks over the terrain. Figure 4.3 shows the effects of widening a mountain and increasing its foothill length.

Mountain agents are the most configurable of all agents, as they introduce most of the interesting features on a landscape. Without them the heightmap would be mostly flat. The designer determines the number of mountain agents that will run, and specifies how many tokens each mountain agent will receive. A single mountain agent will randomly position itself on the map, decide on direction, and begin elevating terrain. It stops when it runs out of tokens or is unable to proceed due to some obstacle. Mountain agents attempt to turn to avoid obstacles, but this ability is limited to ensure that agents do not randomly wander the map.

Mountain agents are given a maximum altitude, and vary the generated height within a specified range below this height. Mountain agents may also be assigned a width and slope, which allows them to either spread out, or to create tall narrow ranges. While mountain agents perform smoothing, they also follow this up by adding noise to restore some of the character lost during smoothing. This noise is specified by a probability of altering a point’s altitude, and a variance. When a point’s altitude is modified during this roughening phase, a random value up to the variance parameter is either added or subtracted from the point’s current altitude.

Mountain agents periodically generate foothills perpendicular to the range axis. The lengths of these are randomly determined from a configurable range, as is the frequency at which these foothills are created.

4.2.5. Hill Agents

Hill agents are a special case of the mountain agent. As with a mountain agent, the designer specifies the number of tokens assigned to each hill agent, indirectly determining the size of each hill. Hill agents generally create very short mountain ranges with a lower altitude, and no foothills, as seen in Figure 4.5. Hill agents may have their altitude range determined by specifying a maximum altitude and a variance. Since hill agents are a special

29
4.2.6. River Agents

River agents create water channels between points near mountains and the ocean. Each agent creates one river. A river agent begins by choosing a random point on the coastline and a random point on a mountain ridge line. It begins at the ocean and moves in the general direction of the mountain point, following the elevation gradient uphill. This causes the river to meander, rather than creating a perfectly straight river. When the agent runs into a mountain it stops and begins to backtrack its path downstream. During this phase of the agent’s operation it will lower a wedge of terrain, similar to the way mountain agents raise heights. The width of this wedge increases as the river moves downstream. This downhill phase stops if the river encounters another river, in effect making one of the rivers a tributary of the other.

The river agent avoids creating rivers that are too short, or which fail to reach the mountains. An agent makes multiple attempts to place a river, but will eventually give up if too many attempts are unsuccessful. As a result, it is possible to specify river agents and not have rivers placed. The primary cause of river agents failing to place a river is that every attempt to run a river uphill encounters a mountain before the river’s minimum length is reached. This is more of a problem with smaller maps, since the mountains dominate small maps more than larger maps and are more likely to be found near coastlines. This the most important parameter for a river agent is probably the minimum length. River agents attempt to place rivers on the terrain, running from a mountainous region to the ocean. The
minimum length parameter causes the agent to abort attempts to place rivers smaller than this value, and to instead search for a place to create a longer river.

River length is a function of the coastal and mountain points selected, as well as the height of the terrain between these points. Rivers are created uphill, and will stop when the terrain becomes too mountainous, even if the preselected mountain point has not been reached. The alternative would be to allow the river agent to create a channel through one mountain to reach the other. The designer is able to specify the maximum altitude of the terrain allowed before the river terminates. Other altitude limits are the maximum allowed altitude on the shoreline (to prevent waterfalls at the coastline), and the minimum altitude of the mountain point (to prevent rivers from starting at foothills or on flat terrain). A minimum distance from the coastline to the mountain point may also be specified, which prevents rivers from starting near the coast but heading towards a more remote part of the coastline.

River agents require a backoff parameter that causes them to back away from mountains when the altitude limit is exceeded. This prevents rivers from climbing a cliff and then starting their downhill building phase. In effect, this type of behavior would allow rivers to start at the top edge of cliffs. When the agent’s altitude limit is reached, the agent will back off a number of vertices along its path, and then begin cutting the river into the landscape.

River agents may optionally lower the terrain on the mountain side of the river. This causes a deeper channel to be formed, as opposed to gentler sloping river banks. Rivers are assigned an initial width, which then increases over time as the river runs downhill, simulating the accumulation of runoff water. This width, the frequency of widening, and the downhill slope of the cut channel may all be specified. Figure 4.6 shows two generated rivers. The river on the left is a dry riverbed created by using a shallow initial altitude drop, and a shallow slope. The rivers on the right were created by specifying multiple river agents, plus a narrow target altitude range for the coastline.
4.3. Implementation and Evaluation

The terrain generator executes runnable agents in a random order, and for a random slice of time within a window. It assumes that atomic locking exists at the vertex level. The first phase of agents, those that produce the coastline, do not share this scheduling system, but there is nothing in these agents which depend on other agents. The purpose of this scheduling framework is to demonstrate the independence of agents, as there can be no predictable ordering of agent activity. The simplistic behavior of these agents results in complex interactions among agents, and that the terrain is an emergent result.

This agent-based terrain generator lends itself to implementation in either a purely procedural environment, or in a designer-centric environment. In the former a game could use my technique to generate terrain on-the-fly, guided by agent settings provided by the publisher in advance or in real time. In the latter environment in which the publisher requires more control over the content, a designer could use my technique to generate terrains that are first screened and/or modified by a human being before being distributed. Note that unmodified terrains lend themselves to easy distribution since our terrains are uniquely specified by the parameters and the initial random number seed. I created a prototype tool (see Figure 4.4) that lets designers experiment with the various agent parameters and view the resulting terrain in 3D using the SAGE engine [54], which was used during development. Note that the parameters tend to be high-level designer-centric values that correspond in an intuitive way to terrain features.

I provided a list of attributes earlier which I believe any good procedural content generation system should have. I note that others have provided similar lists which I contrast with my own. For example, Ong et al. [50] suggested that the qualities should be as follows:

- Adaptive
- Innovative
- Scalable
- Intuitive

I believe my agent-based approach satisfies each of these traits. Ong’s definition of \textit{adaptive}
is met as my use of agents allows different terrain types to be created, and new terrain types may be added by writing new agents. I believe that the innovation requirement is met since terrain features are an emergent property of the intra-agent interaction. The agent-based system is scalable as the map size, the number of agents, and the number of tokens given to each agent may be changed. I believe a set of designer-specified parameters results in intuitive control of terrain generation. For example, using the graphical front end the designer may manipulate sliders that correspond to terrain features instead of being asked to modify abstract parameters that do not directly correspond to the features of the generated terrain.

Saunders [67] suggested the following desirable traits for procedurally generated terrain in his Master’s thesis:

- Require a low degree of human input
- Permit a high degree of human control
- Be completely intuitive to control
• Be able to generate a wide variety of recognizable terrain types and features, in believable relationship to one another
• Produce models at arbitrary levels of detail
• Run quickly enough to be used in real-type, dynamic applications
• Be extensible to support new types of terrain

I believe that my agent-based approach satisfies all but the real-time requirement. The designer is asked to select parameters for generation, and defaults exist for all parameters. As a result, the designer may generate multiple heightmaps with the same parameters without providing any further input. Since my parameters deal with geographic features rather than abstract concepts like noise, I feel that they are intuitive and accessible to a non-technical designer. When the effect of a parameter is not obvious, I feel that it is easily learned by experimentation since the parameters are grouped by geographic feature type. This agent-based technique allows the creation of arbitrary levels of detail by varying the size of the generated heightmap, although other parameters will need to be similarly scaled. The variety of agents in the current implementation allows for a variety of recognizable terrain types, and as new agents may be added the system is extensible.

I do not feel that real-time is a necessary requirement for an ideal terrain generation system, unless one adopts a loose definition of real-time. If an application has terrain available when it is needed, even if the terrain was generated offline, I believe that the terrain generator has performed its job. While there are applications that require that terrain be generated as a player moves around the world, there are many successful commercial games that have shipped with pre-generated terrain. In order to preserve generality in my list of traits, I therefore dispense with a real-time requirement.
Figure 4.5. Hill agents produce hills, similar to the way mountain agents produce mountains.

Figure 4.6. River agents generated a dry river bed (left), and three rivers that meet at the ocean (right).
CHAPTER 5

ECONOMICS

No one is entertained by economics. Michael Moore

While role playing games such as Oblivion, Fallout, Everquest, and World of Warcraft allow players to trade goods and services with computer controlled non-player characters (NPCs), acceptable prices in these games have been selected a priori by game designers and typically remain fixed throughout the game. Game economies differ from real world economies in that interesting behavior is more desirable than accurate modeling, and static prices do not seem as interesting as those that change as the result of game events. Little work has been done to apply traditional economics to role playing games, therefore I developed a price update system, grounded in traditional economics, that may be used in such games.

Economic simulation has a second use, which is determining the number of NPCs in a particular role who may be supported in an area. This is needed when creating communities, to ensure that the distribution of roles makes sense. For example, having 100 blacksmiths in a small town makes no sense, as there couldn’t be enough work for all of them. This type of analysis can be done off-line, as part of worldbuilding. So even if the economic simulation is not made part of the game server, this technique has utility for procedural content generation.

The previous listed role playing games (RPGs) all use static economies created by developers before the game ships. One of the drawbacks of a static economy is that nothing changes, and in particular the player is unable to cause changes to occur. Under a static economy, vendor preferences do not change and as a result one sees situations where vendors have a seemingly infinite supply and demand for commodities. In a dynamic economy, trades fulfill needs leading to the adoption of new preferences and new behaviors. In particular I expect traders to buy only those commodities that are useful to them, and if they are unable to trade profitably to go out of business and find another line of work. This is the mechanism
which determines an acceptable number of each role within a community.

A game economy consists of a set of players and NPC agents that periodically trade with each other. These agents take on the role of vendors, as any of these agents could at a particular point in time offer to buy or sell a commodity or service from any other trader. This set is referred to as a market, and the participants is known as traders. Offers to buy a good or service is referred to as bids, and offers to sell a good or service is known as asks.

While the most general form of an economy involves the trade of both goods and services, I choose to simplify this model without loss of generality by considering all trades to involve only goods, referred to as commodities. I are able to do this because a service can be mapped to a tradable token such that the token may later be exchanged for the performance of the service. In the real world we see this same type of conversion used with postage stamps and gift cards for pure services such as haircuts.

In addition to the determination of prices, supply, and demand, an economic system will determine the allocation of resources. I consider both commodities and the NPC traders themselves to be resources, the latter because the market is able to generate wealth through them. Each NPC trader may be allocated or assigned one of several roles that govern its behavior, and I will show later that if the market is in equilibrium the allocation is Pareto-optimal (meaning that no trader may improve its position by modifying an action without making another trader worse off, see Pareto [57]). This is a useful observation since it gives us the ability to determine a reasonable distribution of roles within a community. The simulation is thus able to determine the exact number of agents for each role that may be supported given current market conditions, and also to determine when an agent is no longer contributing to the economy.

Given an allocation of NPCs to various roles within the game, I indirectly determine what commodities will be purchased or sold. Supply and demand determine what roles are profitable, and the allocation of NPCs to these roles determines future supply and demand. Thus the simulated economy is a feedback loop, ideally with the behavior of all traders being interrelated.
In evaluating my experimental results, I consider long-standing works in economics by Adam Smith [74] and Vernon Smith [75, 76] and find the behaviors of our model to be sufficiently consistent with classic economic theory to confirm the usefulness of my approach. I have also studied the price update behavior of over 1 billion simulated economies, and am satisfied that the variability and consistency of my results allows designers to create economies that are sufficiently interesting to appeal to game players.

I will demonstrate an economic system with the following properties:

- It determines internally consistent prices for a variety of commodities.
- It adapts to external perturbations and shocks.
- It determines allocations of agents to roles.
- It is not dependent on any one set of game rules.
- It requires little memory or computation per transaction.

The goal of a game economy is to provide the player with an interesting experience that changes over time, and one in which they feel they have some measure of influence. My work differs from traditional computational economics by stressing interesting behavior at the expense of realism. As with advances in areas such as computer graphics, I note that players are willing to suspend their disbelief if they enjoy the game experience. I show how to implement a system that can provide interesting reactive behavior, is easily added to games, and has low resource requirements. This type of economic behavior is lacking in most current role playing games, and it is my belief that future games would benefit from a richer set of economic behaviors.

5.1. Related Work

Researchers at Iowa State University have done a lot of work in the area of agent-based computational economics, and this has drawn our attention, particularly the price resolution technique found in Nicolaisen, Petrov, and Tesfatsion [47]. Their work tends to focus more in qualitative areas, such as learning relationships between factors in a market, while we are concerned primarily with quantitative results: what is a good price for a commodity at
this point in time? Steiglitz [78] has also performed agent-based simulations, however based on inspection of the source code I feel that my agents behave more rationally (for example, their agents appear to purchase as much of a commodity as they can afford without regard to the price or the agent’s need).

The TAC-SCM (Trading Agent Competition - Supply Chain Management) competition has been run annually since 2003, and produced many papers in the area of supply-chain management (see, for example, [10, 12, 32, 56]). In this competition, agents buy and sell commodities and produce products for resale. They attempt to predict changes in prices, and operate in a profitable manner. While supply-chain management is a substantially different problem, some of the techniques used by these simulations are applicable to RPG economics.

Roth and Erev [63] used reinforcement learning (RL) to learn prices in a simulated economy. In particular they used the acceptance or rejection of offers to provide reinforcement of a trading agent’s pricing policy. I considered this approach, but was concerned that the amount of state I would need to consider would make policy convergence impractical. Flores and Garrido [20] similarly used RL, and I experimented with their technique of linearly interpolating prices using weights on the low and high end of the price range.

One advantage of reinforcement learning is the ability to update policies in an environment where agents do not know their current state. Value-based approaches like Q-learning or temporal difference (TD) learning do not work well in environments with hidden state, as agents need to know the current state in order to select a corresponding action. However, one is able to create an equivalence set of states based on observations of the system, and estimate which set likely contains the current state. Dahl’s work with poker [15] shows that RL can work with hidden state, however for this problem the current visible state allows one to know the exact trajectory through action space that has been taken so far. This is an unusual situation when using RL.

Price determination for a set of commodities is a significant problem, and a variety of techniques have been used with other problems. Several TAC-SCM entrants (for example, [10, 55, 68]) attempted to predict winning bids. The Botticelli trader estimated the proba-
bility of filling orders, and adjusted offer prices until the expected trade volume matched its ability to fill them. The probabilities are updated based on trading experience, which makes price a function of market history. Many more factors could go into pricing (utility, pricing trends, market supply and demand). There is also the question of whether probability is a linear function. Within a small range, linear approximations are adequate, but with larger uncertainties a nonlinear update may be more appropriate.

Pardoe and Stone [55] used a Bayes classifier to estimate the probability of an offer being accepted, and trained their classifier using data from prior TAC scenarios. In the case I considered, the state of other agents is not known, limiting my ability to estimate prices. I chose instead to base offers on each agent’s belief in the true price of a commodity, and not consider whether other agents might agree. I chose this approach to more closely model the imperfect information real traders have, and in turn I believe this leads to more realistic results.

Shapire [68] modeled price changes as a conditional density estimation problem. A price range was discretized into a set of bins, and a probability distribution was created over this set. This technique also modeled future prices as a function of historical prices, which works well if there are no other factors that might affect prices.

Wellman et al. [91] used a novel approach by estimating future demand for a commodity and adjusting prices in advance of the market. The success of this approach is dependent on the ability to make good predictions.

Ketter [31] inferred market regimes (conditions such as oversupply of a commodity) based on the results of attempted trades. Gaussian mixture models (GMMs) were fitted to historical price data, and used as classifiers. I did not require these types of predictions, since my problem was defined as allowing agents access to market statistics for resolved offers, or to access market price history. Ketter’s system modeled price as a function of demand (similar to Wellman) and estimated trade volume as being similar to previous rounds.

