1ST INTERNATIONAL NORTH-AMERICAN CONFERENCE ON INTELLIGENT GAMES AND SIMULATION

GAMEON-NA 2005

EDITED BY

Hans Vangheluwe

and

Clark Verbrugge

AUGUST 22-23, 2005

McGILL UNIVERSITY
MONTREAL, CANADA

A Publication of EUROSIS-ETI

Printed in Ghent, Belgium

Original cover art produced by David Levy, Ubisoft, Montreal, Canada

1ST International North-American Conference
on
Intelligent Games and Simulation

MONTREAL, CANADA
AUGUST 22-23, 2005

Organised by
EUROSIS

Co-Sponsored by

Ghent University

GR@M

UBISOFT

Larian Studios

GAME-PIPE

Hosted by

McGill University

Montreal, Canada

EXECUTIVE EDITOR

PHILIPPE GERIL
(BELGIUM)

EDITORS

General Conference Chair
Hans Vangheluwe
McGill University
School of Computer Science
Montreal, Canada

General Conference Chair
Clark Verbrugge
McGill University
School of Computer Science
Montreal, Canada

PROGRAMME COMMITTEE

Abdennour El Rhalibi, Computing and Mathematical Sciences, Liverpool John Moore University, United Kingdom
Adam Szarowicz, School of Computing and Information Systems, Kingston University, United Kingdom
Alice Leung, BBN Technologies, USA
Chris Darken, Dept. of Comp. Sci., Naval Postgraduate School, USA
Christian Bauckhage, Center for Vision Research, York University, USA
Christian Reimann, C-LAB, Universität Paderborn, Germany
Christian Thurau, Applied Comp. Sci., Universität Bielefeld, Germany
Christos Bouras, Computer Engineering and Informatics, University of Patras and RACTI, Greece
Dottie Agger-Gupta, Human and Organization Development, Fielding Graduate University, USA
Ian Marshall, Engineering, Mathematical & Information Science, Coventry University, United Kingdom
Ingo Steinhaeuser, , Binary Illusions, Braunschweig, Germany
Javien Marin, Electronic Digital Systems, Universidad de Malaga, Spain
Jorg Kienzle, School of Computer Science, McGill University, Canada
João Tavares, Dept. of Engineering, Universidade do Porto, Portugal
Leon Rothkrantz, Data and Knowledge Engineering, Delft University of Technology, The Netherlands
Leon Smalov, Digital Entertainment and Creativity Department, Coventry University, United Kingdom
Maja Pivec, Learning and Knowledge-based Systems, Institute for Information Technology, Austria
Marco Gillies, Dept. of Computer Science, University College London, United Kingdom
Marco Roccetti, Computer Science, University of Bologna, Italy
Marcos Rodrigues, Materials and Engineering Research Institute, Sheffield Hallam University, United Kingdom
Mark Riedl, Institute for Creative Technologies, University of Southern California, USA
Markus Koch, C-Lab, Universität Paderborn, Germany
Michael Young, Center for Digital Entertainment, NC State University, USA
Michael Zyda, School of Engineering's GamePipe Laboratory, USC Viterbi, USA
Mike Katchabaw, Comp. Sci., University of Western Ontario, Canada
Oliver Lemon, School of Informatics, Edinburgh University, United Kimgdom
Olli Leino, Media Studies, U. of Lapland, Finland
Oryal Tanir, Knowledge Engineering and Simulation, Bell Canada
Paolo Remagnino, Digital Imaging Research Center, Kingston University, United Kingdom
Pedro Demasi, UFRI, Rio de Janeiro, Brazil
Robert Askwith, Network Security, Liverpool John Moore University, United Kingdom
Robert Zubek, Comp. Sci., Northwestern University, USA
Ruck Thawonmas, Department of Human and Computer Intelligence, Ritsumeikan University, Japan
Stephane Assadourian, UBISOFT, Montreal, Canada
Stephen McGlinchey, Artificial Neural Network Research Group, University of Paisley, United Kingdom
Sue Greenwood, Department of Computing, Oxford Brookes University, United Kingdom
Tina Wilson, Coventry University, United Kingdom
Victor Bassilious, Division of Software Engineering, University of Abertay-Dundee, United Kingdom
Volker Paelke, Dept. of Computer Science, University of Hannover, Germany
William Swartout, Information Sciences Institute, University of Southern California, USA
Yoshihiro Okada, Department of Informatics, Kyushu University, Japan

LOCAL ORGANISING COMMITTEE

Marc Lanctot, Local Chair, gr@m

Felix Martineau, Local Organizer, gr@m

Miriam Zia, Social Activities Organizer, MSDL

Alexandre Denault, Publicity Helper, gr@m

Gregory Paull, Remote/Commercial Contact and Publicity Person, Secret Level

Special thanks to:

Stephane Assadourian and David Levy, Ubisoft, Montreal, Canada

For their support to this conference.

EUROSIS is a Division of ETI Bvba, The European Technology Institute, Torhoutsesteenweg 162, Box 4, B-8400 Ostend, Belgium

All EUROSIS-EIT publications are indexed

EUROSIS-ETI Publication

ISBN: 90-77381-19-8

GAME'ON-NA
2005

Preface

Welcome to Game-On 'NA 2005, the North American premiere of the well-established European Game-On conference series on AI and simulation in computer games. Montreal is a highly appropriate location to start off Game-On 'NA. With several universities, the presence of many key computer game companies, and of course Montreal's creative ``bouillon de culture,'' the environment clearly reflects the growing importance of computer games, now a major part of our social and economic environment.

Academic interest in computer games has been growing for some time. Creating and managing large, increasingly complex game projects poses many practical and research challenges. Academic contributions have a lot to offer in this area where pragmatic approaches have been ``de rigeur'' for many years. The works presented here covers a wide variety of efforts to both improve specific parts or aspects of game design or implementation, and to define game problems and the associated game research issues.

Core game concerns, such as game AI and content-generation are continuing problems for industry. Several papers and more than half of the invited talks focus on these issues, and provide research and results based on Bayesian learning [Bauckhage et al.], realistic NPC reactions and behaviour [Gruenwoldt et al.], and the design of a high-level pattern catalog for game scripting [Onuczko et al.]. Other papers concentrate on building a basic research infrastructure for analysis and exploration. These include environments for examining automatic game difficulty adjustment [Bailey and Katchabaw], real-time strategy game design [Buro and Furtak], massively-multiplayer game research [``Mammoth'' workshop], and the design and verification of computer narratives [Pickett et al.]. Automatic generation of game physics from Modelica models [Vangheluwe and Kienzle] demonstrates the introduction of techniques used in real-time simulation of mechatronic systems. Student sessions further provide a field study of the use of games in military training [McDonough and Strom], and an analysis of the concept of immersion in games [Arsenault].

As well as peer-reviewed papers, Game-On 'NA 2005 features a number of invited talks by both academic and industry leaders. Paul Kruszewski from BioGraphics Technologies and Gregory Paull from Secret Level give industry viewpoints on crowd simulation and cover-finding strategy AI respectively, Duane Szafron from the University of Alberta's GAMES group discusses behaviour patterns, and Nicholas Graham from Queen's University gives some insights into issues in teaching game development at the university level.

Game-On 'NA 2005 is also about making contacts in the computer game research community. Several social events are planned, including an opening party, a conference dinner, and a tour of Ubisoft's Montreal studios. We hope you find your time at this first Game-On 'NA productive and enjoyable.

Hans Vangheluwe and Clark Verbrugge

X

CONTENTS

SCIENTIFIC PROGRAMME

Is Bayesian Imitation Learning the Route to Believable Gamebots?

Christian Thurau, Tobias Paczian
Applied Computer Science
Bielefeld University
P.O. Box 100131, 33501 Bielefeld, Germany
{cthurau,tpaczian}@techfak.uni-bielefeld.de

Christian Bauckhage
Centre for Vision Research
York University
4700 Keele St, Toronto, ON, M3J 1P3, Canada
bauckhag@cs.yorku.ca

ABSTRACT

As it strives to imitate observably successful actions, imitation learning allows for a quick acquisition of proven behaviors. Recent work from psychology and robotics suggests that Bayesian probability theory provides a mathematical framework for imitation learning. In this paper, we investigate the use of Bayesian imitation learning in realizing more life-like computer game characters. Following our general strategy of analyzing the network traffic of multi-player online games, we will present experiments in automatic imitation of behaviors contained in human generated data. Our results show that the Bayesian framework indeed leads to game agent behavior that appears very much human-like.

Introduction

Statistical machine learning is gaining more and more attention as a means of enhancing game AI. While the majority of contributions considers supervised learning or reinforcement learning, the paradigm of *imitation learning* has drawn little attention so far. However, learning through imitation is a powerful mechanism for the acquisition of behaviors. Since it avoids tedious and futile trial and error strategies and does not require labeled training data, imitation might be a short cut on the route to more engaging artificial game agents.

In a recent contribution, Rao, Shon & Meltzoff (2004) introduced a Bayesian model of imitation learning for applications in robotics. Based on experiments in developmental psychology, their probabilistic framework models four stages of imitative abilities that were observed in infant behavior. Since a basic understanding of these stages helps grasping the approach discussed below, we shall summarize them briefly: By means of postnatal *body babbling* infants acquire a model of their body. They learn which muscular actions lead to what kind of limb configurations and thus acquire a vocabulary of useful *motor primitives*. This enables the *imitation of body movements* where infants map observed actions onto their own body. At the age of several weeks, for instance, they can mimic facial expressions they have never seen before. In a third stage, infants start *imitating actions on physical objects* such as toys which are external to their body. By the time they are 1.5 years old, infants are experienced in interacting with other humans. Consequently, they can acquire models of agents with intentions. Forward models alow them to *infer the goals of an agent* even if they only observe unsuccessful demonstrations; inverse models are used to select motor commands that will achieve undemonstrated but inferred goals.

In this paper, we will describe how similar mechanisms embedded in a Bayesian framework can produce more life-like computer game agents (henceforth called *gamebots*). Using the game QUAKE II® as a platform for our research, we investigate behavior acquisition from imitating how human players steer their avatars through 3D game worlds (often called *maps*). Since QUAKE II® represents the arguably most popular genre of First-Person-Shooter games, a player's overall goal is to score as many points as possible by shooting enemy players. Situation dependent behaviors and subgoals arise from the current game context. Factors that influence context dependent behavior are the internal state of a player's character, the behavior of opponents, as well as various items placed all over the map. Low avatar energy, for instance, might be compensated by picking up armor or health packages.

Within the limits imposed by the game physics, games like QUAKE II® allow for all kinds of movements, strategies and cunning. As this kind of *intelligent* gameplay requires some sort of cognitive capabilities, it comes with little surprise that gamebots which would behave truly human-like are still not available. In the next section, we will discuss this problem in more depth and shall roughly survey related work. Afterwards, we will discuss our approach to Baysian imitation learning of human-like behavior. As we shall see, our contribution is fourfold: (i) we apply concepts from the theory of edge reinforced random walks to impose an adequate topology on the space of internal states of game agents; (ii) we make use of clustering methods from statistical machine learning to acquire a vocabulary of action primitives for game agents; (iii) we extend the model of Rao et al. (2004) in order to guarantee the temporal coherency of the actions selected for game characters; (iv) we integrate these three techniques within a Bayesian learning framework. In the fourth section, we will present and discuss first experimental results which underline that this framework indeed provides an auspicious avenue to believable gamebots. Finally, a conclusion will close this contribution.

Related Work

It is a widely accepted notion these days that life-like behaving gamebots are the next big step towards ever more exciting game experience. This holds even in the age of online gaming where human players engage each other in virtual battle over the Internet. The nature of popular genres such as massively multiplayer online role playing games (MMORPGs) simply calls for non-player characters to set forth the action.

