Code Generation for AI Scripting in Computer Role-Playing Games

M. McNaughton, J. Schaeffer, and D. Szafron,
Department of Computing Science, University of Alberta
Edmonton, AB T6G 2E8, Canada
{mcnaught,jonathan, duane}@cs.ualberta.ca

D. Parker
Electronic Arts (Canada) Inc.
4330 Sanderson Way
Burnaby, BC V5G 4X1, Canada
dparker@ea.com

J. Redford
BioWare Corp.
200, 4445 Calgary Trail,
Edmonton, AB T6H 5R7, Canada
jamesr@bioware.com

Abstract

Scripting custom content for computer role-playing games requires the designer to tell a story by writing small fragments of computer code distributed among the characters, props, and locations of the game world. The main challenge of these games is to create believable motivations and behaviors for the dozens or even hundreds of characters and rooms with which the player may interact. We present the ScriptEase model of game scripting. This model is pattern-template based, allowing designers to quickly build complex game worlds without doing explicit programming. This is demonstrated by generating code for BioWare's game Neverwinter Nights.

Introduction

Players of computer role-playing games (CRPGs) are demanding more sophisticated artificial intelligence (AI) and storylines to complement the rich graphics of today. CRPG producers also enhance their sales by bundling software tools that users can use to augment the game, in essence using it as a platform for their own creations. For example, BioWare's hit game Neverwinter Nights (NWN) [7] shipped with the Aurora toolset, which BioWare's own designers used to develop the game content. This includes the C-like scripting language, NWScript, used to control the characters and special items in the game. Enthusiasts without training as programmers were frustrated by the complexity of the scripting system. We saw an opportunity to tap the creativity of non-programmers by providing them with a tool they could use to script the AI and the story without explicit programming.

We present ScriptEase [12], a tool that makes CRPG scripting easy for the non-programmer without depriving expert programmers of the power they demand. To satisfy this wide range of needs for learnability, expressiveness, and extensibility, ScriptEase uses generative design patterns [11] to operate on three levels of programming abstraction. ScriptEase also tackles the problem of knowledge management in virtual game worlds by helping to track and organize the thousands of scripts scattered among the objects in the world. ScriptEase supports testing by encapsulating scripts into tested and reusable pattern artifacts. We are certainly not the first group to design a programming tool for non-programmers [1, 4]. Nor are we the first to provide such a tool for a computer game [2, 5]. However, the wide scope and power of ScriptEase is new. We do not believe that the mechanism we use to achieve reuse and knowledge transfer from expert programmers to non-programmers has been used in a game context before. BioWare called ScriptEase "the answer to our non-programming dreams" [6].

The remainder of this paper is organized as follows: we discuss requirements for a visual programming tool in general and in particular for the control flow and data structure needs of CRPG scripts. We discuss the structure of ScriptEase scripts and extensions of ScriptEase to better support rich and reusable AI behaviors. Lastly we discuss its usability.

Problem

The definitions of the colloquial terms "scripter" and "programmer" are contentious, but the distinction is real. Many scriptable products and scripting tasks do not require any familiarity with the programmers' tools of the trade, such as algorithmic complexities, class hierarchies and polymorphism, pointers, or parameter passing conventions.

The scripting tasks in CRPGs are enormously challenging. Due to the CPU requirements and programming complexity required for strong AI in the traditional sense, convincing behaviors have to be hand-coded for all cases that are likely to arise in game-play situations. For example, in NWN, once the PC leaves the Hall of Justice at the start of the game, an NPC approaches to ask for help with finding her family. The NPC has no model of being lonely and missing its family, nor does it deduce that the PC is a great hero and may be able to help. The game is simply scripted by the designer to monitor the PC's exit from the building and to force the NPC to run towards the PC and initiate a conversation which has been scripted by the designer. The game is full of hundreds of such vignettes, each of which requires several carefully-orchestrated scripts. Creating and testing such scripts is very difficult and time consuming. This example required nine scripts distributed between the NPC, an invisible tripwire drawn on the floor, and the dialog file attached to the NPC. The capabilities of the AI are limited by the immense amount of time it takes to create and test scripts. Our approach manages this complexity and provides infrastructure that will support AI. NPC characters will be able to learn, remember events in the game, and have rich behaviors.