Studies have been made of the economies of various massively multiplayer online role-playing games (MMORPGs) by Simpson [73] and Castronova [11]. While these do not tell
us how to simulate game economies, we can see that game economies do behave similarly to real-world economies.

Meadows [45] developed models and a simulation to study social systems, however one could argue that these were economic models since they addressed resource allocation, and growth. This is also an excellent overview of model creation and simulation, I note in particular how one must describe the type of information a model is intended to produce. Following Meadows’ categorization, my model provides projections of dynamic behavior modes. I omit the term *imprecise* as the prices reported represent actual trades in the simulation, and would presumably be actual NPC offer prices when used in a game.

5.2. An Economic System

A simulated economic system serves four major functions as shown in Figure 5.1. In addition to determining the prices of commodities it also determines order quantities based on supply and demand for each commodity, the production and consumption of commodities which is indirectly related to supply and demand, and an allocation of commodities and roles to participating agents. This mirrors the properties of real economic systems, in particular the coupling of price with supply and demand as discussed by Adam Smith [74].

Each agent maintains a set of price beliefs for each commodity it is able to buy or sell. These price beliefs are represented as an upper and lower price bound, with the agent believing the price to be somewhere in this interval. Any time the agent needs to make a price estimation, for example during offer creation, it will select a uniformly random value in this interval. The outcome of a trade will provide either positive or negative reinforcement to this belief. Positive reinforcement will result in the agent shrinking this interval around the mean, negative reinforcement may result in the interval increasing about the mean and possibly being translated to a different mean. A designer interested in creating an economic system would need to decide when these updates occur, and the magnitude of the changes.

Periodically agents will need to submit trade offers to the clearing house in order to buy or sell commodities. When an agent wishes to create an offer, it will need to determine the commodity to trade, a fair price, and the quantity of the commodity to trade. A
designer may choose to have agents buy only commodities they use for production and sell commodities they produce. In this case, an agent would create bids when the inventory of needed commodities drops below some threshold, and create asks anytime it has inventory to sell. The Create Bid routine creates an offer to buy at most limit units of Commodity, and Create Ask creates an offer to sell at least limit units of Commodity.

Create Bid(Commodity, limit)
1  bid-price ← PriceOf (Commodity)
2  ideal ← Determine-Purchase-Quantity(Commodity)
3  quantity-to-buy ← Min(ideal, limit)

Create Ask(Commodity, limit)
1  bid-price ← PriceOf (Commodity)
2  ideal ← Determine-Sale-Quantity(Commodity)
3  quantity-to-sell ← Max(ideal, limit)

The determination of offer quantities is based on an agent’s need, the inventory on hand, and the observed market price for that commodity. An agent might determine that it has no need to trade in a particular commodity, or that a need is present but current market prices are unfavorable and trades should be avoided. If an agent believes that a commodity is either overpriced or underpriced, it will adjust the quantity in its order depending on whether the agent is buying or selling. The quantity is scaled based on the location of the current market price within the trading range that the agent has observed. Agents that trade more frequently will have observed more trades, and will therefore have a better idea of the trading range. Agents that trade infrequently are more likely to make mistakes in pricing, however the resolution of these trades will cause the price history to be updated and the agent will improve its performance in future trades.

While there are likely many different means of determining trade quantity, I have had success basing this number on how far the agent’s price belief is from the observed market
average. This introduces an important but subtle distinction, as prices may be expressed in two different forms. A historical average price represents successful trades that have occurred in the past. Agents should be aware that past performance is no guarantee of future results, and should therefore trust their own beliefs more than the historical average. However, agents should question their belief if these values diverge and their offers are being rejected.

**Determine-Sale-Quantity** *(Commodity)*

1. \( mean \leftarrow \text{historical mean price of Commodity} \)
2. \( favorability \leftarrow \text{position of mean within observed trading range} \)
3. \( amount\ to\ sell \leftarrow favorability \times \text{excess inventory of Commodity} \)
4. return \( amount\ to\ sell \)

**Determine-Purchase-Quantity** *(Commodity)*

1. \( mean \leftarrow \text{historical mean price of Commodity} \)
2. \( favorability \leftarrow \text{max price - position of mean within observed trading range} \)
3. \( amount\ to\ sell \leftarrow favorability \times \text{available inventory space} \)
4. return \( amount\ to\ sell \)

An economic model may be used as a tool for allocating resources, determining trade volumes, or estimating commodity prices. These factors may be expressed as a set of coupled functions of the other factors. In general, supply and demand determine price, and price determines supply and demand (Smith [74]). For example, in an economy where wheat is sold, the amount of wheat traded on the market is a function of the bid and ask prices, and the quantity traders are willing to trade at these prices.

The amount of each commodity that is produced and consumed is determined by having each agent **Perform-Production**. In the most general form the agent will attempt to transform one basket of commodities into a second basket of commodities. The commodities that are consumed represent the raw materials used up during production, and the commodities that are produced represent the products of the agent’s labor. Production
may therefore reduce an agent’s inventory of raw materials, and increase its inventory of products. As a result of an inventory reduction, the agent may later find that its inventory is too low and create bids in an attempt to replace this inventory. This creates demand in the market, and the agent competes against other buyers for whatever supply is available on the market. Similarly, any increase in inventory may prompt the agent to create asks in an attempt to sell excess inventory. This creates supply in the market, and the agent competes against other sellers for whatever demand is available. There is therefore a strong relationship between the results of PERFORM-PRODUCTION and the supply and demand for various commodities in the market. This in turn implies a strong relationship with future prices for these commodities, as an increase in supply will tend to drive prices downwards and an increase in demand will tend to drive prices upwards.

It is important to note that there is no single correct price for a commodity, but rather a price that is acceptable to the community at a particular point in time. The individual trade prices may not be an optimum price, nor indicate a market equilibrium, as noted by Vernon Smith [75]. Traders will sometimes trade at non-optimal prices, and that they will learn from their experiences and adjust their future behavior, or they will fail and be removed from the market. While the actions of individuals are not optimal, they are usually

Figure 5.1. Responsibilities of an economic system.
rational and reflect the self-interest of each individual. These individuals tend to adjust their behaviors until they collectively behave in a Pareto-efficient manner.

An economic system also serves to allocate resources within the market. We have seen how agents compete to buy and sell commodities, and as a successful offer results in a trade it will also result in an allocation of the commodity traded to the buyer. A buyer who offers a higher price will have their offers accepted before lower priced offers, and therefore the market can be seen to allocate resources first to those who will pay more for them.

Each trading agent is assigned a particular role or profession when it is created, and maintains this role during its lifetime. This role determines the production rules that an agent will use when Perform-Production is called, and subsequently the commodities that the agent will trade in. In the most general case, agents would not have these restrictions, but as we are also interested in determining a stable distribution of agent types, requiring agents to adhere to a limited set of rules allows us to make statements about the ability of a particular ruleset to support a given number of agents.

As the simulation progresses, successful agents will buy raw materials and sell their products. Unsuccessful agents will fail in their attempts to buy or sell, and therefore generate no cash flow. We have found that it is helpful to assess some fixed overhead, either in the form of required consumption, or in taxes, to pressure each agent to be productive. Under such a system unsuccessful agents will eventually go bankrupt as their money supply is exhausted, while successful agents will earn a profit above their expenses. When an agent goes bankrupt we choose to replace the agent with one of a profitable type, thereby adjusting the distribution of roles within the population of agents. This represents a market allocation of agents to roles.

5.3. Price Belief Updates

Agents will update their price beliefs in response to their offers being accepted or rejected, therefore agents that make offers gain more information on the market than those who make no offers. These price beliefs are represented by a lower and upper bound for a price interval, with the agent believing the true price of a commodity lies within this interval.
Agents are able to expand, contract and translate this interval as desired. An agent’s price beliefs are updated after each of the agent’s offers has been resolved. During this resolution, the system determines if a trade occurs, the number of units that traded, and the trade price. The offer resolution mechanism will be discussed in detail in the next section.

When an agent’s offer is accepted, this is taken as evidence that the agent’s price belief is accurate, and when an offer is rejected the agent learns that their price belief is inconsistent with the market. Even in the case of an accepted offer, it is beneficial for an agent to anticipate future price changes due to supply/demand imbalances. The price belief update performed will depend on whether the agent has generated a bid or an ask. The update procedures are described by Price-Update-From-Bid and Price-Update-From-Ask. These updates take into account partially filled orders, the difference between the offer price and the historical market average, and the current supply and demand for the commodity being traded.

When an offer is accepted, the may only need to take supply and demand into account for a minor change. When an offer is rejected, the agent has a more difficult choice. The agent may have offered far from the mean, causing its offer to be placed far enough down in the offer book that no matching offer could be found, or the offer may have exceeded the limits of the matched agents. No seller will agree to sell a product below the cost to produce that product, nor will any agent agree to pay significantly above the observed trading range. In this case the rejected agent will want to reevaluate its price belief, translating its price range towards the mean and increasing the size of the interval to reflect its lack of confidence in the belief.

Agents that are very low on inventory and have had their bid rejected will make a more aggressive adjustment of their price belief in an attempt to leapfrog their competitors.

If none of these special situations exist, a rejected agent will examine the current round’s supply and demand for the commodity and if there is a large imbalance adjust their prices in anticipation of price adjustments by potential trading partners.
Price-Update-From-Bid(Commodity)

1 if at least 50% of offer filled
2    then
3        Move price belief limits inward by 1/10 of upper limit
4    else
5        Increase upper belief limit by 1/10 of current value
6 if less than full market share and inventory < 1/4 capacity
7    then
8        displacement ← price diff from mean/mean
9        Translate belief range upwards by displacement
10 elseif offer price > trade price
11    then
12        Translate belief range downwards by 110% of overbid
13 elseif supply > demand and offer > historical mean price
14    then
15        Translate belief range downwards by 110% of overbid
16 elseif demand > supply
17    then
18        Translate belief range upwards by 1/5 historical mean price
19    else
20        Translate belief range downwards by 1/5 historical mean price
Price-Update-From-Ask(Commodity)

1  weight ← percent of order unfilled
2  displacement ← weight * price mean
3  if No units sold
4     then
5       Translate believe range downwards by 1/6 displacement
6  elseif Have less than 75% of market share
7     then
8       Translate believe range downwards by 1/7 displacement
9  elseif offer price < trade price
10     then
11       Translate belief range upwards by 120% of weight * overbid
12  elseif demand > supply
13     then
14       Translate belief range upwards by 1/5 historical mean price
15  else
16       Translate belief range downwards by 1/5 historical mean price

5.4. Experimental Procedure

Experiments were performed for both the general example and a large number of examples using random production rules. I created several simulators that allow computer controlled trading agents to buy, sell, produce, and consume commodities. These agents were assigned roles so that an equal number of agents were initially in each role. Between 1000 and 10000 rounds of simulation were performed, during which time each agent interacts with other agents either as a buyer or a seller, and unsuccessful agents are replaced by new agents in different roles.

The individual trading agents were designed to provide realistic behavior by maintaining unique state that is updated based on their individual experiences. I required agents
to be partitioned into classes based on role to take into account for the human tendency to specialize labor. In the real world an individual typically holds one job at a time, although the individual may change jobs over time. I restricted agents from trading in commodities that they did not require for production, nor produce. This is not a significant restriction as a rule could be added that converted a commodity into itself, making an agent technically a producer without performing any real production. For example, a rule that transforms one unit of commodity A into one unit of commodity A would allow the agent to buy and sell this commodity, without changing the amount present in the world. This rule would allow an agent to speculate in commodity A. An assumption that an agent will act to maximize profits with only knowledge that it personally gained indicates that agents will act in a rational and fair manner.

The following assumptions are made regarding the trading agents:

- Traders are heterogeneous, having unique pricing beliefs, roles, inventories, and money on hand.
- Traders follow role-specific rules for consuming and producing commodities.
- Traders use an arbitrary unit of currency as a standard for pricing commodities.
- Traders only trade in commodities that they personally produce or consume.
- Traders are allowed to maintain a limited inventory of each commodity.
- Traders act to maximize their long-term profits.
- Traders do not have perfect knowledge of the market.
- Traders learn from personal experience.

The random number generator was assigned a unique seed for each run. The use of random numbers to determine prices within a confidence interval, or to determine if an unexpected event occurs caused the simulator to produce different results, but each similar in the general behavior. As we will discuss in a later section, our simulation exhibits a chaotic sensitivity to small changes in the initial conditions.

In each round of simulation each agent performs a production operation, generates offers to buy or sell certain commodities, and delivers these offers to the auction house. The
central auction house collects these offers and stores them in separate offer books (one book for bids, one book for asks). Once all agents have had an opportunity to enter their offers, the auction house resolves the trades using a distributed double auction, as described by Steiglitz, Honig, and Cohen [78].

**Simulation-Loop**

1. for round ← 1 to number-of-rounds
2. for each trading agent
3. do perform production
4. generate offers
5. resolve offers

Production can be generalized as the conversion of one set of commodities (referred to as a *basket*) into another basket of commodities. The details of how this is performed is implementation dependent, but in general one verifies that the necessary materials are on hand, removes these from an agent’s inventory, and adds the production product to the agent’s inventory. I therefore assume that each agent maintains a separate inventory capable of holding an arbitrary number of each commodity. In practice a game may place limitations on the size of this inventory.

An agent creates offers by examining all commodities that it either consumes or produces. If the agent is running low on a commodity that it consumes, CREATE-Bid is called to create an offer to buy an appropriate amount of this commodity. The offer is then sent to the clearing house where it is added to the bid book for that commodity. Similarly, if an agent has produced some commodities that it does not need, CREATE-Ask is called to create an offer to sell an appropriate amount of this commodity. This offer is sent to the clearing house and is added to the ask book for that commodity.

Once each agent has had the opportunity to add a set of offers to the appropriate offer books, the offer books are shuffled to remove any bias due to the order the agents were
processed, and both books are sorted by price. The central clearing house will then use a
double-auction to determine the resolution of each of these offers.

I was interested in the quality of the simulations only to the extent that it allows
me to provide prices that appear reasonable to players. So while I have no strict need for
high quality results, I sought techniques that were fast and gave good behavior. Double
auctions were selected both for their efficiency, and their ability to approximate theoretically
predicted behaviors (see, for example, Smith [75], and Gode and Sunder [23]). The use of
these auctions in experimental economics for the past fifty years gives me confidence that
they represent a sound technique.

In this type of auction, RESOLVE-OFFERS matches the highest bid with the lowest
ask, a trade occurs, the offers are updated to reflect the quantity of a commodity exchanged,
and any offers with zero units unfilled are removed from the book. This process continues
until either the bid or asks book is emptied. The offer books are shuffled at the beginning
of a round to eliminate bias among agents with the same offer price. Note that when this
matching stops one of the books will likely have offers remaining, and these are reported to
the issuing agent as being rejected. During offer resolution, the minimum of the bid and ask
quantities are exchanged at the average of the bid and ask price as discussed by Nicaolaisen
et al. [47]. Inventories of each agent are adjusted by the amount of the trade, and the amount
of currency each agent has is also adjusted.
**Resolve-Offers (Commodity)**

1. Shuffle both bid and ask books for Commodity
2. Sort bid book in order of decreasing offer price
3. Sort ask book in order of increasing offer price
4. **while** both books are non-empty
5. **do**
   - buyer ← the first bid in the book
   - seller ← the first ask in the book
   - quantity traded ← Min(units offered by seller, units desired by buyer)
   - clearing price ← Average (seller’s offer price, buyer’s offer price)
   - **if** quantity traded > 0
     - reduce units offered by seller by quantity traded
     - reduce units desired by buyer by quantity traded
     - transfer quantity traded units of Commodity from seller to buyer
     - transfer clearing price * quantity traded from buyer to seller
     - both seller and buyer update their price model
   - **if** quantity offered by seller = 0
     - remove the first ask from the book
   - **if** quantity desired by buyer = 0
     - remove the first bid from the book
5. Remaining offers are rejected and the issuing agent updates its price belief

At the end of each round, agents are notified of the quantity of commodity traded as a result of their offer. This notification contains market statistics for the current round, such as trade volume, the average price for trades, the high and the low price for the commodity in the offer. Agents will then update their personal price models to reflect their belief in
the true value of this commodity. Note that there is no single true value for a commodity, but rather a set of beliefs held by each agent that trades in a commodity. Over time it is observed that the agents converge to a single shared belief in a commodity’s value, although external events (shocks) can cause the market to shift to a new shared belief.