However, when it comes to gamebot control, the gaming industry still mostly relies on seasoned deliberative AI techniques like finite state machines or the A* algorithm. While A*-search was introduced almost four decades ago (Hart, Nilsson & Raphael 1968), output generating finite state machines date back even further (Mealy 1955, Moore 1956). Of course, the problem with these techniques is not their maturity but rather their lack of life-likeness. Once human players get used to a game, common gamebots are perceived to miss the element of surprise human opponents would provide; they appear to behave 'dumb' (Cass 2002).

The picture could be different, if gamebots were to learn from experienced human players. Work in this direction was first reported by Sklar, Blair, Funes & Pollack (1999) who collected the key-strokes of people playing *Tron* in order to train neural network based game agents. Neural networks also proved to perform satisfiable in First-Person-Shooter games. Trained on the data contained in the network traffic of multiplayer games, neural architectures learned different aspects of engaging gameplay. They were shown to reproduce convincing *reactive* behavior (Bauckhage, Thurau & Sagerer 2003, Thurau, Bauckhage & Sagerer 2003) as well as *tactical* decisions such as context aware weapon selection (Bauckhage & Thurau 2004).

Other machine learning techniques have been applied as well: Spronck, Sprinkhuizen-Kuyper & Postma (2003) incorporate reinforcement learning into rule selection for agent behavior in a role playing game and Le Hy, Arrigioni, Bessière & Lebeltel (2004) describe action selection for a commercial game using Baysian networks which were trained by means of human generated input.

Next, we will present a different approach which underlines that –given suitably preprocessed data– even simple Bayesian statistics provides a powerful tool for game AI programming.

Bayesian Imitation Learning

In earlier work (Thurau, Bauckhage & Sagerer 2004), we realized movement behaviors for QUAKE II® gamebots using a straightforward, probabilistic model $P(a_t|s_t)$. The values of the random variable s_t are given by vectors that encode the state of an agent and its surroundings at time t. The values of the random variable a_t denote the most appropriate (re)action. The discrete sets of possible state vectors $S = \{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_M\}$ and action vectors $A = \{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_P\}$ result from clustering the data contained in recordings of

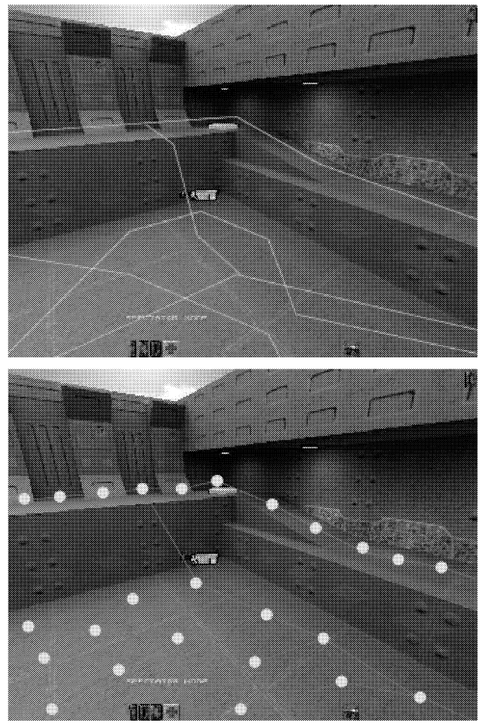


Figure 1: If applied to data representing the paths a player did run during a match, Neural Gas clustering will result in a structured set of waypoints which captures the topology of the corresponding map.

matches played by human players. Although goal orientation was not explicitly modeled in this approach, it yielded movement sequences that human players identified as goal driven behavior. Still, the movements lacked strategic sophistication and were convincing only in certain constrained situations.

The Bayesian model of imitation learning proposed by Rao et al. (2004) also accounts for goals and subgoals. Consequently it can be expected to avoid the shortcomings of our initial attempts on probabilistic behavior modeling. Within this Bayesian framework, the probability for the execution of an action $\mathbf{a}_i$ at time step t depends on the current state s_t, the next desired state (or subgoal) s_{t+1} and the overall goal state s_g. Using Bayes' theorem, it amounts to

$$\begin{aligned} &P(a_t = \mathbf{a}_i | s_t = \mathbf{s}_i, s_{t+1} = \mathbf{s}_j, s_g = \mathbf{s}_k) \\ &= \frac{1}{C} P(s_{t+1} = \mathbf{s}_j | s_t = \mathbf{s}_i, a_t = \mathbf{a}_i) \, P(a_t = \mathbf{a}_i | s_t = \mathbf{s}_i, s_g = \mathbf{s}_k) \end{aligned}$$

where the normalization constant C results from marginalizing over all possible actions:

$$\begin{aligned} C &= P(s_{t+1} = \mathbf{s}_j | s_t = \mathbf{s}_i, s_g = \mathbf{s}_k) \\ &= \sum_m P(s_{t+1} = \mathbf{s}_j | s_t = \mathbf{s}_i, a_t = \mathbf{a}_m) P(a_t = \mathbf{a}_m | s_t = \mathbf{s}_i, s_g = \mathbf{s}_k). \end{aligned}$$

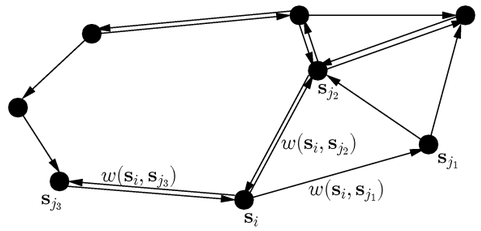


Figure 2: Didactic example of a state transition graph.

In order to obtain reasonable probabilities for the selection of an action a_t, the desired subgoal s_{t+1} and the global goal s_g need to be known. Note, however, that in the game domain winning the game is the only true global goal. Also note that interpreting successive states as subgoals does not alter the model but eliminates the need of mining for subgoals in the training data. Indeed, experienced human players act reasonable and with implicit subgoals in mind. Therefore, reasonable goal states will emerge from clustering the network traffic of recorded matches and automatically provide a game agent with suitable choices[1].

A Structured State Space

State vectors $\mathbf{s}_i$ should contain a problem specific description of a gamebot's internal state and the surrounding game-world. For instance, if imitation learning is applied to simple classical maze problems (i.e. finding goal directed paths in a maze world), recording a player's position vectors $\mathbf{p}_t = [x_t, y_t, z_t]^T, t = t_1, \ldots, t_{\text{end}}$ provides all necessary information.

In order to derive a discrete approximation of the state space, we cluster the state vectors recorded from the network traffic using Neural Gas clustering, a technique introduced by Martinetz, Berkovich & Schulten (1993). Since we may expect to find certain topologies within the state space (see Fig. 1), Neural Gas clustering is well suited for our problem because it is known to yield superior results in recovering topological structures of a dataset.

Knowledge of state space structure may facilitate computing some of the probabilities required in the Bayesian framework. In order to provide the resulting discrete state space with a useful structure, we therefore compute a state transition graph. In this graph, a directed edge between two state prototypes indicates a possible state transition. Moreover, edges are labeled with transition counts (see Fig. 2). As the idea for the transition graph was inspired by the theory of *edge reinforced random walks* known from statistics (Diaconis 1988), we use the term weights when referring to state transition counts. Formally, the transition graph is thus given as a triple $G = (V, E, w)$ where $V = \{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_M\}$ is a set of vertices, $E \subseteq V \times V$ is a set of directed edges and $w : E \to \mathbb{R}^+$ is a function assigning weights to the edges. Edges are drawn based on state transitions observed in the training data; i.e. we have

$$(\mathbf{s}_i, \mathbf{s}_j) \in E \iff \exists s_t, s_{t+1} : s_t = \mathbf{s}_i \wedge s_{t+1} = \mathbf{s}_j \tag{1}$$

Transition counts are recovered from traversal frequencies in the training data; the weight w of an edge $(\mathbf{s}_i, \mathbf{s}_j)$ corresponds to the number of times a human players was observed to move from state space position $\mathbf{s}_i$ to position $\mathbf{s}_j$.

Given this weighting scheme, suitable upcoming (sub)goals for Bayesian behavior synthesis can be determined either from a roulette wheel or a maximum a-posteriori selection over the state transition probabilities

$$P(s_{t+1} = \mathbf{s}_j | s_t = \mathbf{s}_i) = \frac{w(\mathbf{s}_i, \mathbf{s}_j)}{\sum_k w(\mathbf{s}_i, \mathbf{s}_k)} \tag{2}$$

Imposing a discrete state space structure like this provides certain advantages over an unstructured collection of prototypes. For instance, limiting the choice for a successor to those states that are connected to the current state considerably lowers the computation time but still allows for a reconstruction of all observed state sequences[2]. What is left is determining an adequate discrete set of motor or movement primitives $\mathbf{a}_t$ to generate an action dependent forward model $P(s_{t+1} | s_t, a_t)$.

Finding Movement Primitives

From the point of view of a gamebot, the only way to traverse the state transition graph lies in choosing appropriate actions. For instance, if the next desired state would correspond to a position direct in front of the gamebot, it simply had to move forward in order to reach that state.

The set of available actions is given by *movement primitives*, prototypical actions, which are extracted from recorded matches. For the experiments, presented below, a movement primitive is a 5 dimensional vector

$$\mathbf{a} = \begin{bmatrix} \text{yaw angle} \\ \text{pith angle} \\ \text{forward velocity} \\ \text{sideward velocity} \\ \text{upward velocity} \end{bmatrix}$$

Prototypic movement primitives result form applying k-means clustering to the given movements of human players. We found a number of $k \in [150, \ldots, 200]$ movement primitives sufficient for synthesizing smooth looking motions.

In the context of Bayesian behavior learning, we are interested in the effects of movement primitives on state transitions. This will provide a forward model for choosing movements in order to reach specific subgoal states. However,

[1] The resulting (sub)goals are of course highly player and playing style dependent.

[2] Obviously, some state transitions might become impossible for the game agent just because no human player was observed performing the state traversal. However, this is what human-like behavior is all about.

not all actions are reasonable in any game context. Sometimes, an otherwise frequently used movement could lead into a wall or down a ledge. The forward model for action selection thus will have to depend on the current state.

Following the proposal of Rao et al. (2004), we apply the mechanism of *body-babbling* to estimate a forward model $P(s_{t+1}|a_t, s_t)$. Again, we make use of the widely available demo data.

Synthesizing Action Sequences

Human players control their avatars using mouse and keyboard. This implicitly limits their choice of actions. In QUAKE II®, for example, instant turns are impossible for a human player. As the input modalities thus constrain the sequencing of actions and as differences between movement primitives which are generated using mouse and keyboard are rather small, motions of human controlled avatars usually appear to be smooth.