The NWScript Language

NWScript is a C-like scripting language used by NWN to control everything from the NPCs' behaviors and the storyline of the game to weather changes and lighting effects. Notably missing are pointers and full support for aggregate data structures like C structs and arrays. Long-term data is stored in string-indexed dictionaries attached to game objects. Aurora provides NWScript with an interface to the game engine through a set of library functions. The game's execution model is event-driven. Scripts run in response to events in the game world. When a particular event occurs, its associated script executes. Scripts that do not terminate in a fixed number of virtual machine instructions are forcibly aborted. Almost every object in the game has a set of events associated with it. For example, creatures recognize 12 events, such as when its way is blocked, when it is attacked, and when it sees something new. Other objects like furniture, rooms, and doors have scriptable events as well. Notably, conversations can have scripts attached. A conversation is a tree of all things an NPC character could say to a PC, and all responses the PC could choose from. Not all nodes are appropriate given the state of the storyline and the circumstances, so a script may be attached to a conversation node to control whether it may be said. Also, a script may be run when a conversation node is uttered in-game, either by an NPC or by the PC.

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 head of the queue and executed. An action can be something like walking to a target location, commencing an attack, opening a chest, or resting for a while. The scripting language is capable of modifying an action queue by either clearing it, temporarily locking it so new actions cannot be added, or adding new actions to the end. The ability to have a callback fire when an action is cleared or completed would be a welcome addition.

User Requirements

To determine the capabilities ScriptEase would require, we looked at the approximately 2000 scripts in "Chapter 1" of the campaign that shipped with NWN to find recurring patterns of control-flow structure, patterns of variable use, and any often-used set of scripts. Most scripts are only a few lines long. We also looked at the questions asked by non-programmers in the NWN scripting web forum [8] hosted by BioWare to determine what kinds of conceptual difficulties people have with programming. The typical conceptual and practical roadblocks were problems like:

- What is a variable?
- What kind of brackets should I use?
- Why doesn't my script compile?

There are many others as well. A typical approach by a non-programmer attempting to write a script is to request script fragments on the forum or to browse through libraries of scripts that are all but finished and only require customizing a few parameters. A non-programmer who is trying hard can often identify the parts of a script that need to be changed, but if a combination of portions of different scripts is required to get a custom set of effects the task may become too difficult.

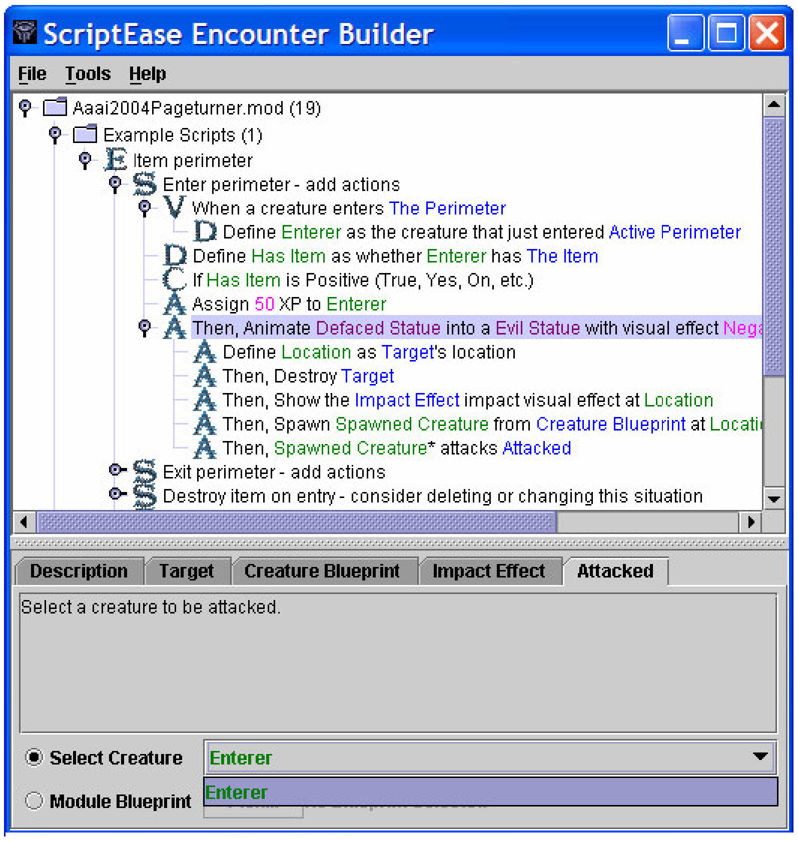
We saw that the requests for scripts by non-programmers could usually be filled by scripts having a few simple properties:

- They never need to assign a variable one value and then assign it a different value later.
- If the control path branches at a conditional, one branch usually exits immediately.
- If the branch doesn't exit immediately, it doesn't rejoin the other path until the end of the script.
- Loops are rarely needed. When they are, they typically iterate over all game objects satisfying a simple predicate, then either: stop upon finding the first one satisfying some further predicate, or perform a simple action on all of them. Nested loops are very rare.