Although individual agents in the world maintain no personal history, the clearing house does maintain some historical information that is available to all traders in the market. Agents are therefore required to adjust their personal beliefs about the value of a commodity based on this public information and any information privately learned from prior offer resolution. This public information consists of:

- The average price for each commodity within some user-defined window.
- The average quantity of each commodity offered for sale within some user-defined window.
- The average quantity of each commodity bid on within some user-defined window.

5.5. Agent Replacement

Ideally, agents in a game should change roles when there is economic pressure to do so. If I treat these roles as professions, I may evaluate an agent’s performance by their profitability, and have them decide to change jobs when the agent is no longer profitable. These changes in roles are necessary to provide variations in supply and demand for commodities, as commodities high in demand will attract new agents and in turn increase the available supply. My experiments have shown that the simulated markets move towards a set of agents supportable under the current economic conditions, and reallocate agents when market conditions change. In practice, simulated market conditions are constantly changing so the market never converges to a stable set. This automatic reallocation of agents is a benefit to the game designer, as it allows adjustments in the population of NPCs without explicitly coding for causative events. For example, an interruption in supply for a commodity (such as timber) will affect industries directly dependent on the commodity (shipbuilders for example) as well as indirectly (farmers who provide food to the shipbuilders). These external events may, depending on the magnitude and duration, cause agents to go bankrupt.
An agent that is unable to remain profitable will eventually go bankrupt, and be replaced with a new agent of the currently most profitable type. This profitability statistic is a moving average of profits for a particular type of agent over some user-define number of prior rounds, ensuring that recent performance is evaluated. I have seen good results with windows between 8 and 15 rounds, but a particular set of production rules may work better with other values. It is a reasonable assumption that a recently bankrupt agent was not in a profession that was doing well, and therefore this replacement strategy acts to maintain a constant population size but varies the composition of agent types. As a result, as long as bankruptcies occur, the simulation will make adjustments to the distribution of agent types. Ideally, absent of some external disruption, there will be a point where no future bankruptcies occur, as the market is capable of supporting each agent indefinitely.

My results are consistent with accepted economic theory. Adam Smith [74] theorized that people trading in an open market would lead to the production of the proper quantities of commodities and the division of labor. My results support this belief, since agents that are not profitable become bankrupt and are replaced by more profitable agents.

The first fundamental theorem of welfare economics states that a market with a supply/demand equilibrium leads to a Pareto-efficient allocation of resources, meaning that no change to the resource allocation can be made without making at least one trader worse off [4]. This would suggest that when the market is in equilibrium the allocation of agents to roles will over time tend to an optimal value [41]. In practice no market ever moves into equilibrium, but instead will move into a neighborhood that is near equilibrium and oscillate about the equilibrium point [75].

The economic system will constantly attempt to determine a distribution of agent roles that results in market equilibrium, however it never reaches this goal and instead orbits around equilibria until market conditions establish new equilibria points. At this time, the agent distribution is seen to adjust and move towards these new equilibria.
5.6. A General Example

My simulation used various techniques to exercise this economic system. The most general form of production was to allow the simulator to call agent-specific routines that would update inventory. This allowed me to implement complex production rules without restriction, while updating the price models in a manner consistent with an actual game. One such ruleset allowed agents to be farmers, miners, refiners, woodcutters, or blacksmiths. These agents produced, and consumed food, ore, wood, metal, and tools according to the production rules defined by Farmer-Production, Miner-Production, Refiner-Production, Woodcutter-Production, and Blacksmith-Production.

This example was created to illustrate a typical economy, as might be found in a fantasy role playing game. As the rules were implemented using arbitrary code, the designer is free to create as complex a ruleset as desired.

Farmer-Production

```
1 if has-wood and has-tools
2   then
3       produce 4 units of food
4       consume 1 unit of wood
5       break tools with prob 0.1
6 elseif has-wood and has-no-tools
7      then
8       produce 2 units of food
9       consume 1 unit of wood
10  else
11    agent is fined $2 for being idle
```
**Miner-Production**

1. if has-food and has-tools
2. 
3. produce 4 units of ore
4. consume 1 unit of food
5. break tools with prob 0.1
6. elseif has-food and has-no-tools
7. 
8. produce 2 units of ore
9. consume 1 unit of food
10. else
11. agent is fined $2 for being idle

**Refiner-Production**

1. if has-food and has-tools
2. 
3. convert all ore in inventory into metal
4. consume 1 unit of food
5. break tools with prob 0.1
6. elseif has-food and has-no-tools
7. 
8. convert at most 2 units of ore into metal
9. consume 1 unit of food
10. else
11. agent is fined $2 for being idle
WOODCUTTER-PRODUCTION

1 if has-food and has-tools
2 then
3 produce 2 units of wood
4 consume 1 unit of food
5 break tools with prob 0.1
6 elseif has-food and has-no-tools
7 then
8 produce 1 unit of wood
9 consume 1 unit of food
10 else
11 agent is fined $2 for being idle

BLACKSMITH-PRODUCTION

1 if has-food
2 then
3 convert all metal in inventory into tools
4 consume 1 unit of food
5 else
6 agent is fined $2 for being idle

5.7. Random Generation of Production Rules

In addition to testing with the production rules described above I developed a second simulator that creates random production rules in order to demonstrate that my results are not dependent on any single set of rules. To achieve this I expressed the production rules in a matrix format, allowing the simulator to assign random values to the matrix and then simulate a set of agents operating under these rules. I claim that if I observed acceptable behavior from economies using randomly generated rules, then I have a suitably general
solution that will perform well with rules that a designer might select. I do not claim that all rules will perform well, only that a large set will. In particular rules that express a non-closed economy (where agents consume a commodity that is not produced) are not going to produce pleasing results under any economic system.

The matrix form for production rules defines a rule as a set of commodities that is converted into another set under a probability distribution. For example, the rule shown in Equation (1) allows an agent to convert two units of Commodity$_1$ into one unit of Commodity$_4$. Additionally, the agent is required to possess one unit of Commodity$_3$ that is consumed 10% of the time. In a simulation round, an agent is permitted to perform production using one of these rules. If the agent does not possess inventory listed on the left-hand side of a rule, the production is not allowed and the agent must consider other rules. It is therefore possible for an agent to be unable to perform production in a given round due to inadequate inventory. In this situation the simulator assesses an idleness tax, to ensure that non-productive agents are eventually driven bankrupt. As each type of agent was allowed to select among several rules, I ranked the production rules in order of preference and had agents use the first rule in their set that they were able to execute.

This use of multiple production rules for an agent-type along with probabilities for terms in production rules allows me to model complex behavior including conditionals (such as, does the agent possess a tool or catalyst represented by Commodity$_3$ in this example).

\begin{equation}
2 \times \text{Commodity}_1 + \text{Commodity}_3 \Rightarrow \text{Commodity}_4 + \text{Commodity}_3 (p = 0.9)
\end{equation}

Since these simulations were requiring large numbers of calls to the random number generator, I was concerned that the simulation might exceed the default random number generator’s period and bias my results. I replaced this generator with a Mersenne Twister random number generator (MT19937), which has a period of $2^{19937} - 1$.

Due to the large number of random experiments I performed, I was unable to study all of the results. I therefore established screening criteria to filter out unacceptable results,
with the intention of counting the number of simulation runs that were well behaved. I arbitrarily selected a set of desirable features for a price graph, and then modified the filters until I was seeing only these types of graphs. Many of these features were based on the belief that I need to observe regular trades for each commodity if I am to judge the market’s overall performance. I further wished to see that prices change over time, but wanted some long-term stability in prices. I define stability as the tendency of prices to return to equilibrium, as opposed to diverging to 0 or infinity. The exact values in the filters were therefore determined empirically from a representative set of price graphs. The final criteria used were as follows:

- Each commodity was produced by at least one type of agent.
- Each commodity was consumed by at least one type of agent.
- No commodity goes more than 20% of the total number of rounds without trading.
- The average trade volume for a commodity is greater than one unit per round.
- The variance in commodity price is between 0.025 and 7.5 times the largest trading price.
- The average change in price is between 0.02 and 0.9, the variance of this change is also between 0.02 and 0.9.
- Fewer than 2 price inflections occur per round on average.
- The variance in the time between price inflections is less than 1.2 times the number of rounds.

I am therefore comparing both the variance in price and the variance in the first derivative of the price.

5.8. Experimental Results and Evaluation

When evaluating the performance of a complex system, one realizes that combinatorial growth in the number of possible interactions makes exhaustive analysis intractable. In this economic simulation, however, there are a limited number of ways in which a trader (human or agent) can affect the economy. Buying a large quantity of a commodity can reduce available supplies for other traders, and encourage agents to switch production to this
commodity. Selling a large quantity of a commodity can increase supplies, and discourage agents from producing more of this commodity. And finally, I assume that agents have the ability outside of the market to interfere with supply and demand, by blockading an area or destroying resources. My prime concern is with showing that this economic model recovers from even extreme behaviors, and therefore that the range of possible trader behaviors will not destabilize the system in the long term.

5.8.1. A Representative Ruleset

I first consider the general fantasy-RPG themed example discussed in Section 5.6. I feel this ruleset is representative of the types of rules one might use within a game.

Figure 5.2 shows the behavior of a simulation over 2000 rounds. I arbitrarily decided that each round represents one day’s activity in game time, and therefore have almost 5.5 years of price data. A graph of the supply/demand ratio over time is shown in Figure 5.4, and demonstrates that the economy undergoes the full range of supply and demand imbalances. A concerted effort by multiple players could artificially bring about such an imbalance, but I see no long-term effects. Figure 5.2 demonstrates that the economy recovers once an imbalance is eliminated. This auto-recovery is necessary for long-term stability, and in particular eliminates the need for human intervention.

When looking at this long-term behavior, I observe that trading occurs within a bounded range, suggesting that prices are orbiting a stable equilibrium rather than diverging to either 0 or infinity. The system is constantly attempting to move back into equilibrium, while it is undergoing further perturbation as the result of trades. I consider the graph in Figure 5.2 to show long-term stability in prices as the system recovers from disruptive events on its own. I also note that prices are not precisely predictable, although some relationships can be seen over time. A close up of one part of the graph is shown in Figure 5.3. There is a correlation between commodities that are dependent on one another, as the prices of products move with the price of the raw materials. For example, the price of refined metal (shown with a dash-dot pattern) tends to rise and fall with the price of unrefined ore (drawn with dashes). The magnitude of the changes differs, but the local maxima at rounds 1020,
1060, 1110, 1140, 1170 and 1200 occur in both graphs.

Paradoxically the price of refined metal in Figure 5.3 appears to increase before the price of refined ore (the precursor product). It should be kept in mind that these figures show average prices over time, and an agent may choose to raise its offering price the moment it experiences resistance to a price, even if the market average does not yet reflect the belief in a price change. While I see correlations in prices, I also see the independent movement of prices. The prices of ore and metal in this case are not translations of each other, but vary within a limited range. This long-term stability is desirable, as it shows that the system does not undergo runaway inflation, but instead self-corrects.

The sensibility of prices is a subjective measurement, but as long as the simulated agents behave rationally one must accept that the prices they trade at make sense. I note in particular that as the prices of raw materials go up, the prices of finished goods increases with a slight delay as inventories are used up. Allowing an agent to stockpile a certain amount of a commodity provides a short-term buffer against price changes, and tends to dampen price swings. An internal consistency in prices occurs since the economy is a closed system, and each transaction influences future transactions. This consistency is predicted by accepted economic theory, and to the extent that my results agree with theory I am able to claim that my system’s behavior makes sense.

5.8.2. Applications of Economic Theory

Proponents of general equilibrium theory believe that supply and demand will equalize over time, however my results do not support this. In particular Arrow and Debreu [3] argued in favor of this equalization, assuming that traders in the market had perfect information and responded instantly to market changes. Their argument is intuitive when one considers that an imbalance in the supply/demand ratio should result in price changes that in turn result in changes to supply and demand and return the system to equilibrium. However, I believe that in the real world Arrow and Debreu’s constraints do not hold, nor do they hold in my simulation. Traders create offers based on their belief in the market price, but without knowing the beliefs of other individuals. Traders are able to estimate the beliefs of others
Figure 5.2. The system has stable long-term behavior, moving in and out of equilibrium.

Figure 5.3. Detail of the previous figure, showing similar behaviors among commodities. Based on their observations of trades that complete, but only have perfect information on their own trades and their own price beliefs. Traders also do not respond instantly to market changes, as they only update beliefs after they have tendered an offer and seen how it was received. This delay, coupled with the time needed for market averages to converge following a shift in belief cause the agent to respond slowly to market changes. I believe that this is
a useful property, as it prevents agents from overreacting to short-term market changes, as well as better reflecting how a trader in the real world would respond.

I look to Vernon Smith [75], a pioneer in the field of experimental economics for an explanation: Smith explains that supply and demand can only set broad limits on the behavior of the market, as any successful trades remove a quantity of supply and demand from the market and therefore alter the supply and demand curves. In a later paper Smith [76] explains that “all information on the economic environment is private; far from having perfect or common information” and “prices and allocations converge quickly to the neighborhood of the predicted rational expectations competitive equilibrium”. So in the ideal case supply and demand would converge to the same value, in real experiments they will only be in the same neighborhood. This agrees with my observations of the supply/demand ratio over time.

In one experiment 500 heterogeneous agents were simulated for 10000 rounds of trading, and the supply/demand ratios were graphed over time. These ratios were not constant,
but instead varies between approximately 0.5 and 2, repeatedly crossing the line $y = 1$ (representing the equivalence of supply and demand). I conclude that the market is constantly trying to make these values equivalent, but overshooting and then correcting itself. As this behavior agrees with Smith’s observations, my confidence in these results is further strengthened.

I have observed that agent profitability tends to zero over time, as prices for raw materials increase to the point where buyers refuse to bid on them. Adam Smith [74] discusses a similar phenomenon in the Wealth of Nations (Chapter 10, Part II) where he notes that the landlord will raise prices until the tenant is left with “the smallest share with which the tenant can content himself without being a loser, and the landlord seldom means to leave him any more”. If we look at the average agent profit (by type) over time in Figure 5.6 we see profitability orbiting the zero equilibrium, the disruption due to the external event, and the recovery as profits again trend towards zero. While the external event does create a significant disruption, once the event completes the system returns to orbiting the equilibrium. The long-term behavior of the simulated economies therefore agrees with accepted economic theory in this aspect.