In order to recreate such natural motions, we introduce conditional probabilities $P(a_t|a_{t-1})$ for the execution of movement primitives at time t. Again, these can be learned from observing human players. Since the action vector that was executed last can be seen as part of the world state vector, we extend the model of Rao et al. (2004) by introducing a_{t-1} into the equations. Assuming independence of a_{t-1}, s_t and s_g (after all, physical limitations should not affect environmental conditions), this results in the following model:

$$P(a_t = \mathbf{a}_i|s_t = \mathbf{s}_i, s_g = \mathbf{s}_k, a_{t-1} = \mathbf{a}_j)$$
$$= \frac{1}{C} \frac{P(a_t = \mathbf{a}_i|s_t = \mathbf{s}_i, s_g = \mathbf{s}_k)P(a_t = \mathbf{a}_i|a_{t-1} = \mathbf{a}_j)}{P(a_t = \mathbf{a}_i)}$$

where the normalization constant C has to be adapted to the new variables a_{t-1}:

$$\begin{aligned} C =& P(s_{t+1} = \mathbf{s}_j|s_t = \mathbf{s}_i, s_g = \mathbf{s}_k) \\ =& \sum_m P(s_{t+1} = \mathbf{s}_j|s_t = \mathbf{s}_i, a_t = \mathbf{a}_m) \\ & P(a_t = \mathbf{a}_m|s_t = \mathbf{s}_i, s_g = \mathbf{s}_k, a_{t-1} = \mathbf{a}_j) \\ =& \sum_m P(s_{t+1} = \mathbf{s}_j|s_t = \mathbf{s}_i, a_t = \mathbf{a}_m) \\ & \frac{P(a_t = \mathbf{a}_m|s_t = \mathbf{s}_i, s_g = \mathbf{s}_k)P(a_t = \mathbf{a}_m|a_{t-1} = \mathbf{a}_j)}{P(a_t = \mathbf{a}_i)} \end{aligned}$$

Finally, given a current game state and a desired goal state, the conditional probability for the execution of a movement primitive $\mathbf{a}_i$ can be written as follows:

$$\begin{aligned} P(a_t =& \mathbf{a}_i|s_t = \mathbf{s}_i, s_{t+1} = \mathbf{s}_j, s_g = \mathbf{s}_k, a_{t-1} = \mathbf{a}_k) \\ =& \frac{1}{C} P(s_{t+1} = \mathbf{s}_j|s_t = \mathbf{s}_i, a_t = \mathbf{a}_i) \\ & \frac{P(a_t = \mathbf{a}_i|s_t = \mathbf{s}_i, s_g = \mathbf{s}_k)P(a_t = \mathbf{a}_i|a_{t-1} = \mathbf{a}_j)}{P(a_t = \mathbf{a}_i)} \end{aligned}$$

Next, we will present experiments which indicate that this Bayesian scheme for goal oriented action selection indeed results in gamebot behavior that appears to be human-like.

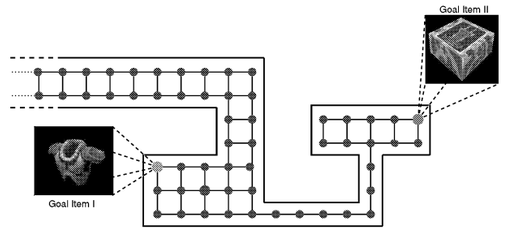


Figure 3: Schematic representation of an extended simple maze task. Two points in the waypoint map that results from clustering recorded player movements correspond to the locations of desirable items on the map. In the our experimental setting, a subgoal of the game agent is to increase its amour value. It therefore has to devise a path trough the abstract higher dimensional state space that accords with a path through the 3D waypoint map leading to the armor item.

Experiments

In order to test the Bayesian framework and its usefulness for the game domain, we carried out a series of experiments in motion planning. The basis of all experiments was of course formed by recordings of data generated by human players. The experimental goal was to reproduced all observable movement behaviors.

Maze Problem

The first experiment basically tested the functionality of the approach. Several examples of a goal directed movement sequence were recorded; in each sequence the player's motion ended at the same map position. Due to the simplicity of the task, the state vectors we considered here only contained observed player positions x, y, z.

After connecting the agent to a game server, it was supposed to reproduce human-like movements by selecting subgoals based on the probabilities encoded in a graph model learned from the training data.

Since the agent traverses a discrete lattice of prototypical positions, the number of clustered state prototypes plays a pivotal role. Our experiments revealed that smaller maps require between 50 and 100 prototypes to allow for collision free navigation. Given this, the Baysian approach performed as expected and believable human-like movements were recreated.

Extended Maze Problem

Our second experiment considered an extended maze problem; Fig. 3 sketches its setting. The training data we considered here displayed several instances of a human player first picking up goal-item 1 (an armor) and then continuing to goal-item 2 (a better weapon).

The state space dimensionality was extended to account for the player's inventory. The additional dimensions encode

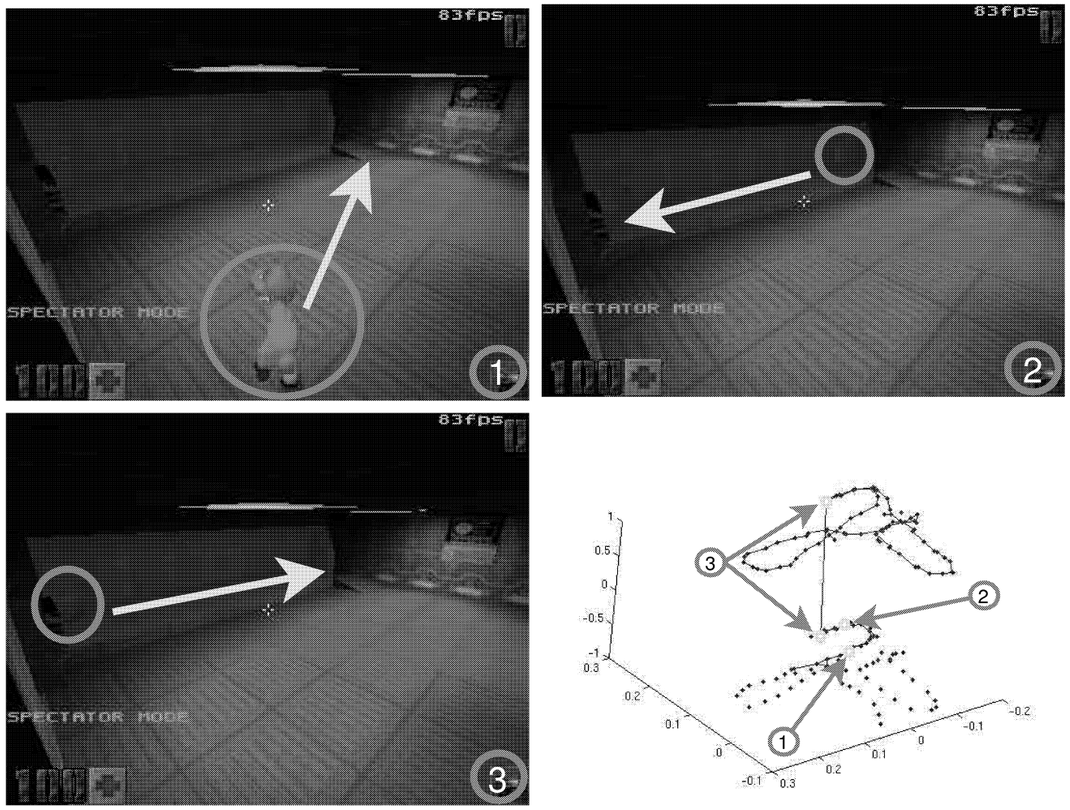


Figure 4: Screenshots showing an artificial player solving an item pickup task and visualization of the corresponding transitions in state space. The screenshots display the agent's actions in the game world; the plot in the lower right corner shows what is going on in a subspace of the state space considered in this experiment: the X and Y axes denote the gamebot's (x,y) positions, while the Z axis represents inventory item information. Movements in the 3D gameworld and resulting changes of the agent's internal state correspond to movements between nodes of the state graph. First, in (1), the agent is seen moving along the graph closer to a node whose (x,y) coordinates coincide with a goal item. In (2), the agent is strafing around the corner of a transparent wall. Finally, as seen in screenshot (3), it reaches the item and continues its way to the next item. The item pickup in (3) considerably increases the inventory value for this item and thus results in a 'jump' along the Z axis of the subspace shown here. Although, in visualizations like this, such state space discontinuities appear random to the human eye, they are not. In fact, the screenshots in this figure show but a part of a longer sequence of actions. At the beginning of this sequence, the agent determined its next suitable subgoal represented by a state s_{t+1} and the item pickup in this figure is actually a planned action to reach this graph node s_{t+1}.

information about the internal armor value and the weapon currently hold.

Given state graphs computed under these conditions, we expect to observe state sequences which reflect the order of the item pickups. In order to reach a state with higher armor values, the agent has to obtain the armor item. Since in the demo data the armor pickup precedes the weapon pickup, states of higher armor value must lie on the way through state space that leads to the overall goal state. On the map, the agent therefor has to visit the location of the armor prior to continuing to the final destination (note that state space paths must not be confused with paths in the 3D gameworld). Figure 4 shows an example of a trained gamebot trying to accomplish this task and a three dimensional projection of the corresponding state graph transitions.

Again, the number of state prototypes is crucial. Using a 5 dimensional state space and a mid-sized game-map, we found a number of 150-200 prototypes sufficient for our purpose. All in all, the observed behaviors were imitated very convincingly (see the examples of state space trajectories in

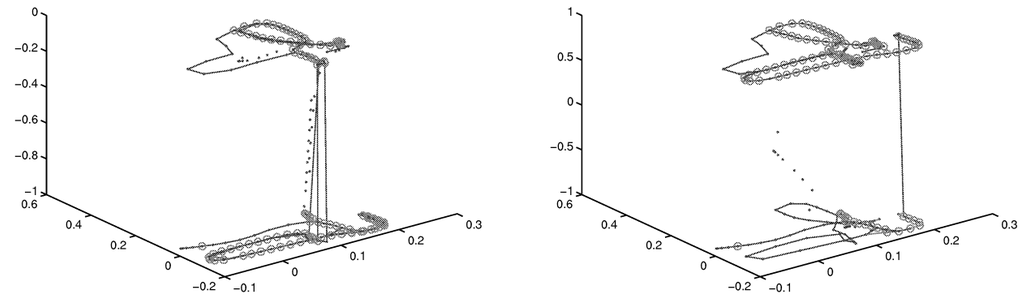


Figure 5: Examples of synthesized movements (plotted in state space coordinates). Again, the X and Y axes correspond to x and y positions of the agent whereas the Z axis denotes inventory armor values (left figure) and inventory rail gun values (right figure). The blue dots correspond to sequences of state vectors generated by human player while 'solving' the task of item cycling considered in our second experiment. Red circles indicate state prototypes reached by the artificial player.

Fig. 5). The agent managed to reach both goals in the predefined order. Additionally, the movements appeared to be smooth and showed characteristics mostly seen in human players. For instance, the agent "slides" around corners thus preserving observed player habits.

Discussion

Generating action primitives by means of training data clustering and estimating the consequences of their execution in terms of state space transitions can effectively reproduce various sorts of movement behavior observed in humans. This even holds for more complex movements such as swimming or the famous rocket-jump[3] (like simple movements, these activities are as good as any other action and might be necessary to achieve certain goals). The convincing reproduction of single complex movements already leads to a very lifelike appearance, because such movements naturally appear as if generated by human players. However, in conjunction with probabilistic action selection based on goal states this impression is further strengthened: longer sequences of individual human-like movements resulting from the Bayesian approach convey a strong impression of intelligently planned and purposeful behavior.

Of course, the number of state prototypes necessary to produce smooth, humanlike movements is map- and task-dependent. While for the simple maze problem a set of 50 to 100 state vectors was sufficient to create convincing behavior (trials with a lesser number failed to do so; adding more prototypes did not further improve the appearance of the movements), solving the extended maze problem required 150 to 200 state vectors. This raises the question, if the presented approach will scale to more complex behaviors and more demanding contexts? Up to a certain level of complexity it is in fact reasonable to assume it will: even if the dimension of state space increases, the state transition graph ensures that action selection will be based upon *local evidence*. The graph structure ensures that, even if there is a growing number of states, not all of them will have to be considered in the necessary computations. As this locality constraint is thus desirable, future work should further explore techniques for manifold learning and state space structuring. For a full blown state space dimensionality (which is inevitable for a full featured artificial game agent), however, a single state transition graph graph might not be sufficient any more. Especially the interplay of strategic, tactical and reactive behavior may not be sufficiently captured by means of a monolithic graph. A separation into several graphs each targeting the emulation of different behaviors may overcome this problem. However, while the approach presented in this paper is solely based on automatic data analysis and learning, a technique combining several state spaces will surely require expert knowledge to be introduced into the model selection mechanism. Again, this remains as a topic for further research.

