University of Alberta

Library Release Form

Name of Author: James Michael Redford

Title of Thesis: A Visual Tool for Generative Scripting in Computer Role-Playing Games

Degree: Master of Science

Year this Degree Granted: 2003

_____

Signature

James Michael Redford
Apartment 311, 10145 - 121 Street
Edmonton, Alberta
Canada, T5N 1K5

Date: _____

University of Alberta

A Visual Tool for Generative Scripting in Computer Role-Playing Games

by

James Michael Redford

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

Department of Computing Science

Edmonton, Alberta
Fall 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled *A Visual Tool for Generative Scripting in Computer Role-Playing Games* submitted by *James Michael Redford* in partial fulfillment of the requirements for the degree of *Master of Science*.

Duane Szafron
Co-supervisor

Jonathan Schaeffer
Co-supervisor

Jim Hoover

Craig Montgomerie
External Examiner

Date: _____

Abstract

It is common for computer game developers to release world-building toolsets to the public, packaged with their games. This allows home users to create and share custom game modules. Many toolsets utilize text-based scripting languages to specify in-game character behaviors. While text-based languages are generally powerful, non-programmers find them difficult to learn and use. This is unfortunate since many professional game designers and home users are not proficient programmers. In this dissertation, we present our research in developing a powerful and practical visual scripting tool, called *ScriptEase*, which automatically generates code in an underlying text-based scripting language. In *ScriptEase*, users specify information through a series of questionnaires and context-sensitive menus. Currently, *ScriptEase* generates scripts for Bioware's computer role-playing game, *Neverwinter Nights*, but in the future, it could be extended to work with a variety of different games.

Acknowledgements

I would like to thank the following people:

- My fiancé, Melissa, for her love, support, and patience while I played video games in the name of academic achievement
- My parents, for the encouragement they have provided throughout my life
- Matt McNaughton, my partner in crime, for his substantial and essential contribution to *ScriptEase*, our productive discussions, and generally letting me bug him about every little thing
- My supervisors, Duane Szafron and Jonathan Schaeffer, for their guidance, collaboration, and generous financial support
- The other member of my examination committee – Jim Hoover and Craig Montgomerie
- Dominique Parker, for his contribution to *ScriptEase* and his collaboration over the summer
- Mark Brockington, Don Moar, and everyone else at Bioware who took time out of their busy schedules to provide us with valuable feedback
- Sean McQuillan, for his efforts in testing *ScriptEase*
- John Anvik, Calvin Chan, Maria Cutumisu, Patrick Earl and all of the other students and faculty in the Software Systems Group, for their friendship, and for providing a comfortable and stimulating research environment

Contents

List of Figures

Chapter 1

Introduction

In 2002, the commercial computer gaming industry generated over $10 billion US in the United States alone [20]. In the past ten years, advances in computer graphics hardware and software have led to a cascade of games with increasingly more impressive graphics. Accordingly, high-quality computer graphics has been the primary selling point of most computer games. Recently, with faster processors, larger memories, and more sophisticated graphics cards, this has reached a saturation point [16]. As a result, game developers are beginning to shift their focus towards other areas, such as artificial intelligence (AI) to provide a more realistic gaming experience [16]. *The Sims* [60] and *Black and White* [45] are notable examples of successful commercial computer games that incorporate AI as a major feature.

Computer role-playing games, or CRPGs, are one particular genre that is making strides in the area of AI. These games involve a large virtual world populated by a host of characters. In order to make the characters believable, developers need to provide them with realistic and complex behaviors. Additionally, conversations between characters and the player are vital to the gameplay of most CRPGs. Creating natural conversations is another daunting task that developers face. Finally, CRPGs cast the player in an interactive story, and like all stories, CRPGs require a plot to be specified. To further complicate matters, CRPG plots are non-linear and interactive. All of these features require special development tool support.

Typically, the virtual game worlds of state-of-the-art CRPGs are vast and detailed. Specifying the spatial layouts and graphics for the environment, as well as the 3-D prop and character positions, is complicated. Developers create world-building toolsets that their designers use to specify all of this information. Toolsets also provide special support for specifying the AI-related character-behaviors, conversations, and plots. Most CRPGs script behaviors using rule-based systems, finite state machines, or scripting languages. Scripting languages provide a powerful behavior- and plot-specification mechanism in computer games, and many game toolsets utilize them. Games like *Neverwinter Nights* [56] and *Morrowind* [47] provide powerful text-based scripting languages in their toolsets. Other games, like *Starcraft* [61] and *Unlimited Adventures* [66], incorporate easy-to-use visual scripting tools into their toolsets.

State-of-the-art scripting tools for computer games suffer from a trade-off between power and ease-of-use. Power is necessary in order to create sophisticated and realistic behaviors. Easy to use visual tools are desirable since game designers are not necessarily programmers. Additionally, it is becoming

more common for game toolsets to be released to the public. Home users are also not likely to have much, if any, programming experience, making ease-of-use even more desirable.

Modern visual scripting tools lack power because they are inflexible. The set of behaviors that they can represent is small and inextensible by end users. For instance, the design tool included in *Unlimited Adventures* [66] allows designers to choose from thirty-six different high-level event types, such as combat and finding treasure, which can be customized and added to a game module. However, event types that were unanticipated by the original game designers are not included in this list, and end-user designers cannot extend the available events. Therefore, designers are restricted to a small set of possible behaviors that can be scripted. The root of the problem lies in the high level of abstraction, at which visual scripting tools work. Lower level mechanisms are needed to build custom high-level visual structures.

On the other hand, text-based scripting languages are quite powerful. The scripting language for *Neverwinter Nights* [56] is a procedural C-like language, which provides designers with access to the game engine's state through a set of basic library functions. Designers build complex behaviors using these library functions. Text-based scripting language work at a much lower level of abstraction than visual scripting tools, which is the basis for their power. At the same time, the low level of abstraction is one of the main reasons that text-based languages are hard to use.

Our aim is to develop a method for specifying behaviors in an easy-to-use visual environment, while maintaining a level of power close to that of a text-based language. Our approach involves working on multiple levels of abstraction. The highest level allows the average non-programming designer to script behaviors by instantiating and customizing high-level behavioral templates, called *Patterns*, in a purely visual programming environment. At this level, all information is provided using context sensitive menus and questionnaires. No coding is required. The lower levels allow designers to create new patterns and specify new basic actions that the game's characters can perform. Some programming skill and/or coding may be required at the lower levels, but not nearly as much as a text-based language requires. Finally, we automatically generate code in a text-based scripting language – the lowest level of abstraction – based on the information specified in the visual tool.

In this dissertation, we present our research in creating an easy-to-use and powerful visual scripting tool called *ScriptEase*. *ScriptEase* allows game designers to build complex scripts in a purely visual environment for the state-of-the-art computer game *Neverwinter Nights* [56], by Bioware. Information is specified in *ScriptEase* using reusable patterns and context-sensitive menus. Code is then automatically generated in *Neverwinter Nights*' text-based scripting language. *ScriptEase* includes a *Pattern Builder*, which allows new patterns to be

created, and an *Atom Builder*, which allows new primitive conditions and actions to be defined. These lower level support tools add flexibility to the system by facilitating extension.

ScriptEase augments the *Neverwinter Nights* toolset, called *Aurora*, by replacing all functionality normally specified using the scripting language, including behaviors, plotlines, and some parts of conversations. *Aurora* is still needed in order to create the game modules, specify environments, place props and characters, and define the structure of conversations. Based on internal testing and a small usability review, we are confident that *ScriptEase* is a powerful and practical visual scripting tool. We plan to release *ScriptEase* to the public in the near future. We hope *ScriptEase* will be able to capitalize on the vast base of *Aurora* users already established.

In this chapter, we have offered the motivation behind our research and a briefly presented *ScriptEase*, the visual scripting system that we have developed to reach our goals. In Chapter 2, we define and discuss computer role-playing games, including the challenges faced by designers and potential research opportunities. We also introduce *Neverwinter Nights* and its toolset, *Aurora*. In Chapter 3, we describe previous work relating to our research. We present two computer game toolsets other than *Aurora* – *Unlimited Adventures* and *The Elder Scrolls Construction Set* – and the way in which they support scripting. We then mention four visual programming environments: *Prograph*, *Sanscript*, *Starlogo*, and *CO_2P_2S*. Next, we discuss various behavior specification techniques, including scripting languages, finite state machines, and force fields. We conclude the chapter with a look at the automatic code generation mechanisms of generative design patterns, generative questionnaires, and a novel generative learning technique. In Chapter 4, we present *ScriptEase* in detail and discuss the usability review that we performed. Finally, in Chapter 5, we discuss the issues addressed by *ScriptEase* and present possibilities for future work.

Chapter 2

Computer Role-Playing Games

2.1 Background

2.1.1 Definition

In order to discuss computer role-playing games, we first need to define them. Providing a definition of computer role-playing games (or CRPGs) is not easy, and there are a number of varying opinions on the subject [28][30][34][35]. In general, we can define a CRPG as a game where the player controls a character that is the protagonist in an interactive story. Unfortunately, this definition is not specific enough and can also be applied to adventure games such as *Resident Evil* [59], first-person shooters such as *Quake* [58], and general action games such as *Super Mario Brothers* [62]; none of which are considered CRPGs. So rather than attempting to provide a formal definition, we will instead list several features that seem to be present in most CRPGs, especially in the fantasy sub-genre of CRPGs.

Abilities

"Speechcraft allows you to influence others by admiring, intimidating, and taunting them. Listeners are more willing to divulge information or to entrust important tasks to the skilled speaker." **– *Morrowind*** [47]

One feature that seems to appear in every CRPG is a mechanism for characters to acquire abilities and/or improve them. The variety of abilities that might be included in a CRPG is quite extensive. Some examples could be casting a magic spell, picking a lock, long-distance running, or playing a musical instrument. Some abilities are represented by integer-valued attributes; the higher the value, the more skilled the character is at using that ability. There are other abilities that a character either does or does not have; there are no levels of skill associated with them. CRPGs employ a number of different mechanisms for improving skills. Two examples are given below.

In *Neverwinter Nights* [56], characters have an integer-valued attribute that is called *experience*. Characters gain experience by accomplishing tasks in the game such as killing monsters and completing quests. When a character's experience score reaches certain levels, a number of their abilities might improve and they may even gain new abilities.

In another example, *Hero's Quest* [54], a character possesses a number of physical attributes such as strength and intelligence, and skills such as climbing and throwing, all of which are represented by integer values between 0 and 100. As a character uses a particular skill in the game, that skill score gradually increases, as do the physical attributes associated with that skill.

Exploration

"Welcome back, oh illustrious adventurers! Long has been thy sojourn in this strange realm, though 'tis a fitting respite for great heroes." **– *Ultima 3: Exodus*** [64]

Characters are placed in a virtual world that needs to be explored. In the fantasy sub-genre, these worlds are vast and inspired by the medieval fantasy worlds of *Dungeons and Dragons* [9] or *Lord of the Rings* [21]. The landscapes are dotted with cities, towns, castles, and dungeons, which are populated by a variety of colorful characters ranging from simple farmers, to valiant warriors, to fierce monsters. This feature is not unique to CRPGs, but CRPG worlds are typically larger and more detailed than in other game genres. To fully explore a world, the player must usually take advantage of other CRPG features. For instance, a player might have to gather information about a particular location, by talking with some of the supporting characters, before traveling there. Alternatively, a player might have to acquire a specific item or accomplish some complicated task before being allowed to enter a certain area.

Equipment and Treasure

"Amulets are necklaces with some form of large decoration or symbol. Most are ornamental, but some are infused with magic." **– *Neverwinter Nights*** [56]

An important aspect of CRPGs is the constant search for more powerful equipment. In the fantasy sub-genre, this primarily includes weapons, armor, and other magical items. Characters also acquire wealth, usually in the form of gold coins, which can be used to purchase equipment from shopkeepers. Gold and equipment are typically found on the bodies of monsters that the character has killed and in treasure chests or other containers hidden in hard to reach locations.

Monsters

"Also called Illithid, the brain devouring Mind Flayers are hideously alien creatures of the Underdark. Evil beyond redemption, they will consider you a slave or simply food, if they consider you at all." **– *Baldur's Gate 2: Shadows of Amn*** [44]

Fighting with monsters is a theme that runs through all fantasy CRPGs. Not only is it fun, it also usually provides the primary mechanism for gaining treasure and improving abilities. Again, battling monsters is not a unique feature of the CRPG genre, but it incorporates other CRPG features. For instance, when fighting, characters can usually use certain skills or abilities that they have acquired, such as the ability to trip an opponent, or to cast a magical spell. Characters also utilize

a variety of weapons, armor, and other offensive and defensive equipment that they have acquired throughout their adventures.

Supporting Characters and Conversations

"It is almost impossible to solve thy quests without talking to virtually all people in each towne." ***– Ultima 4: Quest of the Avatar*** [65]

CRPG worlds are populated with a number of supporting characters, often referred to as non-player characters or NPCs. Some of these characters play mundane background roles such as farmers, peasants, artisans, politicians, tradesman, and all of the other miscellaneous roles found in everyday life. These background characters typically execute very simple and limited behaviors when interacting with the player. Other NPCs play important plot-dependant roles and have complex interactions with the player.

The player can also talk to NPCs. The characters in some CRPGs, such as *Final Fantasy* [50] and *Ultima 1* [63], speak a single line of text when the player approaches them. There is no real interaction between the player and the NPC in this model. Games such as *Hero's Quest* [54], *Gorasul* [53], and *Ultima 4* [65], allow the player to type in subjects to ask the NPC about. This conversation model offers a much more interesting challenge to players, as they have to discover the relevant topics of conversation for each NPC. Finally, most of the more recent CRPGs, such as *Baldur's Gate* [43], and *Neverwinter Nights* [56], present conversations as a series of dialog windows that include the words spoken by the NPC and a list of sentences from which the player can select a response for their character. Another notable game is *Morrowind* [47], which uses elements from all three conversation models.

Puzzles and Investigation

"The man who invented it, doesn't want it for himself. The man who bought it, doesn't need it for himself. The man who needs it, doesn't know it when he needs it." ***– Baldur's Gate 2: Shadows of Amn*** [44]

Many CRPGs challenge players with puzzles. Some are as simple as answering a riddle, others are more complicated. For instance, a puzzle found in *Baldur's Gate 2* [44] involves 12 items and 12 statues. The player must place the correct item in the hands of each statue. Each statue includes a hint indicating which item should be given to it. Another example, from *Shadows of Undrentide* [57], involves a grid on the floor. The player is required to place objects called rune stones in certain cells of the grid, forming a particular pattern. The room containing the grid also contains various clues as to what this pattern should be. When these puzzles are solved, the player gains some type of reward.

Many quests involve an amount of investigation as well. For example, a quest in *Baldur's Gate 2* [44] begins when a city guard informs the player that a series of

murders has occurred in a particular area of the city. The player then attempts to find the killer by investigating that area of the city; interviewing witnesses, gathering evidence, and such.

Plot

"A city dead for a thousand years. A city I had to see with my own eyes. The end of Yuna's journey. The last chapter in my story." **– *Final Fantasy X*** [51]

As mentioned earlier, CRPGs are a form of interactive story where the player is the protagonist. As with all stories, CRPGs have a plot. Some games, like *Final Fantasy X* [51], immerse the player in a very rich and involved story written by a professional writer. The plot is completely linear and unfolds in the exact same manner every time through the game. Many players dislike linear plots such as this, but *Final Fantasy X*'s story is so well written and interesting, that most players do not mind.

Other games, such as *Baldur's Gate* [43] and *Neverwinter Nights* [56], have one essential plotline that runs through the entire game, and dozens of non-essential sub-plots that appear from time to time. These sub-plots (sometimes called mini-quests) are usually unrelated to the main plotline, so the player is not required to follow them. Typically, a number of these plotlines possess a small degree of non-linearity. Depending on the player's choices, these plots unfold in different ways. For instance, a mini-quest might indicate that the player is supposed to acquire a magical ring for his king, and that the mighty dragon Trogdor the Burninator [29] possesses this ring. The player might have several options. The character could slay the dragon and take the ring. The character could sneak into Trogdor's lair and steal the ring. The character could negotiate with the dragon to trade something for the ring. The character could even create a forgery of the ring and attempt to pass it off to his king as authentic. There could be a multitude of other possibilities that the game designers had included, as well.

2.1.2 Design Challenges

Now that we have identified several features that define computer role-playing games, we can discuss why research in this area is interesting. There are a number of challenges that the game designers face when creating CRPGs. They are listed below. In Section 2.2.2, we will discuss how the *Aurora Toolset* can be used to meet these challenges. This is the tool used by Bioware to design their game, *Neverwinter Nights* [56].

World Layout

CRPG worlds can be very large, consisting of hundreds of different areas. The game developers need to be able to specify the layout of these areas. For instance, there may be a large city that consists of hundreds of houses, businesses,

streets, markets, plazas, and parks. The city may also be decorated with streetlights, flags, trees, signs, fences, gates, monuments, and many other miscellaneous fixtures. The game developers must be able to create and layout the components of this city easily and relatively quickly (after all they also have a hundred other areas just as complex as this one to design).

Non-Player Characters (NPCs)

The world also needs people to populate it. Game developers need to specify the physical appearance and clothing of an NPC, as well as any attributes and abilities used by the game engine. For instance in *Neverwinter Nights*, the developers had to specify attributes for each NPC, such as their name, gender, intelligence, and voice among many others. They also had to specify abilities, such as the magic spells that the NPC could cast, the different weapons and armor that the NPC could use, and the various skills that the NPC possessed. Finally, these NPCs have to be placed in the world so they can take on a role in the story.

Specifying Behaviors

In order to make NPCs interesting, developers need to tell them what to do. For example, Clancy is a castle guard and he guards the King's treasure chest. When anyone other than the King or another guard enters the room, Clancy tells them to get out. If the intruder does not leave immediately Clancy sounds an alarm by ringing a large gong in the corner, then attacks the intruder. If Clancy does not see another person for one hour, he gets tired and falls asleep. At 12:00, Clancy gets hungry and wanders off to the kitchen to get food. Clancy may have many more behaviors as well. Specifying a complex set of behaviors such as this, for an NPC, can be a time-consuming process for developers.

Encounters

The previous section describes a set of behaviors applied to a single principle actor, but there are other situations in CRPGs where interactions between multiple participants need to be defined. We call these interactions *Encounters*. The following is an example of a typical encounter. The participants in this encounter are a treasure chest, a sword, a statue, a demon, and the player's character. There is a room in a castle with the sword inside the treasure chest and the statue beside the chest. If the player's character opens the treasure chest and takes the sword, the statue transforms into a demon and then attacks the character. The behavior of the demon attacking the character could be part of the encounter specification, or it could be part of the demon's personal set of behaviors.

Conversations

Talking to NPCs is essential in CRPGs. Conversations are the player's primary source of information when trying to solve the various quests in the game. In

Section 2.1.1, we presented three common conversation models used in CRPGs. The third model where conversations are presented through a series of dialogs is the one we are most concerned with in this dissertation. Some conversations are quite large and consist of dozens of possible dialog windows that could be presented to the player. A conversation can be represented by a directed graph, where the nodes contain the text that an NPC speaks to the player, and the edges represent the player's possible responses to what the NPC says.

Plots

Specifying plots, especially non-linear plots, is not a simple task. Plot can be represented by a directed graph. Nodes in the graph represent different states along the plot line, and edges represent actions the player can take to advance the plot in some way. Developers must identify and specify plot states, indicate which actions the player can perform in order to affect the plot, and then indicate how the plot is to be advanced. They are also challenged with ensuring that no player actions can break the plot, causing a nonsensical event to occur, or even worse, making the game impossible to finish. Some games provide fail-safes to fix some plot problems that might occur. For instance, *Neverwinter Nights* provides the player access to a container called a Divining Pool. If the player ever loses some plot-critical item, it will appear in the Divining Pool so the player can retrieve it.

2.1.3 Pen and Paper Role-Playing Games

Computer role-playing games are inspired by their pen and paper counterparts, such as *Dungeons and Dragons* [9]. Some games, such as *Baldur's Gate* [43], *Icewind Dale* [55], and *Neverwinter Nights* [56] are based directly on various *Dungeons and Dragons* rule sets. Pen and paper RPGs involve a human Dungeon Master that is responsible for running the game. The *Dungeons and Dragons* third edition *Player's Handbook* defines the Dungeon Master, or DM, as "the player who controls non-player characters, makes up the story setting for the other players, and serves as a referee." [9] The DM is responsible for describing the game world and situations that the players are in by verbally describing them or using visual aids such as miniature figurines, maps, and other props. In turn, the players describe the actions that they wish to take to the DM, and then the DM determines what the consequences of those actions are and relates them back to the players. Computer role-playing games attempt to emulate this experience by having the computer take on the role of DM. Both pen and paper, and computer RPGs have their advantages. Pen and paper RPGs give players the benefit of a human DM capable of improvisation, and CRPGs provide a visually exciting and interactive graphical environment for the game.

2.1.4 Research Opportunities

The last four challenges mentioned in Section 2.1.2 - behavior specification, encounters, conversations, and plots - offer particularly good research opportunities since all four of these elements are still limited in state-of-the-art games.

Plots often consist of only a handful of states and any non-linear plotlines are not very complex. However, it is hard to criticize CRPGs for this. The truly non-linear storylines that players enjoy in pen and paper role-playing games require the imaginative and improvisational interaction between a human Dungeon Master and the players. In computer role-playing games, the Dungeon Master is a computer program that is only capable of doing what the designers instructed it to, during development. The imagination of the players is also limited by the game's user interface. The game is not capable of allowing the player to perform absolutely any action they would like, because in order for the game to respond to a particular player-action, that action must have been anticipated by the game designers during development, and an appropriate response programmed. It is unrealistic to expect computers to be as capable Dungeon Masters as humans in the near future, but there are opportunities for improvement in creating non-linear CRPG plotlines.

Conversations in CRPGs often feel very unnatural. The reason is that the amount of information that an NPC is capable of conveying to the player is rather limited. Typically, NPCs have only a select few pieces of information that they are designed to reveal to the player. This lack of information can lead to an NPC repeating phrases several times within a single conversation. It also means that the player cannot talk to an NPC about any arbitrary topic; just the ones the NPC was designed to talk about. The reasons for this are that designers do not want to bog the player down in conversations full of useless information, and it would also take a long time to specify very verbose conversations. The length of a conversation is especially relevant considering most CRPGs are released in several different languages. However, the creation of realistic conversations in CRPGs is certainly a good opportunity for research.

The behaviors exhibited by creatures are often predictable and not very intelligent. For instance, it is common for a player to enter, say, a tavern, open some crates and barrels, and then take the items inside. In reality, the proprietor of the tavern would consider this stealing and attempt to stop the player by attacking them or calling the city guards. In many CRPGs, the player receives no negative feedback in situations like this. Typical CRPG worlds can contain hundreds of buildings, all of which would require the developer to put in the effort to specify behaviors for the occupants to protect their belongings. Usually, this type of behavior is not critical to the plot of the game, so it is understandable why developers might overlook it and spend their time defining more important behaviors. There are a great many non-critical behaviors such as this that

developers just do not have the time to implement by hand, but would add a great deal of realism to the game. Development tools that can automatically generate and attach these behaviors to characters would be useful.

Finally, encounters are so pervasive in CRPGs that they can often be categorized into groups or patterns. For instance, the encounter example given in Section 2.1.2, where the statue turns into a demon when the player's character takes a sword out of a treasure chest, is part of a pattern of similar encounters. Variations of this pattern might involve changing the participants. For instance, there could be an encounter where a kitten transforms into a bear when an amulet is removed from a pedestal. Variations could also be changes in the actions performed by the participants. For instance, when an emerald is placed inside a desk drawer, the bookcase behind the desk disintegrates revealing a secret passage. Specifying these patterns in such a way that they can be quickly customized to fit the many different variations required by the game designers is an interesting problem.

2.2 Neverwinter Nights

2.2.1 The Game

Neverwinter Nights [56] is a state-of-the-art computer role-playing game developed by Bioware. Since its release in the summer of 2002, *Neverwinter Nights* has been a critically-acclaimed best seller and has won over eighty awards [26]. The game is based directly on the third edition rules of the pen and paper role-playing game *Dungeons and Dragons* [9], and contains all of the features of a CRPG listed in Section 2.1.1. The vast game world is full of fantastic treasures and horrible monsters. The variety of abilities that a character can possess includes hundreds of skills, feats, and magic spells. There are hundreds of supporting characters, all of which have their own set of behaviors and are capable of conversation with the player. The game has a major plotline that runs through it and dozens of mini-quests for the player to complete, many of which include puzzles and investigation. A screen shot from *Neverwinter Nights* is shown in Figure 2.1. It includes the player's character Na'thal Vale, several NPCs, and a monster – the bear – in an interior setting. Na'thal is equipped with a green-glowing magical sword. The circle of light on the floor with a halo above it is a magical transportation portal. Bioware developed and used the *Aurora Toolset*, presented in the next section, to specify all of this information.