What users need are small code fragments requiring some customization by different parameters, and to know where to put the fragment. We observed that when knowledgeable programmers respond to requests for help, they typically do so with insufficient detail for a non-programmer. For example, they may give just a fragment of a script, or a just a function name, each requiring the recipient to customize it appropriately by substituting parameters or knowing how to invoke the function with the arguments in the correct order.

Our major contribution with ScriptEase is the formalization of this knowledge transfer process. ScriptEase allows programmers to package their knowledge into code fragments that formally describe their data and control inputs and outputs. For the user who just wants a simple script to do a simple task, this is more powerful than any amount of clear commenting and prose exposition. ScriptEase makes it impossible for the end user to make a syntax error. Only syntactically correct statements can be made. Put another way, ScriptEase enables encapsulation of code at very fine levels of abstraction.

We've come up with important functional requirements for the user interface (UI), but in this paper our focus is the model of reusable AI code fragments, not UI implementation. Briefly, the concepts of variables, and parameter aliasing are tricky. Models of AI code fragment reusability exist [9]. These models depend on emergent properties of the behavior model to make convincing backdrops to the story, and to fill in the space with "extras"



```
ScriptEase Encounter Builder
File  Tools  Help
Aaai2004Pageturner.mod (19)
  Example Scripts (1)
    Item perimeter
      Enter perimeter - add actions
        When a creature enters The Perimeter
          Define Enterer as the creature that just entered Active Perimeter
        Define Has Item as whether Enterer has The Item
        If Has Item is Positive (True, Yes, On, etc.)
        Assign 50 XP to Enterer
        Then, Animate Defaced Statue into a Evil Statue with visual effect Nega
          Define Location as Target's location
          Then, Destroy Target
          Then, Show the Impact Effect impact visual effect at Location
          Then, Spawn Spawned Creature from Creature Blueprint at Locati
          Then, Spawned Creature* attacks Attacked
      Exit perimeter - add actions
      Destroy item on entry - consider deleting or changing this situation

Description | Target | Creature Blueprint | Impact Effect | Attacked
Select a creature to be attacked.
(●) Select Creature   Enterer
( ) Module Blueprint  Enterer
```

Figure 1 - Main ScriptEase Window

in movie parlance. This is not good enough to carry a story. Our challenge is to integrate AI with the plot constraints of an interactive story in a tool usable by non-programmers. Believable NPCs can have a wide variety of behaviors. However, when a game designer's desires for how the story should unfold are taken into account, the challenge of integrating AI with story requirements is difficult. ScriptEase attempts to meet this challenge.

The user builds the module in the Aurora toolset by picking objects from a palette and painting them on the ground. Once all of the objects have been created and placed, the user closes the toolset and starts ScriptEase.

Structure of ScriptEase

The traditional script is mirrored by the *Situation* in ScriptEase. A Situation is composed of components called *Atoms*, which are encapsulated pieces of scripting code. ScriptEase constrains the user to combine Atoms in syntactically valid ways. ScriptEase has five types of Atom:

- **Event atoms**, describing the types of events in the game to which scripts may fire in response.
- **Action atoms**, which wrap code that can be used to change the game world.
- **Definition atoms**, which wrap code for gathering information about the game world.
- **Condition atoms**, which use information about the game world to direct the control flow of the Situation.
- **Option atoms**, which encapsulate enumeration types in the game engine API.

Figure 1 contains a Situation, marked with an "S" icon and labelled "Enter perimeter - add actions". The Situation is composed of a linear sequence of executable atoms.