Now, if state space manifold identification appears to be the predominant factor for a satisfying performance of our framework, then why, after all, consider Bayesian imitation learning? The answer is clear: without identifying a vocabulary of action primitives and recovering probabilities for their use in different contexts, traversing the state graph would reduce to a deterministic process similar to the ones produced by finite state machine. State transition based on Bayesian probability theory introduce flexibility and surprise. Compared to the usual Baysian techniques found in game AI programming, however, imitation learning as presented here has two noticeable advantages: first, incorporating (sub)goals

[3] An expert player's move, where a rocket is fired on the ground while the player jumps. It results in a much higher altitudes than reachable by ordinary jump movements.

into the process of action selection produces unpredictable short term but nevertheless goal driven long term behavior. Second, where common Baysian approaches to game AI rely on preprogrammed sets of behaviors whose a-priori probabilities are predetermined by programmers, clustering of network data provides a natural set of action primitives. Given these, *body babbling*, i.e. a training phase of random invocation of these actions in different states to learn about their effects, provides suitable estimates for the priors in life-like behavior generation.

Conclusion

Imitation learning is a powerful yet so far underexploited mechanism for behavior acquisition for game agents. In this paper, we applied a Bayesian formulation of imitation learning to game AI programming.

By means of the example of the game QUAKE II®, we could show that artificial players can convincingly imitate human movement behavior. Our results are based on a state and (sub)goal dependent model. Useful state sequences are synthesized using a transition graph that structures a codebook of state vectors extracted from the network traffic of the game. Incorporating temporal context into the probabilistic action selection mechanism leads to especially smooth and realistic movements.

Given these results, it seems that imitation provides an auspicious approach to game AI programming. Currently, we are extending the technique presented in this paper to more complex behaviors. Although the focus of the presented paper is on the reproduction of strategical, i.e. state dependent, item pickups, first experiments applying the presented approach to more tactical (movements relative to an enemy player) and reactive (close combat and shooting) behaviors are work in progress. As first results are encouraging, Bayesian imitation learning in conjunction with learned movement primitives may indeed be the route to believable, life-like gamebots.

Acknowledgments

This work was supported by the German Research Foundation (DFG) within the graduate program "Strategies & Optimization of Behavior".

REFERENCES

Bauckhage, C. & Thurau, C. (2004), Towards a Fair 'n Square Aimbot – Using Mixtures of Experts to Learn Context Aware Weapon Handling, *in* 'Proc. GAME-ON', pp. 20–24.

Bauckhage, C., Thurau, C. & Sagerer, G. (2003), Learning Human-like Opponent Behavior for Interactive Computer Games, *in* B. Michaelis & G. Krell, eds, 'Pattern Recognition', Vol. 2781 of *LNCS*, Springer-Verlag, pp. 148–155.

Cass, S. (2002), 'Mind games', *IEEE Spectrum* pp. 40–44.

Diaconis, P. (1988), Recent Progress on de Finetti's Notions of Exchangeability, *in* J. Bernardo, M. DeGroot, D. Lindley & A. Smith, eds, 'Bayesian Statistics', Vol. 3, Oxford Univ. Press, pp. 111–125.

Hart, P., Nilsson, N. & Raphael, B. (1968), 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths', *IEEE Trans. on Systems Science and Cybernetics* **4**(2), 100–107.

Le Hy, R., Arrigioni, A., Bessière, P. & Lebeltel, O. (2004), 'Teaching bayesian behaviours to video game characters', *Robotics and Autonomous Systems* **47**(2–3), 177–185.

Martinetz, T., Berkovich, S. & Schulten, K. (1993), 'Neural Gas Network for Vector Quantization and its Application to Time-Series Prediction', *IEEE Trans. on Neural Networks* **4**(4), 558–569.

Mealy, G. (1955), 'A Method for Synthesizing Sequential Circuits', *Bell System Technology J.* **34**, 1045–1079.

Moore, E. (1956), Gedanken-experiments on sequential machines, *in* 'Automata Studies', number 34 *in* 'Annals of Mathematical Studies', Princeton University Press, pp. 129–153.

Rao, R., Shon, A. & Meltzoff, A. (2004), A Bayesian Model of Imitation in Infants and Robots, *in* K. Dautenhahn & C. Nehaniv, eds, 'Imitation and Social Learning in Robots, Humans, and Animals: Behavioural, Social and Communicative Dimensions', Cambridge University Press,.

Sklar, E., Blair, A., Funes, P. & Pollack, J. (1999), Training intelligent agents using human internet data, *in* 'Proc. Asia-Pacific Conf. on Intelligent Agent Technology', pp. 354–363.

Spronck, P., Sprinkhuizen-Kuyper, I. & Postma, E. (2003), Online Adaptation of Game Opponent AI in Simulation and in Practice, *in* 'Proc. GAME-ON', pp. 93–100.

Thurau, C., Bauckhage, C. & Sagerer, G. (2003), Combining Self Organizing Maps and Multilayer Perceptrons to Learn Bot-Behavior for a Commercial Computer Game, *in* 'Proc. GAME-ON', pp. 119–123.

Thurau, C., Bauckhage, C. & Sagerer, G. (2004), Synthesizing Movements for Computer Game Characters, *in* C. Rasmussen, H. Bülthoff, M. Giese & B. Schölkopf, eds, 'Pattern Recognition', Vol. 3175 of *LNCS*, Springer-Verlag, pp. 179–186.

CREATING REACTIVE NON PLAYER CHARACTER ARTIFICIAL INTELLIGENCE IN MODERN VIDEO GAMES

Leif Gruenwoldt, Michael Katchabaw
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada
E-mail: lwgruenw@gaul.csd.uwo.ca, katchab@csd.uwo.ca

Stephen Danton
Horseplay Studios
Seattle, Washington, USA
E-mail: stephen_m_danton@hotmail.com

KEYWORDS

Reactive artificial intelligence, relationship modeling in video games, reputation systems

ABSTRACT

Realistic and reactive non player characters that respond appropriately to activity in their game world would be welcomed by game developers and players alike. Unfortunately, despite significant progress made by in this area of research, this goal has still yet to be fully achieved.

In this paper, we present a new Realistic Reaction System, based on our previous work in this area, which provides reactive artificial intelligence through the use of an underlying relationship system that binds together players, non player characters, and objects in the game world. Through proper maintenance, manipulation, and querying of the relationship system, we can effectively and efficiently augment the decision processes used in artificial intelligence controllers in games to provide a richer and more immersive experience to players.

INTRODUCTION

Game developers have long sought to have meaningful and logical interactions between human players and non player characters driven by their games' artificial intelligence. Unfortunately, except for restricted circumstances, typically violent confrontations, this has not materialized successfully, leaving the players feeling isolated or disconnected from the world in which they are playing (Laramée 2002).

A key element to overcoming this problem is the development of artificial intelligence capable of dynamically reacting to players and player actions in reasonable and realistic fashions. Doing so would require a sense of relationship or social network binding the characters and objects in the game world to one another, a sentiment expressed in (Lawson 2003) and elsewhere. Without this, developers have to rely upon static or scripted methods of implementing behaviours and events to mimic realistic character reactions, which is ultimately quite limiting.

Work in developing reputation systems for games has drawn some attention recently in an attempt to address this problem. For instance, the games Ultima Online (Grond and Hanson 1998) and Neverwinter Nights (Brockington 2003) provide reputation systems, but do so with some restrictions and drawbacks; for example, in Ultima Online, reputation changes have unrealistic immediate global effects, regardless of who witnessed the actions precipitating the changes. The work in (Alt and King 2002) shows promise, but requires more flexibility and generality; for example, it supports only a limited relationship set, does not track relationships between non player characters, and has not been applied to game world objects and complex group situations.

In (Gruenwoldt 2005), we introduced a Realistic Reaction System (RRS) for modern video games to overcome these issues. This system models and maintains the relationships between players, non player characters, and objects in the game world over time dynamically, and provides methods by which characters can query the relationship network to formulate appropriate reactions in behaviour, dialogue, and so on. While this served as an important first step in providing reactive characters in games, more work was needed to fully address the problem at hand.

Our current work builds upon this previous work from (Gruenwoldt 2005) to provide a complete and integrated solution to the above problems. In particular, in this paper, we focus on the logic and mechanisms required to link the relationship system from our previous work with artificial intelligence controllers to create reactive non player characters. Doing so effectively poses significant challenges, as a well populated world with a rich collection of relationship types can produce a relationship network large enough to overwhelm both artificial intelligence programmers and scarce game resources at runtime. This was learned first hand in integrating our previous work into the action/adventure/role-playing game Neomancer (Danton 2004; Katchabaw 2005) that we are currently co-developing. Consequently, mechanisms must be in place to help manage, filter, and aggregate relationship information appropriately according to the needs of the game in question.

This paper presents a new extended architecture for RRS for creating reactive non player character artificial intelligence, and discusses our work in implementing and using it to date. We begin with a brief overview of relationships and augmenting artificial intelligence controllers with this information. Following this, we

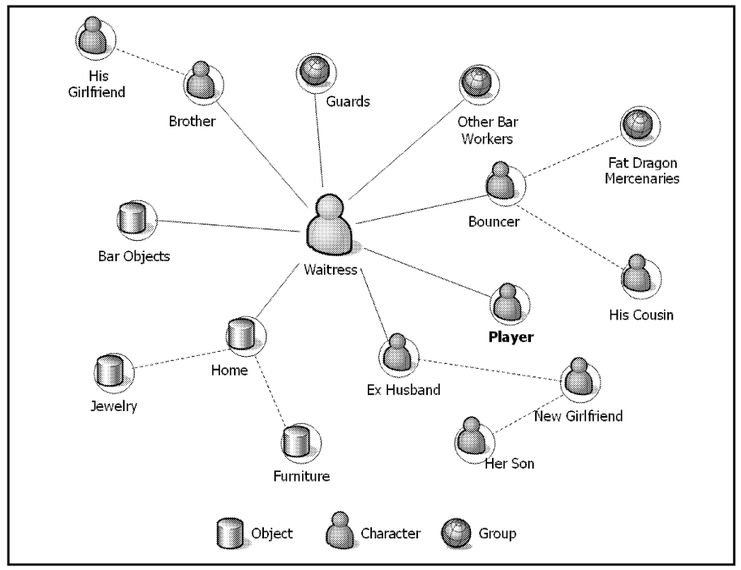


Figure 1: Example Relationship Network from Neomancer

provide architectural details of our new approach, and outline its implementation using Epic's Unreal Engine (Epic Games 2004). We then discuss our experiences with using this new system both in the context of an Unreal game mod (Castaneda 2005), and our Neomancer project (Danton 2004; Katchabaw 2005), currently under development. We finally conclude the paper with a summary, and a discussion of directions for future work in this area.

RELATIONSHIP MODELING AND USE IN GAME ARTIFICIAL INTELLIGENCE

Before examining the details of the newly extended RRS, we first provide background on modeling and manipulating relationship data for use in games, and how we can augment artificial intelligence controllers using this information.

Relationships for Games

A relationship network models all of the relationships between all of the characters, groups of characters, and objects of interest in the game world. One can envision this network as a graph-like structure, with the characters, groups, and objects as nodes in the graph, and the various relationships that exist between them as edges (directed or undirected, depending on the relationship). An example of this kind of relationship network from the Neomancer game project (Danton 2004; Katchabaw 2005) is presented in Figure 1.