2.2.2 The Aurora Toolset

When Bioware released *Neverwinter Nights*, they packaged the *Aurora Toolset* with the game. *Aurora* is the world-building tool that was used by Bioware's

Figure 2.1: A screenshot from *Neverwinter Nights.*

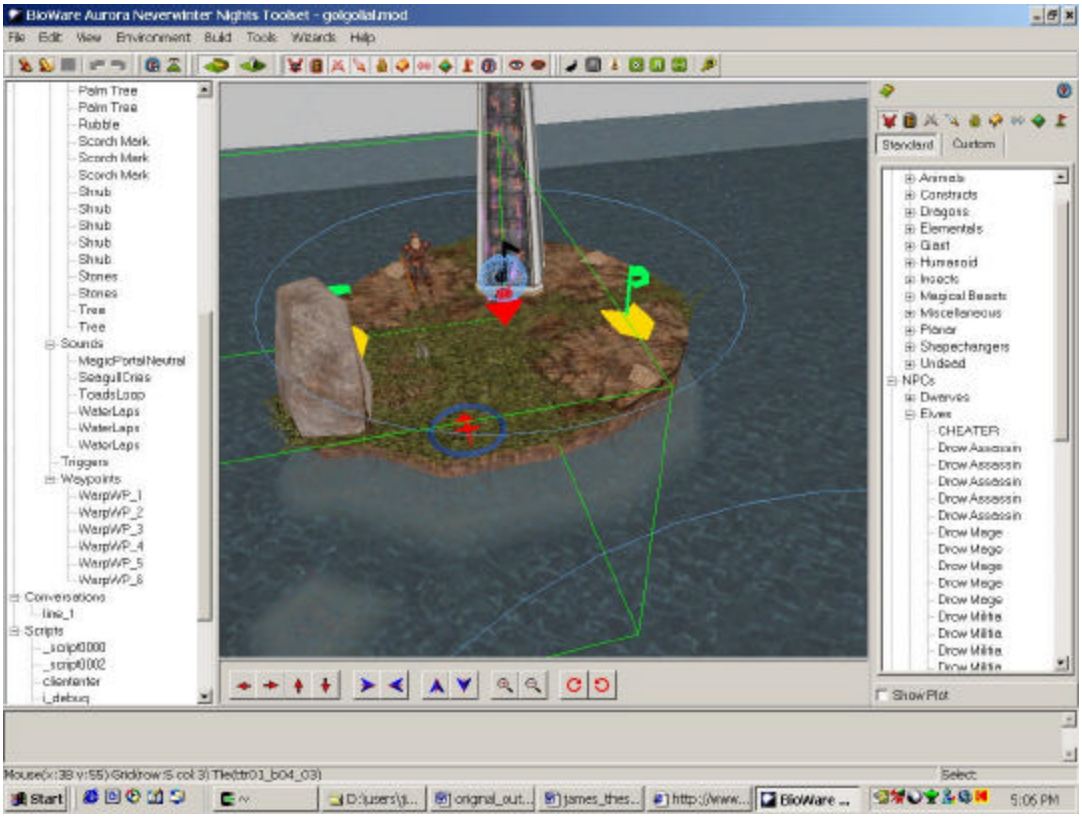

Figure 2.2: A screenshot from the *Aurora Toolset*.

game designers to create the entire *Neverwinter Nights* game world. *Aurora* allows home users to create their own worlds and adventures using the *Neverwinter Nights* game engine. While *Neverwinter Nights* is not the first game that has included world-building tools, *Aurora* is groundbreaking in the power that it gives to its users and the ease with which it can be used. *Aurora* provides solutions to all of the design challenges listed in Section 2.1.2.

World Layout

The appearance of each individual area is governed by an environmental template, called a tileset. The *Aurora Toolset* provides the user with about a dozen different indoor and outdoor tilesets to choose from, including forests, deserts, castle interiors, and caverns. Once a tileset is chosen, the user lays out the terrain features such as trees, roads, walls, and water. The terrain features available are dependant on the chosen tileset. In addition to the general terrain layout, the user can drop miscellaneous objects (called placeables) into the world such as statues, wagons, crates, and treasure chests. These objects are independent of the tileset and can be placed into any area by simply pointing and clicking. A screenshot from the *Aurora Toolset* is shown in Figure 2.2. It shows an NPC, an obelisk (placeable) and a boulder (placeable) on a small island surrounded by water. The panel on the left shows all objects that have been placed in the world, as well as all defined conversations and scripts. The panel on the right is the palette of all terrain types and placeable objects that are available to be placed.

Non-Player Characters

Aurora provides an extensive list of predefined creatures (NPCs and monsters) that can be placed into the world. The user can define new creatures and edit their properties through a tabbed gui dialog, shown in Figure 2.3. Creatures, like any game object, can be dropped into the world by just pointing and clicking. Creatures can also be placed at run-time using the *Aurora Toolset*'s scripting language, described in Section 2.2.3.

Specifying Behaviors

Behaviors for creatures are specified using a scripting language – *NWScript* – described in Section 2.2.3. Every creature receives a default set of behavioral scripts that handle many simple tasks, such as opening a door that blocks the creature's path, and some more complex tasks, such as the artificial intelligence (AI) for combat. The user can modify or replace any of the creature's scripts they wish. Creatures are not the only objects with scripts. Creatures, placeables, doors, and areas are some of the objects that have a set of scripts associated with them. Even the game module itself contains a set of scripts.

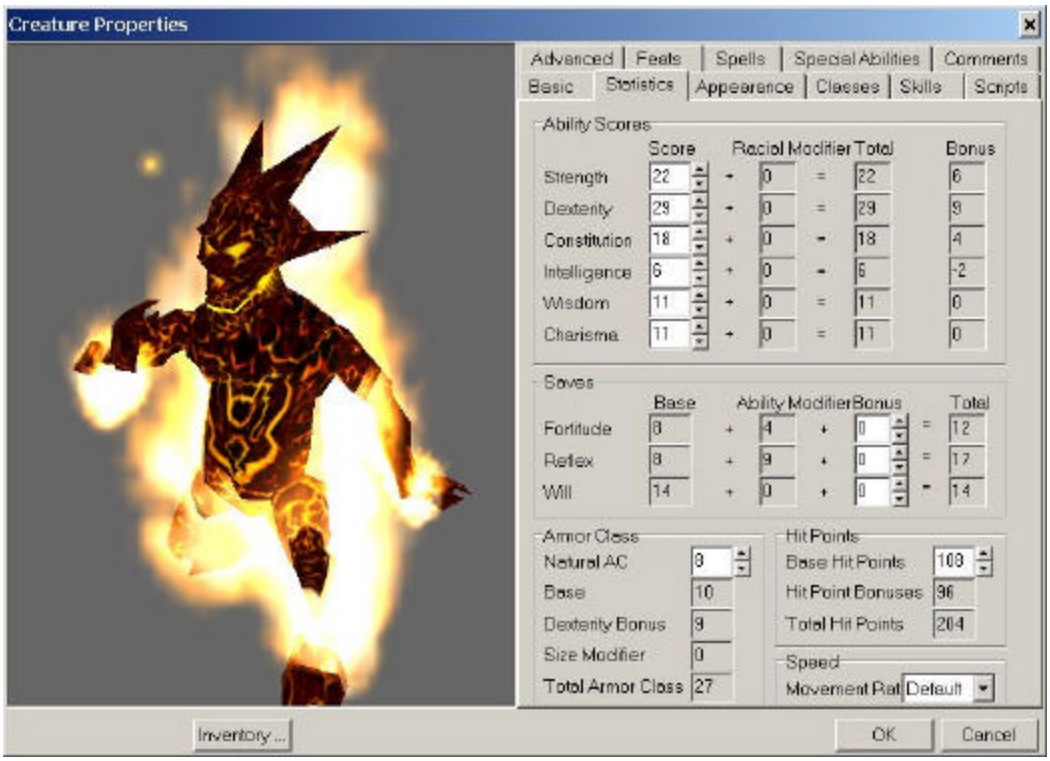

Figure 2.3: The tabbed properties dialog for a monster - The Elder Fire Elemental

Encounters

Aurora provides special support for one particular encounter pattern, where a number of creatures are created (or spawned) when a character enters some perimeter drawn on the ground. The user specifies the perimeter, the spawn point, a list of creatures, and a difficulty rating. The difficulty rating indicates how hard it will be for the player to defeat the creatures in battle, however the spawned creatures need not be hostile. The list of creatures can be further customized to indicate the minimum and maximum numbers of each type of creature to spawn. The game then randomly spawns an appropriate number of each creature to match the difficulty rating indicated by the user. *Aurora* refers to these particular encounter patterns simply as *Encounters* and the tool used to build them as the *Encounter Wizard*. Note that our use of the word "encounter" is used in the broader scope as described in Section 2.1.2. In order to specify other types of encounters in *Aurora*, such as the example given in Section 2.1.2, the scripting language must be used.

Conversations

In *Neverwinter Nights*, conversations are presented to the player as a series of dialog windows, where each window shows the text spoken by an NPC and a list of possible responses for the character. The player's choice of response affects which text the NPC will speak next. *Aurora* uses a tree type interface for specifying conversations, as shown in Figure 2.4. However, the underlying data

Figure 2.4: This is an example of a conversation specified using the *Aurora Toolset*. Text spoken by the NPC is shown in red, possible responses by the player are shown in blue, and links to other nodes in the tree are shown in gray.

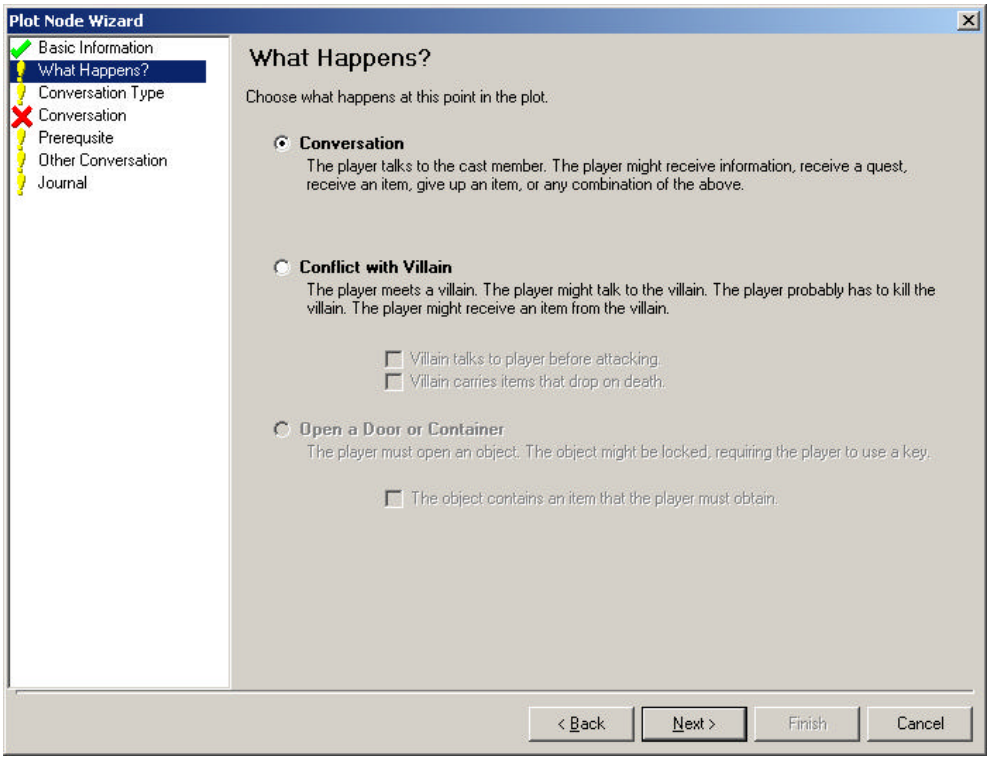

Figure 2.5: The *Aurora* plot wizard allows the user to specify a plot as a series of plot nodes. Each node involves a conversation, a conflict, or opening a door or container.

structure is actually a directed graph, since leaf nodes in the conversation tree are capable of pointing to any other node in the tree. Nodes in the conversation graph can have scripts attached to them, indicating the actions to be executed when that node is reached. Additionally, nodes can have a conditional script attached to them called a "Text appears when" script. This script contains a Boolean function that determines whether the text in the node is displayed or not.

Plots

Originally, *Aurora* contained no special support for plot. Plot was specified *ad hoc* using the scripting language, by modifying global variables. Later, Bioware released a patch that added a plot wizard to *Aurora*. The plot wizard offers a mechanism for specifying some simple plotlines. In the wizard, events that can advance the plot are limited to a conversation with a character, conflict with some villain, or opening a specific door or container, as shown in Figure 2.5. In order to implement more complicated plotlines, the user must still use the scripting language.

2.2.3 The Scripting Language - NWScript

In this section, we will describe *NWScript* (detailed in [38]), the scripting language used by the *Aurora Toolset*. However, before we begin, we need to describe some important aspects of the *Neverwinter Nights* game engine that are integral to *NWScript*.

First, we should define exactly what a script is. A script is a set of instructions that the game engine interprets as actions to perform.

Second, the *Neverwinter Nights* game engine is event-based. Almost every object in the game has a set of events associated with it. For instance, every creature handles the events shown in Figure 2.6. Each of these events can have a script attached to it. When a particular event occurs, its associated script executes.

Finally, every object has a first-in-first-out action queue, which contains all of the actions it is supposed to execute. Actions are removed from the front of the queue and executed accordingly. The scripting language is capable of modifying these queues.

NWScript is a procedural C-like language, supporting variable assignments, arithmetic operations, conditionals, loops, and function calls. *Aurora* provides *NWScript* with an interface to the game engine through a set of library functions. The best way to illustrate *NWScript* to the reader is to give a short example; consider the code in Figure 2.7. This code defines a function that teleports a player's character to a given location, as long as the distance moved is no further

Event	Description
OnBlocked	Occurs whenever a creature's movement is blocked by a door.
OnCombatRoundEnd	Occurs at the end of every round, while the creature is in combat.
OnConversation	Occurs whenever the creature finishes a conversation.
OnDamaged	Occurs whenever the creature is damaged.
OnDeath	Occurs whenever the creature dies.
OnDisturbed	Occurs whenever the creature's inventory changes in any way.
OnHeartbeat	Occurs approximately every six seconds.
OnPerception	Occurs whenever the creature sees or hears another creature.
PhysicalAttacked	Occurs whenever the creature is initially attacked by another creature.
OnRested	Occurs when the creature rests.
OnSpawn	Occurs when the creature is first created.
OnSpellCastAt	Occurs whenever a spell is targeted on the creature.
OnUserDefined	This event is user defined.

Figure 2.6: The list of events associated with a creature.

```
 1 void TeleportPC(object oPC, location lDest) {
 2   float distance;
 3   location lPlayerLocation;

 4   if( GetIsObjectValid(oPC) ) {
 5     lPlayerLocation = GetLocation(oPC);
 6     distance = GetDistanceBetweenLocations(
                    lDesc, lPlayerLocation );

 7     if( distance < 50.0 ) {
 8       AssignCommand(oPC, ClearAllActions());
 9       AssignCommand(oPC, ActionJumpToLocation(lDest));
10      }
11    }
12  }
```

Figure 2.7: Sample NWScript code.

than fifty meters. The parameter `oPC` is a reference to the character and `lDest` is the location to which the character should be teleported.

Those familiar with C-programming should have no trouble understanding most of the code. The functions used on lines 4, 5, and 6 are part of the function library provided by Aurora. `GetIsObjectValid` tests to see if an object is valid, `GetLocation` returns the location of the given object, and `GetDistanceBetweenLocations` returns the distance in meters between two given locations. Lines 8 and 9 modify the character's action queue. `ClearAllActions` empties the calling object's action queue. `ActionJumpToLocation` places an action in the caller's action queue, which will instantly transport the caller to a given location when it is executed. In this example, we do not want to modify the caller's action queue; we want to modify the character's action queue. *NWScript* provides a function called `AssignCommand`, which allows us to specify another object's action queue to

modify. Lines 8 and 9 modify the action queue of the character, referenced by the oPC variable.

NWScript is very powerful. The number of different behaviors and encounters that can be programmed by the user is virtually limitless. However, since *NWScript* resembles a traditional programming language, non-programmers find it very hard to learn and use. The fact that home users can create their own worlds and adventures is one of the primary selling points of *Neverwinter Nights.* This is unfortunate since the majority of *Aurora*'s users are not programmers and they are unable to take full advantage of the power that the scripting language offers. In Chapter 4, we present our visual programming tool called *ScriptEase*, which does allow non-programmers to specify complex behaviors.

Chapter 3

Related Work

Our research incorporates ideas from several different areas, including computer game toolsets, visual programming environments, behavior specification, and automated code generation. In this chapter, we present related research done in these areas.

3.1 Toolsets

When toolsets started to appear in the late 1980s, they were released as novelties that appealed to only small groups of dedicated gamers, due to their complexity, limited power, and lack of a distribution medium. Today, game developers spend as much time, if not more time, developing powerful and easy-to-use toolsets as they do on the actual game. This is in large part due to the size and complexity of most state-of-the-art computer games, which require sophisticated tools to specify spatial layouts and game mechanics, for their own internal development. Recently, toolsets have become so popular that it is common for developers to release them to the public and use them as promotional points when marketing their game. Toolsets released to home users increase the replayability of a game by allowing thousands of users to create and share their own custom game modules. The pervasiveness of the Internet in modern culture permits mass distribution of user-created game modules. We presented the *Aurora Toolset* for *Neverwinter Nights* in Chapter 2. In this section, we will present toolsets for two other CRPGs, *Unlimited Adventures* and *Morrowind*. There is a multitude of game toolsets available. We chose to present *Unlimited Adventures* because it was one of the first toolsets promoted as the major selling point of a game. Additionally, its behavior specification mechanism is very similar to *ScriptEase*. We chose to present *Morrowind*'s toolset because it is a state-of-the-art alternative to the *Aurora Toolset*.

3.1.1 Unlimited Adventures

Unlimited Adventures (or *UA*) [66] is a *Dungeons and Dragons* CRPG that was released in 1993 by Strategic Simulations. The player controls a party of up to six characters. *UA* is a turn-based game that uses a very simple 3D maze-like graphics engine. The game world is a grid, where each cell in the grid is a position that the party can occupy. Each of the four sides of a cell can be a wall, a

Figure 3.1: A screenshot of the *Unlimited Adventures* editor.

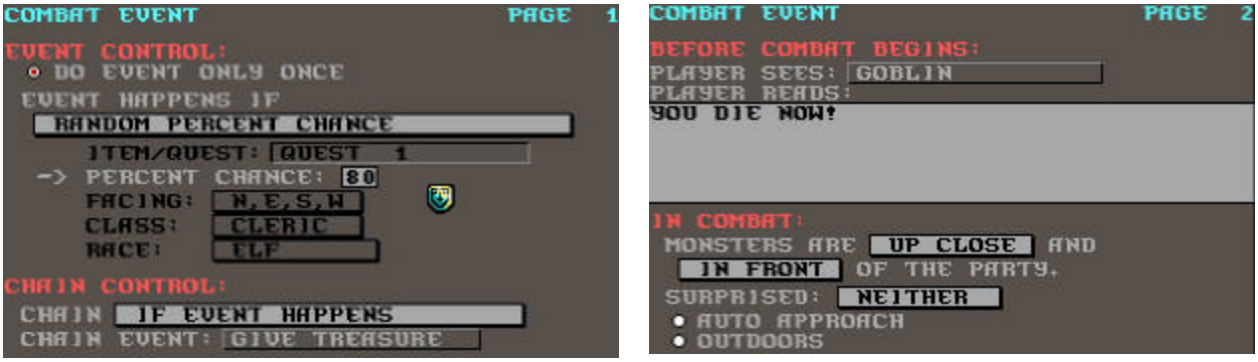

Figure 3.2: Two pages from a combat event questionnaire in *Unlimited Adventures*.

door, or simply open space. The game editor places the designer in an empty 3D maze (all cells are open). The designer can navigate the maze using the arrow keys in the same way a party moves through the maze in the actual game. The designer can place walls and doors in the cell that they currently occupy. First, the designer selects the wall image that they wish to use, such as brick, wood, or even trees, and then it is added by indicating which side of the cell should become a wall. Figure 3.1 shows a screenshot of *Unlimited Adventures*. The 3D maze view is shown on the left. The centre shows a top-down grid view of the map with a compass indicating the direction currently being faced. Information about the currently occupied cell is shown on the right. Other miscellaneous information, about the game world and the designer's current actions, is shown along the bottom.

Unlimited Adventures supports limited behavior and plot specification through a feature called events. The designer can choose from thirty-six different event

types, including combat, having an NPC talk, and finding treasure. The designer places an event in a map cell, thus the party entering an event's cell is an implicit condition for that event to fire. After an event is selected, an event-specific multi-paged questionnaire is presented, which allows the designer to modify various options in order to customize the event. Figure 3.2 shows two pages of a combat event's questionnaire. These images show several event options that the designer can modify, including an explicit condition for the event to fire, and some text to display before the battle. In this example, the explicit condition specifies that the event has an 80% chance of occurring and the text displayed before the battle is "You Die Now!". Note the *Chain Control* section in the first image. This feature allows events to be linked together. In this example, if the combat event occurs, then a *Give Treasure* event is fired right after. Some event types even allow conditional linkage to one of several possible events. Events in *UA* are similar to *ScriptEase's* situations, which we present in Chapter 4.

While *UA*'s events are not as powerful as a scripting language such as *NWScript*, it is far easier to use. The user can completely specify an event by answering several questions in an easy-to-read menu-driven questionnaire. The average menu contains between five and twenty entries. Unfortunately, the thirty-six event types cannot be modified or added to. The ability to define new behaviors that were unanticipated by the game designers would add significant power to the tool. We discuss how *ScriptEase* addresses this problem in Chapter 4.

There is one final interesting feature of *Unlimited Adventures* that we will mention in this section. The toolset is capable of showing the designer the cells in the grid that are immediately accessible to the party, the cells that are accessible only if the party has some special ability or equipment, and the cells that are completely inaccessible to the party. The usefulness of this feature is two-fold. First, if the designer intended for the party to access a particular area in the world, but forgot to create a door or some other transition mechanism for entering that area, then this feature would catch that mistake. Second, most of the mazes created with the toolset have a series of connected rooms and corridors but do not use every single cell in the grid. In other words, some of the cells are never meant for the party to enter. If the maze has a leak – a missing section of wall that gives the player access to the unused space – then this feature will catch it. Error-detection is an important, yet often ignored, feature in computer game toolsets.

3.1.2 The Elder Scrolls Construction Set

As with *Neverwinter Nights*, *The Elder Scrolls 3: Morrowind* [47] is a state-of-the-art CRPG released in 2002 with all of the CRPG features listed in Section 2.1.1. The game comes with a toolset called *The Elder Scrolls Construction Set*, or *TESCS*, that addresses all of the design challenges listed in Section 2.1.2. Terrain features, structures, and miscellaneous objects can be selected and placed in a similar fashion to *Aurora*. Supporting characters and props can be defined and customized using tabbed dialogs similar to those used in *Aurora*. *TESCS* is a

```
1 If (GetDeadCount "Trogdor" > 0 )
2   StartCombat Player
3   SetFight 100
4 endif
```

Figure 3.3: Sample of the *Morrowind* Scripting Language.

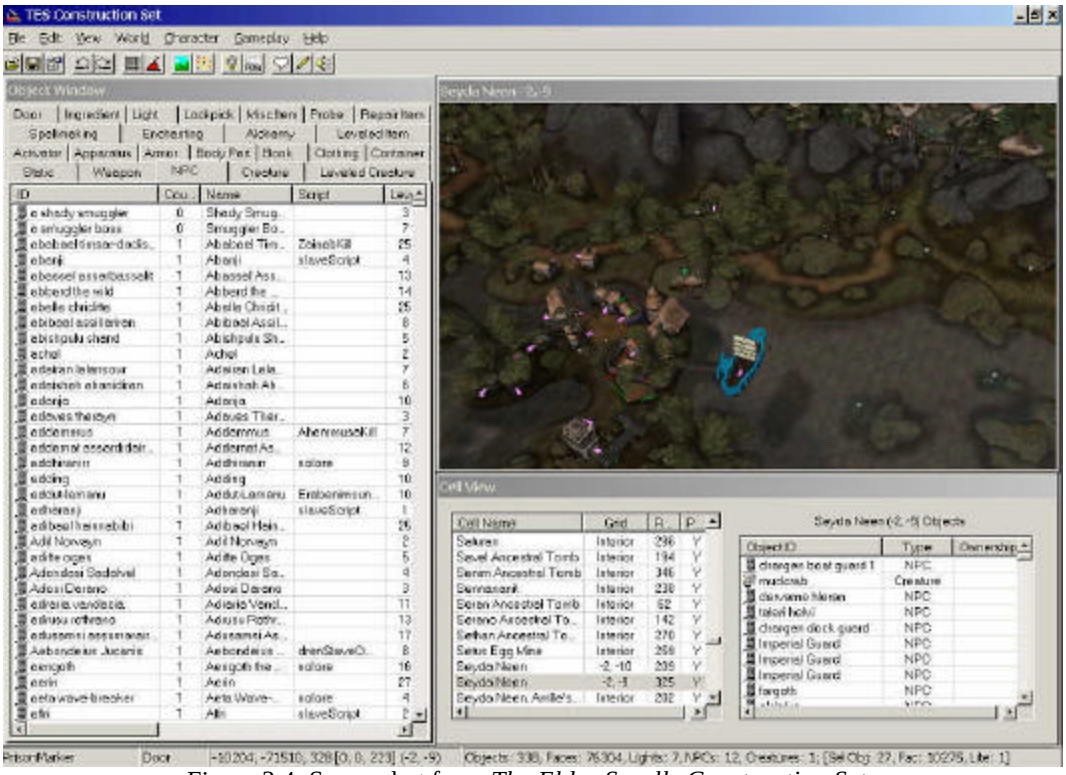

Figure 3.4: Screenshot from *The Elder Scrolls Construction Set*.

bit more complex and harder to use than *Aurora* but also gives the user some more power over the physical characteristics and 3D positioning of characters and props. Character behaviors, encounters, and plot must all be specified using *Morrowind*'s scripting language. There is no special support for these, such as the *Neverwinter Nights* plot and encounter wizards. The toolset also includes a conversation editor.

Like *NWScript*, *Morrowind*'s scripting language is procedural, and accesses the game engine through a set of library functions. A sample is shown in Figure 3.3. The code on line 1 tests if a creature called "Trogdor" is dead. If that condition is true then the executer of this script attacks the player (line 2) and sets its own morale to 100% (line 3), meaning it will never flee from the battle.