5.8.3. Response to Extreme Stimuli

My system is able to adapt and recover from external perturbations and shocks. This is a useful feature since it addresses the type of market manipulation that players might choose to engage in. Figure 5.5 shows the effect of a short term interruption in the supply of wood. Between round 600 and round 700 woodcutters were unable to harvest wood, simulating a forest fire that has eliminated the resource. We see the price of wood remain fixed during this period, as there are no sales there will be no price updates, since a price is only established when an offer is accepted. The prices of ore and tools begin to rise during this period as both products use wood (to shore up mine shafts and for tool handles) and existing wood inventories are depleted. While wood production resumes on round 700, prices continue to rise for another 50 rounds as the unsatisfied demand greatly exceeds the limited supply. The prices of products that depend even indirectly on wood increase, although there
An interruption in wood supply and subsequent recovery.

is a delay before the market adjusts these prices. In this case the delay is due to accumulated inventories being drawn down, and acting as a moderating force on prices. These prices will eventually resume their previous pricing behavior, but there is a time lag as inventories are depleted and agents start to believe that wood is no longer scarce. Even a serious shock to the economy such as the fire creates no long-term harm, as eventually we observe the system returning to equilibrium at approximately round 950.

One should keep in mind when looking at these graphs that the prices are partially a function of random chance. Tools break at random times, and agents enter bids based on random guesses within their price confidence interval. As a result there can be large price fluctuations if enough of these random events occur in a short interval. This can be a good thing for both the designer and the player, as it means that one may never exactly predict price behavior. However, the overall price trends do follow patterns, and do react to major events (such as the forest fire in Figure 5.5). It should be possible for players to engage in arbitrage if they so choose. A knowledgeable player who becomes aware of the lack of wood could buy up tools and ore and wait for the market prices to increase, then sell them at a profit. Short term price shocks are therefore not a problem as long as the long-term behavior of the economy is consistent.
Figure 5.6. Agent profitability by type for the forest-fire scenario, demonstrating an orbit about the equilibrium.

The allocation of roles to agents is an important product of this model, as it is necessary to know how many agents may be supported in a role. If we consider a community with N agents and M roles, one might need to know how to partition the N agents into M sets such that the agents remain profitable. This type of question comes up when we consider adding a new NPC (the N+1th agent) to the game and wish to know what role this NPC should take on. A village populated entirely by woodcutters would raise the question of how these woodcutters find food, or where all of this wood is going. One may avoid these situations by ensuring that all population distributions are viable, that there is a need for each of the agents and that each of the agent’s needs are met. I can obtain this information from the simulation at minimal cost by observing the profitability of roles over time.

At any point in time there will be an ordering of the M roles by profitability. The more demand there is for a role, the more profit one will expect for agents in this role. This is the result of high demand driving up prices for the products of this role faster than the materials needed by agents in this role. Conversely agents in a role that is not in demand will find it difficult to execute trades, yet will still have overhead (food in this example). When a new agent is added to the simulation, if I bias the role selection by the profitability of the
roles, I ensure that high demand roles gain agents and low demand roles lose agents. At any point in time the population of agents represents a viable community, since agents will be removed when they are no longer able to provide for themselves. Figure 5.7 shows changes in the distribution of agents by role over time, starting from an arbitrary (and unsustainable) distribution. In this example farmers and woodcutters are in high demand since both provide resources needed by other agents, so it makes sense that new agents would favor these roles. The exact number of agents at a point in time is a function of random events, but also of the ruleset and ontology.

Short-term random events prevent the market’s role allocation from reaching equilibrium; however there are stable patterns observed. I also observe that roles producing commodities that are high in demand will have more agents than those producing less needed commodities. As supply and demand change over time, the need for certain roles changes over time, and the market moves towards a different allocation. At each point in time, the number of agents in a particular role approximates the number of profitable agents under current market conditions. As a result, a census of agents in the market allows one to determine a reasonable distribution of agent types, and we are therefore able to create a community of N agents and know that the market will reallocate the roles until an acceptable distribution is found. Furthermore one is able to start the simulation with an arbitrarily chosen distribution (I initially assigned equal numbers of agents to each role), and know that the system will quickly reallocate these agents into more appropriate roles.

5.8.4. Experiments with Random Economies

In addition to my experiments with the general example, I performed a large number of experiments using random economic rules. As discussed earlier, I applied a filtering function to test price graph characteristics and decided if a particular set of rules produced acceptable results. I used UNT LARC’s cluster of PS3 consoles to evaluate these random economies, with the simulation performed on the Cell processors’ SPE units. The high degree of parallelism and the high performance of these Cell processors allowed me to evaluate millions of distinct rule sets in a few hours. I tested approximately 1.5 billion random
Figure 5.7. The distribution of agent roles over time.

economies and found 2.7 million (0.19%) that passed all of the filter criteria. These economies appear to be uniformly distributed, as I observed roughly the same fraction for multiple smaller runs using different random number seeds. I calculated that there were approximately $2^{160}$ distinct matrices, and assuming that 0.19% of these have acceptable performance we have more than $2^{150}$ economies to choose from. Figure 5.8 shows two representative price graphs, demonstrating price behavior under random rules. Systems using random rules are less likely to be well behaved than those a human designer might create. Based on the large number of well-behaved random economies, I believe that the behavior of this economic system is independent of any single ruleset, and I am confident that a designer would be able to create a ruleset that performs better than random.

The calculations required to update the economy can be carried out very quickly. A single 3.2GHz SPE can perform approximately 200000 agent updates per second, including simulator overhead such as data logging. Most games do not have this many NPCs, nor do they require them to be constantly buying and selling. Only two floating point values are required per commodity per agent, which I feel to be particularly light. I am confident that adding an economic simulation to a game will not add a significant burden in terms of either processing time or memory requirement, and as a result this technique is feasible.
5.8.5. Observed Chaos

Chaos is defined as sensitivity to initial conditions affecting the outcome, and this describes behavior seen in our simulation. I first became aware of the issue when I observed differences in the output when the compiler’s optimizer was turned on. My investigation of this phenomena concluded that even though I used double-precision floating point in my implementation, discretization errors were present in my intermediate results due to the inability of the compiler to express certain floating point values (such as 0.1) as exact values in binary. Furthermore, the optimizer was reordering floating point operations, causing these discretization errors to propagate differently than they would in an unoptimized version.

The magnitude of the sensitivity was demonstrated by a one-time addition of $10^{-9}$ units of currency to a single agent during a simulation. I observed that several agents went bankrupt who would otherwise have remained solvent. In addition, after 500 rounds of simulation the price of certain commodities varied by around 20% from the normal runs. This magnitude of error is within the observed discretization error for 0.1, and can be expected to occur normally during simulation when calculating moving averages.

This chaotic behavior is not an error, but can be expected in an iterative simulation that employs feedback. My model was tested with production rules that coupled different agent types. This means that each type of agent produces a product needed by at least one other type of agent, and uses a product produced by at least one other type of agent. As a result of this coupling, any perturbation of one agent will propagate to other agents. Furthermore, an amplification effect occurs as a result of continuous errors becoming discrete.
errors. Say that one agent is saving money to buy a needed tool, and in one case the agent comes within $\epsilon$ of being able to afford the tool before going bankrupt. In another run, due to chaotic effects, this same agent may gain an extra $\epsilon$ of currency and buy the tool. By owning this tool, the agent may become profitable and remain in business. As a result, this agent continues to have an impact on the economy, buying and selling goods and affecting prices on these goods. A small error of $\epsilon$ has a much larger affect, once it results in a discrete change (the number of tools owned by the agent changing from 0 to 1).

I believe that this behavior is desirable, since it makes the impact of player actions harder to predict. It is important to note that the changes to the economy outside of the $\epsilon$ error are justifiable under the production rules.

Oxley and George [52] note that economics can be chaotic. Rosser [62] also gives a good explanation of economics as a complex dynamic system. My model does indeed have the following characteristics found in chaotic complex systems (see Arthur, Durlauf, and Lane [5]):

- Disperse interaction: Agents interact with a subset of other agents
- No global controller: No single agent may control the market
- Tangled interactions: The production models are usually coupled.
- Continual adaptation: Agents constantly update their beliefs about prices
- Perpetual novelty: In a chaotic phase, markets are created and destroyed as the agent mix stabilizes. Also until agents’ beliefs in commodity prices converge commodities will frequently trade at prices that contradict these beliefs.
- Out of equilibrium dynamics: Prices may not move to an attractor, but may orbit perpetually.
CHAPTER 6

QUESTS

If life is a video game, then most of us have no chance of winning, if by winning you mean succeeding in a quest or saving a princess.

Douglas Lain

Quests provide players with purpose and direction, and encourage the players to explore the world and interact in ways which might not otherwise occur to the player. A quest is a task which challenges the player to complete goals in return for some reward [17]. Most quests for computer RPGs are created with an intended solution in mind. This differs significantly from a tabletop RPG where more than one solution is to be expected.

A GM in a tabletop game will typically create a scenario which players will then play out [19]. In CRPGs, these scenarios are prepared by designers, and players select which they will experience. The tabletop GM has feedback during gameplay, and can adjust the scenario as needed. The CRPG quest is created a priori and is static.

A GM will want to provide players with a large assortment of quests so that the player always has new experiences available. These quests provide motivation for the player to continue playing the game. For commercial games, such as CRPGs and MMORPGs, this type of content can motivate the player to continue financially supporting the publisher.

A player who is unsure of what to do next may appreciate a choice of quests that allows them to select from a set of possible goals. Quests also serve as a story telling device, since a player completing a quest is required to interact with NPCs who can explain why various tasks need to be performed and entertain the player with stories set in the game world. When I looked further into the factors that can motivate a game player I considered Harackiewicz et al. [26] who provided a list of factors that determine intrinsic motivation. In particular I wish to focus on the concept of perceived task value, which was defined as a belief that a task is meaningful or important. In the context of a computer game I interpret this as meaning that in order to maintain a player’s interest the player must associate meaning with their actions. I believe meaning for a player comes from the belief that their actions
have an impact on the world, in other words that the world state has somehow changed as a result of their actions.

When I decided to investigate quest generation, I naturally wanted to answer the question “what is a quest” in great detail. I studied over 3000 quests from several games, before setting on a quest model. At that point a detailed study of 753 quests from four games (Everquest, World of Warcraft, Vanguard Saga of Heroes, and Eve Online) provided examples of representative quests which I used to update my model. I stopped my analysis after 753 quests because I was no longer seeing new structure, or quests which did not fit the model.

Patterns of actions were seen repeated among most quests, although specific details varied. For example, the task of delivering an item to another NPC is commonly seen, but the specific item and the specific destination change. Once the quests were abstracted from these sorts of details, the action patterns became easier to distinguish.

6.1. Previous Work

Quests play a significant role in RPGs both in terms of providing narrative, and in giving players an opportunity to interact and modify the game world. Jeff Howard [28] discusses a process for translating literary narratives into quests, which supports the idea that quests are a form of narrative. While Howard’s definition of a quest appears to be broader than the one we use, his paper is useful as it looks at the problem of quest design from a different viewpoint. Wardrip-Fruin [90] discusses the concept of player agency, which I extend by using social psychology to show that agency is part of what contributes to player motivation. Henry Jenkins has argued that narrative has a proper role in games, although his definitions seem overly broad [29]. Jenkins lists a number of game elements which could be incorporated into a story. Michael Mateas responds to this by pointing out Jenkin’s omission that player’s actions are story elements [43]. Since I do not directly address narrative in my work, there is no need for us to speculate on Jenkins’ position, although it seems he would agree that quests are a valid narrative tool.

Previous research on RPG quest analysis falls into two categories: structural and
functional. Autonomous generation of quests involves not only a suitable form for a quest, but a means for the generator to know when to generate a quest and the ability to ensure that the quest makes sense in the current game state.

Although some classification studies have been performed [1, 80, 81], these do not necessarily have the goal of autonomous generation of quests. Sullivan has also classified player actions, and has came to similar conclusions [81]. Ashmore and Nitsche [6] propose procedural quest generation using key-lock structure quests, rather than NPC interactions. This structure has potential for autonomous generation, but lacks the sense of purpose found in quests derived from NPC goals. Jill Walker [89] classifies World of Warcraft quests into exploration and combat quests, which is significant as it is an early attempt to classify quests. We believe our classification is more general, in that it can express additional types of quests. Dickey [16] provided a classification system which appears suitable for autonomous generation, but does not claim to provide complete coverage for RPG quests nor does it address how quests could be generated based on this system. My quest decomposition is closely related to world state, which makes it fairly straightforward to select a quest type which modifies some piece of state. Aarsath discussed quests in general, although using a somewhat liberal definition which could be applied to first-person shooters [1]. Quests were divided into three categories: place-oriented, time-oriented, and object-oriented. My categorization using various NPC motivations is more fine-grained, which is desirable for a generative taxonomy. One interesting finding was the relationship between story and space constraints, which suggests that story oriented games limit player agency more than other games. This seems reasonable, as the limitations of telling a prewritten story require players to follow a script, perhaps with the illusion of some amount of free will. This observation would not hold if a game were able to generate story arcs and plotlines on the fly, assuming the story generator was capable of adapting to changes in the world state.

Gruenwoldt also discusses reactive NPCs in terms of networks, where a state change which affects one entity in the world would encourage appropriate reactions from related NPCs [24]. Such a system could be used in a game that uses our generation framework, but
the two concepts do not overlap. Gruenwoldt goes into depth in the area of how attitudes change, where we avoid specifying how or how much of a change occurs as the result of an action. In other words, we are not concerned at present with how an NPC decides that it is threatened, but instead what type of quests might be appropriate for an NPC which enters this state.

Sullivan et al. [80] discuss the state of MMORPGs, noting the lack of player choices. They note that in general such games have no central story arc. They present a taxonomy of player actions which is similar to ours, the differences may amount to nitpicking. Their quest generation system is interactive, requiring the assistance of a human author, and therefore places the creative burden on the user.

The University of Alberta’s ScriptEase system has been used to create quests (actually complete scripts for NeverWinter Nights), but is an interactive system [14]. In contrast, our system is autonomous as it does not rely on human input while the generator is running. The distinction in authorship is important, as one of our implied claims is that computers can create content (not just format it). There is similarities in our approaches, since both systems are pattern-based and expand on predefined patterns. Their paper is significant in the introduction of adaptation operations. This is similar to our expansion of abstract quests (increasing the length of a quest). One of our contributions is a list of abstract patterns which we claim cover quests in a series of production games. An earlier paper from this group [44] discussed their use of patterns, but did not enumerate the list. A follow up paper [51] also discussed patterns, but did not enumerate them.

Paiva et al. discusses the relationship between users and characters, and the relationship to believability. Believability is an important goal, but the emotional response is not directly relevant to quest generation. This may be more appropriate for dialog generation, in particular pragmatics [53]. Another paper on emotional state of agents discusses motivation, desires and goals [39]. This paper is interesting, as it addresses social networks among agents (a subject of future work), however it focuses on planning and character behavior which would be more appropriate in a discussion of runtime AI than content generation.
But their use of character modulations is very similar to our use of motivations.

6.2. Quests as Narrative

Ideally the quest can be seen to tell a story. In a tabletop game, the story is the collaborative product of the GM and the players. In a CRPG the story is fixed before play begins, if such a story exists in the first place. Most RPGs have a large number of quests that do not involve any storyline. For example, the side quests in Skyrim have no storyline involvement.

In general we do not know what will happen until the quest plays out. “The role-playing game exhibits a narrative, but this narrative does not exist until the actual performance.” [40]. The story unfolds as more of the quest is explored. A human GM is able to adapt to unexpected actions by the player, allowing for potentially unlimited outcomes. CRPG quests limit the ability of the player to deviate from a predetermined solution, their story is therefore immutable.

“Consistency is the key to a believable fictional world. When they go back into town for supplies, the PCs ought to encounter some of the same NPCs they saw before. Soon they’ll learn the barkeep’s name – and she’ll remember them as well.” [13].