There are numerous possible types of relationships that exist between entities in the relationship network. Each of these types can have subtypes, and so on, resulting in a hierarchical tree of relationship types. For example, main types of relationships can include: emotional, familial, business, leadership, ownership, membership, and so on. If we were to expand the membership branch, for example, there exist relationships to denote belonging to groups in the game, such as ethnicity, social caste, profession, community residence, and so on. This hierarchy can be easily expanded with additional types and sub-types as necessary.

Furthermore, each relationship has several attributes. These attributes include origin, history, regularity, strength, polarity, and validity. Relationship-specific attributes can also be assigned where appropriate.

Relationships can be affected in numerous ways. The most direct method is by filtered and processed game events. In other words, when one entity in the game world observes the actions of another, those actions can directly impact the relationships between those two entities, and the appropriate relationships must be added, updated, or removed. Relationships are also affected by game events indirectly, by their propagation through the relationship network. Depending on the nature of the event and how it affects entities in the network directly, the event can be felt by other related entities. Time also affects relationships. Given enough time, relationships drift towards a neutral state, in the absence of events or interactions that would otherwise act to strengthen them.

Augmenting Artificial Intelligence Controllers with Relationship Data

We now examine how to examine how to augment artificial intelligence controller using relationship data from a relationship network. Since scripting, state machines, and rule-based systems are the most widely used techniques in implementing game artificial intelligence (Champandard 2004), and scripting tends to be too static to be truly reactive, we will focus our attention on these last two techniques in this paper.

Generally, an artificial intelligence controller can be augmented to use relationship data in its decision process by querying the relationship network and using this data as additional game state to regulate state transitions or rule firings. In a state machine, this will require additional specialized transitions and/or specialized states; in a rule-based system, this will require additional rules with specialized firing conditions.

Care must be taken in using this relationship data, however. Using raw relationship information will result in an explosive increase in the size and complexity of state machines or rule systems using it, because the number of possible relationships and relationship attribute values could be quite large. This would make programming artificial intelligence for games quite difficult and tedious. If relationship information, however, was filtered and aggregated, much of this complexity can be removed, resulting in state machines and rule systems that are more manageable and easy to use.

Augmenting State Machines with Relationship Data

Consider, for example, the fragment of a state machine shown in Figure 2 for a guard in the guards group depicted in the relationship network in Figure 1.

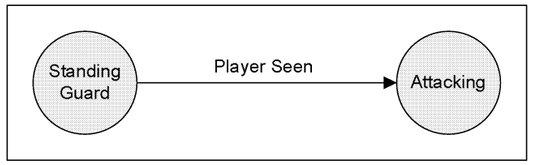


Figure 2: State Machine Fragment without Relationship Data

When the player is observed, this event causes the guard to switch to an attacking state to attack the player, regardless of the player's prior activities or relationship to the guard in question. While this reaction might appear realistic if the player's behaviour warranted an attack, it would appear oddly out of place if not. Such out of place behaviour can break the player out of immersion, and have a detrimental effect on the player's enjoyment of the game (Bates 2004; Rouse 2004).

Now consider the state machine fragment that is shown in Figure 3, based on the previous state machine. In this case, the relationship data between the player and guard in question has been distilled and aggregated for simplicity into one of three possibilities, negative, positive, and neutral, resulting in three possible transitions from a new state in the machine. (This new state can be avoided if an extended state machine is used that can have transitions triggered by multiple events or pieces of state information.) This allows player behaviour to influence the relationship with the guard, and the guard's reaction to the player as a result.

Suppose that guards in the guards group that was shown in Figure 1 are friends with the waitress that is the focal point of that relationship network. If the player hurts the waitress, the guards would have a negative view of the player and react with hostility towards the player. If the player, on the other hand, was helpful to the waitress, the guards would have a positive view of the player and help the player in return. With no prior contact with the waitress, the guards would have a neutral view of the player, and act accordingly.

Consequently, using relationship data, we can now have artificial intelligence that reacts in a realistic fashion determined by player behaviour. This results in a better overall experience to the player, as the player is left with the impression that his or her actions actually have an impact on the game world and its inhabitants (Bates 2004; Rouse 2004).

Augmenting Rule-Based Systems with Relationship Data

In this section, we examine augmenting rule-based systems with relationship data, much like state machines were in the previous section. Using the situation calculus notation of (Russell and Norvig 2003), we can express the original state machine fragment given in Figure 2 using a

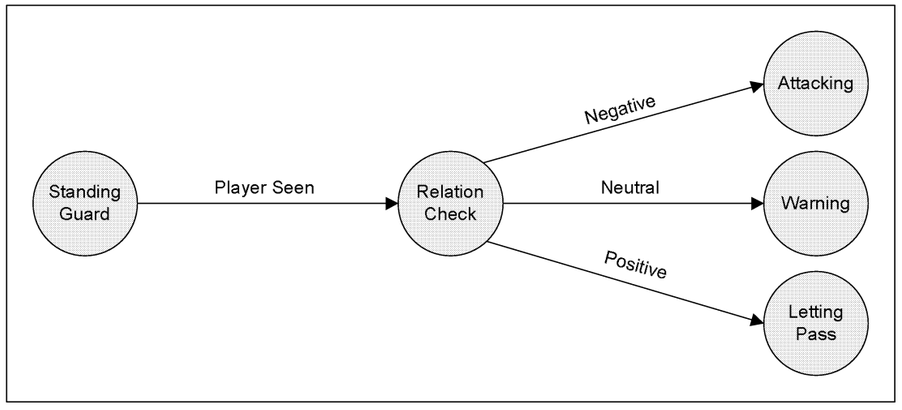


Figure 3: State Machine Fragment with Relationship Data Considered

rule system quite easily. Note that for simplicity, we are omitting effect axioms from the rule systems given in this paper. In the end, we are left with the rule system presented in Figure 4.

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \Rightarrow Poss(Attacks(Guard,Player),s) \quad (1)$$

Figure 4. Rule System Fragment without Relationship Data

The rule system in Figure 4 consists of one very simple rule. In essence, this rule states that if the guard is standing guard in a given situation and sees the player, then it becomes possible for the guard to attack the player from that situation. This is a fairly direct translation of the state machine from Figure 2.

Adding distilled and aggregated relationship data to the rule system in Figure 4 is quite straightforward. Following the example in the previous section, we augment this rule system with negative, positive, and neutral relationship data, resulting in a system with more specialized rules, as shown below in Figure 5.

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Negative) \Rightarrow Poss(Attacks(Guard,Player),s) \quad (1)$$

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Neutral) \Rightarrow Poss(Warns(Guard,Player),s) \quad (2)$$

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Positive) \Rightarrow Poss(LetsPass(Guard,Player),s) \quad (3)$$

Figure 5: Rule System Fragment with Relationship Data Considered

With three possible aggregated relationship states, the rule system in Figure 5 now must have three rules, each with a specialized firing condition corresponding to one of the possible states. Rule 1 is fired from a situation in which the guard is standing guard, sees the player, and has a negative relationship with the player. In this case, it is now possible for the guard to attack the player. Rule 2 is fired from similar circumstances, except that the guard has a neutral relationship with the player. In this case, the player is only given a warning. Lastly, Rule 3 is fired when the guard has a positive relationship with the player, and lets the player pass as a result. With these three rules in place, the artificial intelligence can be more reactive to player behaviour and produce results that are more in line with player expectations.

More Advanced Use of Relationship Data

As can be seen from the examples in previous sections, augmenting state machines and rule-based systems with relationship data is not difficult when aggregated relationship data is used. If raw data were to be used instead, however, both techniques could become significantly more complex, as discussed earlier.

That said, the use of raw relationship data can result in more advanced artificial intelligence controllers for non player characters that are able to fine tune their response to particular situations using more specific relationship data. This would result in even more realistic reactions and a better overall player experience.

Continuing the previous examples, if the player has a lucrative financial arrangement with the guard in question, the guard may still let the player pass, even if the guard has an overall negative or neutral opinion of the player. If we were to add this logic to the rule system from Figure 5, we can do so by adding the new specialized rules in Figure 6. (Other specialized rules may need to be added or used to replace existing rules for completeness, but the rules provided in Figure 6 suitably illustrate what would be required.)

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Negative) \wedge RelationshipExists(Guard,Player,Financial) \wedge FinancialRelationshipValue(Guard,Player,High) \Rightarrow Poss(LetsPass(Guard,Player),s) \quad (4)$$

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Neutral) \wedge RelationshipExists(Guard,Player,Financial) \wedge FinancialRelationshipValue(Guard,Player,High) \Rightarrow Poss(LetsPass(Guard,Player),s) \quad (5)$$

Figure 6: New Rule System Fragment with More Advanced Relationship-Based Decisions

Naturally, similar extensions can be made to state machines, such as the one shown in Figure 3, to capture such behaviour as well. This will again result in additional states and/or transitions, but would result in a more sophisticated and realistic controller.

In the end, we have the ability to construct both simple and specialized artificial intelligence controllers using relationship data as shown in the examples given above. This allows us to trade off complexity for expressiveness when required to suit the needs of the game in question.

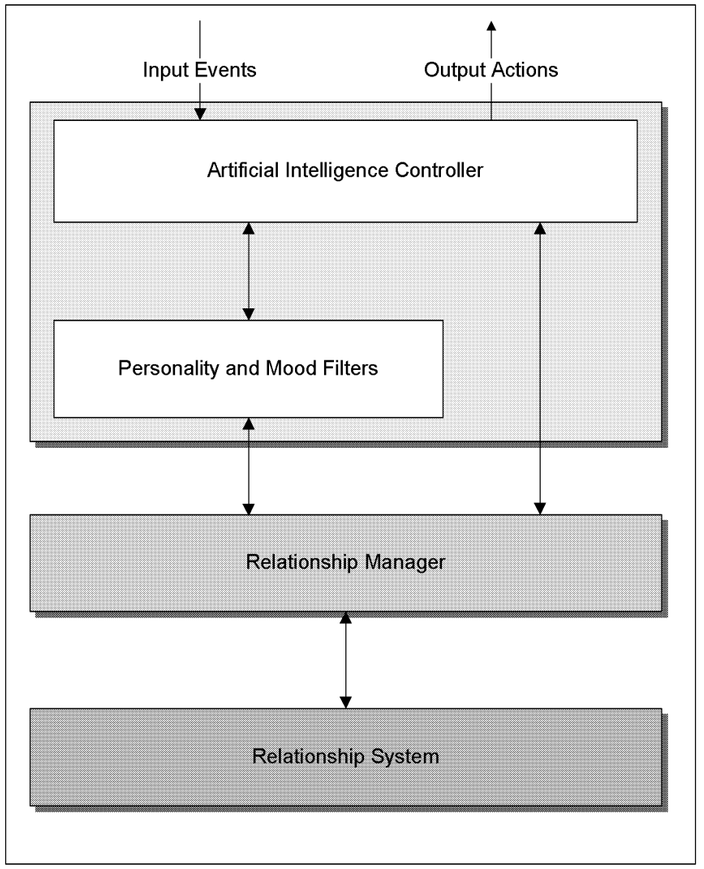


Figure 7: Reactive Artificial Intelligence Architecture

ARCHITECTING REACTIVE ARTIFICIAL INTELLIGENCE

To create reactive non player character artificial intelligence, we have developed the new architecture for RRS shown in Figure 7. The various elements of this architecture will be discussed in detail in the subsections below.

Relationship System

The relationship system is used to maintain and provide access to the relationship network constructed in the game. It provides raw access to this relationship data to the relationship manager, which can then provide a more specialized interface to simplify data access. For details on the internal design of the relationship system and its relationships and relationship network, the reader is urged to consult (Gruenwoldt 2005) for more information.