Like *Neverwinter Nights*, *Morrowind*'s game engine is event-based. However, unlike *Neverwinter Nights*, events in *Morrowind* do not modify a character's actions directly. Instead, *Morrowind*'s events merely modify the state (or memory) of a character. A character executes its script repeatedly as quickly as

possible. The script determines which actions the character should perform, based on its memory and global information provided by the game engine.

A screenshot from *TESCS* is shown in Figure 3.4. The tabbed *Object Window*, on the left, shows lists of all of the objects in the game, including NPCs, conversations, magic spells, clothing, etc. The *Cell View* window on the lower right displays a list of all of the areas in the game world as well as a list of objects that exist in the selected area. Finally, the render window displays the selected area in 3D. The user can insert, remove and modify objects by selecting them in the render window.

Morrowind's gameplay is very different from *Neverwinter Nights*, due mostly to their difference in perspective. *Morrowind* uses a first-person perspective and a *driving*-type movement interface, where the player uses the keyboard to move forward, backwards, and side-to-side. The player uses the mouse to turn and interact with the world. *Neverwinter Nights* has a floating third-person perspective and movement is accomplished fully through a point-and-click interface, where the player uses the mouse to select a location and the character then moves there. It is interesting to note that despite the games' somewhat disparate gameplay models, the functionality of their toolsets – *Aurora* and *TESCS* – is very similar.

3.2 Visual Programming Environments

Besides computer game toolsets, there are dozens of other visual programming software packages available. We mention them here because *ScriptEase* is essentially a visual programming tool, even though it is specialized for CRPG scripting. Some of these tools are general-purpose visual programming languages, such as *Prograph* [39] and *Sanscript* [37], while others, such as *Starlogo* [36], are designed to handle a specific family of problems. Others still, such as *CO_2P_2S* [14] [15], provide a graphical front-end that generates code in an underlying text-based language. Dominique Parker has written a survey of various visual programming packages [17], including *Prograph*, *Sanscript*, and *Starlogo*. In this section, we will briefly summarize these three tools. We will mention *CO_2P_2S* in Section 3.4. *Starlogo* and *CO_2P_2S* are prototypes. *Prograph* and *Sanscript* are commercial development tools. With the exception of a few very simple programs, such as those listed in [33], none of these tools are used in real-world applications.

Prograph and *Sanscript* are full-fledged programming languages as opposed to graphical front-ends that generate code for a text-based language. They both represent programs visually as data-flow graphs. The nodes in the graph represent operations to perform on the data. Each node has a set of input and output points, which can be connected to other nodes using edges. There are nodes

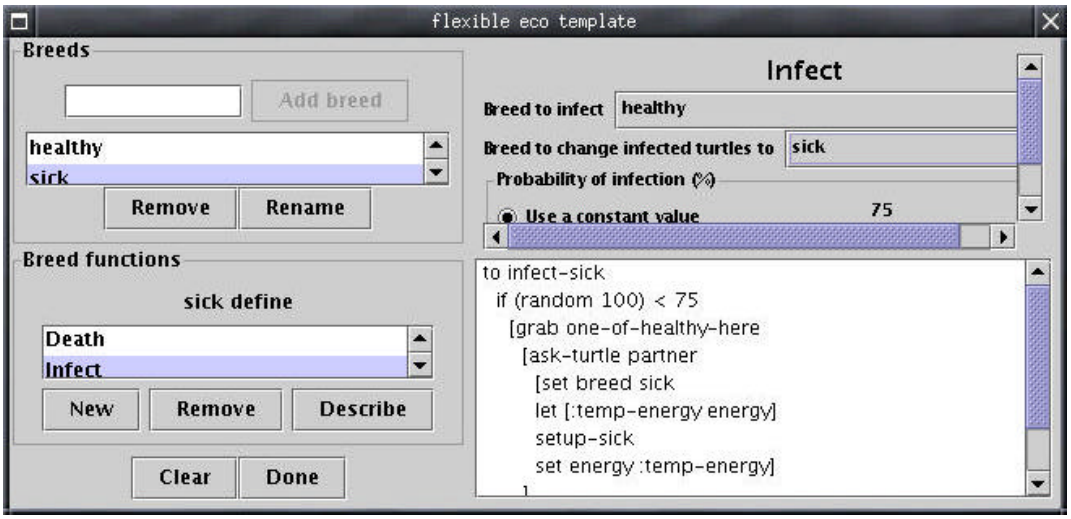

Figure 3.5: *Starlogo*'s Ecology Template.

corresponding to all of the syntactic structures found in most text-based languages, such as conditionals, arithmetic operators, loops, and function calls. Additionally, both languages feature a visual interface for creating data-types. *Sanscript* is a visual-procedural language and, in addition to the standard set of types, supports record types, analogous to records in Pascal or structs in C. *Prograph* is object-oriented and supports the creation of classes and their organization into a hierarchy.

Starlogo is a visual programming tool that focuses on simulations involving multiple independent agents that interact with each other and their environment. *Starlogo* is fundamentally a text-based language. Agent-control programs can be written entirely in text, but at runtime, the agents and their environment are represented visually. The user can also create GUI widgets, such as buttons and sliders, capable of giving the agents commands or changing variable values dynamically during the program's execution. In addition, *Starlogo* code can be generated using purely visual parameterized templates. For instance, Figure 3.5, taken from [17], shows an instance of the *Ecology* template that creates two breeds of turtles: sick and healthy. The user defines new breeds of turtles using the *Breeds* section of the template dialog. The list of functions that apply to a particular breed can be edited in the *Breed functions* section. These functions refer to the actions that a turtle is capable of performing, such as walking, eating, and dying. The window in the upper-right contains function-specific parameters that allow the user to customize the simulation. The lower right window contains the generated code for the selected function and it is updated dynamically as the user makes changes.

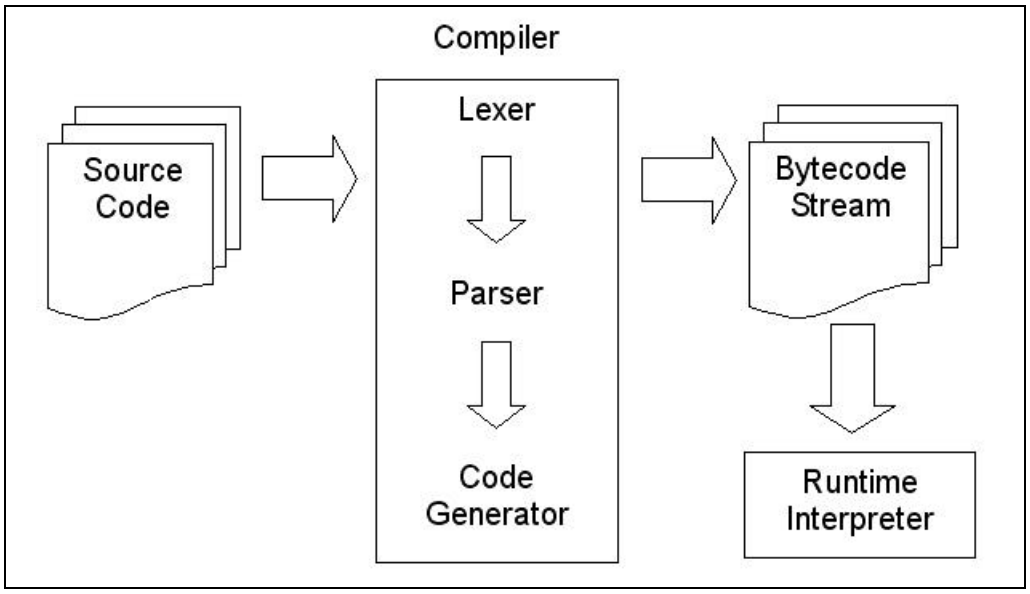

Figure 3.6: Dataflow of a Script Compiler.

3.3 Behavior Specification

All CRPGs, and most other computer game genres, have a set of characters that interact with the player and each other in the game world. In this section, we will mention three interesting behavior-specification mechanisms used in commercial computer games: scripting languages, finite state machines, and force fields.

3.3.1 Scripting Languages

Scripting languages are a common method for specifying behaviors in computer games. We discussed the *Neverwinter Nights* scripting language, *NWScript*, in Section 2.2.3 and the *Morrowind* scripting language in Section 3.1.2. In this section, we will discuss scripting languages in general.

The basic architecture of a scripting language is shown in Figure 3.6 (recreated from [3]). A scripting language consists of two components: the actual language and the scripting engine that interprets it. Source code, written in the scripting language, is compiled into a format interpretable by the scripting engine. The compiled code is called the bytecode stream, and it is formatted in such a way that the scripting engine can decode and execute its instructions quickly. Lexical analysis and parsing of source code can be an expensive process, especially if the language is complicated. By moving the lexer and parser into an offline compiler, the performance hit to the game at run-time is greatly reduced.

A scripting language adds a small amount of overhead to the run-time performance of a game. However, this is offset by several advantages that a scripting language provides as opposed to hard coded behaviors. First, as long as

the interface between the scripting language and the game engine is not changed, the game's designers and developers can work independently. As a result, changes to character behaviors do not require that the entire game source be recompiled; only the relevant scripts. Second, the chance of the game crashing due to erroneous script code is minimized since the code is interpreted and run in a protected environment. In the worst case, the script simply fails and the character does nothing. Non-crashing code is especially important if the designers, which may include home users, are not particularly good programmers. Finally, scripting languages are generally simpler to use than standard programming languages such as C++, making them amenable for use by both game designers and home users.

Several commercial games use an existing general-purpose scripting language, such as *Python* [11][40]. Some of these games include *Blade of Darkness* [46], *Frequency* [52], and *EveOnline* [48] (according to [11]). Using a language like *Python* has two major advantages. First, *Python* includes several advanced features such as complex data types, class inheritance, and garbage collection, all of which have been rigorously tested and used by thousands of users in the past. Since custom scripting languages lack the years of evolution that *Python* has endured, they cannot hope to include these advanced features and be as stable as *Python*. Second, *Python* has an enormous online user-community. There is a vast collection of documentation, tutorials, message forums, code libraries, sample code, and development tools immediately available to script writers.

Custom scripting languages have advantages as well. General-purpose languages, such as *Python*, are typically quite large and may seem more daunting to the home user than a small simple custom language. Another concern is if a general-purpose scripting language is used, the developer is under pressure to interpret the language correctly according to some standard. This may introduce unneeded complexity and performance issues to the game via the runtime script-interpreter. Often, the tasks that are specified using a scripting language do not require the complex features included in general-purpose languages. A simple language with few or no complex features is easier to learn and understand. Additionally, it may be difficult to translate a general-purpose language's bytecode stream into a form recognizable by the game engine. Likely, the general bytecodes are more complex than those used by the game engine. A custom language can generate a game-interpretable bytecode stream directly.

Readers interested in more information on scripting language design for commercial computer games should see [2], [3], [4], [5], [6], [18], and [22].

3.3.2 Finite State Machines

A finite state machine, or FSM, is a simple and elegant behavioral model for controlling character AI. An FSM is composed using the two following features [27]:

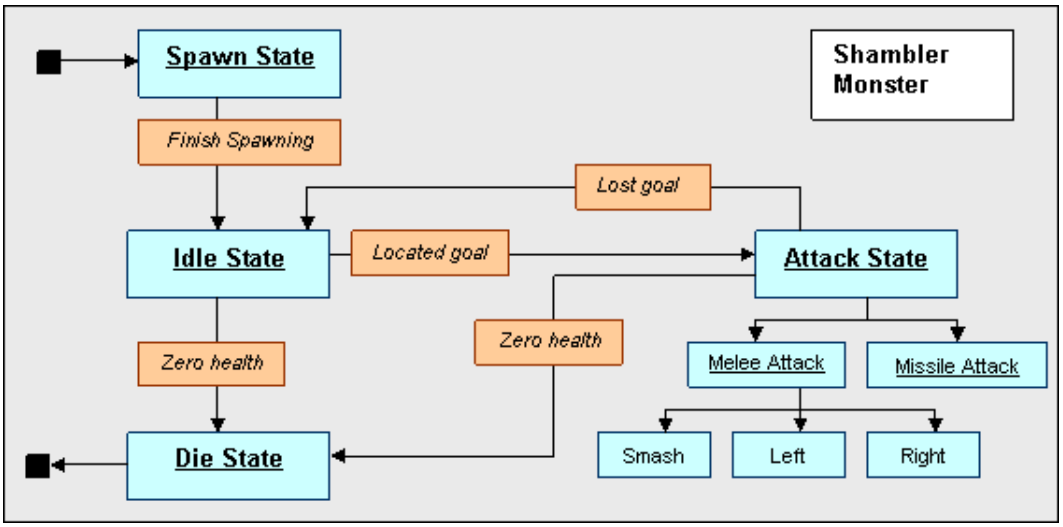

Figure 3.7: An FSM representation of a Shambler monster from *Quake*, taken from [27].

1. **Behavioral States:** Characters occupy a single state at a time. Each state defines a single behavior or set of behaviors that the character can utilize.
2. **State Transitions:** Characters may move from one state to another if there is a state transition defined between those two states. Transitions have rules or conditions attached to them. A state transition occurs when the conditions for that particular transition are met.

Figure 3.7 shows a behavioral FSM for the Shambler monster in *Quake* [58]. This figure was taken from [27]. The small black boxes on the left side of the diagram are the entry and exit points to the FSM. The light blue boxes are the behavioral states of the FSM. The black arrows are state-transition arcs. The orange boxes contain conditions for the state-transition arcs to which they are attached. There are four major states in this example: *Spawn*, *Idle*, *Die*, and *Attack*. Each of these states can have its own mechanism for determining which action the Shambler could perform. This mechanism could even be another FSM, permitting arbitrarily deep nesting of FSMs. The attack state in Figure 3.7 illustrates this. Once in the attack state, the Shambler must decide to perform a melee attack (close range) or a missile attack (long range). The decision could be made based on a random number, a script, another FSM, or any other decision-making mechanism. Similarly, if a melee attack is chosen, the Shambler must then decide to either attack with its left arm, right arm, or smash with both arms.

One disadvantage of FSMs is that they can be predictable, especially if the FSM is small. Adding non-determinism to an FSM model is important when attempting to create realistic and unpredictable behaviors. To this end, a fuzzy logic architecture [1][24] can be applied to FSMs, creating Fuzzy State Machines [27], which add an element of randomness to the state transition arcs.

3.3.3 Force Fields

The disadvantage of FSM-based architectures (fuzzy or otherwise) is that, by definition, a character can only be in one state at a time. Consider our dragon friend, Trogdor the Burninator. Trogdor might have several needs and desires, such as food, rest, treasure, security, and entertainment. In an FSM model, there would be a state for every one of these motivations and Trogdor would be in one of them at a time. For instance, if Trogdor was hungry he might fly over to the largest herd of cattle that he can find, even though it takes him dangerously far away from his lair, where some brave adventurer could sneak in and steal his treasure. If we wanted Trogdor to consider multiple needs when choosing his actions, the FSM would have to grow exponentially to handle every possible combination of needs.

Force fields are a behavioral model used in simulation games like *The Sims* [13][60], and researched in sports games such as *FIFA Soccer* [23][49]. In a force field architecture, characters have a set of drives, like Trogdor's needs listed earlier. Each object that has an effect on a drive generates a force field attracting or repelling the character. For instance, food sources would generate fields attracting Trogdor to them, and renowned dragon-slaying heroes would generate repulsive force fields. The strength of a given force field is determined by the urgency of its associated drive and the distance of the force field from the character. All of the force fields influence the character's actions. The action taken is determined by the sum of all of these influences. Referring to Trogdor again, he may find a smaller herd of cattle closer to his lair, satisfying both his hunger and security needs simultaneously.

Another advantage of force fields is that they are well suited to machine learning. When something undesirable happens, such as Trogdor's treasure being stolen or a goal being scored in a soccer game, the urgency levels of the force fields can be adjusted in order to prevent the same thing from happening again. Learning is particularly useful in sports games, which often have *sweet spots* or a series of actions that the player can perform that scores a goal almost every time. Jack van Rijswijck has implemented a force field AI system in a research version of EA Sports' FIFA Soccer 2002 [23].

3.4 Automated Code Generation

A great deal of research has been done in automated code generation, primarily with generative design patterns. In this section, we will mention two automated code generation techniques: generative design patterns and generative questionnaires. We will also mention a novel generative learning technique.

3.4.1 Generative Design Patterns

Gamma *et al.* [12] define design patterns as *"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."* A design pattern is a high-level description of a solution to a family of recurring design problems. The implementation of the solution is left to the user, since only they understand the specific variant of the problem. To reiterate, design patterns do not provide specific implementations of their solutions, since their solutions require application-specific information.

Generative design patterns are an attempt to generate framework code for a specific solution from the family of problems, based on specification parameters provided by the user [15]. MacDonald *et al.* [15] discuss why design patterns are not generally used as generative constructs that support code reuse, and give three major reasons. First, it is difficult to generate a single body of code that adequately solves each problem in the family of a particular design pattern. Second, defining and modifying generative patterns is hard. Finally, the lack of a tool-independent representation for generative design patterns hinders the creation of a shared-repository that would make patterns more available to both users and pattern developers. MacDonald *et al.* present an approach to solving these problems using their tools, *CO_2P_2S* [14] and *Meta-CO_2P_2S* [7]. *CO_2P_2S* is a visual tool that allows the user to easily specify the various options of a particular pattern, as well as insert application-specific code. *CO_2P_2S* then generates object-oriented framework code based on the options selected. *Meta-CO_2P_2S* is a visual tool that allows users to create new generative design patterns. Budinsky *et al.* [8] have developed a web-based tool that generates code for design patterns. The tool generates code based on implementation trade-offs selected by the user. The tool also provides full documentation for each pattern through a series of cross-referenced html pages. For the reader interested in generative programming in general, Czarnecki and Eisenecker [10] have written a comprehensive book on the subject.

3.4.2 Generative Questionnaires

We mentioned questionnaires earlier, in Section 3.1.1, with regard to the game toolset for *Unlimited Adventures*. A generative questionnaire presents the user with a series of questions about what they would like their program to do and then automatically generates code based on the answers. As opposed to generative design patterns, generative questionnaires do not generate frameworks, but instead generate fully functional stand-alone code that requires no extension by the user. Questionnaires have this capability because they are more restrictive, than generative design patterns, in the power that they give to the user. This restriction is acceptable (and sometimes preferable) in behavior-specification tools for commercial computer games, since the majority of users for such tools are non-programmers. *Unlimited Adventures*, the *Aurora Toolset's* plot wizard,

Starcraft's [61] trigger editor, and *Lilac Soul's NWN Script Generator* [31][32] are examples of such tools.

3.4.3 A Generative Learning Technique

Pieter Spronck *et al.* [19] have devised a novel learning technique for combat AI in CRPGs. Their approach involves giving a character a set of weighted hand-scripted if-then-else rules such as *"If I'm critically wounded then drink a healing potion"*, *"If an enemy is a magic user then cast a deafening spell on them"*, or *"If an enemy is within 5 feet of me then attack them"*. The learning algorithm extracts a subset of these rules, based on their weights, and generates a combat script for the character using the selected rules. After a battle, the weight of each rule used by the character is modified depending on how well they performed. Spronck *et al.* have used *Neverwinter Nights* [56] and a custom-built simulation environment that resembles *Baldur's Gate 2* [44] as testbeds for their research.

3.4.4 Summary

Automatic code generators must make a choice between power and ease-of-use. Generative design patterns are quite powerful, since the user is required to submit application-specific code. These tools make the correct implementation of design patterns easier for programmers, but do little to allow non-programmers to benefit from design patterns. On the other hand, generative questionnaires are easy and accessible to non-programmers due to their purely visual interfaces, but lack in power for the same reason. Our tool, *ScriptEase*, aspires to be both powerful and easy to use by non-programmers, in the specific domain of CRPG scripting. We present *ScriptEase* in Chapter 4.

Chapter 4

ScriptEase

4.1 Introduction

4.1.1 Goals

In many ways, a CRPG is like a movie. It has a cast, sets, props, special effects, a script, etc. All of these elements are important when designing a CRPG. We would like to provide all of these facilities in an easy-to-use visual environment. Fortunately, much of the work has already been done. Casting characters, designing sets, and placing props is easily accomplished using existing CRPG toolsets such as *Aurora*, presented in Section 2.2.2, and *TESCS*, presented in Section 3.1.2. Therefore, our focus is on the game's script.

Unlike a movie script, a CRPG script is typically non-linear and always interactive. The dilemma faced by designers is that the player's character is unscripted and therefore unpredictable. Designers must be careful not to allow any player actions to break the plot of the game. To account for this, a CRPG script is divided into a list of rules. Each rule defines how a character, prop, or the environment reacts to a particular action by the player. For instance, a rule associated with a bartender might indicate the following: *When a player character enters my tavern, greet them and offer them a tankard of ale*. In order to support a very large set of complex rules, state-of-the-art CRPGs utilize text-based scripting languages. Not only are these languages hard to learn and use, they are also difficult to read and understand. A movie script can be written in plain English, and read easily. We would like to represent a CRPG script in a similarly easy-to-write and easy-to-read format.

In Section 2.1.4, we discussed four design challenges in computer role-playing games that offered good research opportunities: behavior specification, conversations, plots, and encounters. All of these challenges are related to the game's script. The three primary goals of our research are the following.

1. To provide a powerful and easy-to-use visual alternative to a text-based scripting language for CRPGs that requires no programming knowledge on the user's part.
2. To provide special support for specifying encounters using patterns.
3. To serve as a basis for future work in specifying behaviors, conversations, and plots.

In this chapter, we give a detailed presentation of our research and the visual scripting tool that we developed, *ScriptEase*. The current version of *ScriptEase* is specific to the CRPG *Neverwinter Nights* and its toolset, *Aurora*.

4.1.2 Overview

Since the *Neverwinter Nights* scripting language, *NWScript*, is text-based, it is very difficult for non-programmers to use, such as game designers and home users. *ScriptEase* provides a visual scripting alternative to *NWScript* that requires no programming knowledge to use. To reiterate, the average *ScriptEase* user never writes code. We presented our first prototype version of *ScriptEase* in [15], and since then, it has evolved into a powerful and practical scripting tool.

Users specify information in *ScriptEase* using easy-to-understand questionnaires whose answers are specified using context sensitive GUI components, such as lists, menus, and trees. Once specified, the information is presented back to the user in easy-to-read English sentences.

In order to provide users with a degree of power close to that of a text-based scripting language, *ScriptEase* works on multiple levels of abstraction. At the highest level, users instantiate high-level reusable patterns. At the second level, new patterns can be created using a tool called the *Pattern Builder*. Neither of these two levels of abstraction requires any coding by the user. The third level of abstraction allows new fundamental scripting components, called *Atoms*, to be created using the *Atom Builder*. At this level, some coding is required. An atom's designer must enter the code that the atom generates. With a large enough base of atoms, the average user will never use the atom builder. However, those users that are capable of programming in text-based languages can create new atoms. Additionally, non-programming users can post requests for new atoms in public Web-based forums where users capable of programming can fulfill the request, taking the burden off our shoulders. The *Pattern Builder* and *Atom Builder* facilitate extension in *ScriptEase*, alleviating the lack-of-power problems that other visual scripting tools, such as *Unlimited Adventures* (discussed in Section 3.1.1), suffer from. Finally, ScriptEase automatically generates *NWScript* code – the lowest level of abstraction – based on the information provided by the user at the higher levels. The generated code is automatically documented using the same English language descriptions used to present information to the user. For the most part, we expect users will never look at the generated code, but it is comprehendible should they decide to.

Ideally, *ScriptEase* would be integrated into the *Aurora Toolset* in order to provide a seamless interface between the two. We did not have access to *Aurora*'s source code, so total integration was not feasible. Fortunately, the *Neverwinter Nights* file-formats [25][41] and a stand-alone command-line *NWScript* compiler are publicly available. Therefore, *ScriptEase* can load data from the preexisting

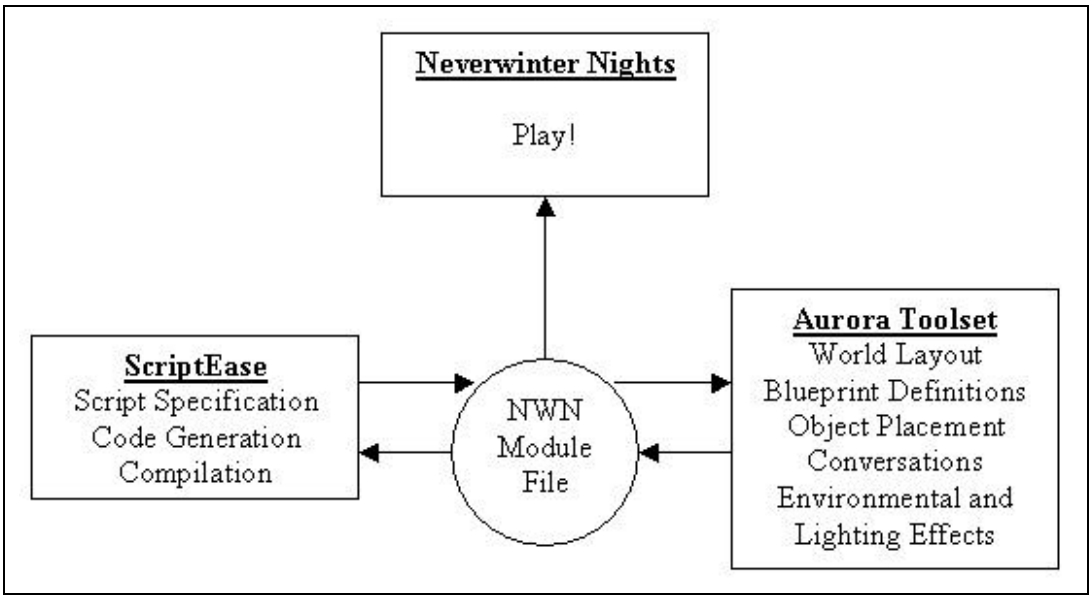

Figure 4.1: A dataflow diagram illustrating the interactions between the *Neverwinter Nights* game engine, the *Aurora Toolset*, and *ScriptEase*.

module file, have the user specify scripts based on that information, generate *NWScript* code, compile it using the stand-alone compiler, and then insert the compiled scripts back into the module file. Figure 4.1 illustrates how *ScriptEase* interacts with the *Aurora Toolset* and the *Neverwinter Nights* game engine. The *Aurora Toolset* is used to create the module file, define object blueprints,[1] layout the world, place objects, and do everything else unrelated to the scripting language. *ScriptEase* handles all of the tasks normally done by *NWScript*. It is used to specify scripts in a purely visual environment, and then it automatically generates *NWScript* code, compiles it, and inserts it back into the module file. The *Neverwinter Nights* game engine loads the module file and executes it.