6.3. Quests as Functional Game Elements

To the player, a quest is an in-game task they have the option to perform. This is the case for pencil and paper games, as well as CRPGs. Quests consist of some objective, a potential reward, the items used during the quest, the characters used during the quest, and any special locations used during the quest.

A quest also consists of a set of milestones which must be met in order to complete the quest. Some may be optional, but usually these are required. The player is not always required to complete these activities in a particular order. These milestones may not be explicitly known by the player.

Quests must therefore contain dialog which in-game characters can use to explain and direct the player. Often this dialog is in the form of hints, rather than explicit directions.
Part of the challenge of a quest is to figure out how a particular goal can be met.

As a player begins a quest, their participation and progress must be tracked. At each stage in a quest, new options and opportunities open up for the player. Player actions potentially affect quests which are in progress, it is up to whoever is managing the game to determine the outcome of these actions. A human GM has the most flexibility, computers are limited to events and outcomes for which they are prepared.

6.4. Analysis

One of the first steps needed, when I was analyzing a quest, was to abstract away the details. Consider for a moment the 'Going Postal' series of quests in Everquest. These quests have the League of Antonican Bards ask the player to deliver mail to another city. If accepted, the player is given a letter and told to take it to a particular NPC in another city. This second NPC will then pay the player upon receiving the letter.

The specific details, such as the start and ending locations, the dialog spoken by NPCs, the fact that the item to be delivered is a letter, are not important. In the abstract, this quest involves delivering some arbitrary item to a remote NPC in return for some reward.

Consider the quest 'Delivery to Nurel', also found in Everquest. The player is asked to deliver a 'Scroll of Tul’Nartek' to the NPC Nurel Uralu, who will give the player some potions as a reward. This quest also involves delivering an item to a remote NPC in return for a reward.

Both of these quests are, at some level, identical. Only specific details have changed, but the actions taken by the NPCs and players remain the same. The details, such as dialog, are important, since they provide the illusion of a unique quest. But the common structure is the critical element. One could reproduce the sequence of actions and create new delivery quests just by changing the dialog, items, and locations used.

The goal of quest analysis is therefore to study quests from existing games and identify their abstract structure. Then study a set of these abstracted quests and recognize common patterns. These patterns are the basis of the strategies referred to in Chapter 12, when I discuss quest generation.
6.5. An Example

An example of one of the quests studied is shown in Figure 6.1; one of the quests from Everquest. The quest starts at (a) with the player learning of the quest from a character named Cindl in the town of Halas. She asks players to bring her polar bear skins in return for armor pieces. In part (b) the player has located a polar bear cub, and is in the process of fighting it. If the player is lucky, in part (c) the bear will have dropped a polar bear skin (the white bear pelt), as opposed to the common ruined bear pelt. In part (d) the player returns to Cindl and brings her a polar bear skin. She thanks the player and exchanges the skin for a random armor piece plus a gold coin.

The significance of this quest is that a new player has no armor, nor money to purchase either the armor or the materials to make armor. This quest provides a level-appropriate challenge which results in both armor and money for the player. The rarity of the polar bear skin introduces an element of luck into the quest, as well as making the quest last longer.

Abstrating away the details of this quest gives me a list of actions (discussed in more detail in Chapter 8). I see that the following actions are required of the player:

1. Exchange information
2. Go to an entity
3. Damage an entity
4. Exchange items with an entity
5. Exchange items with an entity

In step (1), the player talks to the NPC and learns of the quest. In step (2) the player locates a polar bear cub, on the tundra. In step (3) the player kills the polar bear cub. In step (4) the player acquires a polar bear skin from the corpse of the cub. And in step (5) the player exchanges the polar bear skin for the armor and gold.

The game server needs to keep track of the progress of the player through the quest. But steps (2) and (3) do not need to be explicitly checked. If the only way the player has of obtaining a polar bear skin is by killing a polar bear cub, then acquisition of the skin implies the previous steps were completed. When server-side tracking is covered in Chapter 13, this
Figure 6.1. Cindl’s Polar Bear Collection (from Everquest)
type of optimization will be seen to reduce the overhead of tracking quests.

This sort of analysis allows me to see the basic structure of a quest, and to in turn identify common sequences of actions. This is the beginning of the formation of strategies.
CHAPTER 7

NPC MOTIVATIONS

... the NPCs should be as complex and richly detailed as the PCs – although the focus should be on motivation and personality, not game statistics.

D&D Dungeon Master’s Guide.

In addition to recording the structure of quests, I determined the motivation of the NPC assigning the quest based primarily on the NPC dialog used during the quest. This is a subjective assignment, but for the most part it was clear why the NPC wished the player to undertake a quest. The list of observed motivations is shown in Table 7.1.

By studying the motivations of the NPCs who offer quests I was able to form an association between a character’s motivation and the types of tasks that character wishes the player to perform. This association is useful to a generator, as it would likely start with some world state and need to generate an appropriate quest. If an NPC lacks food, I am able to take advantage of prior observations that NPCs lacking food requested delivery or courier quests, where the player would be asked to obtain food and deliver it to the NPC. The advantage of this technique is that rather than attempting to try various actions and see if they address the NPC’s need, I am able to go directly to some viable quest structure.

The generator consults its knowledge base to determine the currently dominating motivation for an NPC. This information could be updated dynamically by the server, but is not presently being changed. It is sufficient to provide the generator with an NPC and one of the motivations in Table 7.1.

I am not attempting to model the mental state of an NPC, but recognize that there are many viable ways of performing this modeling that can be used to identify the currently dominant motivation for an NPC. The value of such a mental model is that it provides the NPC with a purpose. This usage of motivation is as a limited representation for NPC goals. Events in the game may cause an NPC to adopt a motivation, and this should be maintained until other events cause the NPC to adopt a new motivation. In this manner, NPCs may
become reactive to events in the game, which enhances the realism and believability of the game.

I believe this use of NPC motivations to create quests is novel, significant, and reactive. While other work has been done in the area of mental models, there does not appear to be substantial use of NPC motivations by an autonomous quest generator in the prior literature. This use of NPC motivation is significant, since it guarantees that quests have a purpose consistent with an NPC’s state. And finally, this generation technique is reactive since generator adapts indirectly to changes in the world. Events modify NPC state, and world state such as item prices. These values then influence the quest generation.

7.1. Knowledge

Knowledge is important to NPCs since it enables them to perform actions they would not otherwise be able to. For example, knowing an enemy’s location allows an NPC leader to plan an attack or retreat. Asking a player to help gather knowledge on various subjects is a good way to encourage a player to explore areas of the world they might otherwise avoid, and interact with NPCs they might never have met.

Knowledge can be gained by interacting with NPCs, Items and world locations. Specifically, one can observe or talk to NPCs, collect items and either study them or deliver them to someone else for study, and explore locations in the world. Table 7.2 summarizes these strategies.
### Table 7.2. Strategies for gaining knowledge

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivering an item for study</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
<tr>
<td>Observing</td>
<td>[go to location]</td>
</tr>
<tr>
<td></td>
<td>[use skill]</td>
</tr>
<tr>
<td></td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange information]</td>
</tr>
<tr>
<td>Interviewing NPCs</td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange information]</td>
</tr>
<tr>
<td></td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange information]</td>
</tr>
<tr>
<td>Using an item in the field</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[go to location]</td>
</tr>
<tr>
<td></td>
<td>[use an item]</td>
</tr>
<tr>
<td></td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
</tbody>
</table>

### Table 7.3. Strategies for increasing comfort

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtaining luxuries</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
<tr>
<td>Killing pests</td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[damage an NPC]</td>
</tr>
</tbody>
</table>

#### 7.2. Comfort

While most of the NPCs studied did not express concern for their personal comfort, a few did. These NPCs employed two main strategies: obtaining luxury items and removing pests, which are summarized in Table 7.3.

#### 7.3. Reputation

It was very common to find NPCs who desired to increase their reputation with a faction. These NPCs would ask players for help with some task that would bring fame to the NPC, such as killing powerful enemies, or obtaining rare and valuable items.

Quests that increase fame are nontrivial tasks, whose completion demonstrated something about the NPC (for getting someone else to perform this task), or the player (for suc-
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtaining rare items</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
<tr>
<td>Killing enemies</td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[damage an NPC]</td>
</tr>
<tr>
<td>Visiting a dangerous place</td>
<td>[go to location]</td>
</tr>
</tbody>
</table>

Table 7.4. Strategies for increasing reputation

cessfully completing the task). Examples include killing powerful enemies, obtaining difficult to find or well guarded items, and demonstrating bravery by visiting some dangerous place in the world. Table 7.4 summarizes the strategies used to increase fame and recognition.

7.4. Serenity

Peace of mind is a nebulous concept, but we use it to categorize quests that result in some NPC feeling better, and where other motivations do not appear to apply. Capturing or killing criminals, and avenging a wronged ally are good examples, as there is not necessarily a shift in power. This is in contrast to the desire to defeat enemies, which is not emotionally driven but instead seeks a change in relative power. Recovery of lost or stolen items, particularly when these items have no intrinsic value are also good examples, as an NPC can miss a memento and desire its return. A related motivation is the desire to recover the bodies of fallen comrades, or family.

The mechanism for capturing an NPC varies from game to game, and will determine the exact form of the strategy used. We provide a couple of examples that correspond to different mechanisms used by Everquest and World of Warcraft. The first form compels the NPC to follow the player, the second incapacitates the NPC who may then be carried in the player’s inventory.

7.5. Protection

Protection means removing threats, restoring damage caused by past threats and guarding against future threats Table 7.6 shows common strategies used to protect a faction and its resources.
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revenge, Justice</td>
<td>go to NPC [damage an NPC]</td>
</tr>
<tr>
<td>Capture Criminal(1)</td>
<td>go to NPC [exchange an item] [go to NPC] use an item [go to NPC]</td>
</tr>
<tr>
<td>Capture Criminal(2)</td>
<td>go to NPC [exchange an item] [go to NPC] use an item [exchange an item] [go to NPC] [exchange an item]</td>
</tr>
<tr>
<td>Check on NPC</td>
<td>go to NPC [exchange information]</td>
</tr>
<tr>
<td>Recover lost/stolen item</td>
<td>go to NPC [exchange an item]</td>
</tr>
<tr>
<td>Rescue captured NPC</td>
<td>go to NPC [damage an NPC] [defend NPC]</td>
</tr>
</tbody>
</table>

Table 7.5. Strategies for increasing Serenity

The removal of threats usually requires killing NPCs, or destroying items that threaten a faction. Players can help repair damage by using an item or a skill on a damaged NPC, item or world location, for example healing an NPC with a magic spell or by applying bandages, using potions to cleanse contaminated springs, repairing a damaged wall. Players may help protect against future threats by building fortifications and acting as guards.

7.6. Conquest

We have observed quests that ask players to kill enemy NPCs; destroy enemy fortifications, equipment, and supplies; or steal equipment and supplies from an enemy. Players who undertake these actions weaken an opposing faction, which serves the interests of the NPC originating the quest. Table 7.7 summarizes these strategies.

7.7. Wealth

NPCs may attempt to increase their wealth by having players gather raw materials for the NPC to assemble into salable items. An NPC may also ask the player to steal some
item or make an item for the NPC to sell. Table 7.8 summarizes these strategies.

7.8. Ability

NPCs may attempt to improve their skills, or have allied players improve skills in order to increase the power of a faction. This may be done by creating new items that allow a skill to be practiced, such as fabricating a weapon or lockpicks; or practicing a skill when the appropriate tools are already available, such as attacking enemy NPCs or inanimate targets. The player may need to collect raw materials so that a tradeskill may be practiced. NPCs may also request that items be collected for research. Table 7.9 summarizes these strategies.

7.9. Equipment

NPCs may increase their power by acquiring additional supplies. Players may assist by assembling new items, or obtaining items elsewhere and returning them to the NPC. Table 7.10 summarizes these strategies.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack threatening entities</td>
<td>go to NPC</td>
</tr>
<tr>
<td></td>
<td>damage an NPC</td>
</tr>
<tr>
<td>Treat or repair</td>
<td>exchange an item</td>
</tr>
<tr>
<td></td>
<td>go to NPC or item</td>
</tr>
<tr>
<td></td>
<td>use an item</td>
</tr>
<tr>
<td>Treat or repair</td>
<td>go to NPC or item</td>
</tr>
<tr>
<td></td>
<td>assemble, repair or heal</td>
</tr>
<tr>
<td>Create Diversion</td>
<td>exchange an item</td>
</tr>
<tr>
<td></td>
<td>go to NPC or item</td>
</tr>
<tr>
<td></td>
<td>use an item</td>
</tr>
<tr>
<td>Create Diversion</td>
<td>go to an NPC or item</td>
</tr>
<tr>
<td></td>
<td>damage an NPC or item</td>
</tr>
<tr>
<td>Assemble fortification</td>
<td>go to an item or location</td>
</tr>
<tr>
<td></td>
<td>assemble, repair or heal</td>
</tr>
<tr>
<td>Guard Entity</td>
<td>go to an NPC, item or location</td>
</tr>
<tr>
<td></td>
<td>defend an NPC, item or location</td>
</tr>
</tbody>
</table>

Table 7.6. Strategies for increasing Protection
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack enemy</td>
<td>[go to NPC]</td>
</tr>
<tr>
<td></td>
<td>[damage an NPC]</td>
</tr>
<tr>
<td>Steal supplies and equipment</td>
<td>[go to an NPC or item]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
</tbody>
</table>

Table 7.7. Strategies for increasing Conquest

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather raw materials for tradeskills</td>
<td>[go to an NPC or item]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
<tr>
<td>Steal valuables for resale</td>
<td>[go to an NPC or item]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
<tr>
<td>Make valuables for resale</td>
<td>[assemble, repair or heal]</td>
</tr>
</tbody>
</table>

Table 7.8. Strategies for increasing Wealth

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assemble tool for new skill</td>
<td>[assemble, repair or heal]</td>
</tr>
<tr>
<td></td>
<td>[use an item]</td>
</tr>
<tr>
<td>Obtain materials for training</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[use an item]</td>
</tr>
<tr>
<td>Use existing tools</td>
<td>[use an item]</td>
</tr>
<tr>
<td>Practice combat</td>
<td>[damage an NPC or item]</td>
</tr>
<tr>
<td>Practice skill</td>
<td>[use a skill]</td>
</tr>
<tr>
<td>Research a skill(1)</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[use an item]</td>
</tr>
<tr>
<td>Research a skill(2)</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[exchange information]</td>
</tr>
</tbody>
</table>

Table 7.9. Strategies for increasing Ability

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequence of required actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assemble equipment and supplies</td>
<td>[assemble, repair or heal]</td>
</tr>
<tr>
<td>Deliver supplies</td>
<td>[exchange an item]</td>
</tr>
<tr>
<td></td>
<td>[go to NPC, item or location]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
<tr>
<td>Steal supplies</td>
<td>[go to an item]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
<tr>
<td>Trade for supplies</td>
<td>[go to an NPC]</td>
</tr>
<tr>
<td></td>
<td>[exchange an item]</td>
</tr>
</tbody>
</table>

Table 7.10. Strategies for increasing Equipment
CHAPTER 8

ACTIONS

*Good actions give strength to ourselves and inspire good actions in others.*

Plato

I have identified a small set of player actions that allow me to express quests in an abstract notation. These actions serve as an alphabet for a quest language. These actions are abstractions of the steps players would need to take to complete the quests, and always result in some change in the world state when performed. For example, the quest of delivering a letter to an NPC might require the player to obtain the letter, move to the NPC, and then give them the letter. As a result of performing these actions the player’s position changes, and the NPC gains possession of the letter, and whatever knowledge that letter might contain.