Relationship Manager

A relationship manager provides an interface to relationship data from the relationship system to one or more non player characters in a game. Typically, a relationship manager provides filtered or aggregated access to this data to simplify accessing and dealing with relationships in the artificial intelligence controllers of the corresponding characters. For example, the augmented artificial intelligence controller logic that was presented earlier in this paper made use of relationship data aggregated into three possible values: positive, neutral, and negative. If the controllers had to make use of raw relationship data, they would be orders of magnitude more complex to deal with the large number of possible relationships and relationship attributes as was discussed. This was learned first hand in directly using relationship data from the first prototype of RRS from our previous work (Gruenwoldt 2005) in constructing new artificial intelligence controllers; as the relationship system grew more robust and more expressive, the controllers grew in complexity quickly to the point where they became exceedingly difficult to program.

Depending on the needs of the game and its non player characters, relationship managers can aggregate data in a number of ways. A manager can provide a single relationship measure, or multiple relationship measures, and these measures can of a variety of types. For example, instead of providing positive, neutral, or negative values to the controllers presented earlier in this paper, a relationship manager could provide a numeric hostility score with

different values resulting in different transitions or rule firings. While providing a filtered or aggregated view of relationship data, a relationship manager must also still provide access to raw relationship data to allow the construction of more robust artificial intelligence controllers able to respond to more specific situations, as discussed in the previous section.

No constraints are imposed on how filtering and aggregating are to take place in this architecture; the selection of algorithms and heuristics is up to the implementer, as this process can be game specific. Filtering and aggregating in a relationship manager can occur when relationship changes are submitted to the relationship system, to have this data pre-computed for when queries are made, or in an on demand basis, when queries for data are actually received. Which of these approaches performs better likely depends on the game in question and the mix of relationship changes versus queries; fortunately, since this process is encapsulated within each relationship manager, a relationship manager can tune its own behaviour at run-time to provide the best overall results and performance without affecting the rest of the system.

In the end, we take an object-oriented approach to relationship managers, providing a base framework from which game-specific managers can be derived. In fact, game-specific managers can be further derived to allow variations from one class of non player characters to another, or even to the point of having specific managers for specific characters within the game, if required. This allows for a great deal of flexibility in filtering and aggregating data for use in a game.

Artificial Intelligence Controller

The artificial intelligence controller provides the core decision making functionality for a particular non player character in the game, accepting input events and formulating appropriate actions in return. As discussed earlier in this paper, in a gaming environment, this module will likely be driven by a state machine or rule-based system of some kind. For more details on controller design and implementation, (Champandard 2004) serves as a good reference.

Personality and Mood Filters

Personality and mood filters are used to provide further customization and specialization to artificial intelligence controllers to allow for more varied non player characters. These filters work by modifying the way input events are processed into relationship changes or by modifying results from relationship queries before they are processed by the controllers. This lets different characters record and retrieve relationship data differently, allowing different behaviours even when the characters are driven by fundamentally the same controller. This also allows character behaviour to be tuned over time as their personality develops throughout the game, or as their moods shift. As necessary, however, these filters can also be bypassed by the artificial intelligence controller, to provide direct contact with the relationship manager for the character.

Different personality traits can be parameterized and used in these filters, such as intelligence, aggressiveness, attentiveness, disposition, prejudices, and so on. For example, a character that is generally pessimistic could have its relationship changes adjusted more negatively than a character would have otherwise. Different moods and emotional states can also be used in filters for similar effects. For example, a character that is in an exceptionally good mood at a given time could have results of its relationship queries modified in a positive fashion, causing it to act better towards other characters it might not have otherwise.

IMPLEMENTATION AND EXPERIENCE

A prototype of the new RRS architecture from the previous section has been developed for Epic's Unreal Engine (Epic Games 2004) in UnrealScript. UnrealScript has many of the features of a traditional object-oriented language, providing excellent support for extensibility for the future. Games built on the Unreal Engine can take advantage of this system by either extending a new game type and new pawn and controller classes, or by embedding the appropriate hooks into existing game code. In addition, our earlier work with the Unreal Engine provided additional console commands to support manipulation of relationships manually from within the game (Gruenwoldt 2005). This allows game developers and designers to add relationship information during production from within the game itself, allowing easy debugging and initialization of content.

Artificial intelligence controllers in the Unreal Engine are in essence state machines, allowing them to be augmented with relationship data as described earlier in this paper. Using the existing relationship system, relationships, and relationship network from (Gruenwoldt 2005) as a foundation, a new relationship manager was derived from its base class to aggregate this relationship data together using a collection of simple heuristics. A small number of personality and mood filters were also developed to provide more varied tuning of relationship changes and query responses. As simple examples, optimistic and pessimistic filters were implemented to adjust the positivity and negativity of relationship changes respectively when required within a game.

After development, initial validation of the new RRS took the form of individual test cases. More extensive validation took the form of modifying the existing LawDogs game modification to Unreal Tournament 2003/2004 (Castaneda 2005). LawDogs was chosen primarily because its setting included a bar scene, which follows in line closely to the relationship network example presented in Figure 1, and introduced originally in (Danton 2004). LawDogs also had reasonably simple gameplay with straightforward and traditional artificial intelligence, making it a suitable first deployment for our work. Our experience with LawDogs demonstrated that the non player characters in the game were able to react according to player interactions quite well.

Figure 8: Screenshot from Neomancer with Test Character

Based on this success, we are currently augmenting the artificial intelligence developed for the Neomancer project (Danton 2004; Katchabaw 2005), an action/adventure/role-playing game being co-developed by the University of Western Ontario and Seneca College. It features richer characters and gameplay than LawDogs, and is better suited towards larger scale deployment and testing of our current work.

We have encountered similar success with test non player characters in Neomancer using the new RRS, and are currently expanding use of the system as the artificial intelligence controllers and characters in the game continue to be developed, refined, and enhanced. (The Neomancer project is expected to take up to three years to complete, and we have only completed the initial year of the project.) A screenshot of a test character in the Neomancer world is shown in Figure 8.

CONCLUDING REMARKS

By capturing game relationships and facilitating more appropriate character responses through linking relationship data to artificial intelligence controllers for non player characters, our new Realistic Reaction System can provide more immersive and compelling gameplay in modern video games efficiently and effectively. In the end, non player characters will be able to react to player behaviour in a more realistic fashion, leading to a better overall gameplay experience for the player.

Experimentation with an Unreal-based implementation of this system to date has proven successful, both in small and mid-size deployments of the system. Larger scale deployment in a commercial-grade game is currently underway and progress to date has been excellent. This new Realistic Reaction System demonstrates great promise for future development efforts.

In the future, there are many possible directions for research to take. This includes the following:

- We plan to complete our current development and deployments efforts with Neomancer and port the new RRS to other games and platforms for further research and development.

- To meet stringent performance constraints we further plan to investigate techniques to optimize RRS and minimize run-time overhead in manipulating and querying relationships in the system. While we have found that proper filtering and aggregation in relation managers can greatly improve performance, additional performance improvements would always be quite beneficial.

- Finally, we also intend to extend our library of relationships, relationship managers, and personality and mood filters to allow RRS to support a wider variety of artificial intelligence behaviours in games by default.

REFERENCES

G. Alt and K. King. "A Dynamic Reputation System Based on Event Knowledge". *Appeared in AI Game Programming Wisdom.* Charles River Media. 2002.

B. Bates. *Game Design.* Second Edition. Thomson Course Technology. 2004.

M. Brockington. "Building a Reputation System: Hatred, Forgiveness and Surrender in Neverwinter Nights." *Appeared in Massively Multiplayer Game Development.* Charles River Media. 2003.

G. Castaneda, et al. LawDogs UT2003-UT2004 Modification. *Available from project home page online at http://www.planetunreal.com/lawdogs.* February 2005.

A. Champandard. *AI Game Development.* New Riders Publishing. 2004.

S. Danton. *Neomancer Game Design Document.* Horseplay Studios Technical Report. November 2004.

Epic Games. *Unreal Engine 2, Patch-level 3339.* November 2004.

G. Grond and B. Hanson. "Ultima Online Reputation System FAQ". *Origin Systems Technical Document (available at http://www.uo.com/repfaq).* 1998.

L. Gruenwoldt, M. Katchabaw, and S. Danton. "A Realistic Reaction System for Modern Video Games". In *the Proceedings of the DiGRA 2005 Conference: Changing Views – Worlds in Play.* Vancouver, Canada, June 2005.

M. Katchabaw, D. Elliott, and S. Danton. "Neomancer: An Exercise in Interdisciplinary Academic Game Development". *In the Proceedings of the DiGRA 2005 Conference: Changing Views – Worlds in Play.* Vancouver, Canada, June 2005.

F. Laramée. "Nine Trade-Offs of Game Design". *Appeared in Game Design Perspectives.* Charles River Media. 2002.

G. Lawson. "Stop Relying on Cognitive Science In Game Design - Use Social Science". *In Gamasutra Letter to the Editor (available online at http://www.gamasutra.com/php-bin/letter_display.php?letter_id=647).* December 2003.

R. Rouse III. *Game Design: Theory and Practice.* Second Edition. Wordware Publishing, Inc. 2004.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Second Edition. Prentice Hall. 2003.

AN EXPERIMENTAL TESTBED TO ENABLE AUTO-DYNAMIC DIFFICULTY IN MODERN VIDEO GAMES

Christine Bailey and Michael Katchabaw
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada
E-mail: cdbailey@csd.uwo.ca, katchab@csd.uwo.ca

KEYWORDS

Auto-dynamic difficulty, difficulty adjustment in games

ABSTRACT

Providing gameplay that is satisfying to a broad player audience is an appealing goal to game developers. Considering the wide range of player skill, emotional motivators, and tolerance for frustration, it is simply impossible for developers to deliver a game with an appropriate level of challenge and difficulty to satisfy all players using conventional techniques. Auto-dynamic difficulty, however, is a technique for adjusting gameplay to better suit player needs and expectations that holds promise to overcome this problem.

This paper presents an experimental testbed to enable auto-dynamic difficulty adjustment in games. Not only does this testbed environment provide facilities for conducting user studies to investigate the factors involved in auto-dynamic difficulty, but this testbed also provides support for developers to build new algorithms and technologies that use auto-dynamic difficulty adjustment to improve gameplay. Initial experiences in using this auto-dynamic difficulty testbed have been quite promising, and have demonstrated its suitability for the task at hand.

INTRODUCTION

The goal of producing a game is to provide many things to players, including entertainment, challenge, and an experience of altered state. Ultimately, however, a game must be fun. One major source of polarization among players on this issue is the level of difficulty in a game. There is a great degree of variation in players with respect to skill levels, reflex speeds, hand-eye coordination, tolerance for frustration, and motivations.

In (Csikszentmihalyi 1996), the concept of "flow" is used to refer to an individual's "optimal experience". In a state of flow, the individual experiences intrinsic enjoyment from undertaking a task that feels almost effortless and natural, while also causing the individual to feel focussed and challenged. One facet of flow is that there is a balance between the challenge presented by the task and the increasing skill of the individual, as discussed in (Falstein 2004). This closely resembles the concept of a zone of proximal development as discussed in (Woolfolk et al. 2003), in which a balance between skill and challenge is needed in educational settings in order for learning to take place. This zone of proximal development was found to be different for each student, and that tasks considered easy by some may be too difficult for others. Assigning difficulty levels in games must address this problem so as to hit the "optimal experience" for as many players as possible, each of which may have very different zones of proximal development. These issues are important because, as noted in (Miller 2004) and (Rouse 2004), a game must balance challenging and frustrating the player to provide the best overall level of satisfaction and enjoyment.