- An Event atom which says when the Situation may execute, and introduces relevant game entities to the scope. For example, an event may fire a script whenever any creature enters a polygonal area (called a perimeter). In the example of Figure 1, the event is the node marked with a V icon, "When a creature enters The Perimeter". The event provides the rest of the Situation with information like the identities of the Active Perimeter and the creature that entered it. These take the form of Definition atoms, described next.
- An optional sequence of Definition atoms gathering more information about the game world. In the example, the event contains its own

read-only Definition, and the user has added another one, “Define Has Item…”, right after the event.

- An optional sequence of Condition atoms performing Boolean tests on the information gathered by previous atoms. Tests with a false result cause the Situation to stop executing.
- An optional sequence of Actions that manipulate the game world. The current version of ScriptEase has simple Action atoms and Complex Actions. A Complex Action is a parameterized sequence of other actions. It can be manipulated just like an atom, or expanded, and its internals modified by a non-programmer, unlike an action atom. An Action atom may introduce game entities into the scope. For example, an action that spawns a new creature in the game can make it available as a parameter to the actions following.

The Situation has a linear control flow starting at the Event and flowing down through the Definitions, Conditions, and Actions. Note that a Definition is just a special case of an Action that must introduce an entity and is prohibited from having side effects in the game (though there is no mechanism for enforcing this). We chose to restrict the Situation to a single linear flow of control not only for simplicity of implementation in the UI but also because more complex flow is confusing for the audience we are trying to reach.

Now we discuss the structure of atoms in detail before moving on to Encounter Patterns. The Action atom has:

- A name which briefly tells what the Action does. This name is displayed in the menu of available actions when the user is building a script.
- The entities that the action introduces to its scope, either by creating them or retrieving handles to them, with their default label strings.
- A list of named parameters and their types. These are one-to-one with the parameters to the underlying function call.
- A "specific description" which contains text substitution codes that are replaced with actual parameter values of the action. For example, an action atom called "Assign XP to a creature" has two parameters: the creature to receive the XP, and the quantity of XP assigned. The specific description is "Assign <p2> to <p1>". In Figure 1, we can see that the specialized description of this instance of the atom is "Assign 50 XP to Enterer”.
- The function name and scripting code body that actually implements the action's semantics in the game engine's scripting language.

The other atom types, barring Option atoms, have a similar set of properties. Atoms are constructed by a skilled programmer. Option atoms are minor pieces of glue, being just a list of the underlying code symbols and English descriptions of same.

ScriptEase patterns are offered in two forms, the Encounter Pattern and the Complex Action. The Complex Action contains a sequence of actions and a list of parameters available to those actions. After instantiating a Complex Action, the user can set the parameters and edit the action list however desired. Figure 1 shows a Complex Action labelled "Then, Animate Defaced Statue..." that the user has added to the script. The top-level node for the Complex Action (CA) is shown selected, and the tabs in the bottom pane are properties for the CA. The "Description" tab contains documentation on the purpose and use of this CA, and the remaining tabs are parameters to the CA. This CA turns a "placeable" ("Target") (in NWN, essentially a piece of furniture like a chest or a statue) into a creature ("Creature Blueprint") while showing a flashy visual effect ("Impact Effect") and makes it attack someone ("Attacked"). In Figure 1 the *Attacked* parameter tab is selected. The user can choose parameter values from two sources. "Module Blueprint" allows the user to select a statically placed entity in the game module. "Select Creature" has the user select a game object introduced dynamically by a preceding atom. Here, the only creature in scope is "Enterer", introduced by the Event as the creature that crossed “The Perimeter”. “The Perimeter” is a parameter of the "Item perimeter" Encounter pattern, the node marked with an "E" icon above the Situation node.

The Encounter Pattern is similar in spirit to the Complex Action. Like the Complex Action, it has a list of parameters available to its components. The difference is that it has a list of Situations rather than Actions. After instantiation, the parameters can likewise be set and the Situations edited however desired. Encounter Patterns let the user quickly fill out the details of common scripted interactions that recur through RPGs. These recurring patterns are not limited to NWN. For example:

- “Spell request conversation” lets the user quickly script a conversation that allows the PC to request spell casting services from an NPC.
- “Automatically close and lock door” generates scripts for a door that closes and locks itself.
- “Barrier perimeter” generates scripts for an invisible force field.

Encounter Patterns have options for common variants and can be augmented with functionality unforeseen by the designer.

An advanced non-programmer can create new Encounter and Complex Action Patterns using the ScriptEase Encounter Designer (not shown), a graphical tool similar to the Encounter Builder of Figure 1. The Encounter's structure of Situations and Atoms is built using the same editing menus. The difference is a panel that allows the user to create the Encounter's parameter list.