To preserve generality, I further abstract these actions by referring to the letter as an item. The abstract form of this quest is for the player to obtain an item, move to an NPC and then give them the item. If I replace the letter by another item, say a key, I have another quest that is identical in structure, but not in the details. The value of such an abstraction is that it allows quest designers to work at an abstract level and then to move to a concrete representation of their design. This technique will be useful when automatically generating quests.

I believe that this list of actions is complete, in the sense that all of the quests I have studied could be expressed as partially ordered sequences of these actions.

1. *Damage an NPC or item* – Damaging an NPC or item can indicate combat or sabotage. The degree of damage is not specified. Some quests require only minimal damage, others require total destruction of the target.

2. *Assemble, repair an item, heal an NPC* – An assembly or repair action either creates a new entity, or restores an existing entity. This is essentially the opposite of the damage action.

3. *Exchange information (talk, read, write)* – An exchange of information may occur
between the player and an NPC, or the player and an item. This action combines
talking, listening, reading and writing. This information transfer may be one way,
or bidirectional.

(4) **Exchange items with an entity** – An exchange of items is similar to the exchange of
information, although physical objects are involved. While a piece of information
may be given to multiple people, an item can only be possessed by one entity at a
time. This action combines taking, giving, and trading items.

(5) **Defend an entity** – The defense of an entity implies that the entity (an item, NPC,
or location in the world) is unable to defend itself, and is expected to come under
attack. A quest will usually combine a defense action with a set of combat events
challenging the player’s ability to successfully hold off attacks on the entity.

(6) **Go to an entity** – Players are often required to move to another location, possibly
the current location of an NPC or item, and possibly a fixed location in the world.
The player may or may not know where this location is, so the discovery of a location
is a common subquest when a quest requires movement.

(7) **Use item or skill** – The use of an item or skill can encompass a large set of actions
that do not fall into other categories. Tool use is a common activity in role playing
games, and obtaining the proper tool is a common subquest. The use of a skill is
not necessarily tied to tool use.

(8) **Wait for an event** – Players are rarely required to wait for some event to occur,
such as during the Test of Patience which occurs in Everquest’s Shaman Epic quest.
In this quest the player is asked to wait in a remote location for an event to occur,
while being tempted to leave by various NPCs.

These actions are intended to be independent of any particular game genre, and often have
specific details abstracted away. For example, the quest of delivering a letter to an NPC
could be expressed as the sequence:

- exchange an item (obtain the letter)
- go to an NPC (make your way to the recipient)
- exchange an item (deliver the letter)

The value of expressing quests using these abstracted actions is that we are able to see a simple structure once all of the details have been removed. This is an example of a common delivery quest.

Since the list of actions presented above is sufficient to represent all the quests in our database, I claim that this set is complete. My belief in the completeness of this set is based on the size of my database, and the likelihood of encountering a new quest which is not representable by these actions. In such a case, the list may be extended, preserving my claim. This quest structure is capable of describing any role playing game quest.
CHAPTER 9

ENCOUNTER METRICS

*Measure what is measurable, and make measurable what is not so.*
Galileo Galilei

When creating combat encounters as part of a quest, it is important for these encounters to have an appropriate difficulty. The encounter should neither be too easy for the players, nor too difficult. Everquest provides certain characters with crowd control abilities, which can reduce encounter difficulties. And it is often possible for players to divide groups of enemies so that they can be fought separately. However this does not eliminate the need for a quest designer to keep encounters balanced.

A measurement of the difficulty of an encounter should be based on an estimate of the resources players would need to expend in order to complete this stage of a quest. Resources can include health, magic, time, and monetary investment. It is clear that these are dissimilar concepts, and raises questions like “how much health is one unit of time worth?”

For purposes of encounter generation, I introduced the concept of a point as the basis for measuring difficulty. I eventually settled on the arbitrary choice of 1 point equaling 15 minutes of gameplay. Other point assignments are made relative to this value.

It should be noted that 15 minutes of high level gameplay is quite different than 15 minutes of gameplay for a new level 1 player. The concept of a point should in some way involve the level of the player.

A number of statistics were gathered from Sony’s Everquest servers. It was observed that on average a player spends 3.25 hours per level for the first 80 levels. This begins to indicate the value of time.

Mob point values are assigned based on level: $Points = 1.1^{Level}$

At some point I needed to know the difficulty of an encounter which would be appropriate for a player. To get a rough idea of this value I created an encounter in-game, and collected a few statistics. One of my experiments was designed to study how encounter
Figure 9.1. Point scaling with level

difficulty scales with the number of opponents, and so I re-ran a basic scenario changing only
the number of opponents faced. For this study I noted player’s resource usage (health and
magic) during the encounter, and graphed the results. Even when running only a few sce-
narios, it was clear the graph of resources versus number of opponents was always a sigmoid
curve.

I created a simulator which used EQEMu’s combat resolution algorithms and basic
NPC AI code to automate both sides of an encounter. I forced all participants to stand
their ground until one side of the fight was eliminated. This produced very smooth sigmoid
curves.

Figure 9.2 shows 8 such experiments, where the player level was increased by one
for each experiment. A simulated player of levels 30-37 was ran against waves of 0-100
opponents. All of the opponents were level 10 creatures. The result is a series of similar
sigmoid curves with different roots. In this particular graph, the player levels increase from
left to right – level 30 is on the far left, and level 37 is on the far right. The root (or X-
Figure 9.2. Waves of level 10 opponents vs level 30-37 characters intercept) of a curve indicates the break-even point of an encounter. This is the point when both sides are evenly matched. This value is a good indicator of the overall power/ability of a player or opponent creature. By changing attributes or equipment of one of the participants, a scenario can be reran and a shift in the root will occur. This shift is due to the power difference caused by the change to the scenario.

In this manner, it is possible to compare values that would not otherwise be comparable. The relative value of a point of health vs a change in weapon speed, for example.

The graph seen in Figure 9.2 shows the effect of a linear increase in player level on combat effectiveness.
CHAPTER 10

THE KNOWLEDGE BASE

Some people drink from the fountain of knowledge, others just gargle.
Robert Anthony

The knowledge base is a collection of facts about the game world, and rules used to infer other facts. I consider these rules to be separate from the rules in the rulebase, as these rules are used only to express the knowledge base in a more compact and general form. The generator and the knowledge base were implemented in Prolog, and allow the use of Prolog’s inference engine to answer queries about the world, by making use of this knowledge base. Given an assertion as a query, Prolog will either prove the assertion to be true via example, or will prove the query to be false.

Facts are entered into the knowledge base in the form of functor/argument tuples. For example, Figure 10.1 shows facts regarding alligator eggs found within the world. The first fact associates the string “alligator egg” with the database key 10160. This is the key in the items table, discussed elsewhere. The functor is item_id/2, and the two arguments are “alligator egg” and 10160. Prolog does not enforce any interpretation of facts, therefore it is up to the program, in the form of rules, to determine how these facts are interpreted. One could interpret the first fact to mean that there are 10160 alligator eggs in the world, or any other possible meaning. It is only due to convention within the knowledge base that I choose to interpret the second argument to be a database key used with the items table.

The second fact indicates that alligator eggs are edible. This allows rules requiring edible items to consider using alligator eggs.

The third fact associates the indefinite article ’an’ with singular alligator eggs. This rule is used to help create correct natural language references to the items, such as in character dialog. By convention, all items in the knowledge base use the indefinite article ’a’ unless overridden by article/2.
item_id('alligator egg', 10160).
edible(10160).
article(10160, an).
source(10160, drop−crescent).
source(10160, drop−moors).
source_mobs(10160, [395204,395198,395197,395195,395192,395145,
395136,395119,395114,395102,395101,395093,395056,
395055,395050,395033,394063]).

Figure 10.1. Prolog rules describing an alligator egg

The last three facts indicate possible sources of alligator eggs. We learn that the eggs
can drop (as in be left behind as treasure) from mobs in two zones: Crescent Reach, and
Blightfire Moors. The fact source_mobs/2 indicates a list of mobs which might drop an
alligator egg when killed. The second parameter in this fact is a list of database keys, in this
case keys into the npc_types table.

All objects usable in quests must have similar facts defined. The knowledge base
ended up requiring about 25000 lines of Prolog.

In addition to items, facts need to be defined for all static NPCs who can be part of
a quest. This is a limited set, which requires the generator to be able to dynamically create
new NPCs. Dynamically created NPCs will have their knowledge base entries created along
with the NPC. Figure 10.2 shows the facts provided for one of the merchants in the town of
Crescent Reach.

NPCs use npc_id similarly to the way items use item_id. In this case, the second
argument is the database key into the npc_types table. This particular NPC is an example
of a unique NPC, as there will only be one in the world at any time.

The second rule defines the location of this NPC. This rule shows a newer form of
indicating an asset type. In the previous example, the item IDs were listed as plain integers.
The location_of functor requires that IDs be supplied with the asset type, a minus sign
(the -/2 operator), and the integer ID. This convention is relatively new to the codebase, and
I have not gone back and modified older facts and rules to adopt it. The second argument to
location_of indicates the general area in which this NPC can be found, in this case area a14
in Crescent Reach. I divided both zones that I use for testing (Crescent Reach and Blightfire Moors) into separate areas based on natural boundaries and the difficulty of content found in these regions. Area a14 is the first of 3 floors which contain the town proper. Location information such as this allow the generator to create either global location descriptions (e.g. “on the 1st floor in Crescent Reach”) or local descriptions (“on the 1st floor”).

Initially, I only needed to have `location_of` for an NPC, but I later ran into cases where I needed to know the exact location of the NPC. For example, if I needed to create another asset nearby, or display a particle effect at some point during the quest. The `coord_of/2` an entity provides its X,Y,Z coordinates. These can be floating point numbers if greater accuracy is desired, but are currently express as integer values. In addition to NPCs, `location_of/2` and `coord_of/2` can be used with item-ID to obtain the initial location of items created by the generator.

Most of the remaining facts should be easy to understand. Faction is one of the ten sets of NPCs which were arbitrarily created for testing. This divides the NPCs into groups who are allied or opposed to each other. Alignment is a numerical measure of how good an NPC is. This is currently used only to restrict quests involving theft to NPCs with low alignment scores.
Chapter 7 discusses NPC motivations. It is expected that this value would be dynamically managed by the server, and adjusted as events occur. For development purposes, motivations were fixed for each NPC, and tests were ran by changing the initial NPC used by each quest.

Figure 10.2 includes two Prolog rules using the `merchant/2` functor, that are easily distinguished from each other. The first form associates an NPC ID with a database key for a merchant table (of goods available for sale). While the generator does not currently query the SQL database for information in this table, there are utility rules outside of the generator which do make such queries. Typically this information is used to update facts for items, for example creating source facts indicating that a vendor sells the item. The second form of the merchant functor is used to indicate the type of items sold by this merchant. In this case, Merchant Odimari sells general goods. If the generator later creates a dynamic item in this class, this merchant would be considered a candidate to sell that new item. Note that class uses the `';'` operator (`:2`) to separate a data type from its value, in this example “class” and “general”.
CHAPTER 11

THE RULE BASE

*If you obey all the rules, you miss all the fun.*  
Katharine Hepburn

The rule-base is a collection of Prolog rules used to generate quests, and which make use of external input and the knowledge base. I consider these rules separate from the rules used to describe knowledge (ie. the knowledge base).

I am not going to list all of the rules in the rule-base, but will highlight some of the more important concepts.

Static entries in the knowledge base have properties, such as `id_of/2`, which provide a database key into a table appropriate for the asset type. There are a set of utility rules which replicate this behavior for dynamic assets. If a search of the static entries fails then these utility rules will search the dynamically created entities for matching properties, as shown in Figure 11.1. This mechanism allows the set of entities to grow during quest generation, while still using the rules written for static entries. The dynamic entries are not preserved across quest generation, as these dynamic entries are created especially for a single quest and are not intended to be shared with other quests. This means that removing a quest from the database is possible because there is no chance of another quest using dynamic assets from the quest which is being removed.

There are also several utility rules which simplify common queries, such as looking up the zone-id of an NPC, as shown in Figure 11.2. These rules were created to remove duplicate code, and allow updates to apply to the entire generator with minimal work.

```prolog
edible(Item_ID) :-
    get_dynamic_attribute(item~Item_ID, template:Template),
    edible(Template).
```

**Figure 11.1.** Checking dynamic entities for edible properties

97
zone_id_of(npc-NPC, zone-Zone_ID) :-
    location_of(npc-NPC, Zone_Name-_-Area),
    zone_id(Zone_Name, Zone_ID).

Figure 11.2. Obtaining the zone-id of an NPC

random_member(List, Value) :-
    random_permutation(List, Shuffled),
    member(Value, Shuffled).

random_permutation(List, Random) :-
    key_random(List, Keyed),
    keysort(Keyed, Sorted),
    pairs_values(Sorted, Random).

key_random([], []).
key_random([H|T0], [K-H|T]) :-
    Key_Limit is key_limit,
    K is random(Key_Limit),
    key_random(T0, T).

Figure 11.3. Random selection of items from a list

Random selection of elements from a list is a common operation. It is important that this process is not allowed to continue unbounded, as it can cause the generator to lock up when faced with an impossible query. My solution is to randomly sort the list and then return each value until the list is exhausted. This guarantees that each member is considered one time, and that the process terminates when the list is exhausted. Each element is processed exactly one time, but in a random order.

As shown in Figure 11.3, a random permutation is first calculated, and then a member of this shuffled list is returned. Upon backtracking, member/2 will select the next member of the list. Random permutations are calculated by assigning random keys to each element in the input list. The rule key_random recursively prefixes each list member with a random integer. Duplicates are permitted. The variable Key_Limit is unified with a constant parameter key_limit/0, and is an upper limit on the size of the keys. A list such as [a,b,c] would be transformed into something like [30-a, 1204-b, 243-c]. The list is then sorted using keysort/2, and the keys are removed from the sorted list. The result is a randomly shuffled
list.

Additional rules support the random selection of a list member based on a supplied distribution. Each element in a list is prefixed with a probability, the list is then sorted, and a random check is made for each member of the list. A probabilistic bubble sort is used to randomly sort the list. This acts like a regular bubble sort, with the caveat that each member can be mis-sorted based on the supplied probability. It is unlikely that other applications will be found for this sorting technique, but it is very helpful in this context where we are not looking for an optimal solution, but rather an interesting one. Some minor mistakes increase the size of the solution space.

Random numbers are an important tool for the quest generator. SWI-Prolog supplies a pseudo-random number generator capable of generating values with a uniform distribution. However, sometimes a normal distribution is preferred, and this introduces a significant problem with the tails of the distribution. Tails of a normal distribution should be infinite in length, but most techniques for generating normal distributions have finite tails. Thomas et al. provide an excellent survey of techniques for generating Gaussian random numbers [83]. Box-Muller can create normally distributed values, but cannot generate values more than 6.66 standard deviations from the mean when using 32-bit arithmetic.

Another possible technique is to calculate the average of a sequence of N uniform random numbers, taking advantage of the Central Limit Theorem. This technique limits the tails of the curve by forcing all values to be within N from the mean. Once again, we have finite length tails.

I feel that the tails in the distribution are very desirable (representing exceptional outcomes), and given the difficulties with preserving the tails in a Gaussian distribution I instead chose to use a Cauchy distribution. Calculating a value in a Cauchy distribution requires calculating the ratio of two random numbers with the same distribution. I decided to use an existing Box-Muller generator to provide these seed values, as seen in Figure 11.4.