Several approaches have been used in the past to attempt to provide appropriate difficulty levels in a variety of different ways, such as having a single static difficulty level (chosen either by the designer or through play-testing of the target audience), having several different static difficulty levels to choose from at the start of the game, or providing cheat-codes. However, each method has its drawbacks and limitations, and ultimately cannot provide an appropriate difficulty level to all players, particularly as their skills improve as they play and learn. In the end, this can drastically limit the success of a game.

Auto-dynamic difficulty refers to the ability of a game to automatically adapt the difficulty level of gameplay to match the skills and tolerances of a player. If done properly, this can provide a satisfying experience to a wider variety of players. The concept of auto-dynamic difficulty is not new; it has been used in early arcade games such as *Xevious* to more recent titles such as *Max Payne* (Miller 2004). Typically, however, this technique is used in an ad hoc and unrepeatable fashion, applied to a particular game or gameplay element within a game. Often, there is little regard or understanding for the various factors that influence player experience and how these factors interact with one another; as long as the current game is improved, that is all that matters.

In this paper, we discuss the development of an experimental testbed to facilitate the development of auto-dynamic difficulty enabling technologies for games. This testbed serves two key purposes. The first is to support experimentation to better understand what shapes player experience and how gameplay and difficulty can be altered to produce the best experience possible. The second is to

serve as a vehicle for testing new algorithms and methodologies for supporting auto-dynamic difficulty developed as part of this work. The goal is that this work will provide both a better understanding of how to create more enjoyable and satisfying gameplay experiences for a wider range of players, and that it will deliver enabling technologies to make use of this new understanding in a wide variety of games and gameplay scenarios.

The remainder of this paper is organized as follows. We begin in the next section with a background discussion of auto-dynamic difficulty adjustment, describing what adjustments are possible, and the deciding factors in determining when and how such adjustments should be made. We then present the architecture and implementation of our auto-dynamic difficulty experimental testbed environment. We then provide a brief discussion of our experiences to date in using this experimental testbed. Finally, we conclude this paper with a summary and discussion of potential future work in this area.

AUTO-DYNAMIC DIFFICULTY ADJUSTMENT

Before discussing our experimental testbed environment, it is first important to further explore the key issues behind auto-dynamic difficulty adjustment. Of critical importance is to recognize what adjustments can and should be made, as well as when and how to make these adjustments. Any adjustments must be made with care in such a way that they enhance the satisfaction and enjoyability of the game, without disrupting the game in a negative fashion. (For example, changes that are too abrupt could disrupt the immersion of the player, causing a negative effect on the overall experience.)

What to Adjust

Designed properly, a good portion of a game's gameplay elements can have difficulty that is adjustable dynamically (Bailey 2005). This includes the following:

Player character attributes. The attributes of the player's character can be tuned according to the desired level of difficulty in a game. As examples, to make a game easier, the player could be made stronger, move faster, jump higher and farther, have more health, have better armour, attack with more damage, attack more frequently, and so on. To make a game harder, these attributed can be adjusted in the opposite directions.

Non-player character attributes. Likewise, the attributes of non-player characters controlled by the game's artificial intelligence can change. Not only does this include the attributes affecting the actions they take as above, but this also includes the decision making processes used. To make a game less difficult, non-player characters can make poorer decisions, provided that these decisions do not make the characters appear artificially stupid. As examples, path-finding can be adjusted to make the player harder to find, aiming can be adjusted so that attacks are less successful, and so on. Similarly, steps can be taken to make better decisions that make the game more difficult.

Game world and level attributes. Various elements of how the game world and its levels are designed can affect game difficulty, including both the structure of the levels, and their contents (Bates 2004). With advancements in game engine technologies, it is now possible to do this dynamically from within the game. Adjusting level structure depends heavily on the gameplay occurring within the gameplay. For example, in a platformer-genre game involving a lot of jumping puzzles, level geometry can be adjusted dynamically to make gaps smaller or larger to make the game easier or more difficult. In a shooter game, as another example, the amount of cover can be adjusted appropriately to make the game easier or more difficult as well. Level contents can also be tuned dynamically to adjust difficulty. By adding or removing items such as ammunition, health upgrades, and so on, a game can be made easier or more difficult. Varying the quantity and spawning locations of enemy non-player characters can also affect difficulty.

Puzzle and obstacle attributes. As discussed in (Bates 2004), there are several ways of adjusting the level of difficulty provided by puzzles and obstacles within a game. Fortunately, many of these techniques can be applied dynamically. While it might not always be possible to dynamically adjust the attributes of the current puzzle or obstacle faced by the player (for consistency and other reasons), it might be possible to instead adjust the difficulty in puzzles faced in the future. For example, if a player is finding one type of puzzle to be difficult to solve, in the future, the solution to that same type of puzzle can be placed closer to the puzzle itself, making it inherently easier to solve (Bates 2004).

As discussed in (Miller 2004), most earlier attempts at auto-dynamic difficulty focussed on a restricted subset of gameplay, typically in the adjustment of player or non-player character attributes. With this rationale applied throughout the game, as discussed above, it is possible to create a better overall player experience.

When and How to Adjust

To determine when to adjust game difficulty and how to do so, data must be collected on players and their progression through the game. To provide the best level of challenge, we must have a measure of the current skill level of the player, as well as their success and failure rates at the various elements of gameplay encountered to date in the game. Since different players will tolerate and accept different levels of challenge at different times, we must also have a sense of the player's general type, motivations, frustration tolerance, and emotional state.

Measuring a player's level of skill in a game, as well as their success and failure rates, is inherently tied to the particular game or game genre. Typically, however, there are multiple metrics that are applicable and can be measured from within the game itself. For example, in a platformer game with a sequence of jumping puzzles, the number of attempts before success and time to completion could be useful metrics. In a shooter game, the percentage

of enemies eliminated per level, the amount of damage taken per level, and time to completion could be useful metrics. One must give careful thought to the metrics selected, however, as they could indicate unanticipated styles of play or other player activity, and not the skill of the player. For example, tracking the number of game saves and loads might be problematic. One might think that a high frequency of saves and loads is indicative of an unskilled player, but this pattern of activity could also be encountered by a player playing the game during short coffee breaks (Bailey 2005). Counting the number of player character deaths might also be misleading, as an unskilled player could get frustrated after a single death and quit the game with a relatively low death count only to return later. So, while there might be multiple methods of tracking player progression through a game, care and thought must be put into the process.

Determining a player's type and internal factors is more difficult to do within a game, but not impossible. For example, (Sykes and Brown 2003) found that the pressure of button and key presses correlated strongly to frustration and difficulty levels within a game. The work in (IP and Adams 2002) examined ways of quantitatively measuring levels of "core" and "casual" in a given player. As discussed in (Bailey 2005), elements of player types identified in (Bartle 1996) and (Lazzaro 2004) could be identified by tracking player movement and progress through a game. For example, the explorer type identified in (Bartle 1996) could be detected by observing players lingering in areas of the game world for extended periods of time without paying attention to game goals, while the achiever type could be detected by observing a linear and timely progression through game goals. As pointed out in (Bailey 2005), however, there are ultimately some internal factors that are not easily measurable from within a game world, and we must rely upon external studies and experimentation to calibrate the game and assist in correlating observed player behaviour and emotion state.

Using measurements of player skill, as well as success and failure rates, it is not hard to determine when a player is encountering difficulty with a certain element of gameplay. While these measurements are important, we must be careful to also take into consideration player type and internal factors; otherwise, we again fall into a "one-size-fits-all" mentality that does not produce appealing results to a broad audience. It is also important to consider the impact of characteristics of the gameplay on the motivation of the player, including whether the necessity of the gameplay element, the rewards for success, the consequences of failure, and so on (Bailey 2005).

In the end, it is possible to develop rudimentary rules to guide when difficulty adjustments should be made and how, based on this information. For example, if a player is encountering a challenging task, but is exhibiting characteristics of the achiever type, then difficulty should not be adjusted as this player type is more likely to enjoy the challenge than not (Bartle 1996). However, to assist in the formulation and validation of these rules and decision models, experimentation is necessary. A thorough investigation in this area is clearly warranted. This reality, in part, motivated developing the experimental testbed discussed in this paper.

AUTO-DYNAMIC DIFFICULTY EXPERIMENTAL TESTBED

To facilitate the study of auto-dynamic difficulty, our current work focuses on the construction of an experimental testbed that will enable experimentation with players and development of new technologies to better tune game difficulty automatically to meet their needs. This testbed is depicted in Figure 1, and discussed in more detail in the sections below.

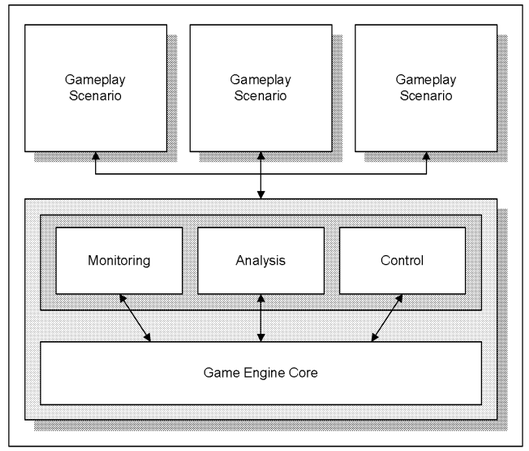


Figure 1: Auto-Dynamic Difficulty Experimental Testbed Architecture

Game Engine Core

The game engine core is used to provide all of the fundamental technologies required to drive a game or gameplay scenario. This includes graphics, audio, animation, artificial intelligence, networking, physics, and so on. One could then layer new gameplay logic and content on top of this engine to have a complete game, without the burden of developing all of the underlying technologies. This saves considerable development time in building the testbed, and also allows the use of professional-grade tools to produce a gameplay experience of very high quality.

At the core of our testbed is Epic's Unreal engine (Epic Games 2004). The Unreal engine is a modern, state-of-the-art game engine that can be used to support a wide variety of game genres and gameplay elements. It also supports rendering in both first and third person views, which makes it easier to support more varied gameplay. The Unreal engine itself is written in C and C++, but provides a flexible object-oriented scripting language, UnrealScript, to make it easy to extend the engine and deliver new functionality. Since this engine is based on leading edge technologies and still in use commercially today, there are no concerns of confounding that could have arisen from using older,

obsolete technologies. (For example, in such a case, one would have to determine if a player had an unsatisfactory experience because of game difficulty or because the game's graphics were not up to the standards set by modern games.) In the end, the Unreal engine was a natural choice of foundation on which to build our testbed.

Monitoring, Analysis, and Control

Monitoring, analysis, and control services are used in the testbed to support both auto-dynamic difficulty experimentation and software developed to implement new auto-dynamic difficulty algorithms and methodologies. These services are used by gameplay scenarios, and directly make use of the game engine core.

To conduct experimentation within a particular gameplay scenario, the experimental environment must monitor and collect the appropriate player and progression data, as discussed in the previous section. The analysis service is used to provide support in the aggregation and correlation of data collected through monitoring. The control service is used to manipulate the experiment in the gameplay scenario, including starting, suspending, resuming, and halting a particular experiment. It is important to note that some aspects of monitoring, analysis, and control may need to be completed offline outside of the testbed software. (For example, augmenting recorded data with audio and video recordings, as well as surveys and interviews currently must be done offline. In the future, it is hoped to add these elements to the testbed software as well for a more integrated solution. Analyses of these elements would still likely require manual intervention, however.)

To support new auto-dynamic difficulty algorithms and methodologies, the monitoring service still collects player and progression data as before. The analysis service in this case is now focussed more on analysing this data to formulate decisions on when and how to adjust game difficulty using rules and decision models formulated based on experience and experimentation conducted using the testbed. The control service in this case still manipulates the gameplay scenario, but is focussed this time on the relevant attributes of the player character, non-player characters, the game world, or game puzzles and obstacles to adjust the game's difficulty according to the decisions developed by the analysis service.