In Section 4.2, we present the primary building blocks of *ScriptEase*, called *Situations*. Section 4.3 describes *ScriptEase*'s type system. In Section 4.4, we discuss *Patterns* and how they are represented and used in *ScriptEase*. In Section 4.5, we describe *Atoms*, which define the various kinds of situation components used in *ScriptEase*. In Section 4.6, we discuss how *ScriptEase*'s code generation works. In Section 4.7, we offer an assessment of *ScriptEase* based on internal usage and a modest usability review. Finally, in Section 4.8, we summarize the chapter.

4.2 Situations

The primary building blocks in *ScriptEase* are called *Situations*. The components of a situation are an *Event*, a set of *Entities*, a set of *Conditions*, and a set of

[1] Object blueprints are descriptions of game objects that can be instantiated and placed in the world. We will revisit them in Section 4.3.

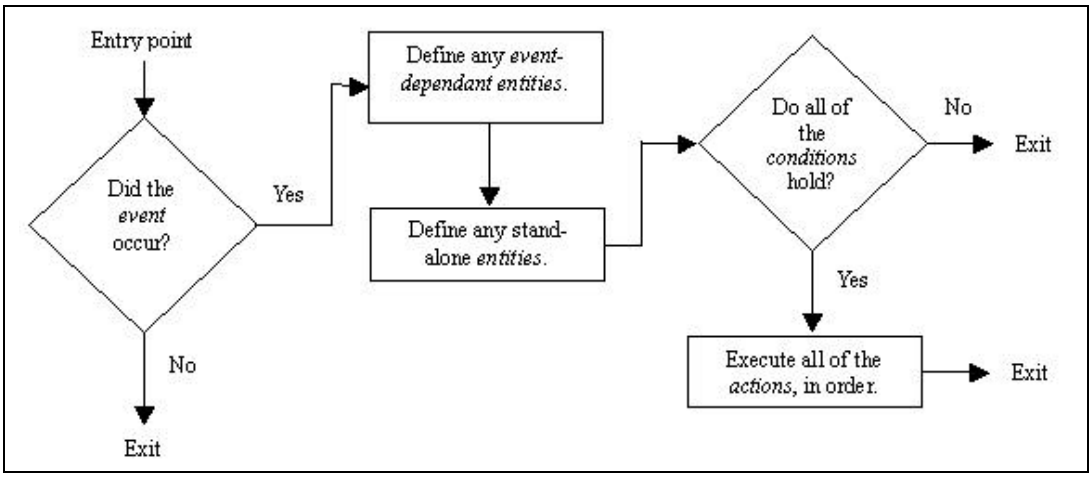

Figure 4.2: The control flow of a *ScriptEase* situation.

Event: When an item is removed from the Aurora-defined placeable called *Pedestal*.

Event-Implied Entities:
Define *Disturbed Item* as the item that was removed from the *Pedestal*.
Define *Disturber* as the creature that removed the item from the *Pedestal*.

Other Entites:
Define *Helm* as the headgear that *Disturber* is currently wearing.

Conditions:
If the *Disturber* is a player character, and
If the *Disturbed Item* is the Aurora-defined item called *Gem of Power*, and
If *Helm* is not the Aurora-defined item called *Wormskull*

Actions:
Define *Pedestal's Location* as the location of the *Pedestal*, then
Destroy the *Pedestal*, then
Show a visual effect called *Harm* at *Pedestal's Location*, then
Create an Aurora-defined creature called *Zombie* at *Pedestal's Location*, hereinafter referred to as *Larry the Zombie*, then
Have *Larry the Zombie* attack the *Disturber*.

Figure 4.3: An example situation, formatted to match its representation in *ScriptEase*.

Actions. The flowchart in Figure 4.2 illustrates how these components are composed to form a situation. A situation reads: when the *Event* occurs, if all of the Conditions hold then, execute the *Actions*. *Entities* are analogous to accessor functions that assign values to variables in a text-based language, and once defined, are used by the conditions and actions of the situation.

This section is split into two parts. In the first part, Section 4.2.1, we discuss each of the components of a situation and how they are presented by *ScriptEase*. While reading Section 4.2.1, the reader will notice that we do not discuss how to specify or edit any of this information. We first want to demonstrate how situations "look" in *ScriptEase*. In the second part, Section 4.2.2, we will describe how situations are modified by creating, removing, and editing components.

We will use the following example to describe the different components of a situation and how the user specifies them in *ScriptEase*. We will assume that blueprints for a placeable called *Pedestal*, a creature called *Zombie*, and items called *Wormskull* and *Gem of Power* have been defined in the *Aurora Toolset*. Our example situation reads: When a player character removes the *Gem of Power* from the *Pedestal*, the *Pedestal* transforms into a *Zombie* that attacks the player character, unless the player character is wearing the helmet *Wormskull*. This situation is typical of those found in CRPGs. Figure 4.3 describes this situation in the terms it components, corresponding to its representation in *ScriptEase*.

4.2.1 Presentation

This section demonstrates how situations are viewed in *ScriptEase*. In the next section, we will describe how information is entered into the situations. Figures 4.4 through 4.9 are screenshots of the pages from *ScriptEase*'s situation questionnaire. The user starts with the *Label page* shown in Figure 4.4. The purpose of this panel is to give the situation a name so that it can be referred to later. In this example, we name our situation *Pedestal-Zombie Transformation*. The user can navigate to the other panels using the *Next* and *Back* buttons shown along the bottom of the panel. Alternatively, the user can choose which panel they wish to examine by selecting it from the list on the left. Next is the *Notes page*, shown in Figure 4.5, which allows the user to include important instructions or documentation regarding this situation. The remaining panels are more complex than the first two, and will require more detailed discussions.

The *Event page* is shown in Figure 4.6. An event is an occurrence within the game that prompts a situation to execute. In our example, the type of event is when an item is added or removed from a container. In order to complete this event, two parameters need to be specified: the container, and the inventory disturbance type (whether the item was added or removed). In Figure 4.6, these parameters have already been set. The event line reads, *"When an item is removed from Pedestal"*. All situation components are presented in plain easy-to-understand English sentences. It is important to note that these sentences are not typed in by the user. They are generated by *ScriptEase* based on menu-driven information provided by the user, which we discuss in Section 4.2.2. The coloured words correspond to the component's parameters. The word *Pedestal* represents the container parameter. It is shown in purple to indicate that it is a game-object defined within the *Aurora Toolset*. The phrase *removed from*

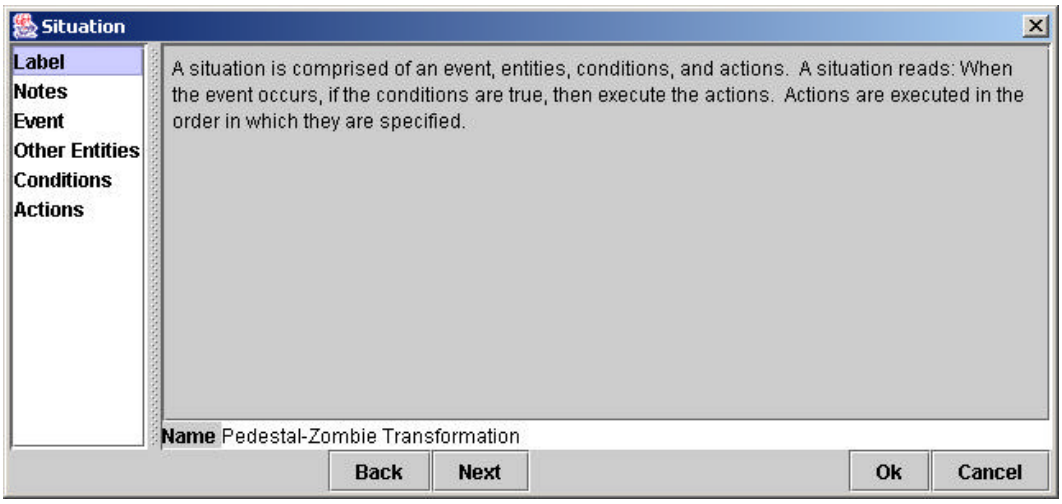

Figure 4.4: The *Label page* of a situation questionnaire.

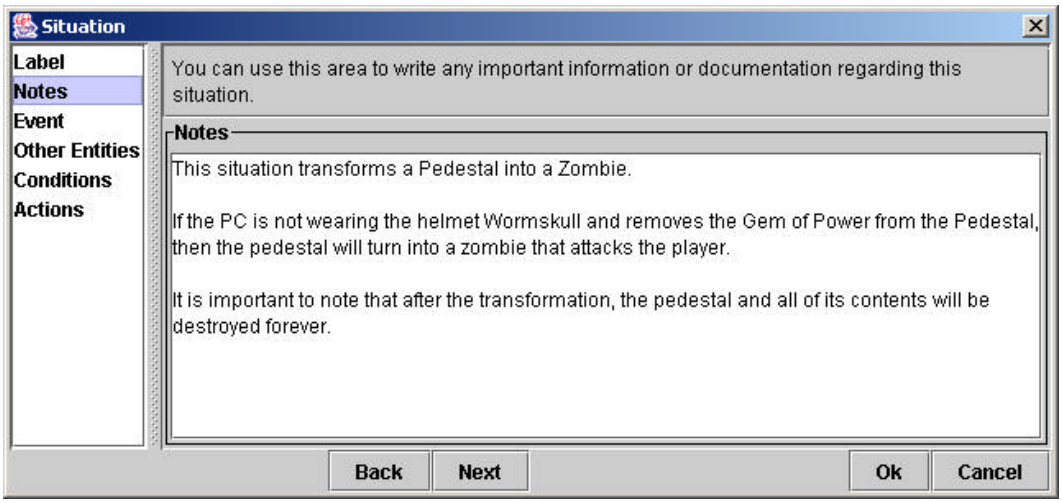

Figure 4.5: The *Notes page* of a situation questionnaire.

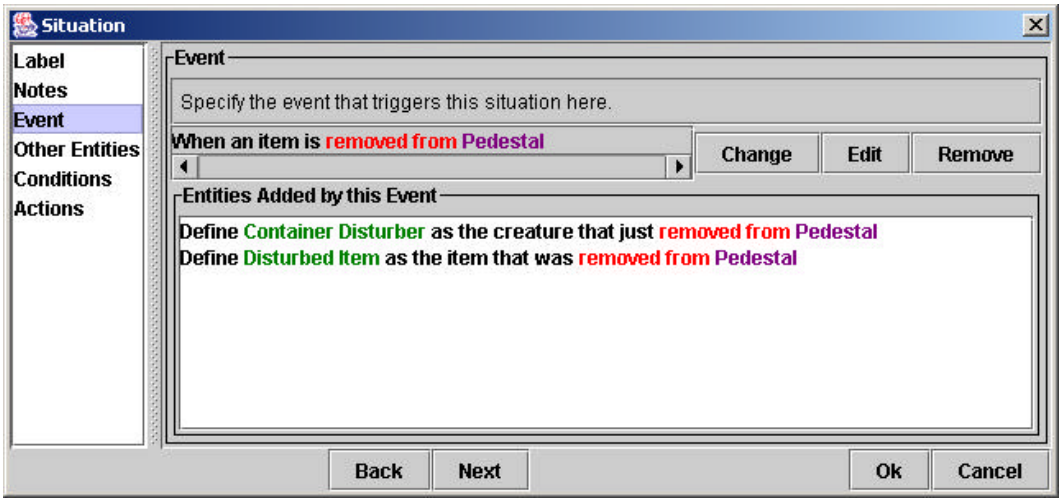

Figure 4.6: The *Event page* of a situation questionnaire.

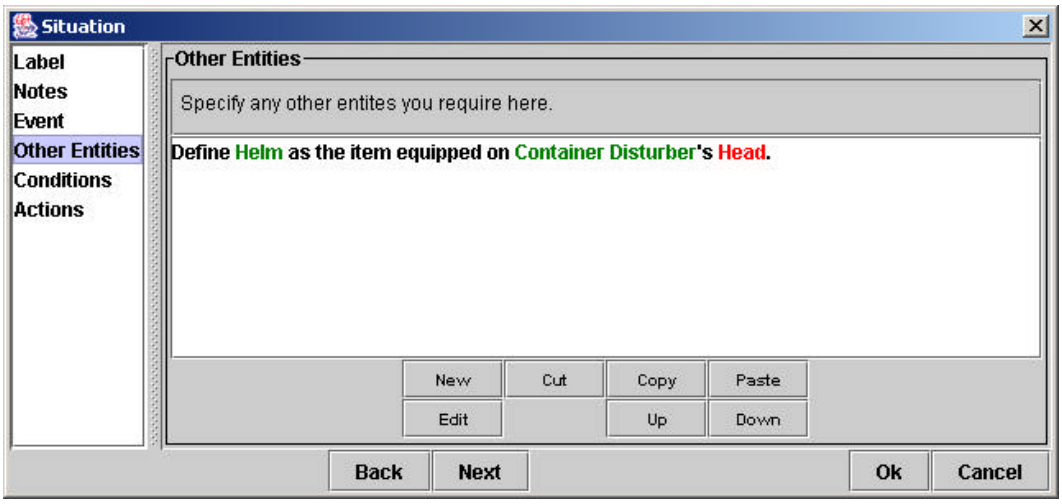

Figure 4.7: The *Entities page* of a situation questionnaire.

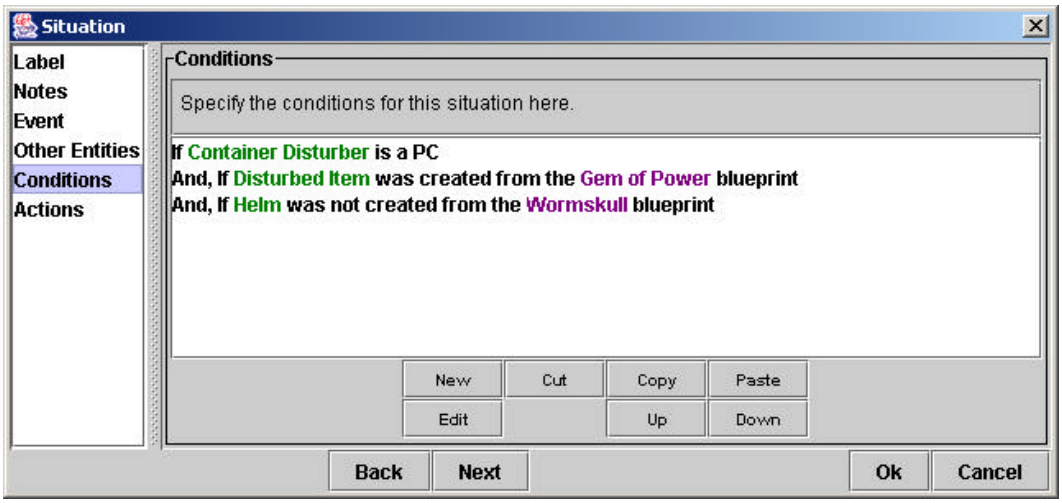

Figure 4.8: The *Conditions page* of a situation questionnaire.

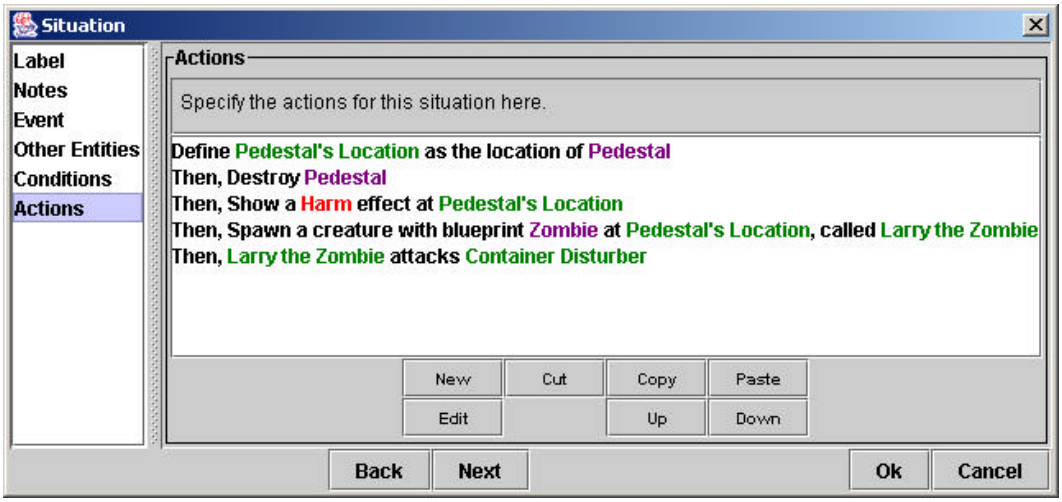

Figure 4.9: The *Actions page* of a situation questionnaire.

represents the disturbance type parameter. It is shown in red to indicate that it is a constant value. In fact, *removed from* is an element of an enumeration type that lists all of the possible item-container disturbances. We discuss enumerations in Section 4.3. In addition to the event itself, there are certain *Entities* that are implicitly defined, based on the event. In this example, the event-implied entities are the creature that disturbed the container's inventory and the item that was disturbed. They are labeled *Container Disturber* and *Disturbed Item* respectively and are listed in Figure 4.6 under the *Entities Added by this Event* heading. Entities (event-implied or stand-alone) are coloured green. These entities are available to the user when they specify conditions, actions, and other entities later. In the next section, we will explain how new events are created and edited.

In addition to the entities implicitly defined by the event, the user can define stand-alone entities, independent of the event. Figure 4.7 shows the *Entities page* of the situation questionnaire. For our example, one additional entity, *Helm*, is defined as the item that *Container Disturber* is wearing on their head. Note that this entity uses the previously defined *Container Disturber* entity in its definition. The word *Head* in the entity's description is coloured red, indicating that it is a constant value. The newly created *Helm* entity can now be used when specifying conditions, actions, or more entities. It is also worth noting that all entities have a type associated with them. The type of the *Helm* entity in our example is *Item*, which means that it is an object capable of being possessed by a creature or placeable. We discuss *ScriptEase*'s type system in detail in Section 4.3. The mechanism used to add and edit entities is presented in the next section.

Figure 4.8 shows the three conditions of our example within the *Conditions page* of *ScriptEase*'s situation questionnaire. The first ensures that the *Container Disturber* is a player character. The second ensures that the *Disturbed Item* is an instance of the *Gem of Power* blueprint. This condition is necessary since the *Pedestal* may contain several items other than the *Gem of Power*, all of which the player should be able to freely manipulate since they are unrelated to this situation. The third condition ensures that the *Container Disturber* is not wearing the helmet *Wormskull*. *Helm* is coloured green because it is an entity. *Gem of Power* and *Wormskull* are coloured purple since they are *Aurora*-defined blueprints.

Figure 4.9 shows the actions from our example within the situation questionnaire's *Actions page*. Actions are executed in order, and only if all of the conditions hold. There are two kinds of actions: entity definitions and game actions. Game actions affect the game world in some way, whereas entity definitions simply access the game's state without changing anything. Therefore, any entity that can be defined in the *Entities page* shown in Figure 4.7 can also be defined in the *Actions page*. Later, we will discuss the reasons for defining entities in different places. The first action in Figure 4.9 defines an entity called *Pedestal's Location*, which retrieves the location of the *Pedestal*. The next action destroys the *Pedestal* (removes it from the game world). The third action displays

a visual effect called *Harm* at the *Pedestal's Location*. This visual effect is purely cosmetic and makes the pedestal-zombie transformation look cool. The fourth action creates a creature using the *Zombie* blueprint as defined in the *Aurora Toolset*. The instance of the *Zombie* blueprint is created at the *Pedestal's Location* and is referred to as *Larry the Zombie*. Finally, the last action instructs *Larry the Zombie* to attack the *Container Disturber*. It should be noted that game actions are capable of defining entities. For instance, the fourth action defines the entity *Larry the Zombie*, and is a game action since it affects the game world by creating a creature.

Entities can be defined in three different places within the situation questionnaire. They can be defined in the event-dependant entities section of the *Event page*, in the *Entities page*, and in the *Actions page*. Entities are available when specifying the parameters of a situation's other components. Entities defined in the *Event page* are separated from the rest since they are dependent on the event and cannot be modified by the user. Event-implied entities are available to be used by every component of the situation except for the event. Entities defined in the *Entities page* are available to all of the conditions and actions of the situation. They are also available to any entity in the *Entities page* that is defined lower in the list. In other words, an entity has access to all of the other entities defined above it, but none of those below it in the list. This visibility issue can cause problems when entities are reordered after being defined. We discuss these problems in Chapter 5. The entities defined in the *Actions page* can only be accessed by the entities and actions defined below them in the action list. The visibility of the entity to the other situation components, influences the designers decision on which list to define the entity in.

4.2.2 Modification

In this section, we explain how situation components are created, removed, and edited in *ScriptEase*. A situation has a single event, which can be changed, removed, or edited using the buttons labeled *Change*, *Remove*, and *Edit* respectively, shown in Figure 4.6. A situation can contain multiple entities, conditions, and actions. These components can be manipulated using the inner button panel shown in Figures 4.7, 4.8, and 4.9. The *New* button creates a new component and adds it to the end of the list. The *Cut*, *Copy*, and *Paste* buttons offer the expected functionality. These functions are local to the page to which they belong. For instance, an entry in the list of conditions cannot be cut and then pasted into the list of actions. The *Up* and *Down* buttons allow the selected entry to be moved in the list. Finally, in order to edit one of these components, the user clicks the *Edit* button or double-clicks on the appropriate list entry. Note that the event-implied entities listed in the event page (Figure 4.6) have no such buttons. These entities are fixed according to the selected event and cannot be modified by the user.

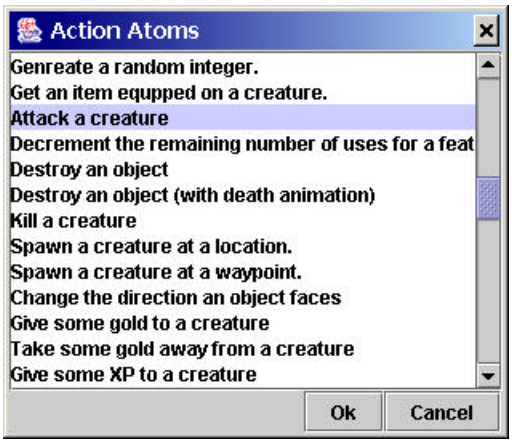

Figure 4.10: A list of possible actions that can be added to a situation.

When the user elects to create a new component, they are presented with a list of possible events, entities, conditions, or actions, depending on which page they are editing. For instance, if the *New* button is clicked in a situation's *Actions page* (Figure 4.9) the dialog in Figure 4.10 is displayed. This dialog presents a list of all possible actions. After the user selects the action that they wish to add, that action's questionnaire is opened. Events, entities, and conditions are created in the same way. If the user elected to edit an existing component, that component's questionnaire would open directly. There are hundreds of different events, entities, conditions, and actions available in *NWScript*. At present, we have included approximately 200 of these components into *ScriptEase*.

We will use the *Attack a creature* action as an example of how a situation component is edited. This is the selected action in Figure 4.10. It involves having one creature attack another. The questionnaire for this action is shown in Figure 4.11. The first page of the questionnaire gives a short description of the action. The remaining pages correspond to the action's parameters. This example has two parameters, the *Attacker* and the *Attacked*. Figures 4.12 and 4.13 are screenshots of the *Attacker* parameter page, which we will explain shortly. Like entities, parameters have a type associated with them. The type of the *Attacker* parameter is *Creature*, which means that only entities of the same type or creature blueprints defined in the *Aurora Toolset* are valid values for this parameter. Accordingly, the *Attacker* parameter page of the action's questionnaire presents both of these options to the user. Figure 4.12 shows the *Known Creatures* option selected. This option allows the user to select a value for this parameter from all of the valid entities, previously defined within the situation. In Figure 4.12, two such entities exist: *Container Disturber* and *Larry the Zombie*. The user may select either of these values from a drop-down menu. In Figure 4.13, the *Aurora Blueprint* option is selected. The box beside the radio button shows the currently selected blueprint, as well as a button labeled *Change*. When the *Change* button is pressed, *ScriptEase*'s *Blueprint Picker* is brought up, also shown in Figure 4.13. This dialog presents the user with all of the custom blueprints defined in the *Aurora Toolset*, within a tree-type interface corresponding to the one used by *Aurora*. The row of icons displayed along the top of the *Blueprint Picker*

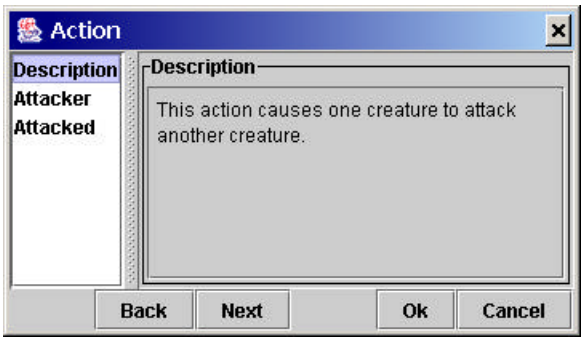

Figure 4.11: The Description page of an *Attack a creature* action's questionnaire.

Figure 4.12: When the *Known Creatures* option is selected, the user can access all of the other creature entities that were previously defined in *ScriptEase*.