There are some utility rules which select assets based on special criteria. An example is seen in Figure 11.5 which assembles a set of NPCs in a given role and then selects a
cauchy(X) :-
  Location is 5,
  Scale is 5,

  % reject any negative values
  repeat,
  rand_gaussian(X1, Y1, 0, 1),
  Y1 <= 0,
  Ratio is X1 / Y1,
  X is Location + Scale * Ratio,
  X >= 0.

Figure 11.4. Generating positive random numbers in a Cauchy distribution

random_npc_with_role(Role, NPC_ID) :-
  findall(NPC, role_of(NPC, Role), NPC_List),
  list_to_set(NPC_List, NPC_Set),
  random_member(NPC_Set, NPC_ID),
  \+ npc_used(NPC_ID).

Figure 11.5. Selecting random NPCs with a given role.

random member of the set. This particular rule also illustrates how asset selection rules
will only allow assets to be used once in a quest. A check is made of the Prolog knowledge
base, and if the entity in question has been previously used the query will fail. Backtracking
then causes the generator to attempt another solution. Eventually either an unused entity
is found, or the set of possible solutions is exhausted. Upon exhaustion, the generator is
capable of creating new entities. Other examples of asset selection rules would be selecting
materials based on NPC role, or items appropriate for an NPC given their motivation.

There are a number of rules used to select appropriate pronouns and adjectives for
entities. Figure 11.6 shows an example third-person object pronoun calculation. These
rules are used to help create boilerplate dialog. Dialog was not a goal of the generator, but
something functional was needed in order to evaluate the resulting quests.

Point allocation to subquests is an important calculation. During node generation a
node profile will indicate the possible types of subquests which may be added, in addition
to any leaf expansion subquests. These are usually additional prerequisites or postrequisites
third_person_object(NPC, 'him') :-
    gender_of(NPC, male).
third_person_object(NPC, 'her') :-
    gender_of(NPC, female).
third_person_object(NPC, 'it') :-
    [+ gender_of(NPC, male),
     + gender_of(NPC, female)]

Figure 11.6. Determining the proper object pronoun.

assign_points(Points, -, []) :-
    Points < min_node_points,
    !.
assign_points(Points, Case_Set, Result_Set) :-
    num_subquests(Case_Set, 0, N),
    split_n_quant(Points, N, Point_Vector),
    !,
    assign_helper(Case_Set, Point_Vector, [], Result_Set).
assign_points(_, Case_Set, Result_Set) :-
    num_subquests(Case_Set, 0, 0),
    length(Case_Set, N),
    fill(N, 0, Point_Vector),
    assign_helper(Case_Set, Point_Vector, [], Result_Set),
    !.
assign_points(Points, Case_Set, [null -0-0]) :-
    format(atom(M1), 'unable to split \w among profile \w',
           [Points, Case_Set]),
    write_trace(M1),
    !.

Figure 11.7. Division of points among subquests.

which can be used to consume extra points. The assign_points/3 rules will attempt to
divide available points among subquests, as seen in Figure 11.7. This allocation fails if
there are not enough points to guarantee that each new subquest node can have at least the
minimum required point value. An exception is made for 0-point subquests. If none of the
allocation attempts succeed, a null subquest is selected so that node generation does not
fail. This null subquest is ignored by the generator. The assign_helper rule is a recursive
distribution of the Point vector among the possible subquests.
Players may need to create new items as part of a quest. To support this, a set of rules concerning item assemblies was created. Figure 11.8 shows an example of an assembly definition for tongs usable by blacksmiths. This record indicates who can use the assembly, who can make the assembly, and an abstract description of the parts one might use. The generator will randomly select parts which match the item types, and create a new tradeskill recipe which converts the parts into the desired item. Tradeskills such as smithing or baking are another mechanism in which new items can be introduced into the game world. A variety of vectors is useful to keep the solution space for the quest generator large.

The rule \texttt{assembly_for} is another example of asset selection for a specific NPC. Since an NPC can fill one or more roles, all roles are collected into a list, and a list of all assemblies appropriate for this role set is created. The list is converted into a set to remove duplicates, and a random assembly is returned.

11.2. Encounters

It is common for quests to require the player to face opposition from enemy NPCs at some point. I have created rules for randomly generating appropriate encounters, based on player level and available points.

At the heart of this process are rules that generate combinations of point values for possible encounters. These are filtered to remove solutions which are outside of the desired

\begin{verbatim}
assembly( tongs, [type=tool, roles=[smith], crafter=[smith],
    parts=[1−2∗t_metal_stock, t_bind, t_cond], result=120213)].

assembly_for(NPC, _Points, Assembly) :-
    findall(R, role_of(NPC, R), Role_List),
    list_to_set(Role_List, Possible_Roles),
    enumerate_assemblies_for_roles(Possible_Roles, [],
        Assembly_List),
    list_to_set(Assembly_List, Assembly_Set),
    random_member(Assembly_Set, Assembly).
\end{verbatim}

Figure 11.8. An example assembly definition, and code to select an assembly
encounter_templates(Points, Results) :-
ground(Points),
Limit is enc_spawn_limit,
all_encounter_vectors(Limit, [], All_Templates),
Low_Limit is Points - enc_point_diff,
High_Limit is Points + enc_point_diff,

% extract the set of encounters within the point limit, and
% tag each template with its point value. These are sorted by
% the point values.

findall(Template, template_aux(All_Templates, Low_Limit,
                                High_Limit, Template), Raw_Result),
sum_and_tag(Raw_Result, Results),

% fail if the empty list is returned
length(Results, Length),
Length > 0,
!.

Figure 11.9. One of the rules used to assemble lists of templates for potential encounters. point range. Figure 11.9 shows one of the rules used to create these potential encounter templates.

The generator selects one of the templates and then instantiates it by selecting area and level-appropriate creatures to replace variables in the template. These creatures must have some association with each other. They can be variations on the same creature type, or different creatures that might be seen together. One common situation using different creatures would be to have intelligent creatures such as an Orc, with guards and servants rounding out the encounter.

There are many rules used to create new NPCs (as opposed to creatures). One of these is shown in Figure 11.10 where the caller provides a name and level for the new NPC. There are variations which create random names, or use existing NPCs as models.
make_static_humanoid3(name:Name, level:Level, npc:NPC) :-
gen_mob_variable(NPC_Var),
NPC_AC is mob_ac_value(Level),
NPC_HP is mob_hp_value(Level),

NPC = [npc, id:NPC_Var, level:Level, ac:NPC_AC, hp:NPC_HP,
template:generic_npc, name:Name],
add_dynamic_entity(npc−NPC_Var, [name:Name,
template:generic_npc]),
assign_random_npc_properties(npc−NPC_Var).

Figure 11.10. One of the rules used to create static humanoids.
CHAPTER 12

QUEST GENERATION

Every act of creation is first an act of destruction.  

Pablo Picasso

12.1. Overview

My quest generator is a Prolog program which generates a quest as a single XML document. This document can be imported into an Everquest server using an importer program which parses the XML and produces both database updates and a PERL script associated with the quest. The server then uses this information to produce the runtime behavior associated with the quest. The separation of generation and import was intentional, to demonstrate that quests could be generated at one site and then transported to and imported into a foreign server. A quest can easily be added or removed from a particular server, allowing server operators the option of sharing quests with other operators. The use of a AAA game as a target platform was a means of demonstrating that the output of the generator looked good in-game as well as on-paper.

The Everquest server uses an SQL database that contains information on world entities (items, NPCs, creatures), and processes like tradeskill recipes. The database is also used to hold information used by the quest runtime to process quests created by the generator, such as quest state. The PERL script is used to hold runtime trigger handlers, which are run when certain conditions are met in-game. These handlers have complex logic which could not easily be encoded in database entries.

12.2. Process

The generator starts with a source NPC, and attempts to create an N-point quest which satisfies the current motivation of this NPC. The technique for selection of this NPC is not important for describing the generator. One might attempt to replace completed quests in one part of the world by generating new quests for NPCs in that area, or one might start with a desired type of quest and then assign an appropriate NPC.
Figure 12.1. An overview of quest generation and playing

Once the initial NPC’s motivation is determined, a top-level strategy is selected. Strategies are essentially entries in a plan library, however a strategy does not represent an entire plan, but typically is a subset of a larger plan. Each motivation will have one or more strategies which can be used to address this motivation. Certain strategies have constraints which may not be met by current conditions, in which case the Prolog runtime will automatically backtrack and attempt a different strategy. The selected strategy implies a base set of actions and their corresponding triggers, and is a trivial solution to a task which addresses the motivation. These strategies can theoretically be expanded to arbitrary complexity, although no studies have been made of the scaling behavior. This expansion is achieved by replacing actions with complex subquests that achieve the same result as the action. Replacement is subject to the points remaining. In addition to action substitution, a
graph node corresponding to a strategy may have additional subquests which are required to be performed before and/or after the main node’s activities. The creation of these subquests also depends on available points. The generator will start a node with a certain number of available points, and will early on subdivide these points to split among the different possible subquests. This random point assignment results in a random weighting of the branches of the graph, and provides variation within solutions to the same NPC-motivation problem. This is more noticeable with higher point quests.

Asset selections will be made early on during the generation of a node for a strategy. For example, if a strategy requires an item and a location, appropriate assets will be selected based on game state. As mentioned earlier, it is helpful if the generator is able to create new assets on-demand. Exhausting the pool of available assets can result in situation where the generator backtracks forever, attempting to solve an unsolvable problem. Unfortunately it is not always possible to know in advance when assets will not be available, since choices which appear acceptable early on can be rejected later causing asset generation to be repeated. Being able to create new assets on-demand solves this particular problem.

12.2.1. Dialog

During node generation, the quest generator will create stock dialog for characters as needed. Currently the generator only produces boilerplate fill-in-the-blank dialog, to give an idea of how a quest will appear. I believe that it is a simple matter for a human to change this dialog after the generator completes. Even creating this boilerplate dialog is difficult for the generator, since the context in which a node will appear cannot always be predicted. I have had good luck pregenerating parts of dialog which are then passed to future node generators, ensuring that dialog flows smoothly between quest nodes.

A quest generally begins with some introductory dialog describing the quest, and directing the player on how to proceed. An example might be “I’ve been wanting some ale, but cannot leave my post. See if you can bring me some.” The top-level binding rules for a strategy accept optional init_text parameters with pregenerated introductory text. If this is not provided, the rule will supply the default introduction for that strategy. This
technique allows strategies to be reused for different purposes.

There are situations where a dialog fragment needs to be inserted prior to the introductory text. An optional `init_prefix` parameter can be used to supply such a dialog fragment. This technique can be used where there is a special purpose behind the quest, for example raising faction. In this case the generator might insert a prefix like “You’ll need to prove yourself worthy of my attention before I will help you.”, which will be said immediately before the introductory text. This type of dialog must be passed in as parameters, since the top-level rule does not know why the strategy is being used, only the caller does.

There is another case where dialog must be pregenerated, and that is when an NPC response is expected. Again, the top-level rule for a strategy does not know the context behind the strategy’s invocation, therefore it cannot always supply an appropriate response. The top-level rule will attempt to generate appropriate dialog, but in many cases better results can be obtained by generating the dialog fragment in the caller. In cases where a response is expected, a caller may pass an optional response parameter with the desired text. This will be spoken at the appropriate time. This technique is used when delivering items to NPCs, as it permits the recipient to properly thank the previous NPC in the quest.

12.2.2. Scripts

The generator also creates PERL scripts which will be called at runtime when a trigger fires. There are several advantages to using PERL for this task: First, the previous quest system for EQEMU uses PERL scripts, second there is a good separation between what needs to occur at runtime, and how this is to occur. Lastly, this technique makes it easy to verify proper quest generation, or experiment with new ideas.

12.2.3. Assets and Triggers

Finally, the generator will create records for assets and triggers which will later result in database updates when the quest is imported into the server. Asset records are created for any dynamically created asset which must be introduced into the game. The generator describes the asset, usually by providing a template asset and then overriding individual
properties as needed. Dynamically generated assets can be either items or NPCs. The quest importer will convert these records into table updates for the items table or the npc_types table.

Triggers are server-side objects which wait for a particular event to occur. Each strategy consists of a partially ordered set of actions which players are expected to complete. Each of these actions will have a corresponding trigger created, which will wait for that action to occur before advancing the quest. A trigger record is a tagged list which will describe the type of trigger, the parameters needed to trigger execution, where the trigger occurs in the sequence of actions, and an optional PERL event handler which will be ran when the trigger fires.

12.2.4. Expansion

The generator recursively expands branches in the node graph until it runs out of points. When the generator runs out of points it will create an XML file describing the quest, and a DOT file which can be used to produce a viewable graph of the quest. The XML file contains the entire quest, and may be installed on any compatible server. I have created a Java-based importer which reads the XML file and performs local database changes. The Java quest importer will also produce a single PERL script containing subroutines corresponding to each trigger in the quest. When the database changes are made and the PERL script is installed in a folder on the server, the quest is ready to be made live. A complete restart of the server is required, mainly because shared memory regions used by the zone servers need to be rebuilt. While it is technically possible to avoid this restart, there is no clear justification for spending the time synchronizing all of the server processes so that the new shared memory contents are handled properly.

12.3. Choice Bindings vs Node Generation

There are a lot of cases within the quest generator when some generation task needs to be done which is similar but not identical to an existing solution. One way to take advantage of this is to split node generation into two phases: One which makes choices that will cause
the node to have a unique appearance, and one which uses these choices to actually create the node. This allows parts of the generator to make their own choices and skip the first part of node generation, resulting in customized node generation for the caller. Every strategy has the ability, though this first phase, to select appropriate choices with little of no extra knowledge. This is one extreme of the available choices. Skipping the random selection and using whatever choices are passed in is the other extreme. A third choice supported by the generator is the ability to preselect certain choices the node might wish to make, and if a value is not preselected then use the random generation process to pick a value.

12.4. Knowledge

Traditionally, knowledge is categorized into several broad categories: [77]:

- Personal (experience)
- Procedural (methods)
- Propositional (facts)

I needed to determine several facts: Where does knowledge come from? What types of information are of interest? Why does a character desire/need a particular piece of information? A knowledge model was constructed to aid with these queries.

Knowledge is currently divided into several categories based on function.

- Lore/History: Historical information, lore.
- Properties: Item properties
- Missing Items: Regarding missing items
- NPC Status: State of NPCs. Well being, location.
- Threat: Information on threats to a faction
- Sources: Information on sources of items or information
- Process: Information on processes used to make items
- Entity Location: The current location of entities

I note that NPC Status and Entity Location have some overlap. Future work on the generator should clarify these distinctions.
Each type of knowledge has a separate distribution of possible sources.

- Maps
- Scrolls
- Books
- Art
- Notes
- Samples

The generator needs to keep track of how current knowledge is. Some information, like entity location, must be current to be useful. The age of knowledge can determine the possible item sources, and locations. Current knowledge is more likely to be found in notes, or samples. Ancient knowledge can be found in documents or artwork.

- Ancient: Before an NPC’s time.
- Contemporary: Within the lifetime of an NPC.
- Current: Very recent.

An NPC must have a reason for wanting a particular piece of information. This is used for dialog generation, and does not otherwise affect the quest generation process. Possible reasons:

- Obtaining status reports.
- New opportunities, such as learning of the existence of some entity.
- External influences on the NPC, such as third party demands
- A technique or solution is needed for a new type of problem.
- Curiosity

The pair \( \langle \text{Category, Reason} \rangle \) implies a unique set of motivations. These are used to create text used for quest descriptions, and do not affect the quest otherwise.