The inspiration for ScriptEase's design and its patterns draws from generative pattern-based programming and prototype-based object systems such as CO_2P_3S [11]. A pattern is a parameterized code artifact that can be instantiated. The instances can be customized but retain the links to the original parameters so that parameter values can be changed without discarding previous customizations.

Usability Assessment

Before we released ScriptEase to the public on November 19, 2003, we performed a small user study to assess how practical it is as a CRPG scripting tool. We re-implemented a part of an area known as The Temple Ruins from the CRPG Baldur's Gate 2 [10]. We chose this area because it is in a commercial CRPG other than NWN, indicating that ScriptEase can be used to script situations independent of a particular game, although the generated code is specific to NWN. Second, this area contains several interesting encounters. By specifying all of them in ScriptEase, we demonstrated its flexibility as a scripting tool. We hired a high-school student to use ScriptEase for two 1-week periods. He built neither his own atoms nor his own patterns. He was familiar with NWN and the Aurora Toolset, but not NWScript. After a week of beta-testing ScriptEase, we requested that he re-create the Temple Ruins using ScriptEase. He spent a little over one day, mostly taken up by one complex conversation involving several riddles with a Sphinx-like character. Not including this conversation, the tester took approximately three hours to script the area and debug it by play-testing it. His implementation contained approximately 30 Situations, both stand-alone and as part of Encounter Patterns. One of the ScriptEase authors implemented the same area by hand using NWScript. It required over 700 lines of code, and despite being expert in the use of NWScript he spent three days writing and debugging. Closer to our release, a professional writer and non-programmer hired to write a tutorial for the public release scripted the tutorial's example module herself and reported it was easy.

Between its release on Nov. 19 2003, and Feb 3, 2004, ScriptEase was downloaded approximately 5000 times. Reaction from the community has been positive. The BioWare forum topics on ScriptEase contain 110 posts. Most misgivings are related to fixable problems of the user interface and not the fundamental programming model of patterns and template code.

Future Work

We want to increase the expressiveness of ScriptEase in terms of control flow and data structures, and develop a more accessible, iconic interface. The next major research direction is in building deeper AI behaviors. We did some initial conceptual work on this [12] and subsequently turned back to strengthen the foundations of ScriptEase at the Encounter level.

ScriptEase is presently written in 38 000 lines of Java, including comments and white space. Of those, 5000 lines are in interfaces to NWN file formats, 7000 lines in generic helper classes, and the remainder is ScriptEase-specific code generation, UI, script representation, and miscellaneous parts. This is a very large project for an RPG developer to take on for its users. For ScriptEase to be useful to the game development community it would have to be released in source form and structured to allow interfaces to other scripting languages to be bolted on. We designed ScriptEase with this in mind.

Conclusions

This paper presented ScriptEase, a scripting tool for computer role-playing games. We are not proposing the ScriptEase model as viable for serious systems programming. However, we do propose its template-based script generation model as appropriate for knowledge-intensive automation tasks. We have learned from our user community that some of the design decisions can be improved. The design of ScriptEase was driven by the needs of non-programmers. There is a need to give NPCs believable behaviors that can interact with plot constraints. We are growing ScriptEase towards satisfying this need. Behavior patterns are the obvious next step for virtual-world inhabitants who must tread lightly on CPU and memory, but at the same time be unique and interesting.

References

[1] MacroMedia, Inc. *Authorware*, (http://www.macromedia.com/software/authorware/)

[2] Micromagic/ Strategic Simulations Inc., 1993, *Unlimited Adventures*

[4] Tufts CEEO, LEGO DACTA, National Instruments, *RoboLab*, (http://www.ceeo.tufts.edu/graphics/robolab.html)

[5] Blizzard Entertainment, Warcraft III, (http://www.blizzard.com/war3)

[6] BioWare Corp., Nov. 19 2003, http://nwn.bioware.com/archive/nwwed.html

[7] BioWare Corp., 2001, *Neverwinter Nights* (http://nwn.bioware.com)

[8] Builders – NWN Scripting (http://nwn.bioware.com/forums/viewforum.html?forum=47)

[9] Massive Animation Software, *Massive* (http://massivesoftware.com)

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

[11] 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)

[12] Matthew 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/~script/scripteasenwn.html)