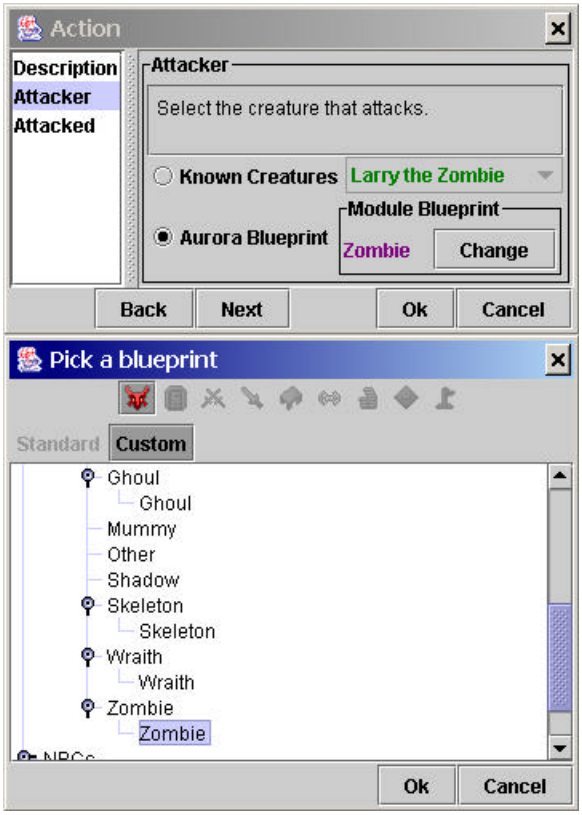

Figure 4.13: When the *Aurora Blueprint* option is selected the user can select any blueprint, defined in the *Aurora Toolset*, by using *ScriptEase*'s *Blueprint Picker*.

Type	Description	Super-Type
Integer	A signed 32-bit integer.	N/A
Float	A 32-bit floating point number.	N/A
String	An arbitrarily long character string.	N/A
Vector	A 3-dimensional positional value.	N/A
Location	A 3-tuple containing an area, a position, and a direction.	N/A
Enumeration	A finite list of values that a variable can take on.	N/A
Area	A 3-D environment that the player's character explores. Areas contain objects with which the player's character can interact.	N/A
Object	Represents anything that can be placed into an area. This includes containers, creatures, placeables, items, doors, perimeters, and waypoints.	N/A
Container	An object capable of holding items. Containers include creatures and placeables.	Object
Creature	An NPC or monster capable of moving about the world and interacting with the player's character.	Container
Placeable	An inanimate prop such as a table, barrel, or tree. While placeables cannot move, they are still capable of interaction with the player's character.	Container
Item	An object that can be held by a container.	Object
Door	A door.	Object
Perimeter	An invisible polygon drawn on the floor.	Object
Waypoint	An object representing a location. Unlike locations, waypoints are capable of interacting with the player.	Object
Blueprint	Contains all of the information required to instantiate a particular object. The creature, placeable, item, door, perimeter, and waypoint types have corresponding blueprint types.	N/A

Figure 4.14: Data types utilized by *ScriptEase* and a brief description of each type.

represent the different object types used by *Aurora*. When the user clicks on one of the icons, the objects with the corresponding type are displayed. The types that are invalid in the context of the given parameter are grayed out, indicating that they are unavailable.

All situation components are edited using questionnaires similar to the one shown in Figures 4.11, 4.12, and 4.13. The questionnaire has a description page and a page for each parameter. The description page for a component that defines an entity asks the user to enter a name for the new entity, in addition to giving a description.

4.3 Types

ScriptEase uses a variety of data types, including the familiar *Integer* and *String* types, as well as some that are not found in traditional programming languages, such as *Creature*, *Door*, and *Location*. A full listing of the types used by *ScriptEase* and their descriptions is given in Figure 4.14. Some of the data types are organized into a hierarchical tree indicated by the *Super-Type* column. Figure 4.15 shows a visual representation of this tree.

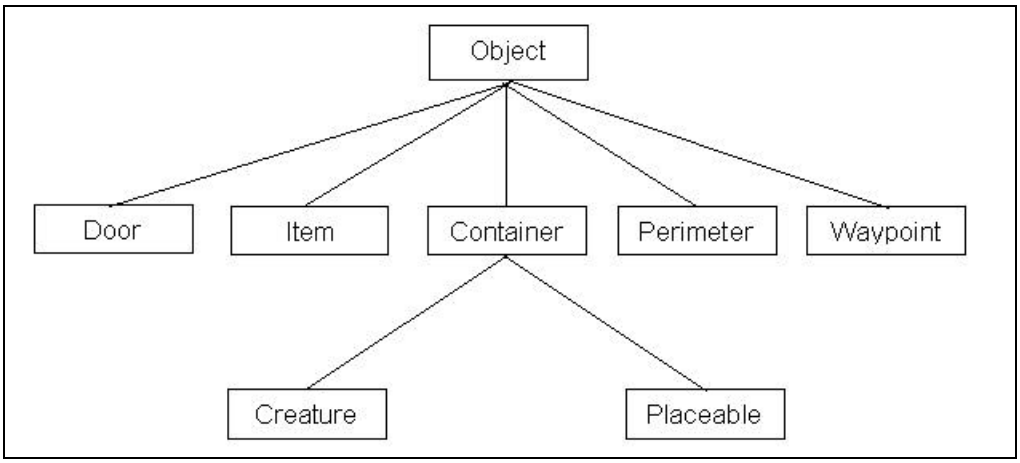

Figure 4.15: This is the object type hierarchy used by *ScriptEase*.

Enumeration Type	Description
Yes / No	This type is appropriate for parameter questions that require an answer of yes or no.
Visual Effect	Visual effects are graphical animations played by the game. There are hundreds of different visual effects. Some include *Lightning Bolt*, which shows a bolt of lighting strike the ground, *Harm*, which displays a pillar of red energy, and *Poison*, which shows a green cloud of mist.
Inventory Slot	Inventory slots are places where a creature can equip items. The possible values of this enumeration are *Head, Neck, Chest, Arms, Left Hand, Right Hand, Left Ring, Right Ring, Waist, Feet, Cloak, Arrows, Bolts,* and *Bullets*.
Inventory Disturbance	An inventory disturbance is a method in which an item can interact with a container's inventory. Its possible values are: *Added to, Removed From,* and *Stolen From.*
Magic Spell	There are over 200 magic spells in *Neverwinter Nights*. Some include *Magic Missile, Haste, Fireball,* and *Meteor Swarm*.

Figure 4.16: Some of the *Enumeration*-types used by *ScriptEase*.

The *Integer*, *Float*, and *String* types are the same as in traditional programming languages. An *Enumeration*-type has a finite list of possible values. There are several different enumeration types used in *ScriptEase*, including those listed in Figure 4.16. A *Vector* is a 3-tuple of floating-point numbers representing a 3-dimensional position in the game world. A *Location* is a combination of a position, a direction, and an *Area*. The position is represented by a *Vector*, and the direction is represented by a floating-point number indicating a directional angle in degrees, where 0^{o} is north. An *Area* represents a closed environment in the game world, such as a section of city streets, a cave, or a room in a castle. *Creatures* move throughout *Areas* and interact with the *Objects* that reside in them.

Instantiations of the types shown in Figure 4.15 have *Locations* and they can be placed inside of *Areas*. The top-level *Object* is a common super-type of the rest. A *Door* is exactly what one would expect – simply a door. An *Item* is an object that can be held in a *Container*'s inventory. A *Perimeter* is a polygonal area

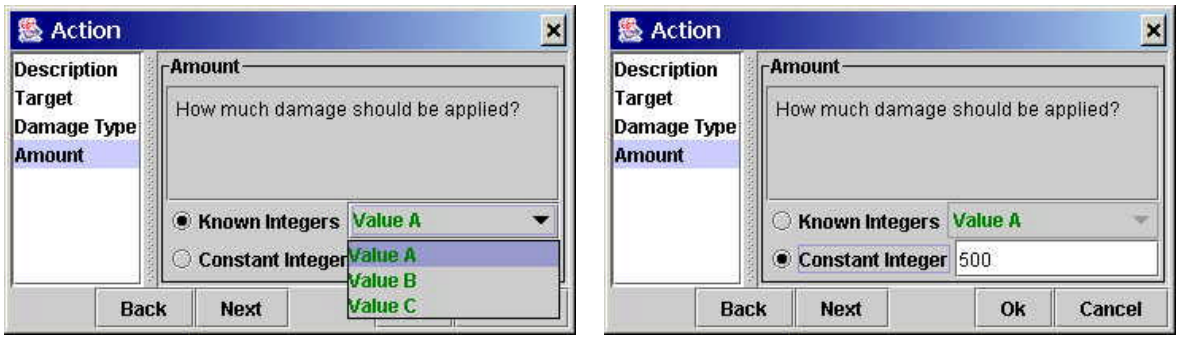

Figure 4.17: An *Integer*-typed parameter's questionnaire page.

drawn on the ground. A *Waypoint* is an object representation of a location and is normally used to specify patrols for *Creatures*. *Waypoints* and *Perimeters* are invisible objects in the game. A *Container* is an object capable of holding *Items.* The *Container* type has two sub-types: *Creature* and *Placeable*. A *Creature* is a player character, NPC, or monster capable of moving throughout an *Area* and interacting with the objects within. A *Placeable* is an inanimate prop such as a tree, barrel, or pedestal. Only *Objects* and *Areas* can have scripts attached to them.

A *Blueprint* is a prototype from which *Objects* can be instantiated. A *Blueprint* contains all of the physical characteristics, properties, attributes, and scripts required to completely specify an *Object*. There are six blueprint-types, one for each of the leaf nodes in the tree shown in Figure 4.15: *Door, Item, Creature, Placeable, Perimeter,* and *Waypoint*.

Entities, *Aurora*-defined objects, and parameters of situation components, all have types. When the user is choosing a value for a parameter, these types serve to restrict the list of possible choices to the compatible entities and *Aurora*-defined objects. For instance, if a parameter required that a *Container* be specified, then the parameter's questionnaire page would present all available *Containers, Creatures*, and *Placeables* to the user. *Creatures* and *Placeables* are included in the possible values since they are sub-types of *Container*, as shown in Figure 4.15. If a parameter's type were one of the non-*Object* types - say *Integer* - then any previously defined *Integer*-entities would still be available. However, instead of providing a GUI widget for selecting *Aurora*-defined objects, the parameter's questionnaire page would instead provide a field for entering a constant integer. Figure 4.17 shows an example of an *Integer* parameter's questionnaire page.

4.4 Patterns

ScriptEase supports two types of patterns, which we present in this section: *Encounter Patterns* and *Action Patterns*.

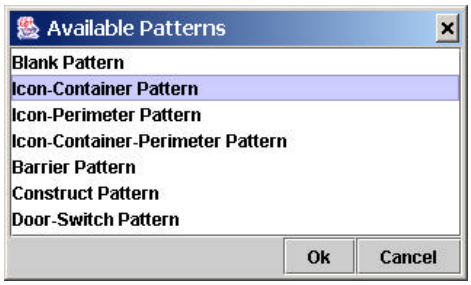

Figure 4.18: A pattern chooser dialog in *ScriptEase*.

4.4.1 Encounter Patterns

CRPGs, *Neverwinter Nights* in particular, contain many similar encounters that can be grouped into families or patterns. We call these patterns *Encounter Patterns* and *ScriptEase* provides special support for instantiating, customizing, and creating them. The structure of an encounter pattern is simple. It has a name, a set of parameters, and a set of situations. The easiest way to demonstrate an encounter pattern is through an example. One very common encounter found in many CRPGs involves something special happening when a particular item is added to or removed from a particular container. We have created an encounter pattern in *ScriptEase* called the *Icon-Container Pattern* to support this type of encounter. This pattern has two parameters. *The Icon* parameter is the item and *The Container* parameter is the container that the icon is to be added to or removed from. The pattern has two situations as well, one that handles adding *The Icon* to *The Container* and another that handles removal.

There is an important distinction between the situations defined inside a pattern and stand-alone situations. Patterns are reusable templates that can be instantiated multiple times in multiple game modules. Patterns and their situations are independent of a particular game module, and as such, they do not reference any specific game objects. However, when a pattern is instantiated, game objects can be bound to the pattern's formal parameters. Specific game objects can also be included in a pattern-instantiated situation when customizing it. In the remainder of this section, we elaborate on these points.

When the user elects to instantiate a pattern, *ScriptEase* presents a list of available patterns, like the one shown in Figure 4.18. The selected pattern in Figure 4.18 is the *Icon-Container Pattern* for our example. Once selected, the pattern's questionnaire is displayed, as shown in Figure 4.19. The first page of a pattern's questionnaire provides a brief description of the pattern and a text-field for the user to name the pattern instance. Following that is a page for each parameter of the pattern. The user specifies parameter values for patterns in the same way as they specify parameter values for situation components (recall Figures 4.12, 4.13, and 4.17). The *Icon-Container Pattern* has two parameters listed on the left of Figure 4.19: *The Icon* and *The Container*. The final page in a pattern's questionnaire contains a list of the pattern's situations. Figure 4.20 shows the *Situations page* for our example pattern. The *Add Icon* situation handles the case

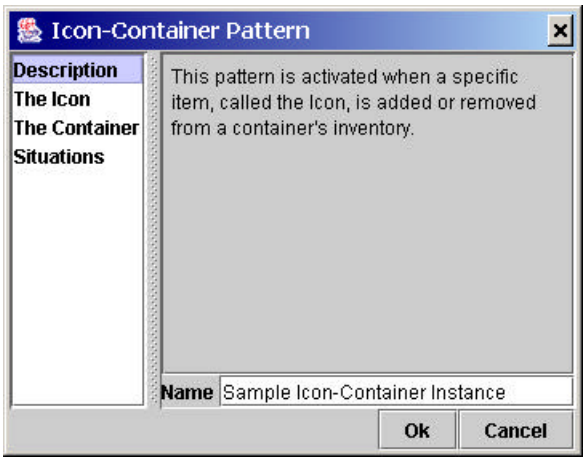

Figure 4.19: The *Icon-Container Pattern*'s questionnaire.

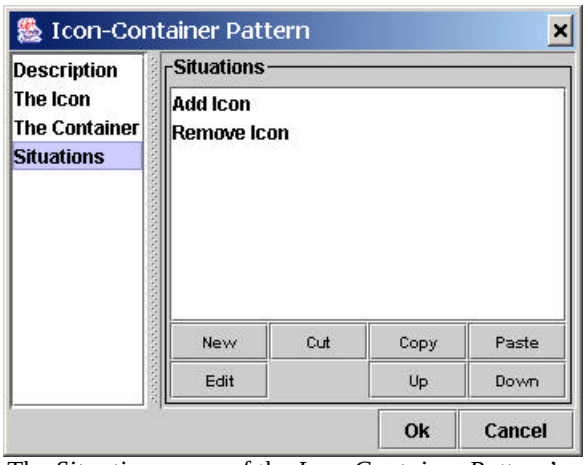

Figure 4.20: The *Situations page* of the *Icon-Container Pattern*'s questionnaire.

where *The Icon* is added to *The Container*. The *Remove Icon* situation handles the opposite case.

Figure 4.21 shows the *Event page* from the *Add Icon* situation's questionnaire. All of the information specified in this panel was automatically added by *ScriptEase* when the pattern was instantiated. The event reads: *When an item is added to The Container*. Notice that *The Container* parameter in both the event description and the entity descriptions is coloured blue. This indicates that *The Container* is a pattern parameter. *The Container* will be replaced with the pattern parameter's user-specified value just before code generation, discussed more in Section 4.6. The *Event page* of the *Remove Icon*'s questionnaire is identical to Figure 4.21 except that the *Added to* constant is replaced with *Removed from*. The *Conditions page*, shown in Figure 4.22, has a condition for ensuring that the *Disturbed Item* is in fact *The Icon*. This condition was also automatically added when the pattern was instantiated. *The Icon*, like *The Container*, is coloured blue to indicate that it is a pattern parameter.

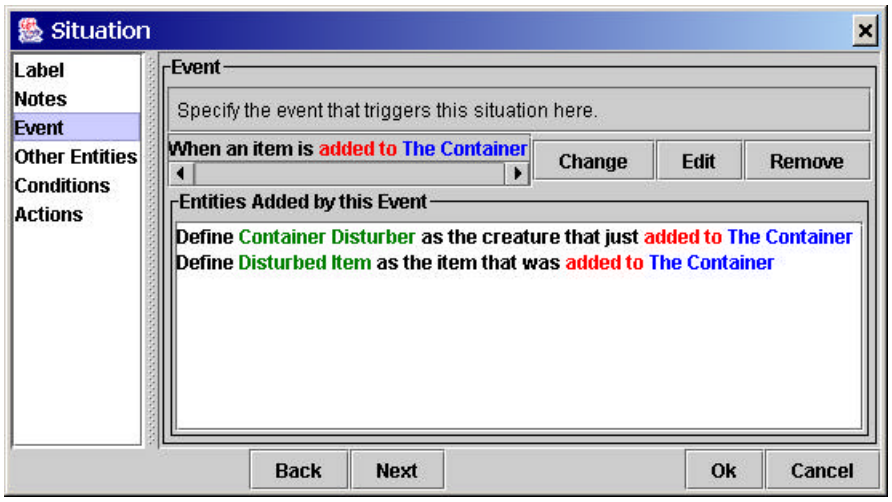

Figure 4.21: The *Event page* of the *Add Icon* situation.

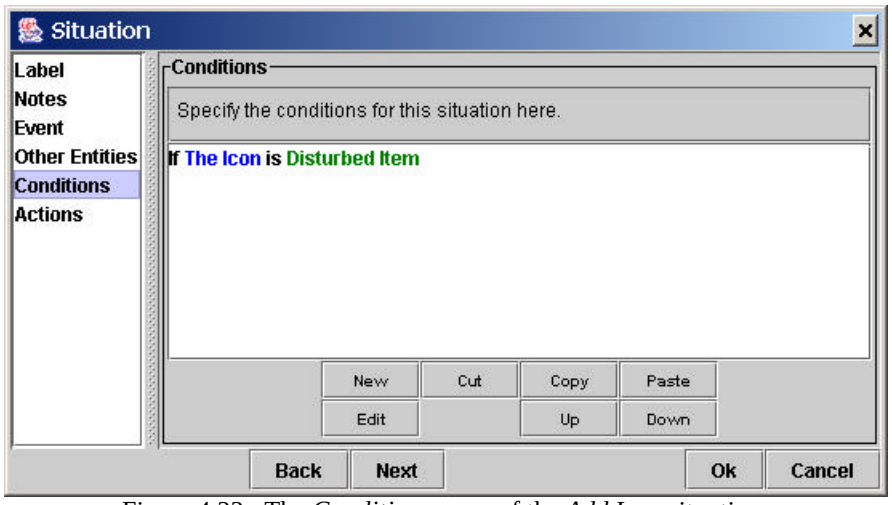

Figure 4.22: The *Conditions page* of the *Add Icon* situation.

Notice that there are no *Aurora*-defined objects in Figure's 4.21 or 4.22 (no purple words). Since patterns are reusable templates, independent of any particular game module, they cannot contain module-specific objects. When a user instantiates a pattern, they specify module-specific objects as parameter values. Parameter specification is the primary method of customizing a pattern instantiation. The secondary method of customizing a pattern instantiation is by modifying its situations, which we discuss next.

The *Entities page* and *Actions page* are left blank in both of the *Icon-Container Pattern*'s situations. Therefore, the situations are incomplete. The user is required to complete these situations by customizing them. Actions, entities, and conditions can be added using the methods discussed in Section 4.2. In fact, the user can even modify the existing pattern-defined situation components, such as the event in Figure 4.21 and the condition in Figure 4.22. Module-specific game

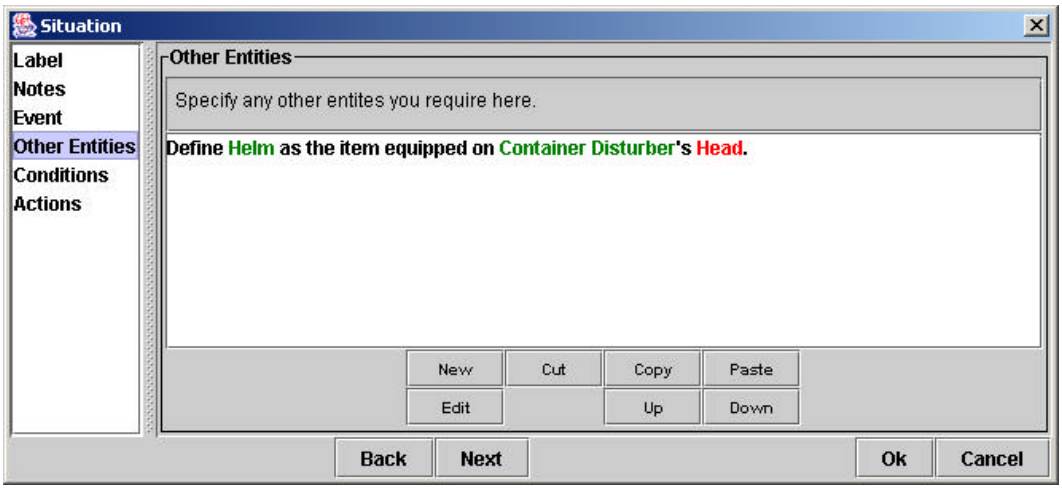

Figure 4.23: The *Entities Page* from the *Add Icon* situation of a customized instance of the *Icon-Container* pattern.

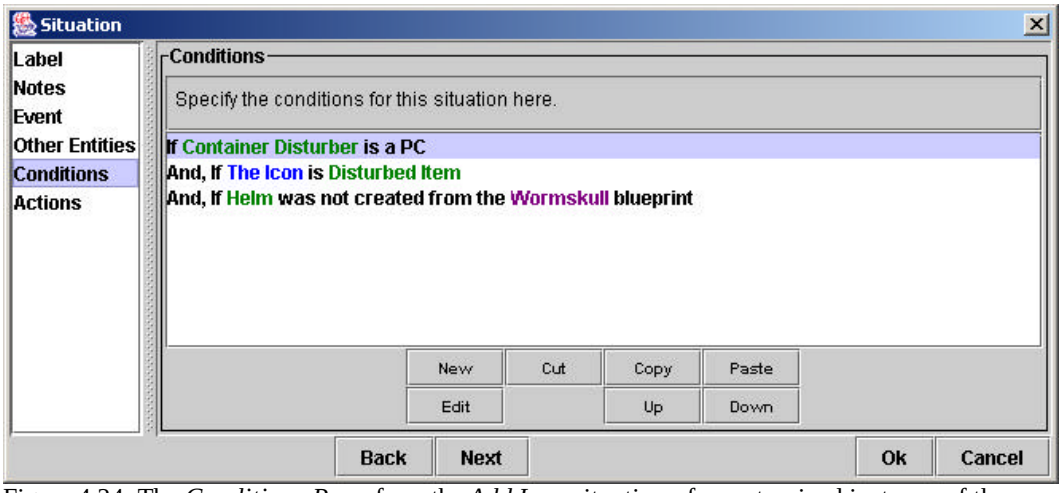

Figure 4.24: The *Conditions Page* from the *Add Icon* situation of a customized instance of the *Icon-Container* pattern.

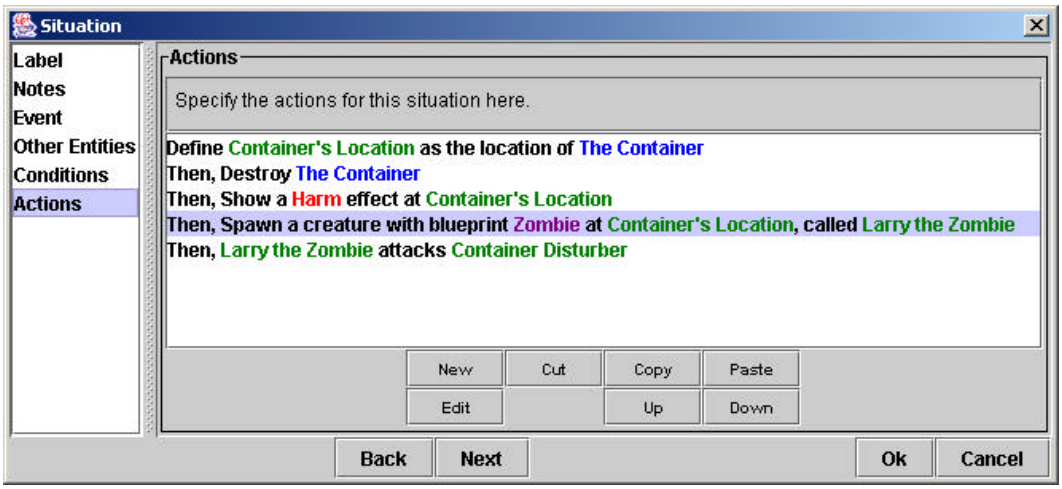

Figure 4.25: The *Actions Page* from the *Add Icon* situation of a customized instance of the *Icon-Container* pattern.

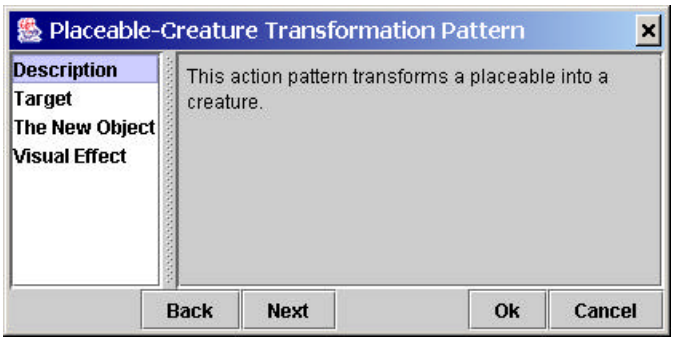

Figure 4.26: The questionnaire for the *Placeable-Creature Transformation* Pattern

objects are valid values for new situation components that are added, since the situations have already been instantiated in the context of a specific game module.