There are various knowledge acquisition techniques available to an agent:

- Conversations with friends or enemies
- Obtaining items from friends, enemies, or the world
• Observation of enemies or world locations

The pair \( \langle \text{Category, Technique} \rangle \) implies a unique set of instructions for the player. The distinction between friends and enemies restrict strategy selection. For example, one does not capture and interrogate an ally.

Once a knowledge category and acquisition method are selected, the remaining choices can easily be made. The result is the text used to introduce a quest to the player. This pairing is also used to pre-select assets used by the quest, generate text used in later parts of the quest, and otherwise constrain future choices by the generator.

Specific types of knowledge are determined based on the role of an NPC. A baker might be interested in processes and materials used by other bakers. A soldier might be interested in enemy strength and locations.

12.5. Items and Equipment

The second major asset type used by the generator is the item. It is common for subquests to create and make use of items. There is a generic rule used to obtain a set of items (subquests are created for each unique item type). There are rules used to create subquests based on item acquisition techniques.

• NPC drops: Items that are obtained by killing NPCs.
• Subquest: An NPC will trade an item for the completion of some personal subquest.
• Tradeskill: The player is required to assemble the item using one of the tradeskills (e.g. blacksmithing, tailoring).
• Vendors: An item can be obtained from an NPC vendor.
• Ground: An item can be found as a ground-spawn. Possibly on an NPC corpse, or in some sort of container.
• Forage: An item can be found by using the forage skill.

There are also rules which govern delivery of a set of items to an NPC.

Item selections are based on the roles of NPCs in a quest. NPCs share an interest in common items, but also have specialized interest in the tools of their trade.
12.6. Item Assemblies

Players may be asked to assemble items during a quest, or to collect items for an NPC to assemble an item. To support this requirement, a system of random item recipes (or assemblies) was created. The generator is able to determine a type of item appropriate for an NPC role, and then to construct a random recipe to make this item. A random variation on the item’s name is also generated. The recipe generation is possible due to template items being defined in terms a range of part types one might use to create an item.

It is important that each NPC role have at least one appropriate assembly type, otherwise the quest generator could attempt infinite backtracking trying to create an assembly where none is possible.

Item assemblies or collections are often associated with a custom container which can be used to combine parts into a new item. The generator is capable of selecting an appropriate container type and size, and then creating a random variation of this container.

12.7. Map Locations

The generator must keep track of different locations on the map, for various reasons. For example, if a generator wishes to place a building on the map then it will need to know the possible coordinates appropriate for this building.

The zone maps used were hand-annotated based on coordinate gathering. I selected likely places on the map, and noted their coordinates. Each individual coordinate is a possible spawn point (for an item or creature), and a collection of nearby points is referred to as a point cluster. When the generator needs to select a location for some piece of content, available clusters are considered. Other factors such as the relative difficulty of the world near the cluster are used to evaluate each candidate cluster. Once a set of acceptable clusters is identified, the generator makes a random choice and marks the location within the cluster as used. This keeps a quest from accidentally reusing a location at different parts of a quest. The generator is otherwise free to reuse clusters.
12.8. Random Names

Random names are helpful in providing quests with a unique feel. I used a random syllable-based approach developed by Samuel Stoddard [79] to which I added custom syllable sets. These were designed to create random words with a distinct sound. The generator would then request a name be generated according to a rule or syllable set.

12.9. Branching and Point Division

The generator tries to create random graph topologies each time it is ran. As discussed earlier, the concept of a “point” is used to limit generator behavior. The point is therefore used as a technique for randomizing graph structure. This is done by randomly dividing available points at each node of the graph, and allocating points to children. If a child is unable to be created, backtracking is used to redo the point division.

12.10. Asset Identification

The generator cannot assign globally unique identifiers to new assets, since it is unaware of what other content has been created on a target server. Indeed, a generator may be creating a single quest capable of being ran on more than one server. The solution to this problem is for the generator to create quest-local identifiers (such as item_0, npc_5). When a quest is imported into the server, the import routine queries the server’s database and selects identifiers appropriate for this server. These identifiers will be used as database keys for future access.

One could use globally unique identifiers (such as GUIDs), but these would make poor database keys. Quests do use GUIDs as a global quest identifier, but this is to prevent the same quest from being loaded twice on the same server. The quest importer and the database serve to serialize quest insertion and guarantee consistency. As a result, each GUID is mapped to a locally unique quest-id.

12.11. Debugging

It is difficult to debug a large Prolog program, since so many things can go wrong. I attempt to test each low-level rule exhaustively, but combinations of constraints can still
cause unexpected problems. A few techniques were developed to assist with this debugging process.

First, the overall structure of a quest can be seen from a graph created by the generator. The generator creates a graph using the DOT graph description language. This graph file can then be processed using graphviz (or another DOT renderer) into a viewable image file. These graphs show the big-picture organization of a quest, with notes on asset selection and reasons for some of the choices made by the generator.

Another valuable technique is tracing. SWI Prolog’s six-port debugger is capable of creating very large trace files, which describe every action taken by the prolog interpreter. This is too much information to effectively process. Early on I attempted to use editor syntax highlighting to help locate information in these traces. A more viable technique was later developed where the generator logged the entry conditions for each strategy generator, along with information on backtracking for these high-level routines. There are some situations when backtracking is unavoidable (like running out of points), but in many situations the existence of backtracking during generation is the sign of a problem with the rule base. Once this tracing facility was introduced, quest generator stability increased significantly.
CHAPTER 13

AN EVALUATION PLATFORM

"Usually, if you have a new idea, you very rarely break through to anything like recognizable development or implementation of that idea the first time around – it takes two or three goes for the research community to return to the topic.

Martin Fleischmann

I selected the game Everquest to serve both as a source of ontology, and as an evaluation platform. Everquest is a AAA MMORPG that was released on March 16, 1999, and has been very influential in later games. Everquest won the 1999 GameSpot Game of the Year [22], and a 2008 Emmy award [48]. I believe successful integration with Everquest demonstrates that my quest generator is capable of working with actual games, rather than just being a good idea on paper.

Everquest was one of the first MMORPGs, and is still being developed and earning revenue for its publisher, Sone Online Entertainment. This game is representative of the MMORPG genre, and features a large body of content. Its mechanics are based on Dungeons & Dragons [25], and it features a large virtual world. Character advancement is similar to tabletop RPGs, as characters gain experience through combat and completing quests. Players have the ability to talk to NPCs using arbitrary text (as opposed to menu-based conversations), with the NPCs responding based on key words.

A high quality server emulator (EQEmu) is available, which allows me to make modifications to the server while still using Sony’s client software. I was able to integrate my generated quests into Everquest, allowing players to experience them alongside Sony’s quests.

The EQEMu software was installed on a Linux host, along with a MySQL database server. Client machines were directed to use this Linux host as their login server, bypassing the Sony servers. Four copies of the Everquest Underfoot client were obtained from Valve Corporation’s Steam service in 2010 and used to connect to the emulated server.
13.1. EQEmu Quest System

The standard quest system provided with EQEMu consists of PERL scripts associated with characters and items in the game. A set of events are passed to the scripts associated with entities in the world, and any matching event handlers are ran. For example, if a player says “Hello” to an NPC, an event is generated. If there is a script associated with this particular NPC, the event is passed to the script. If there is a handler which matches the event, the handler will be executed. The handler scripts can then make changes to the game state required by the quest.

A single quest may involve many different NPCs and items. As a result, the scripts for this quest may be spread over a large number of files, one for each NPC or item used in the quest. While it is possible for a human quest author to maintain these files, the task becomes extremely difficult for a computer program. One of my design goals was to simplify the quest interface, so that my quest generator would create one file per quest rather than one file per game asset. This allows a quest to be added or removed from the server without affecting other quests.

Both the original Sony quest system, and the EQEMu quest system allowed stateless quests, where a knowledgeable player could bypass the early part of a quest by causing events that were expected to occur later. For example, if a conversation were required between a player and an NPC late in the quest, a knowledgeable player could jump into the middle of a quest by initiating the conversation without having performed any of the previous quest steps. Another of my design goals was to create stateful quests, where player progress was actively tracked and events which occurred out of sequence are ignored. Only when the player was at the appropriate stage in the quest would the NPC respond to the required conversation.

13.2. The Quest Manager

This object is one of the primary interfaces used by PERL scripts. It generally forwards requests on to other objects, but also serves as an intermediate between scripts and services available on the server and maintains client state.
13.3. Client State

Both nodes and triggers are shared objects which do not change. Any state associated with a quest is maintained by Client State objects which are persisted in the database. Examples of managed state would be the active nodes for a client, and any internal state for a trigger. For example, if a trigger requires that a player collect 5 items, the client state would keep a count of how many items have been collected so far. This technique allows triggers to be shared among players, while permitting player-specific state to be separately maintained.

Client State objects keep track of active triggers for each client, and implement a rollback mechanism similar to that used by databases. The rollback mechanism allows triggers to optionally checkpoint and later rollback quest state in the event that a quest stage fails. For example, if a quest is advanced by turning in an item, and the next stage involves a battle, then failing at the battle can rewind the state to the point after the item turnin was performed. The alternative is to require that players rerun the quest stages leading up to the failed stage.

Each client state object also keeps track of the state of a client’s quest journal, and takes care of updating the journal as a quest progresses. Some transient data (not persisted in the database) is maintained as part of client state. This data (such as NPCs walking waypoints) can be discarded without penalty when a client leaves a zone.

13.4. Selective Client Modification, and Transient Data

I augmented the game server to enable selective modification of what a client is allowed to see or interact with. It is possible to restrict quest-related content to players running the appropriate stage of the quest, and anyone assisting those players. This helps keep the world from appearing cluttered with quest-specific entities for other players, as well as preventing other players from interfering with a client’s quests. All of the selective modification data is transient, therefore when a quest restarts in a zone it is required to repeat its modification requests.
I created a number of helper objects to assist with selective modification, and general maintenance of transient data. One example is an Entity-Set, which is a container capable of holding an arbitrary number of entity references, such as NPCs or items. Generated scripts may use these sets to provide directions to a group of entities, such as “attack the player”, as well as provide a mechanism where all game objects are cleaned up when a quest is unloaded.

Another helper object is the Selective-Modification, which is a record describing requested changes to the default behavior of entities (or entity sets) during gameplay. These records can perform tasks like changing the distribution of lootable items provided by NPCs, and preventing clients from seeing or interacting with entities. Selective-Modifications can be active in a radius around a particular entity, a radius around a particular location on the map, or zone-wide.

In addition to applying to existing entities, a Selective-Modification can apply to future entities. For example, a spawn-group may be marked as visible only to a set of clients, causing any future NPCs placed into this spawn-group to inherit the modification.

13.5. Client-Side Effects

The server (and therefore the generator) is able to make creative use of facilities on the client-side to create effects not present in the original Everquest. One simple example is long-term particle effects which the server can turn on or off at a location. The particle effect animation is restarted periodically until the server is told to stop the effect.

Another useful effect was the introduction of structures which could be attacked/destroyed. Structures are not normally targetable, nor can they be affected by player action. My solution was to create an invisible proxy NPC who shares the same space as the building. When the player attempts to target the building, they are in fact targeting and later attacking the proxy NPC. When this NPC is defeated, the structure is removed from the world.
13.6. Cache Objects

Runtime node and trigger caches are maintained by separate cache managers. These load nodes and triggers from the database on-demand, and maintain reference counts so that the objects can be unloaded when no longer needed. The node cache manager can respond to common queries about node state.

The trigger cache manager is more complicated, as in addition to maintaining a reference-counted cache, it also maintains mapping tables which allow trigger queries to be processed quickly. For example, one can ask for a list of characters who have a particular trigger active, obtain a list of active triggers, or a list of activation triggers for a zone. There are many queries one might make for a particular trigger, and the ability to inject events into a particular trigger.

13.7. Quest Node Architecture

As quests can be represented by a partially ordered graph, a runtime Node object corresponding to a graph node is used to keep track of player progress. Each node is a static/shared object which has a set of triggers, either optional or required. As triggers complete, or fire, the node moves closer to completion. Once all required triggers in a node have completed, the node completes and advances to the next node in the graph. The completion state, and any other node state is maintained in client-state objects. Node objects do not maintain client state (such as sequence positions for individual triggers), just the shared topology for a quest.

13.8. Trigger Architecture

The main technique used to manage the sequence of expected events in the game is the use of a new set of trigger objects. A trigger is an object which activates when specific conditions exist in-game. Triggers therefore correspond to in-game actions or events. Triggers are static objects, but per-trigger state may be maintained as part of Client State. Triggers are managed by a separate trigger manager, discussed below.

Triggers are categorized by event type (such as dialog or item transfer).
- Null: A null trigger fires immediately upon activation. These are good for initialization of sections of quests.
- Say: These fire when text spoken by a client to an NPC matches a regular expression contained in the trigger.
- Item: Fires when an item-type enters a player’s inventory by any means.
- Drop: Fires when an item-type is discarded by a player.
- Spawn: Fires when a particular type of NPC spawns.
- Death: Fires when a particular NPC, or type of NPC, or member of a spawngroup is killed by a player.
- Proximity: Fires when a player enters or leaves an area centered on an entity or a map location.
- Damage: Fires when an entity’s health drops below a certain ratio due to player combat.
- Use: Fires when an item is used. Can optionally be limited to use on a particular entity.
- Waypoint: Fires when an NPC arrives at a pre-defined waypoint.
- Signal: Fires when an external signal is received. Timer expiration is a good example of such a signal source.

13.9. Trigger Manager

Events in-game are passed to a trigger manager, which then passes the events to any active triggers of the appropriate type. Each trigger examines the event and any client-specific state and decides whether to activate (or fire). More than one trigger may fire as the result of a single event, for example if multiple players are running a quest together and cooperating on mob battles. In this case, a mob’s death will be reported to each player involved in the battle who was also at the proper stage of the quest.

Triggers will only process events when they are associated with the entities referred to in the event. For example, a player must have a quest and be at the appropriate state in the quest before a trigger may fire. Clients may associate with any number of triggers.
For example, a quest may have multiple nodes active, and a node may have multiple active triggers in the event of partial ordering of actions. A single trigger may have more than one associated client, as would be the case if players multiple players are at the same point in the same quest, whether or not they are working together.

A firing trigger will update client-specific state (if appropriate), and will call a trigger-specific PERL handler. This trigger handler can manipulate game objects through the quest manager. It is therefore possible for a single event to result in multiple calls to the same trigger handler, each with different variable bindings.

13.10. Adapting to Another Game

The current implementation of the generator is compatible with the Sony game Everquest, but there is no reason why the generator could not be made to support a different game.

Terrain for most games is stored as a heightmap, but Everquest stores zone geometry as a mesh. This means that overhangs and other areas with multiple locations at the same (x,y) coordinates are currently supported in the generator. Voxel-based games would need a different terrain generation technique, which could be a modification of the one presented earlier. Any game with quests will likely share the same quest structure, as indicated by the use of different games as source material during the quest analysis phase. Different games may require different strategies, but the same types of behavior are seen across different games and genres [17]. All MMORPGs make use of dynamic assets to support game execution, for example items and NPCs. The properties of these objects might change, but the usage by quests should not. Collecting items and killing enemies will always be found in RPGs. Economics should be the same in any game that allows players to buy/sell items and create new items.

The greatest change needed to add support for a different game will be in the knowledge base used by the quest generator. Item properties need to be added to the rulebase, as well as rules needed to support asset allocation: The generator is going to need to know that Item #1000 is used by certain NPCs, and that there are certain techniques for acquiring
that item. The mechanics of obtaining items from NPC drops, ground spawns, item creation, vendor purchases are present in most games. I am not aware of any methods which are not found in Everquest, although there are methods used in Everquest which are not found in other MMORPGs.
BIBLIOGRAPHY