The *Pedestal-Zombie Transformation* situation, described in Section 4.2, could have been created by customizing an instance of the *Icon-Container* pattern. Figures 4.23, 4.24, and 4.25 show the customized *Entities, Conditions, and Actions Pages* of the *Remove Icon* situation from an *Icon-Container* instance. Notice the similarities to the pages shown in Figures 4.7, 4.8, and 4.9. This situation's event (shown in Figure 4.21) and the second condition in Figure 4.24 are components that were automatically added by the pattern. All of the other components were added by the user after the instance was created. The *Pedestal* and the *Gem of Power* from Section 4.2 are replaced with the pattern parameters, *The Container* and *The Icon*. This situation contains references to the game objects, *Wormskull* and *Zombie*, which is allowed since the components to which they belong were added by the user after the pattern was instantiated in the context of a specific game module.

4.4.2 Action Patterns

Encounter patterns involve entire situations and interactions between them. There are other simpler patterns, called *Action Patterns*, which specify a complex action built from simple actions[2]. An *Action Pattern* is composed of a name, a set of parameters, and a set of actions. When composing an action list within a situation, the user may elect to instantiate an action pattern, which will add a series of actions to the action list. We will use the following example to demonstrate. The *Placeable-Creature Transformation Pattern* transforms a placeable object into a creature, by removing the placeable from the game world and creating a creature in its place. Figure 4.26 shows this action pattern's questionnaire. The first page gives a description of the pattern. The remaining pages correspond to the pattern's parameters. This particular pattern has three parameters. The *Target* is the placeable object to be transformed. *The New Object* parameter is a creature blueprint indicating which creature should be created in the *Target*'s place. Lastly, the *Visual Effect* parameter specifies a cosmetic visual effect to show while the transformation is taking place. After the

[2] These simple actions are called *Action Atoms*, which are presented in Section 4.5.

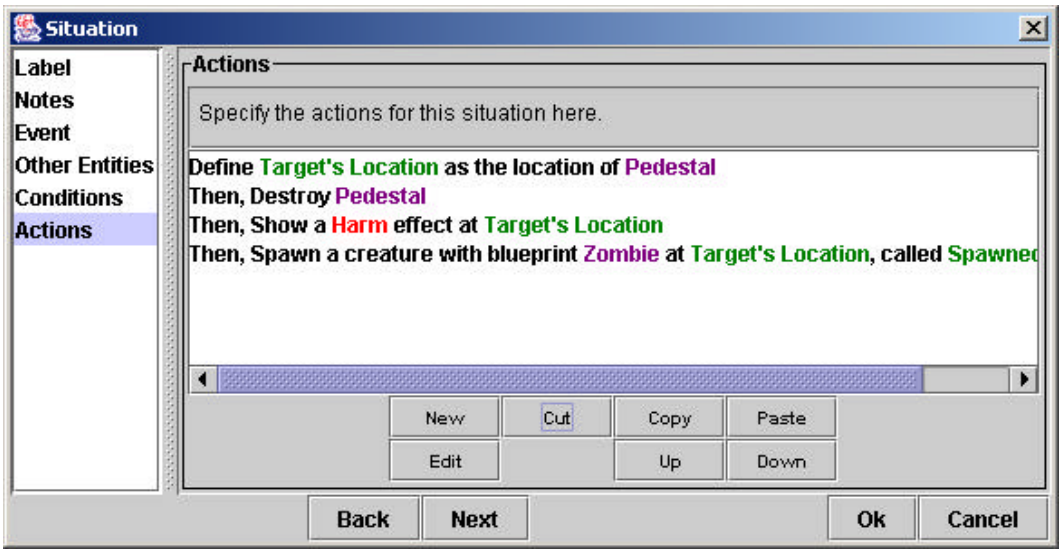

Figure 4.27: The list of actions added to a situation after a *Placeable-Creature Transformation Pattern* was instantiated.

questionnaire is completed, the list of actions associated with this pattern is incorporated into the action list of the situation from which the pattern was instantiated. Figure 4.27 shows the actions generated by the *Placeable-Creature Transformation Pattern*. The parameter values selected by the user in this example are *Pedestal* (the *Target*), *Zombie* (*The New Object*), and *Harm* (the *Visual Effect*). Notice that the added actions are very similar to the first four actions of *Pedestal-Zombie Transformation* example from Section 4.2 shown in Figure 4.9. In fact, the actions from the example in Section 4.2 could have been created using the *Placeable-Creature Transformation* action pattern. Furthermore, the example situation in Section 4.2 could have been constructed by first instantiating an *Icon-Container* pattern, combining it with an instance of the *Placeable-Creature Transformation* pattern, and finally making a few customizations. We discuss pattern combination more in Chapter 5.

Like an *Encounter Pattern*'s situations, an *Action Pattern*'s actions are instantiated from templates, and once they are added to a situation's action list they maintain no reference back to the *Action Pattern* that created them. The user is free to manipulate these actions, once instantiated, in any way they wish.

4.4.3 The Pattern Builder

ScriptEase provides a pattern-creation feature using a separate tool called the *Pattern Builder*. Figure 4.28 shows a screenshot of the *Pattern Builder*'s encounter pattern definition window, with the *Icon-Container Pattern* loaded. The *Name* field along the top is used to specify a name for the pattern. The *Parameters* panel on the upper-left allows the pattern designer to specify the pattern's parameters. The *Situations* panel on the lower-left is used to specify the pattern's situations. Finally, the *Description* panel on the right allows the pattern designer to specify the pattern's description, as seen in Figure 4.19.

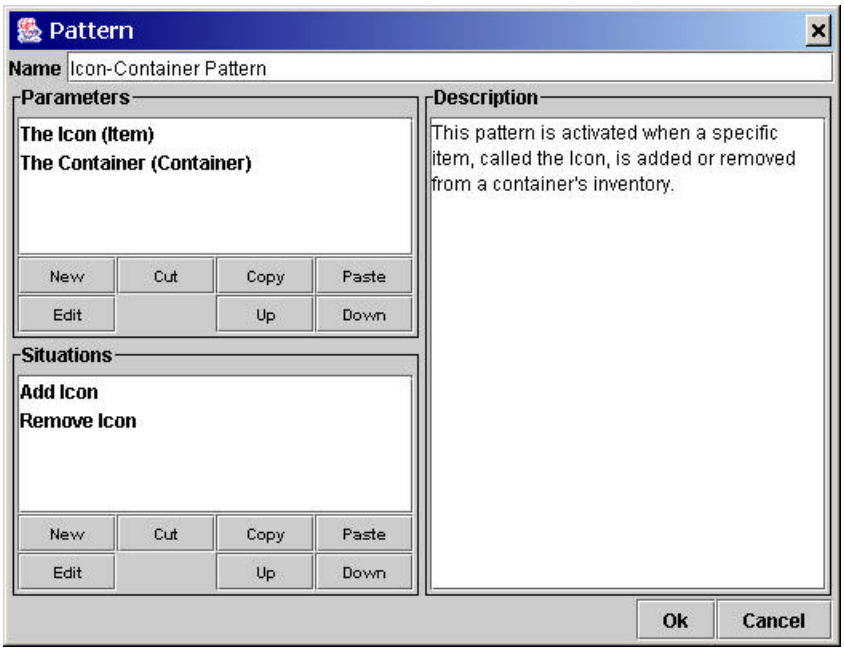

Figure 4.28: The *Encounter Pattern* definition window of *ScriptEase*'s *Pattern Builder*.

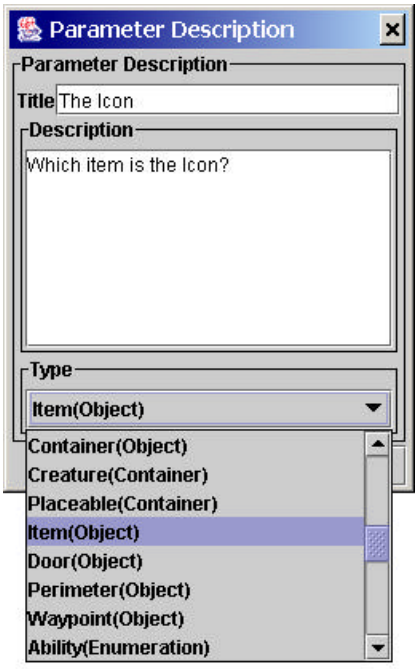

Figure 4.29: A parameter description window from *ScriptEase*'s *Pattern Builder*.

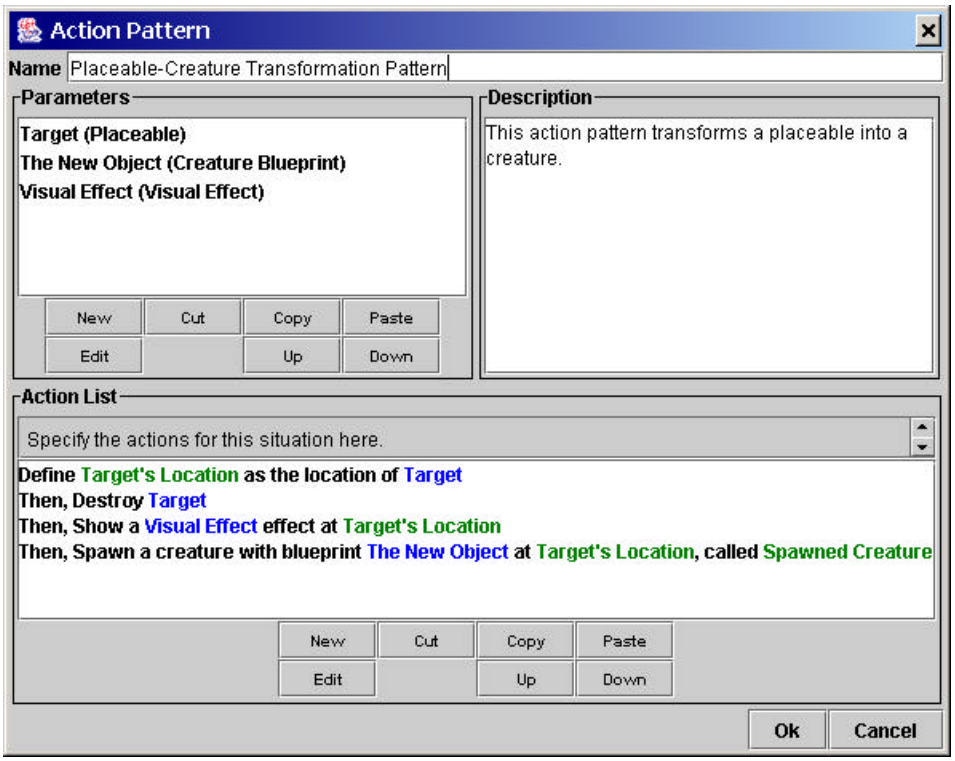

Figure 4.30: The *Action Pattern* definition window of *ScriptEase*'s *Pattern Builder*.

Section 4.2 described how to specify situations in *ScriptEase*. As we mentioned earlier in this section, the major difference when specifying situations within a pattern definition is that patterns exist independent of a *Neverwinter Nights* game module. They are generic components that can be utilized in any game module. Accordingly, no *Aurora*-defined objects are available when specifying these situations. Instead, the user has access to the pattern's formal parameters, such as *The Icon* and *The Container* in the *Icon-Container Pattern*. These formal parameters will not be bound to objects until the pattern is instantiated in the context of a particular game module.

The entries in the parameters list, in Figure 4.28, display their name followed by their type in parentheses. For instance, *The Icon* parameter's type is *Item*. When creating a new parameter or editing an existing one, *ScriptEase* presents the window shown in Figure 4.29. Using this window, the pattern designer specifies the parameter's name, description, and type. The name and description are entered into text-fields, which are used when creating the parameter's questionnaire page. The type is selected from a pull-down list containing all available types. Entries in the type list show the name of the type followed by its super-type in parentheses.

Action Patterns are created using a window similar to the *Encounter Pattern* definition window shown in Figure 4.28. Figure 4.30 shows the *Action Pattern* definition window for the *Placeable-Creature Transformation Pattern*. Instead of

the list of situations defined by an *Encounter Pattern*, an *Action Pattern* defines a list of actions.

4.5 Atoms

An *Atom* is a description of a situation component. *ScriptEase* maintains atoms for events, entities, conditions, and actions. When a user creates a new situation component, say an action, they are first presented with a list of available actions, such as the list of actions shown in Figure 4.10. The entries in the list are actually action atoms, which define the name, the parameters, the English-language description, and other information necessary to instantiate a situation component. For instance in Section 4.2, we used the *Attack a creature* action as an example. This action has two parameters, *Attacker* and *Attacked*. These parameters and the English sentence description (the last action in Figure 4.9) are some of the information specified by the action's atom.

ScriptEase provides a separate tool called the *Atom Builder*, which allows new atoms to be created. Atoms serve as *ScriptEase*'s interface to the underlying text-based scripting language, in our case, *NWScript*. When defining an atom, the atom designer is required to specify the *NWScript* code that *ScriptEase* is to generate during code generation. Therefore, atom designers do require some programming skill and knowledge of *NWScript*. This is in conflict with the first goal we presented at the beginning of this chapter. We accept this because with a large enough base of atoms, the average user will not require additional atoms to be defined, and thus will never even need to look at the *Atom Builder.*

Additionally, by providing the *Atom Builder* feature, the small percentage of *ScriptEase* users who do know how to program can become atom designers, who create new atoms and make them available to the entire user-community over the Web. The *Atom Builder* facilitates extension in *ScriptEase*.

4.5.1 The Atom Definition Window

Figure 4.31 shows the action atom definition window in *ScriptEase*'s *Atom Builder* with the *Spawn a Creature at a Location* action atom loaded. The definition windows for all types of atoms - event atoms, entity atoms, condition atoms, and action atoms - look very similar to the one shown in Figure 4.31. We will discuss each of the window's features.

Name: This field is used to name the atom.

Type: This field defines the type of the entity created by this atom. Only entity and action atoms have this field, since they are the only kinds of atoms that can

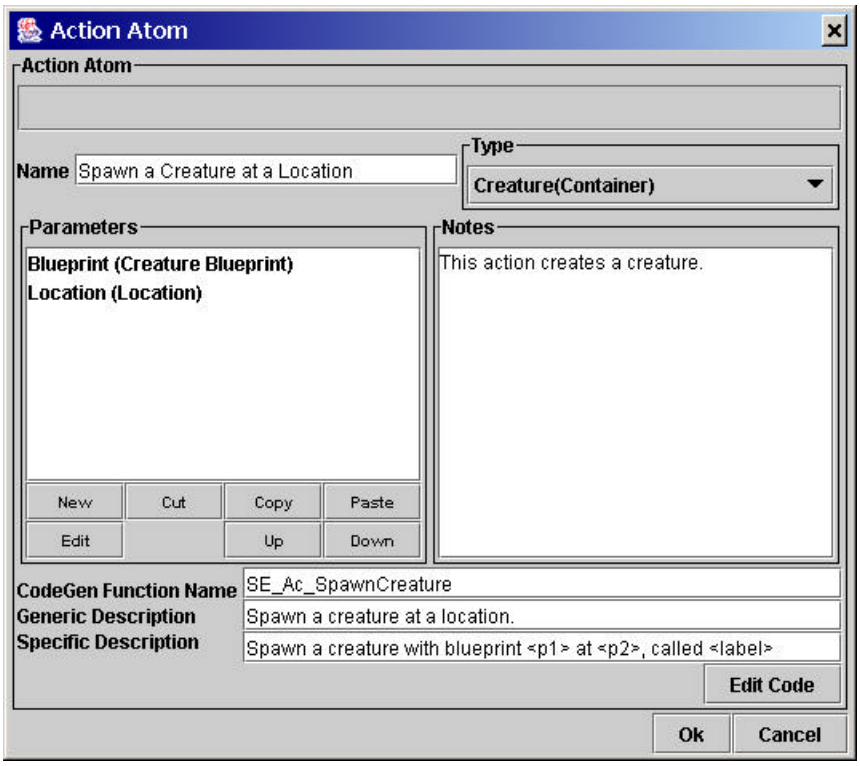

Figure 4.31: The action atom definition window in *ScriptEase*'s *Atom Builder*. The window contains the *Spawn a Creature at a Location* action.

create an entity. Action atoms that do not create an entity, such as the previously mentioned *Attack a creature* atom, indicate this by selecting the type *None*.

Parameters: Parameters are described in exactly the same manner as pattern parameters, which we discussed in Section 4.4.3.

Notes: In the *Notes* panel, the atom creator specifies the description that is shown in *Description page* of the situation component's questionnaire.

Generic Description: Recall the list of actions presented to the user when a new action is added to a situation (Figure 4.10). The text that appears in this list is specified in the *Generic Description* field.

Specific Description: The atom's specific description is used to generate the English-sentence descriptions shown in Figure's 4.6 through 4.9. The tokens `<p1>` and `<p2>` in the *Specific Description* field in Figure 4.31 are substituted by the values of the first and second parameters respectively, when the action is instantiated. In general, a `<pn>` token is replaced by the value of the n^{th} parameter when the situation component is instantiated. The `<label>` token is replaced by the name of the entity that the situation component creates, if it creates one.

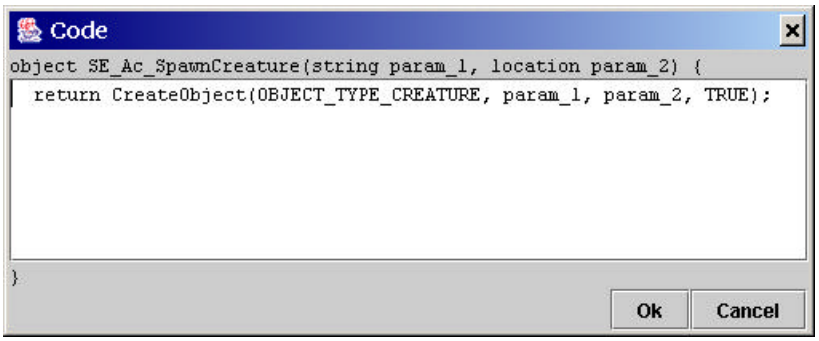

Figure 4.32: The *Code* window of a *Spawn a Creature at a Location* action atom.

CodeGen Function Name: All atoms generate a single function in the underlying text-language during code generation. The name of this function is specified here.

Code: The user specifies the code to generate for this atom by clicking the *Edit Code* button on the bottom right of Figure 4.31. This button opens the window shown in Figure 4.32. Notice that the function's signature is created automatically and displayed along the top of Figure 4.32. The function's name is taken from the *CodeGen Function Name* field. The number of parameters and their types is taken from the list of parameters specified earlier. The return type is taken from the atom's type. The code shown in Figure 4.32 is the *NWScript* code for the *Spawn a Creature at a Location* action specified in Figure 4.28. The atom designer must have enough knowledge of *NWScript* to write the appropriate code, in this case, a single call to the *NWScript* function `CreateObject()`. We discuss code generation more in Section 4.6.

4.5.2 Specifics of Event Atoms

Event atoms require some extra information not needed by the other atom types. Figure 4.33 shows an event definition window for the *When an item is added or removed from a container* event. Two features of this window are not found in the definition windows for the other types of atoms.

Attach Script To: This field allows the atom creator to select one of the event's parameters as the object that the generated script should be attached to during code generation.

Implied Entities: These are the event-dependant entities that are automatically defined by the event. The atom designer specifies these entities in the same way as they would in the *Entities page* of a situation questionnaire (Figure 4.7).

4.5.3 Enumeration Types

Atom designers can define new enumeration types using the atom builder as well. Figure 4.34 shows a series of three windows used in creating new enumeration

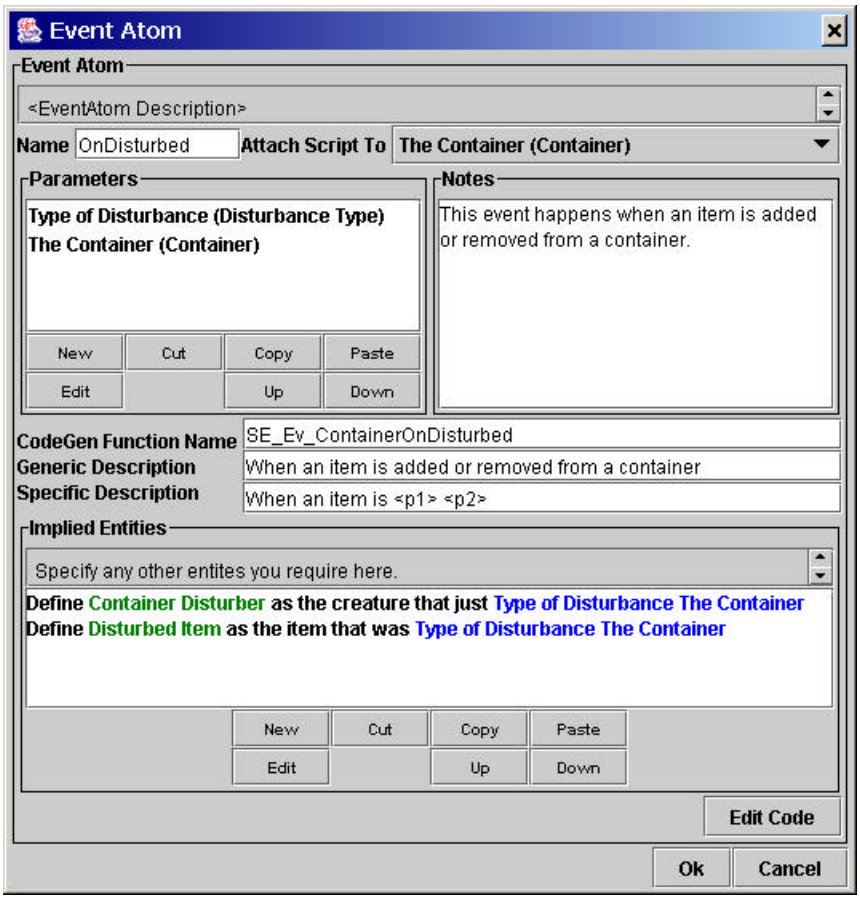

Figure 4.33: The event atom definition window for *ScriptEase*'s *Atom Builder*.

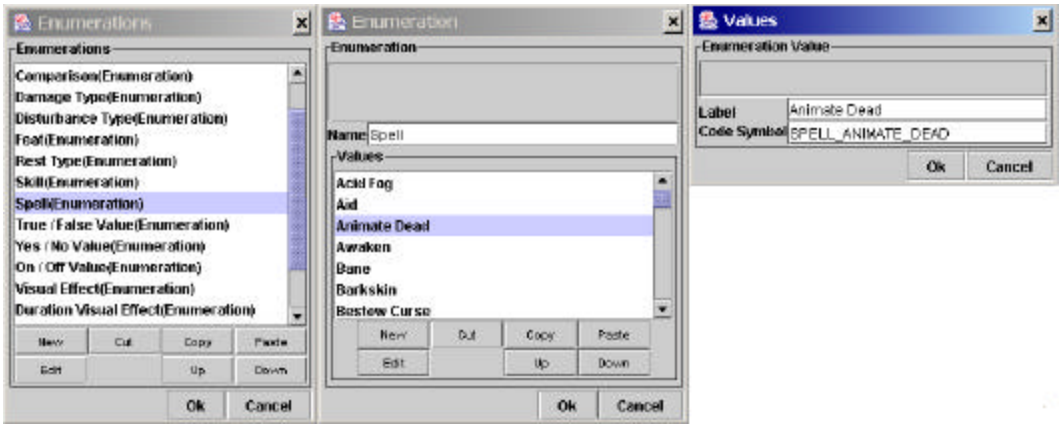

Figure 4.34: A series of windows used in creating new enumeration types with *ScriptEase*'s *Atom Builder*.

types. The left window shows a scrollable list of all of the enumeration types defined thus far. The centre window shows the selected *Spell* enumeration type and all of its possible values. The window on the right is the edit panel for the *Animate Dead* spell. The *Label* field indicates the name of this enumeration value as it appears in *ScriptEase*. The *Code Symbol* field indicates the symbol used to represent this value in *NWScript*. This symbol is used during code generation.

4.6 Code Generation

ScriptEase's code generation involves three steps. The first step is to complete the situations that were instantiated via encounter patterns. These situations contain references to pattern parameters, such as *The Container* in Figure 4.21, that must be bound to their actual values specified by the user. Second, code is generated for every situation in the system. A situation's generated code takes the form of a function, like the one shown in Figure 4.35. The final step is to combine the generated situation functions into scripts that can be attached to game objects.

The first step is self-explanatory. *ScriptEase* scans each situation for references to pattern parameters and replaces them with their user-specified values. The second and third steps require a more in-depth discussion. In Section 4.6.1, we describe how the code in Figure 4.35 is constructed from the situation's components. In Section 4.6.2, we discuss how code for a single component is generated. Finally, in Section 4.6.3, we describe how multiple situation functions are combined into a full script.

4.6.1 Situation Functions

The *NWScript* code shown in Figure 4.35 was generated by *ScriptEase* for the example situation given in Section 4.2, specifically, Figures 4.4 through 4.9. Notice that most lines of code are preceded by a comment that corresponds to the English-sentence descriptions used by the various situation components. These comments are automatically generated to indicate to the reader which lines of code correspond to each situation component. While we do not expect many users to look at the generated code, those that do should have no problem understanding it.

The Function Header

```
1 PedestalZombieTransformation_0() {
```

The name of this situation function is `PedestalZombieTransformation_0`, shown on line 1. This name is automatically generated and is unique among the other situation function names.

The Variable Declarations

```
 6  object Pedestal_SE4;
 7  object DisturbedItem_SE2;
 8  object LarrytheZombie_SE8;
 9  object ContainerDisturber_SE1;
10  object Helm_SE3;
11  location PedestalsLocation_SE5;
```

```
1 void PedestalZombieTransformation_0() {
 2
 3
 4  // The following are all of the variables used in this situation
 5
 6  object Pedestal_SE4;
 7  object DisturbedItem_SE2;
 8  object LarrytheZombie_SE8;
 9  object ContainerDisturber_SE1;
10  object Helm_SE3;
11  location PedestalsLocation_SE5;
12
13  // This script is attached to the following object's OnDisturbed script slot
14  Pedestal_SE0 = OBJECT_SELF;
15
16  // When an item is removed from Pedestal
17  if( ! SE_Ev_ContainerOnDisturbed(INVENTORY_DISTURB_TYPE_REMOVED, Pedestal_SE0) )
18    return;
19
20  // Define Container Disturber as the creature that just removed from Pedestal
21  ContainerDisturber_SE1 = SE_En_ContainerDisturber(Pedestal_SE0,
22                                                    INVENTORY_DISTURB_TYPE_REMOVED);
23
24  // Define Disturbed Item as the item that was removed from Pedestal
25  DisturbedItem_SE2 = SE_En_DisturbedItem(Pedestal_SE0,INVENTORY_DISTURB_TYPE_REMOVED);
26
27  // Define Helm as the item equipped on Container Disturber's Head.
28  Helm_SE3 = SE_En_GetEquppedItem(ContainerDisturber_SE1, INVENTORY_SLOT_HEAD);
29
30  // Main code body - checks conditions and executes actions
31
32  // If Container Disturber is a PC
33  if( SE_Co_IsPC(ContainerDisturber_SE1) ) {
34
35    // If Disturbed Item was created from the Gem of Power blueprint
36    if( SE_Co_CompareObjectsBlueprint(DisturbedItem_SE2, "gemofpower") ) {
37
38      // If Helm was not created from the Wormskull blueprint
39      if( SE_Co_CompareObjectsBlueprintNotEqual(Helm_SE3, "wormskull") ) {
40
41        // Define Pedestal's Location as the location of Pedestal
42        PedestalsLocation_SE5 = SE_En_ObjectsLocation(Pedestal_SE4);
43
44        // Destroy Pedestal
45        SE_Ac_DestroyObject(Pedestal_SE4);
46
47        // Show a Harm effect at Pedestal's Location
48        SE_Ac_ShowVisualEffectAtLocation(VFX_IMP_HARM, PedestalsLocation_SE5);
49
50        // Spawn a creature with blueprint Zombie at Pedestal's Location, called Larry
51        // the Zombie
52        LarrytheZombie_SE8 = SE_Ac_SpawnCreature("zombie002", PedestalsLocation_SE5);
53
54        // Larry the Zombie attacks Container Disturber
55        SE_Ac_AttackCreature(LarrytheZombie_SE8, ContainerDisturber_SE1);
56
57      }
58    }
59  }
60
61 }
```

Figure 4.35: An example of the generated code for a situation in *ScriptEase*.

Next, lines 5 through 11 define all of the variables used in this function. The names of these variables are a combination of the user-defined name used in *ScriptEase* and a trailer string to ensure that no two variable names are the same. For instance, the variable defined on line 5 is `Pedestal_SE4`. The `Pedestal` part is the name used in *ScriptEase* to identify this object. The `_SE4` is the automatically generated trailer string. *ScriptEase* scans the situation and generates a variable declaration line for each entity and *Aurora*-defined object used. Variable name generation is discussed more in the next section.

The Event Code

```
13  // This script is attached to the following object's OnDisturbed script slot
14  Pedestal_SE0 = OBJECT_SELF;
15
16  // When an item is removed from Pedestal
17  if( ! SE_Ev_ContainerOnDisturbed(INVENTORY_DISTURB_TYPE_REMOVED, Pedestal_SE0) )
18    return;
19
20  // Define Container Disturber as the creature that just removed from Pedestal
21  ContainerDisturber_SE1 = SE_En_ContainerDisturber(Pedestal_SE0,
22                                                    INVENTORY_DISTURB_TYPE_REMOVED);
23
24  // Define Disturbed Item as the item that was removed from Pedestal
25  DisturbedItem_SE2 = SE_En_DisturbedItem(Pedestal_SE0,INVENTORY_DISTURB_TYPE_REMOVED);
```

Code generated for the situation's event is split into three parts. The first is shown on lines 13 and 14. This code sets the variable that contains a reference to the object to which the script is going to be attached. *NWScript* refers to this object as `OBJECT_SELF`. *ScriptEase* identifies this variable using the *Attach Script To* feature in the event's atom (refer to Figure 4.33).

The next section of code, lines 16 through 18, is responsible for ensuring that the parameter values for the event match the values provided by the game engine. For instance, the *NWScript*-event that this code is attached to fires whenever any item is added to or removed from the *Aurora*-defined object, *Pedestal*. The *ScriptEase* event is more restrictive in that it requires the inventory disturbance to be a removal. `SE_Ev_ContainerOnDisturbed` is the function specified by the event's atom. It takes a disturbance type and a container as parameters. This function returns true if the disturbance type is a removal and the container is *Pedestal*. If it returns false, the situation function returns immediately, without performing any actions.

Finally, the last part of an event's generated code is the event-implied entities; lines 20 though 25. The variables `ContainerDisturber_SE1` and `DisturbedItem_SE2` correspond to the implied entities shown in Figure 4.6. The functions `SE_En_ContainerDisturber` and `SE_En_Disturbed-Item` are the entities' corresponding code generation functions specified in their atoms.

The Entity Code

```
27  // Define Helm as the item equipped on Container Disturber's Head.
28  Helm_SE3 = SE_En_GetEquppedItem(ContainerDisturber_SE1, INVENTORY_SLOT_HEAD);
```

Lines 27 and 28 contain the generated code for the *Helm* entity, as shown in Figure 4.7. Like the event-implied entities, `SE_En_GetEquppedItem` is the function specified by the *Helm* entity's atom.

The Condition Code

```
32  // If Container Disturber is a PC
33  if( SE_Co_IsPC(ContainerDisturber_SE1) ) {
34
35    // If Disturbed Item was created from the Gem of Power blueprint
36    if( SE_Co_CompareObjectsBlueprint(DisturbedItem_SE2, "gemofpower") ) {
37
38      // If Helm was not created from the Wormskull blueprint
39      if( SE_Co_CompareObjectsBlueprintNotEqual(Helm_SE3, "wormskull") ) {
```

The three if-statements shown in lines 32 though 39 correspond to the three conditions specified in Figure 4.8. The functions `SE_Co_IsPC`, `SE_Co_CompareObjectsBlueprint`, and `SE_Co_CompareObjects-BlueprintNotEqual` are calls to the functions specified in the condition atoms. These functions return true if the condition holds and false otherwise. The generated code for a blueprint parameter value is a constant string. Both `"gemofpower"` and `"wormskull"` are such values. We revisit blueprint parameter values in the next section.

The Action Code

```
41        // Define Pedestal's Location as the location of Pedestal
42        PedestalsLocation_SE5 = SE_En_ObjectsLocation(Pedestal_SE4);
43
44        // Destroy Pedestal
45        SE_Ac_DestroyObject(Pedestal_SE4);
46
47        // Show a Harm effect at Pedestal's Location
48        SE_Ac_ShowVisualEffectAtLocation(VFX_IMP_HARM, PedestalsLocation_SE5);
49
50        // Spawn a creature with blueprint Zombie at Pedestal's Location, called Larry
51        // the Zombie
52        LarrytheZombie_SE8 = SE_Ac_SpawnCreature("zombie002", PedestalsLocation_SE5);
53
54        // Larry the Zombie attacks Container Disturber
55        SE_Ac_AttackCreature(LarrytheZombie_SE8, ContainerDisturber_SE1);
```

The five function calls in lines 41 through 55 correspond to the five actions shown in Figure 4.9. As with the other function calls in this code, these five function calls correspond to the functions defined in the actions' atoms.

4.6.2 Situation Component Code

In order to generate code for a situation, we must generate code for each of its components. In this section, we describe how code is generated for a single

situation component. Every component generates a function call. The name of the function is taken from the *CodeGen Function Name* field of the component's atom (refer to Figure 4.31). For instance, `SE_Ac_AttackCreature` from line 55 of Figure 4.35 is the function name of an *Attack a creature* action.

Each parameter of the function can be a constant, an entity, or an *Aurora*-defined object. Constants include integers, floats, strings, enumerations, and blueprints. An enumeration-typed parameter value generates the symbol specified in the *Code Symbol* field of the enumeration's definition panel (refer to Figure 4.34). `INVENTORY_SLOT_HEAD` on line 28 of Figure 4.35 is an example of code generated for an enumeration value. Blueprints generate a constant string identifier set by *Aurora*. `"zombie002"` on line 52 of Figure 4.35 is an example of the code generated for a blueprint.

Entities and *Aurora*-defined objects are referenced through variables. The variable `Helm_SE3` on line 39 of Figure 4.35 is used as the value of an entity parameter value and is bound on line 28. Each of these variable names is created during a pre-compilation step. This step scans the situation for all entities and *Aurora*-defined objects and associates a unique variable name with each. When a component uses an entity or *Aurora*-defined object, the code generation routine looks up the variable name associated with that object and inserts it into the generated code.

4.6.3 Combining Situation Functions

After the situation functions have been generated, they must be combined into *NWScript* scripts. Every object in *Neverwinter Nights* has a set of script slots that correspond to particular events. For instance, every placeable has an OnDisturbed script slot that corresponds to the following event: When an item is added to or removed from the placeable. Therefore, every one of the situation functions that uses this event must be merged into a single script file. This process is straightforward. Since every situation reduces to a single function, the generated script file simply has to invoke each one of these functions. If there are three generated functions named `situation_0`, `situation_1`, and `situation_2`, all of which use the same event, they must be merged. The final generated script would look like the code in Figure 4.36.

Once these scripts are complete, *ScriptEase* compiles them using a stand-alone *NWScript* compiler, and then inserts them into the game module.

```
void situation_0() {
  // Generated code for this situation
}

void situation_1() {
  // Generated code for this situation
}

void situation_2() {
  // Generated code for this situation
}

void main() {
  situation_0();
  situation_1();
  situation_2();
}
```

Figure 4.36: A script created by merging three situation functions

4.7 Assessment

4.7.1 Setup

In order to assess how practical *ScriptEase* is as a CRPG scripting tool, we performed a small usability review. We hired an experienced game-playing high-school student for two 1-week periods to do testing and a usability review of *ScriptEase*. He had an elementary understanding of procedural programming, typical of that of an average high-school student. He was also familiar with *Neverwinter Nights* and the *Aurora Toolset*, but not the scripting language, *NWScript*. After a week of using *ScriptEase* and testing its features, we requested that he create a small *Neverwinter Nights* game module using *ScriptEase*.

The target module was based on a part of an area known as *The Temple Ruins* from the CRPG *Baldur's Gate 2* [44]. We chose this area for two reasons. First, the situations in this area exist in a commercial CRPG other than *Neverwinter Nights*, indicating that *ScriptEase* can be used to script situations independent of a particular game, although the generated code is specific to *Neverwinter Nights*. Second, this area contains several interesting encounters. By specifying all of them in *ScriptEase*, we demonstrate its flexibility as a scripting tool.

We provided a set of atoms and patterns for the tester to use. He did not use the *Pattern Builder* or the *Atom Builder*.

4.7.2 Patterns Used

The tester used the four encounter patterns, listed below.

Icon-Container

The *Icon-Container* pattern was presented as an example pattern in Section 4.4. Its parameters are an item called *The Icon*, and a container called *The Container*. This pattern defines two situations, one for adding *The Icon* to *The Container*, and another for removing *The Icon* from *The Container*. These situations do not include any actions, leaving them for the user to specify.

Icon-Perimeter

The *Icon-Perimeter* pattern handles cases where a creature carries a particular item called *The Icon*, into or out of a particular perimeter called *The Perimeter*. This pattern defines two situations, one for entering *The Perimeter* and another for exiting. Like *Icon-Container*, the actions executed by these situations are left for the user to specify.

Icon-Container-Perimeter

Icon-Container-Perimeter is an amalgamation of the previous two patterns. It has three parameters: an item called *The Icon*, a container called *The Container*, and a perimeter called *The Perimeter*. This pattern handles the situations where a creature enters or exits *The Perimeter* while *The Icon* is held by *The Container*. The *Icon-Perimeter* pattern could be considered a specialized case of the *Icon-Container-Perimeter* pattern, where *The Container* is the creature entering or exiting *The Perimeter*.

3-Item Construct

The *3-Item Construct* pattern has four parameters. *The First Item*, *The Second Item*, and *The Third Item*, are all items. *The Constructed Item* is an item blueprint. When the three items are placed in any container, they merge into the new *Constructed Item*. This pattern has a single situation that occurs whenever any item is added to any container. If that container happens to contain all three of the specified items, then those items are destroyed, and *The Constructed Item* is created in their place. This pattern requires exactly three component items. A more flexible pattern would allow an arbitrary number of component items. Currently, *ScriptEase* is not capable of specifying an *n-Item Construct* pattern because it does not support list types. We revisit this problem in Chapter 5.

The tester also used the following five action patterns.

XP-Item Reward

The *XP-Item Reward* pattern rewards a creature by giving it some experience points and an item. The pattern has three parameters: *The Rewardee* is the

creature that receives the reward, *XP Amount* is the number of experience to give to *The Rewardee*, and *The Reward Item* is an item to give to *The Rewardee*. Once again, if ScriptEase supported list types, then this pattern could be extended to include a list of reward items, rather than just one.

Damage Punishment

The *Damage Punishment* pattern punishes a creature by applying damage to them. The pattern takes four parameters. *The Victim* is the creature to be punished. *Damage Type* is the type of damage to apply, such as fire, cold, acid, etc. *Damage Amount* is an integer amount of damage to apply. *Damage Effect* is a visual effect to show on *The Victim*.

Death Punishment

The *Death Punishment* pattern resembles the *Damage Punishment* pattern. *Death Punishment* takes two parameters, *The Victim* and *Death Effect*. This pattern shows the visual *Death Effect* then kills *The Victim*.

Spawn and Face

The *Spawn and Face* pattern creates a creature at a given waypoint and faces the creature in the same direction as the waypoint is facing. The pattern's parameters are *The Creature*, which is a creature blueprint, and *The Spawn Point*, which is a waypoint.

Spawn and Attack

The *Spawn and Attack* pattern creates a creature at a given waypoint and orders it to immediately attack a particular object. The pattern's parameters are *The Creature*, which is a creature blueprint, and *The Spawn Point*, which is a waypoint, and *The Target*, which is the object to attack.

4.7.3 Encounters

The following is a list of encounters found in the *Temple Ruins* module, and which patterns were used to specify them.

Monster Encounters

There are several places in the temple where the player is attacked by monsters. Often the monsters spawn when a triggering perimeter is entered, but there are cases where a door being opened or an item being taken causes monsters to spawn. The *Spawn and Attack* action pattern was used in all of these encounters. At times, several monsters were spawned at a time at the same location. It was a

tedious task to instantiate the same pattern repeatedly. Support for lists and loops in *ScriptEase* would do a great deal to alleviate this problem.

The Sun Ray Encounter

One of the more interesting encounters in the *Temple Ruins* involves a pedestal surrounded by a ring of lights, called *Sun Rays.* There is a gem on the pedestal called the *Sunstone*. When the *Sunstone* is removed from the pedestal, the *Sun Rays* go out, and when the *Sunstone* is put back on the pedestal, the *Sun Rays* come on again. The significance of the *Sun Rays* is that any *Shadows* (a type of monster) that enter the ring of lights die immediately if the *Sun Rays* are on. The player character can use the ring of lights as protection, but at some point must remove the *Sunstone* from the pedestal, abandoning the protection of the *Sun Rays*, in order to advance through a locked door in the temple. When the player character approaches the locked door while holding the *Sunstone*, the door explodes in a flash of lightning, allowing the player character to pass.

The tester used the *Icon-Container* pattern to turn the *Sun Rays* on and off in response to the *Sunstone* being added or removed from the pedestal. The *Icon-Container-Perimeter* pattern in combination with the *Death Punishment* pattern was used to kill the *Shadows* that entered the ring of lights, while the *Sunstone* was on the pedestal. The destruction of the locked door was accomplished with an instance of the *Icon-Perimeter* pattern.

The Statue of Amaunator Encounter

There is a statue of a deity named *Amaunator the Sun Lord* in the temple. The player can talk to this statue. The statue tests the player by asking a series of questions about certain rituals pertaining to the worship of *Amaunator*. If all of the questions are answered correctly, an item is given to the player character as a reward. If any question is answered incorrectly, the player character is punished with a damaging flash of fire, and then they must retake the test. Throughout the temple, there are hidden scrolls, which explain the rituals and indicate the answers to the test questions.

The *XP-Item Reward* pattern was used to implement the reward portion of this encounter. The *Damage Punishment* pattern was used to implement to punishment portion.

The Floor Puzzle Encounter

An encounter in the original *Temple Ruins* module from *Baldur's Gate 2* involved a grid on the floor of a room. Each cell in the grid contained a letter of the alphabet. The player character had to walk over the letters spelling out the name *Amaunator* in order to get across to the other side of the room. If the player character stepped on an incorrect letter, they were damaged with a powerful flash

of fire. Unfortunately, the *Aurora Toolset* does not provide the capability to create a grid of letters on the floor. So instead, we covered the floor of the room in scorch marks, except for a narrow winding path. Instead of spelling out the name, the player character simply has to walk the path through the room.

The areas of the floor covered in scorch marks are contained within a perimeter. Therefore, the tester used a *Damage Punishment* pattern, which was activated when the perimeter was entered. If *Aurora* did allow letter-grids to be placed on floors, then the encounter could be implemented as it is in *Baldur's Gate 2* using the same method.

Amauna's Ghost Encounter

There is a stone bed in the temple with a ghostly figure standing beside it. The ghost tells the player that a small child, named *Amauna*, was killed and her grave defiled. He asks the player character to find the child's bones and return them to the stone bed, so she can rest in peace. The bones are hidden somewhere in the temple. Once the bones are found and returned to the stone bed, the ghost of *Amauna* appears and rewards the player character with an item.

The *Icon-Container* pattern, combined with the *Spawn and Face* pattern, was used to spawn the ghost of *Amauna* when the bones were placed on the bed. An *XP-Reward* pattern was used to reward the player character with an item.

Sun Symbol Construction Encounter

The reward items gained from *The Statue of Amaunator* and *Amauna's Ghost* encounters are called *Sun Shards*. These two shards and one other hidden in the temple can be combined into another item called the *Sun Symbol*. Once all three shards are gathered by the player character, they automatically combine to form the *Sun Symbol*. This new item is required to advance further in the *Temple Ruins*.

This encounter is implemented using the *3-Item Construct* pattern. The three *Sun Shards* are the component items, and the *Sun Symbol* is the new constructed item.

4.7.4 Results

We implemented the *Temple Ruins* module in *Aurora* using *NWScript*. Our implementation was over 700 lines of code and took this author, who is an expert user of *NWScript*, approximately three days to write and debug. Our tester had only an elementary understanding of computer programming, and about one week of experience with *ScriptEase*. He scripted the module using *ScriptEase* in a little over one day. However, most of this time was spent on a single frustrating task. The conversation with the statue of *Amaunator* was large, complex, and required

many "Text appears when" scripts[3] attached to its nodes. The tester found getting the conversation to work right, by specifying all of these "Text appears when" scripts and having them interact correctly, was very difficult. In the end, we had to help him complete the conversation scripts. Not including the statue of *Amaunator* conversation's "Text appears when" scripts, it took him only about three hours to script the module in *ScriptEase* and debug by play-testing it. His implementation of the module consists of approximately 12 encounter pattern instantiations and 30 situations, again not including the "Text appears when" scripts. With these scripts, there were over 45 situations in his implementation.

This initial review verified some of the problems that we discuss in Chapter 5, such as attaching scripts to object instances rather than blueprints, arithmetic and string operations, down-casting, and connecting situations. We also identified several additional useful features, such as the ability to delay an action by a certain amount of time, and the need for better support for conversations.

We learned a great deal from the usability review. We plan to hold another, once some of the language and interface issues, discussed in Chapter 5, have been addressed.

4.8 Summary

ScriptEase is written entirely in the *Java* programming language [42]. The source contains approximately 14,000 lines of code not including blank lines, 260 classes and interfaces, and 1700 methods.

ScriptEase allows users to create scripts by specifying patterns and situations in a purely visual environment. Situations are composed of events, entities, conditions, and actions, which are selected from lists and customized by choosing parameters from context-sensitive menus, lists, and trees. A parameter's context is determined by its type. Only entities and *Aurora*-defined objects with appropriate types are presented as valid parameter value choices. Situations are presented in easy-to-read English-language descriptions. Pattern designers can create new encounter patterns and action patterns using the Pattern *Builder*. Atom designers can create new atoms and enumeration types using the *Atom Builder*, although some programming skill and knowledge of *NWScript* is required. Finally, the generated *NWScript* code has a simple structure and is well commented, making it readable to the interested user.

[3] "Text appears when" scripts were mentioned in Section 2.2.2, concerning conversations specified in the *Aurora Toolset*.

Chapter 5

Discussion and Future Work

5.1 Connecting Patterns

Currently, the only support that *ScriptEase* provides for connecting patterns is with *Action Patterns*. An action pattern is instantiated within the context of a situation's action list. When constructing or customizing a situation for an encounter pattern, an action pattern can be instantiated, which adds a series of actions to the situation's action list. In this basic way, encounter patterns and action patterns can be connected. There is no support for connecting encounter patterns to each other.

In Chapter 3, Section 3.1.1, we discussed the CRPG toolset *Unlimited Adventures* and its *Chain Control* feature, which allowed *Unlimited Adventures' Events* to be connected. In the future, we would like to implement similar functionality in *ScriptEase*. Since *ScriptEase*'s situations eventually reduce to a function call during code generation, a natural extension to situations would allow them to invoke other situations. For instance, consider the following situation. There are three fragments of a shattered *Gemstone*, labeled *Frag-1*, *Frag-2*, and *Frag-3*. There is also a magical box, called *Box*, and a perimeter, called *Perimeter*. When all three of the fragments are placed within *Box* and a *Bless* spell is cast upon the box, they are recombined to form the complete *Gemstone*. Alternatively, the fragments can be combined into the *Gemstone* when they are all carried into *Perimeter* by a player character. Figure 5.1 shows these two situations in component form. There is a large amount of repeated information in these two situations. In Figure 5.2, we have added a parameterized third situation, called *Construct Gemstone*. This situation contains the conditions and actions that combine the fragments into the gemstone. The parameter to this situation is the container that holds the fragments. The *Box Situation* and *Perimeter Situation* invoke the *Construct Gemstone* situation, instead of explicitly including the repeated information themselves. Currently, *ScriptEase* does not support parameterized situations or situation invocation. However, with these features, situations could be connected together.

It is unclear how to extend connected parameterized situations into connected multi-situation encounter patterns. However, during a preliminary assessment of the patterns present in CRPGs, we are finding that most potential patterns involve only a single situation. Therefore, it is worth exploring the possibility of redefining an encounter pattern to simply be a parameterized situation. Once the

Name: Box Situation

Event:
When a *Bless* spell is cast on *Box*

Conditions:
If *Box* holds *Frag-1*, and
If *Box* holds *Frag-2*, and
If *Box* holds *Frag-3*

Actions:
Destroy *Frag-1*, then
Destroy *Frag-2*, then
Destroy *Frag-3*, then
Create the *Gemstone* inside *Box*

Name: Perimeter Situation

Event:
When *Perimeter* is entered

Event-Implied Entities:
Define *Perimeter Enterer* as the creature that entered *Perimeter*

Conditions:
If *Perimeter Enterer* is a PC
If *Perimeter Enterer* holds *Frag-1*, and
If *Perimeter Enterer* holds *Frag-2*, and
If *Perimeter Enterer* holds *Frag-3*

Actions:
Destroy *Frag-1*, then
Destroy *Frag-2*, then
Destroy *Frag-3*, then
Create the *Gemstone* inside *Perimeter Enterer*

Figure 5.1: Two example situations with repeated information.

Name: Construct Gemstone

Parameter:
The Container

Event: None

Conditions:
If *The Container* holds *Frag-1*, and
If *The Container* holds *Frag-2*, and
If *The Container* holds *Frag-3*

Actions:
Destroy *Frag-1*, then
Destroy *Frag-2*, then
Destroy *Frag-3*, then
Create the *Gemstone* inside *The Container*

Name: Box Situation

Event:
When a *Bless* spell is cast on *Box*

Conditions: None

Actions:
Invoke *Construct Gemstone* situation with parameter *Box*

Name: Perimeter Situation

Event:
When *Perimeter* is entered

Event-Implied Entities:
Define *Perimeter Enterer* as the creature that entered *Perimeter*

Conditions:
If *Perimeter Enterer* is a PC

Actions:
Invoke *Construct Gemstone* situation with parameter *Perimeter Enterer*

Figure 5.2:Two example situations with their common information extracted into a separate parameterized situation.

initial version of *ScriptEase* is released to the public, user-feedback on this issue will be invaluable.

5.2 Language Issues

5.2.1 Language Structure

The structure of *ScriptEase*'s situations is much more restrictive than the syntax of a text-based language. This is desired because the restrictive structure makes situations easy to construct. However, this also makes future extensions to situations hard to implement. Additionally, it unnecessarily restricts the power of the users who are proficient programmers. Ideally, *ScriptEase* should have a powerful underlying representation that is as flexible as a text-based language. Multiple views could be built on top of the underlying representation in order to provide users, having various levels of programming skill, with an appropriately complex visual scripting tool.

Procedural text-based languages have four primary features: functions, arithmetic operators, variable assignments, and control structures, such as if-then-else blocks and loops. Additionally, these languages support complex data types such as records and lists. A vital avenue for future work is to examine to how each of these features can be included in *ScriptEase*. *ScriptEase* already supports simple if-statements, through *Conditions*, variable assignments, through *Entity* definitions, and a restrictive function structure, through *Situations*. A situation's structure could be extended to allow arbitrary ordering and nesting of conditions and actions, as well as extending conditions to include else-blocks. List types, list iterators, and loops should also be straightforward extensions. These extensions will require us to consider entity-scoping issues. For instance, an entity defined inside of an if-block should only be visible within that if-block.

Figure 5.3 illustrates what the code might look like in this system. Entities, constants, and *Aurora*-defined objects are coloured as they are in the current version of *ScriptEase*, discussed in Chapter 4. It should be possible to extend *ScriptEase* to permit all of this code to be specified using its purely visual interface, while remaining simple to use. In plain English, this code iterates through a list of three creature blueprints, instantiates them, and sets the direction that they face. Thirty-five percent of the time the creatures are placed at a waypoint called *Spawn WP 1*, and the rest of the time they are placed at the waypoint called *Spawn WP 2*. The *Monster List* entity definition, at the top of the figure, demonstrates how a list would be defined. The *For each* line demonstrates a simple list-iterator structure. This statement loops through all entries in the list *Monster List* and defines an entity called *Monster* that references the current list element. The body of the loop is indicated by the text's indentation. The if-statement in this code has an else component, indicated by the *Otherwise*

```
Define Monster List as a list of the following creature blueprints: Skeleton,
    Zombie, and Wraith

For each Monster in Monster List
      Define Random to be a random integer between 0 and 99

      If Random is less than 35
            Spawn a creature from the blueprint Monster at Spawn WP 1,
                called Spawn
            Then, Face Spawn in the same direction as Spawn WP 1
      Otherwise
            Spawn a creature from the blueprint Monster at Spawn WP 2,
                called Spawn
            Then, Face Spawn in the same direction as Spawn WP 2
```

Figure 5.3: A mockup of a sample code segment from a possible future version of *ScriptEase*.

keyword. The *Random* entity is defined within the loop body and should not be accessible outside of the loop. Similarly, the *Spawn* entities inside the if-block and the otherwise-block are unrelated, since they are defined in parallel scopes. Finally, notice that the structure of the code is fairly wide-open. Conditions, actions, entity definitions, and loops can be arbitrarily ordered and nested.

5.2.2 Operators

Arithmetic and string operators are a major problem area in visual programming languages [17]. The problem is that the easiest interface, for specifying equations with multiple terms combined using various operators, is text-based, even for non-programmers. Most people are familiar with seeing these equations in text, from their schooling, and are capable of writing them. Using a visual interface to specifying equations is awkward and unintuitive. Fortunately, the average CRPG script does not require complex equations. However, for the few cases where these equations are necessary, special text-based support would be desirable.

5.2.3 Down-Casting

During our own internal tests, we have experienced problems with down-casting entities. Our concerns were verified during our initial usability review, discussed in Section 4.7. For instance, consider an entity called *Owner*, which is the container that holds the item, *Sword*. The designer knows that *Owner* is a creature, and wants them to attack the player character. Unfortunately, *Owner*'s type is *Container*, not the subtype *Creature*. Containers cannot attack things, only creatures can. Therefore, *Owner* cannot be instructed to attack anything using *ScriptEase*. In a text-based language, *Owner* could be cast into the *Creature* type. Research into how *ScriptEase* could elegantly accomplish down-casting is necessary.

5.2.4 Pattern Parameters

Other automatic code generation tools, such as *CO_2P_2S* [15], use multiple types of parameters. *CO_2P_2S* has four types of parameters: *lexical parameters*, *design parameters*, *performance parameters*, and *verification parameters*. *Lexical parameters* are used to specify syntactic elements in the generated code, such as class and method names. *Design parameters* affect the structure of the generated code. *Performance parameters* affect the performance of the generated code. *Verification parameters* generate code that performs runtime semantic checks on user-supplied code, when turned on. Parameters in *ScriptEase* do not affect the structure, performance, or verification of the generated code at all. They only indicate what code should be inserted into the function invocation that they are associated with. If the structure of a pattern's generated code was not fixed, but dynamic based on some (or all) of the pattern's parameters, then structure, performance, and verification issues could be addressed in *ScriptEase*'s code generation. The disadvantage of this approach is that specifying patterns in the *Pattern Builder* would not be as simple as it is in the current version of *ScriptEase*.

5.3 Interface Issues

ScriptEase was developed as a rapid prototype. Because of this, some of the interface design decisions that we made can be improved upon.

5.3.1 Modal vs. Non-Modal Dialog Windows

A dialog window is modal if when it is opened, all other windows in the application are disabled. In other words, if a modal dialog is open then none of the application's other windows can be edited. Information is specified in *ScriptEase* using a series of modal dialogs. By making all dialog windows modal, we were able to avoid the problems involved with editing multiple views whose data may be dependent upon each other. Text-based programmers often utilize multiple code windows, open simultaneously. Based on our own experience, multiple editable views would be desirable in *ScriptEase*, as well. With a properly implemented *Model-View-Control* [12] architecture, *ScriptEase* could still address these problems and provide multiple editable views through non-modal dialog windows.

5.3.2 Tree Organization

There are several places in *ScriptEase* where trees could be used to better organize information. Atom lists are one of these places. When the user selects an event, entity, action, or condition atom, they are presented with a list. This list

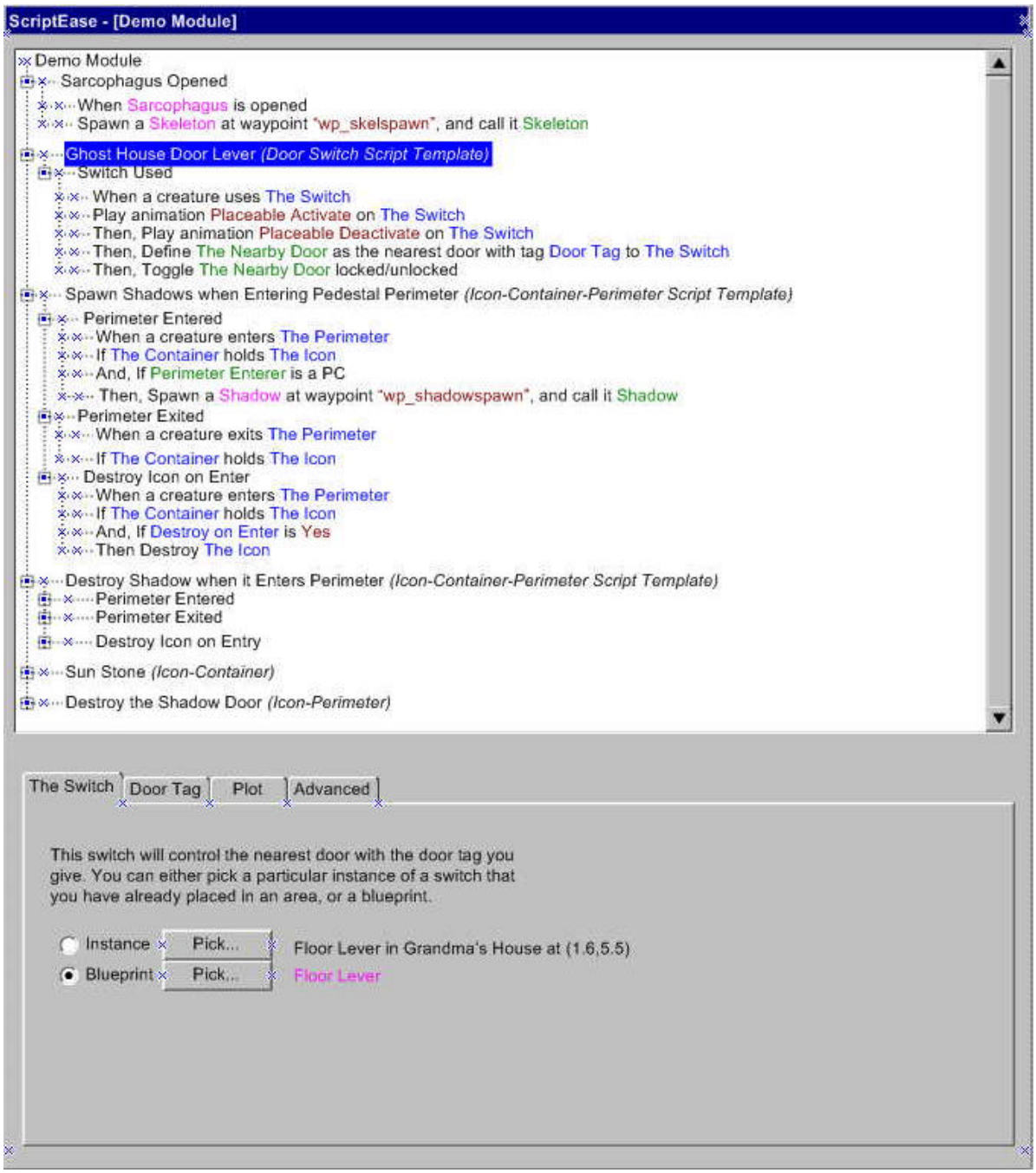

Figure 5.4: Mockup of a possible future *ScriptEase* interface. This figure was created by Matt McNaughton using Microsoft's *Visio*.

can contain hundreds of entries, which makes finding the particular atom that the user is looking for difficult. If atoms were organized in a categorized tree, locating the appropriate atom would be easier.

Generally, *ScriptEase* presents the user with lists of objects, which when selected for editing cause other modal dialogs to open. For instance, if the user selects a situation from a pattern's situation list then that situation's questionnaire opens. If

the user then selects an action from the situation's action list, that action's questionnaire opens. This proliferation of modal dialogs can be eliminated by organizing all of this information into a tree structure. Matt McNaughton and Dominique Parker, two member of the *ScriptEase* team, have been working on a new interface for *ScriptEase*, which addresses this issue. Figure 5.4 shows a mockup of the interface that Matt McNaughton created. This figure shows a pattern instance called *Ghost House Door Lever* selected. The pattern's parameters are shown in the tabbed panel at the bottom of the figure. *The Switch* and *Door Tag* tabs contain pattern parameters. The *Plot* and *Advanced* tabs relate to other future extensions that Matt is exploring. Note how patterns, situations, and situation components are all displayed within a single window, organized in a tree.

5.3.3 The ScriptEase-Aurora Interface

The current version of *ScriptEase* modifies the scripts of object blueprints in *Aurora* then propagates those changes to all of the instances of the blueprints. Because of this, two instances of the same blueprint cannot be accessed separately in *ScriptEase*. In our experience using *ScriptEase*, accessing objects in this way is awkward. Providing *ScriptEase* access to individual object instances is a necessary extension.

5.4 Error Reduction

As a visual programming tool, *ScriptEase* is inherently easier to use and less error-prone then a text-based programming language, which has syntax. Since *ScriptEase* automatically generates compilable code, the user does not have to worry about syntactic or semantic errors. The visual dialogs used to specify parameter values only allow valid values to be selected, further reducing the possible errors that the user could make.

Currently, *ScriptEase* does not handle errors that are the result of a user reordering or removing situation components. For instance, consider the list of actions shown in Figure 5.5. If the user deletes the first action, which defines the *Pedestal's Location* entity, then the third and forth actions become invalid, since they use *Pedestal's Location*. At present, *ScriptEase* does not catch this error. Adding checks that validate the correctness of a situation, after any component is removed or reordered, should be a straightforward extension to *ScriptEase*.

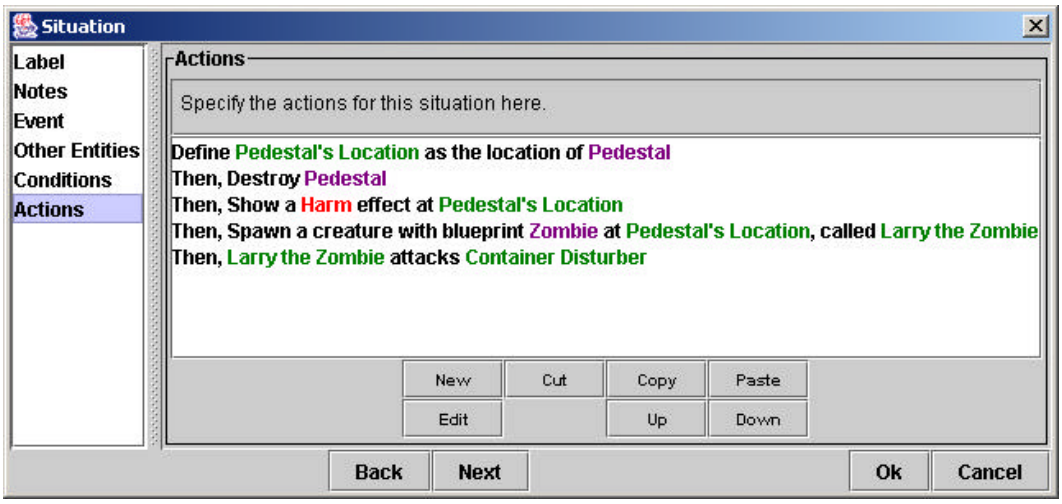

Figure 5.5: The *Actions page* of a situation questionnaire.

5.5 Patterns

Patterns facilitate code reuse in *ScriptEase*. Encounter pattern instantiations work by copying a set of parameterized situation templates into the system. Once created, these situations no longer rely on the pattern from which they were created. This independence allows the user to add to, remove from, and modify the situations in any way they wish. The advantage of this approach is that it gives the user a great deal of power, since they retain total control over the new situations. The disadvantage is that the situations can be modified to such an extent that they no longer make sense in the context of the pattern that they belong to. In other words, the user is capable of using the pattern incorrectly, possibly causing unexpected behavior in the game. The way in which patterns are instantiated, and the way users customize these instantiations, may need to be reevaluated.

5.6 Behavior, Plot, and Conversation Patterns

Currently, *ScriptEase* supports patterns for encounters. In the future, we would like to incorporate behavior patterns, plot patterns, and conversation patterns into *ScriptEase*, as well. We have contemplated behavior and plot patterns and we will present our preliminary thoughts on the subject. We have not yet given any significant attention to conversation patterns or the problems with specifying "Text appears when" scripts that we mentioned in Section 4.7.

A behavior pattern could involve a single principle actor who utilizes a finite state machine. A set of situations is placed within each state, which defines the creature's behaviors while they are in that state. For instance, assume a creature's

FSM includes three states, *Guard*, *Off Duty*, and *Asleep*. The situations in the *Guard* state define all of the actions that the creature performs while guarding a treasure chest, or the door to the treasury, or the king, or any other object or location specified by the user. The *Off Duty* state's situations could define actions for eating dinner, relaxing at home, or going for a leisurely walk. The *Asleep* state's situations would define the actions associated with sleeping. There also needs to be a mechanism for specifying state transitions and their conditions. Additionally, the possibility of nesting FSMs needs to be considered. Finally, we are planning to investigate the incorporation of AI techniques, such as machine learning, into behavioral patterns.

Plot can be specified using a directed graph, where nodes indicate plot states and edges represent conditions that must hold for that state to be reached. We have contemplated representing the plot-advancement conditions as tokens that a character possesses. For instance, in order for a character to open a magical door that advances the plot in some way, there are two conditions: the character possesses the magic key and the character has talked to the king. The magic key is an item that can be carried; therefore, it is a physical plot token. Talking to the king is a task that the character has performed, and is not evidenced by an item being held. Therefore, this condition is represented by giving the character a virtual token indicating that the task was completed. If the character possesses both tokens, the door is opened and the plot advances.

5.7 Extension to Other CRPGs

Ultimately, we would like to evolve *ScriptEase* into a general-purpose CRPG scripting tool, which is independent of a particular game. In theory, we should be able to extend *ScriptEase* to generate code for any CRPG scripting language that is similar to *NWScript*. These are languages that are procedural, event-based, and access the game engine through a set of library functions, such as *Morrowind*'s scripting language [47]. Each game would require its own set of *Atoms*, which correspond to library functions and include the basic building blocks of the source code to generate.

5.8 Conclusion

In Chapter 1, we stated that our aim was to develop a CRPG scripting tool that was both powerful and easy to use. *ScriptEase* accomplishes both of these goals by working on multiple levels of abstraction. *ScriptEase* is easy to use because high-level information is specified through context-sensitive menus and by instantiating reusable patterns. It is also powerful because the high-level

information can then be tweaked by customizing the pattern situations' low-level components. The *Pattern Builder* and *Atom Builder* work on respectively lower levels of abstraction, while still providing easy-to-use visual interfaces. These tools allow users to extend *ScriptEase* by adding their own patterns and atoms, further increasing the power of the system. *ScriptEase*'s initial usability review was the first step in validating its purpose as a practical CRPG scripting tool. Once released to the public, *ScriptEase* will undergo its first substantial trial.

In this chapter, we discussed several outstanding language and interface issues with *ScriptEase*. Clearly, there is room for improvement. Still, we believe *ScriptEase* is more powerful than any other available visual scripting tool for computer games. Furthermore, the *ScriptEase* project is an ongoing effort shared by several colleagues at the University of Alberta. We are confident that future versions of *ScriptEase* will be even more powerful, flexible, and easy to use.

Bibliography

Publications

1. Thor Alexander. An Optimized Fuzzy Logic Architecture for Decision Making, *AI Game Programming Wisdom*, Charles River Media, USA 2002, pp. 367-374.

2. Jonty Barnes and Jason Hutchens. Scripting for Undefined Circumstances, *AI Game Programming Wisdom*, Charles River Media, USA, 2002, pp. 530-540.

3. Lee Berger. Scripting: Overview and Code Generation, *AI Game Programming Wisdom*, Charles River Media, USA, 2002, pp. 505-510.

4. Lee Berger. Scripting: The Interpreter Engine, *AI Game Programming Wisdom*, Charles River Media, USA, 2002, pp. 511-515.

5. Lee Berger. Scripting: System Integration, *AI Game Programming Wisdom*, Charles River Media, USA, 2002, pp. 516-519.

6. Mark Brockington and Mark Darrah. How Not to Implement a Basic Scripting Language, *AI Game Programming Wisdom*, Charles River Media, USA 2002, pp. 548-554.

7. Steve Bromling. Meta-programming with Parallel Design Patterns, M.Sc. Thesis, Department of Computing Science, University of Alberta, 2002. (http://www.cs.ualberta.ca/~systems/Theses/Bromling.MS.ps)

8. Frank Budinsky, Marilyn Finnie, Patsy Yu, John Vlissides. Automatic Code Generation from Design Patterns. IBM Systems Journal, 35(2): 151-171, 1996.

9. Monte Cook, Jonathan Tweet, Skip Williams. *Dungeons and Dragons Player's Handbook, 3rd Edition*. Wizards of the Coast, 2002.

10. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

11. Bruce Dawson. Game Scripting in Python, Game Developers Conference, 2002. (http://www.gamasutra.com/features/20020821/dawson_01.htm, and further discussion at http://www.cygnus-software.com/papers/gamescriptinginpython.html)

12. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

13. Neil Kirby, Solving the Right Problem, *AI Game Programming Wisdom*, Charles River Media, USA 2002, pp. 21-28.

14. Steve MacDonald. From Patterns to Frameworks to Parallel Programs, Ph.D. Thesis, Department of Computing Science, University of Alberta, 2002. (http://plg.uwaterloo.ca/~stevem/papers/PhD.pdf)

15. Steve MacDonald, Duane Szafron, Jonathan Schaeffer, John Anvik, Steve Bromling and Kai Tan. Generative Design Patterns, 17th IEEE International Conference on Automated Software Engineering (ASE) 23-34, September 2002. (http://www.cs.ualberta.ca/~jonathan/Papers/Papers/2002ase.pdf)

16. Matt McNaughton, James Redford, Jonathan Schaeffer and Duane Szafron. Pattern-based AI Scripting using ScriptEase, The Sixteenth Canadian Conference on Artificial Intelligence (AI 2003), Halifax, Canada, June 2003, pp. 35-49. (http://www.cs.ualberta.ca/~jonathan/Papers/Papers/scripting.ps)

17. Dominique Parker. A Survey of Visual Programming Tools, M.Sc. Essay, University of Alberta, July 2003. (http://www.cs.ualberta.ca/~dominiqu/essay/tools.ps.gz)

18. Falko Poiker. Creating Scripting Languages for Nonprogrammers, *AI Game Programming Wisdom*, Charles River Media, USA 2002, pp. 520-529.

19. Pieter Spronck, Ida Sprinkhuizen-Kuyper and Eric Postma. Online Adaptation of Game Opponent AI. The 15th Belgian-Dutch Conference on Artificial Intelligence (BNAIC), The Netherlands, 2003.

20. Everard Strong. Industry Watch: Keeping an Eye on the Game Biz, Game Developer Magazine, April 2003, p. 8.

21. J.R.R. Tolkein. *Lord of the Rings.* George Allen & Unwin (Publishers) Ltd. 1954.

22. Paul Tozour. The Perils of AI Scripting, *AI Game Programming Wisdom*, Charles River Media, USA 2002, pp. 541-547.

23. Jack van Rijswijck. Learning Goals in Sports Games, Game Developers Conference, San Jose, 2003. (http://www.cs.ualberta.ca/~javhar/research/LearningGoals.doc)

24. Michael Zarozinski. An Open-Source Fuzzy Logic Library, *AI Game Programming Wisdom*, Charles River Media, USA 2002, pp. 90-101.

Websites

25. Bioware. Neverwinter Nights: For Developers, http://nwn.bioware.com/developers, 2003.

26. Bioware. Neverwinter Nights Awards, http://nwn.bioware.com/about/awards.html, 2003.

27. Jason Brownlee. Finite State Machines (FSM), AI Depot, http://www.ai-depot.com/FiniteStateMachines, 2003.

28. Crono and Dekar. Console RPGs are Non-Existant. http://www.geocities.com/TimesSquare/Dungeon/2857/edit_nonrpg.html, May 1999.

29. Dragon, http://www.homestarrunner.com/sbemail58.html, 2003.

30. Murray Keir. A Discourse on Computer Role-Playing, http://ptgptb.org/0003/discourse.html, 1998.

31. Lilac Soul. Lilac Soul's Neverwinter Nights Page, http://lilacsoul.revility.com, 2003.

32. Lilac Soul. Lilac Soul's NWN Script Generator, http://nwvault.ign.com/Files/other/data/1044998316652.shtml, 2003.

33. List of Products Created with the Prograph Programming Language, http://www.tritera.com/prograph.html, 2001.

34. LunarAlex. Definition of RPGs: Another Perspective, http://www.rpgfan.com/editorials/old/1999/0006.html, 1999.

35. Jeff Luther. The RPG Experience: Conventions and not Beyond, http://www.gamesfirst.com/articles/jluther/rpg_narrative/rpg_narrative.htm, April 2001.

36. MIT Media Laboratory. Starlogo on the Web, http://education.mit.edu/starlogo, 2002.

37. Northwoods Software. Visual Programming Tools from Northwoods Software, http://www.nwoods.com/sanscript, 2003.

38. NWN Lexicon Group. *Neverwinter Nights* Lexicon, http://www.reapers.org/nwn/reference, 2003.

39. Pictorious Incorporated. Prograph, http://www.pictorius.com/prograph, 2002.

40. Python Language Website, http://www.python.org, 2003.

41. Edward T. Smith. Torlack.com, http://www.torlack.com, 2002.

42. Sun Microsystems. The Source for Java Technology, http://sun.java.com, 2003.

Computer Games

43. *Baldur's Gate*. Bioware Corp. / Black Isle Studios / Interplay, 1998. (http://www.bioware.com/games/baldurs_gate)

44. *Baldur's Gate 2: Shadows of Amn*. Bioware Corp. / Black Isle Studios / Interplay, 2000. (http://www.bioware.com/games/shadows_amn)

45. *Black & White*, Lionhead Studios / Electronic Arts, 2001. (www.bwgame.com)

46. *Blade of Darkness*. Codemasters, 2001. (http://www.codemastersusa.com/blade/)

47. *The Elder Scrolls III: Morrowind*. Bethesda Softworks, 2002. (http://www.elderscrolls.com)

48. *EveOnline*. CCP Games, 2003. (http://www.eve-online.com)

49. *FIFA Soccer*, EA Sports, 2003. (http://www.easports.com/games/fifa2004)

50. *Final Fantasy*. SquareSoft, 1987. (http://www.square-enix-usa.com)

51. *Final Fantasy X*. SquareSoft, 2001. (http://www.square-enix-usa.com)

52. *Frequency*. SCEA, 2001. (http://www.scea.com/games/categories/stratpuzzle/frequency/)

53. *Gorasul: The Legacy of the Dragon*. Silver Style / JoWood, 2001. (http://www.jowood.com)

54. *Hero's Quest: So You Want To Be a Hero?* Sierra, 1990. (http://www.sierra.com)

55. *Icewind Dale*. Black Isle Studios / Interplay, 2000. (http://www.interplay.com/icewind/)

56. *Neverwinter Nights*. Bioware Corp. / Atari, 2002. (http://nwn.bioware.com)

57. *Neverwinter Nights: Shadows of Undrentide*. Bioware Corp. / Atari, 2003. (http://nwn.bioware.com/shadows)

58. *Quake.* id Software, 1996. (http://www.idsoftware.com/games/quake/quake)

59. *Resident Evil*. Capcom, 1996. (http://www.capcom.com/ResidentEvil)

60. *The Sims*, Electronic Arts, 2000. (http://thesims.ea.com)

61. *Starcraft*, Blizzard Entertainment, 1998. (http://www.blizzard.com/starcraft)

62. *Super Mario Bros.* Nintendo, 1985. (http://www.nintendo.com)

63. *Ultima 1: The First Age of Darkness*. Origin Systems, 1981. (http://www.origin.ea.com)

64. *Ultima 3: Exodus.* Origin Systems, 1983. (http://www.origin.ea.com)

65. *Ultima 4: The Quest of the Avatar.* Origin Systems, 1985. (http://www.origin.ea.com)

66. *Unlimited Adventures*. Micromagic / Strategic Simulations Inc., 1993.