In our testbed, the monitoring, analysis, and control services are written in UnrealScript. All three services are integrated into a single new Unreal game type derived from the base Unreal game type class. This new game type provides instrumentation suitable for embedding in gameplay scenarios to enable monitoring, analysis, and control activities. This facilitates the development of new gameplay scenarios and entire games using these auto-dynamic difficulty services, as these new games would simply need to derive their own game type from this new type, instead of the base class.

At present, rudimentary monitoring, analysis, and control services are provided; more sophisticated facilities are currently under development. Currently, the monitoring service can collect time to completion, success and failure rates, and other metrics, the analysis service can support simple correlations and decision rules, and the control service can control experiment operation, and tune certain player and non-player character attributes, as well as selected game world attributes.

Gameplay Scenarios

Gameplay scenarios are used to contain playable elements of games and game content. These can range in scale from mini-games depicting as few as one game activity for the player, all the way up to complete entire games.

In the current version of the testbed, we have implemented a variety of mini-game gameplay scenarios using UnrealScript and UnrealEd (Busby et al. 2005). These include two jumping mini-games (one with fatal consequences, the other with no failure consequences), a timed maze navigation mini-game, a turret mini-game requiring the player to navigate a short hallway lined with automated, indestructible gun turrets, and a fighting mini-game requiring the player to make their way through a room full of heavily armed enemy non-player characters. A screenshot from one of these scenarios is given below in Figure 2. Recognizing the limitations of experimenting with mini-games, as discussed in the next section, we are also building the Neomancer project (Katchabaw 2005) based on our new game type, to provide a complete action/adventure/role-playing game experience for experimentation and development activities.

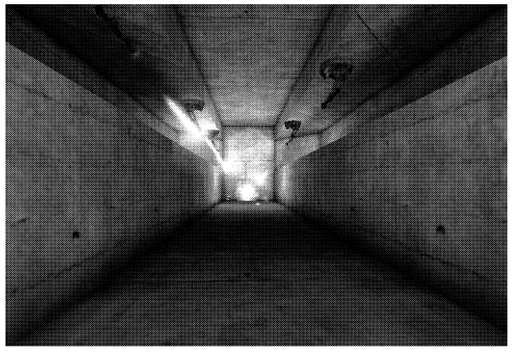


Figure 2: Screenshot from Turret Hallway Mini-Game

EXPERIENCES AND DISCUSSION

Initial user studies and testing using the auto-dynamic difficulty experimental testbed were conducted with a small number of family members and co-workers of researchers at Western. Results of this early experimentation have been rather positive, indicating that the testbed is suitable for the task at hand. While the initial version of the testbed can only monitor a small number of player and progression metrics, analyse through simple correlations and a restricted rule set, and control through only simple operations, we have been able to gather interesting results from experimentation and implement several auto-dynamic

difficulty algorithms. It is clear, however, that more thorough experimentation using a large study group is necessary, both to better understand the interplay of the factors involved in auto-dynamic difficulty, and to develop better algorithms and technologies for games (Bailey 2005).

During initial experimentation using the mini-game game scenarios, it also became apparent that mini-games on their own might not be sufficient for investigating auto-dynamic difficulty fully. Mini-games, by their very nature, do not have a broader story, context, or reward system, which was found to produce a different emotional state in the player than playing a full game. Repetition of mini-games was also found to grow tedious, resulting in a negative impression of the mini-game independent of its challenge or difficulty. Consequently, it is necessary to have a complete gaming experience to fully explore auto-dynamic difficulty. Fortunately, through our development efforts in the Neomancer project (Katchabaw 2005), we have access to a commercial scale action/adventure/role-playing game that will fill this need nicely.

Game performance is a crucial factor to game players and game developers alike. Consequently, it is critical to ensure that there be minimal overhead imposed by auto-dynamic difficulty on the game as it plays. During initial experimentation, frame rate tests were conducted using the Unreal engine's own frame rate monitors, with and without the use of auto-dynamic difficulty and the instrumentation required for monitoring and control. This testing found that there was no measurable difference between frame rates delivered with and without auto-dynamic difficulty in place, and so performance was deemed acceptable.

The approach to auto-dynamic difficulty currently taken in this work is reactive. In other words, once measurements indicate that a game is too easy or too difficult for the player, gameplay can be adjusted to produce a more favourable experience. Unfortunately, a reactive approach means that a player must encounter such problems before any corrective actions are taken, and that the player could lose patience with the game before auto-dynamic difficulty has a chance to become active. It was found during initial experimentation that some mini-game scenarios could be made so easy or so difficult that the player is turned off almost instantaneously, sharply reducing the benefits of reactive auto-dynamic difficulty in these extreme situations. Proactive auto-dynamic difficulty, on the other hand, attempts to adjust game difficulty before a player encounters the above problems through an analysis of non-critical gameplay tasks. Doing so, however, would likely require calibration through more user studies to develop an appropriate predictive model, and introduces other problems if predictions are inaccurate. It would seem, however, that investigating proactive adjustments, perhaps in conjunction with reactive techniques, would be a worthwhile endeavour.

CONCLUDING REMARKS

Delivering satisfying gameplay experiences to a variety of players is a challenging task. To do so, gameplay difficulty must be tuned to suit player needs, as in auto-dynamic difficulty. Our current work is aimed at addressing this, by providing an experimental environment for studying this problem and assisting in the formulation of acceptable solutions. Initial experience through using this auto-dynamic difficulty experimental testbed has been quite positive, showing much promise for the future.

In the future, there are many interesting avenues for continuing research to take. We plan to refine the monitoring, analysis, and control capabilities of the testbed, to enable more thorough user studies. Using this enhanced testbed, we intend to expand experimentation to include a larger, more diverse player population. Based on the results of this experimentation, we will develop additional rules and decisions models for use in the testbed's analysis service to better support a wider variety of auto-dynamic difficulty algorithms. At the same time, we will continue work on the Neomancer project to provide a full length, feature rich gameplay scenario for studies with the testbed. Finally, we plan to continue investigating other open research issues in auto-dynamic difficulty adjustment, including reactive versus proactive techniques.

REFERENCES

C. Bailey. "Auto-Dynamic Difficulty in Video Games". *Undergraduate Thesis. Department of Computer Science, The University of Western Ontario*. April 2005.

R. Bartle. "Hearts, Clubs, Diamonds, Spades: Players who Suit MUDs". *Journal of MUD Research, 1(1)*. 1996.

B. Bates. *Game Design*. Second Edition. Thomson Course Technology. 2004.

J. Busby, Z. Parrish, and J. Van Eenwyk. *Mastering Unreal Technology: The Art of Level Design*. Sams Publishing. 2005.

M. Csikszentmihalyi. *Creativity: Flow and the Psychology of Discovery and Invention*. New York, NY: HarperCollins Publishers. 1996.

Epic Games. *Unreal Engine 2, Patch-level 3339*. Nov. 2004.

N. Falstein. "The Flow Channel". *Appeared in Game Developer Magazine*. May 2004.

B. Ip and E. Adams. "From Casual to Core: A Statistical Mechanism for Studying Gamer Dedication". *Article published in Gamasutra and is available online at http://www.gamasutra.com/features/20020605/ip_pfv.htm.* June 2002.

M. Katchabaw, D. Elliott, and S. Danton. "Neomancer: An Exercise in Interdisciplinary Academic Game Development". *In the Proceedings of the DiGRA 2005 Conference: Changing Views – Worlds in Play*. Vancouver, Canada, June 2005.

N. Lazzaro. "Why We Play Games: Four Keys to More Emotion without Story". *Presented at the 2004 Game Developers Conference*. San Francisco, California, March 2004.

S. Miller. Auto-Dynamic Difficulty. *Published in Scott Miller's Game Matters Blog (http://dukenukem.typepad.com/game_matters/2004/01/autoadjusting_g.html).* January, 2004.

R. Rouse III. *Game Design: Theory and Practice*. Second Edition. Wordware Publishing, Inc. 2004.

J. Sykes and S. Brown, S. "Affective Gaming: Measuring Emotion through the Gamepad". *Proceedings of the CHI 2003 Conference on Human Factors in Computing Systems*. Fort Lauderdale, Florida, April 2003.

A. Woolfolk, P. H. Winne, and N. E. Perry. *Educational Psychology*. Toronto, Ontario: Pearson Education. 2003.

(P)NFG: A LANGUAGE AND RUNTIME SYSTEM FOR STRUCTURED COMPUTER NARRATIVES

Christopher J.F. Pickett Clark Verbrugge Félix Martineau
School of Computer Science, McGill University
Montréal, Canada, H3A 2A7
email: {cpicke,clump,fmarti10}@cs.mcgill.ca

KEYWORDS
Computer Games, Narratives, Petri Nets, Interactive Fiction, Formal Verification, Languages, Compilers, Interpreters

ABSTRACT

Complex computer game narratives can suffer from logical consistency and playability problems if not carefully constructed, and current, state of the art design tools do little to help analysis or ensure good narrative properties. A formally-grounded system that allows for relatively easy design and analysis is therefore desireable. We present a language and an environment for expressing game narratives based on a structured form of Petri Net, the *Narrative Flow Graph*. Our "(P)NFG" system provides a simple, high level view of narrative programming that maps onto a low level representation suitable for expressing and analysing game properties. The (P)NFG framework is demonstrated experimentally by modelling narratives based on non-trivial interactive fiction games, and integrates with the NuSMV model checker. Our system provides a necessary component for systematic analysis of computer game narratives, and lays the foundation for all-around improvements to game quality.

INTRODUCTION

A large number of computer games have strong narrative components. Most notably this includes adventure and role-playing games, but many first person shooters and 3D games also depend on a narrative backbone structure. Unfortunately, as many gamers are aware, complex narratives often contain either outright flaws or more subtly undesireable game properties [Adams, 2005]. Plot holes, non-sequiturs and narrative dead-ends are not uncommon, and difficult to avoid completely when developing a large game. A formal narrative analysis system that can help to determine these problems and otherwise analyse narratives is clearly desireable.

We initially draw on *interactive fiction* (IF) as a source of well-defined, complex narratives; IF is one of the oldest computer game genres, and provides for interactive storytelling at the most basic, fundamental level: in its most common form, the player enters text commands and receives text messages as output [Montfort, 2003]. This setting allows us to focus on "pure" narrative issues, and to separate out user interface and real time, non-deterministic gameplay concerns; it also means we are able to specify *complete* representations of many games.

We use the *Narrative Flow Graph* (NFG) as a formal structure for representing IF games [Verbrugge, 2002], and provide a new interactive NFG interpreter that allows for actual IF gameplay. NFGs are themselves a class of 1-safe Petri Nets (PNs), and thus we can exploit a wealth of available analysis research. At runtime, we feed this low level game format to the NuSMV formal model checking software [Cimatti et al., 2002] to determine game properties.

NFGs are appropriate for formal analysis, but a higher level expression is required for complex game design. We thus introduce the *Programmable NFG* (PNFG) language that accepts a high level game specification. Our language allows for easy expression of game narratives, and we have been able to produce faithful implementations of real, complex IF games relatively quickly, including the complete Scott Adams game, *The Count* [Adams, 1981]. We have also derived IF representations for the two initial chapters of *Return to Zork* [Barnett, 1993], a graphical point-and-click adventure. The PNFG compiler produces NFGs from these narratives, and we are thus able to analyse non-trivial benchmarks. Although we find that narrative complexity soon limits the analysis available, this is early work and a good baseline system for future experimentation.

Together the NFG interpreter and PNFG compiler form the (P)NFG system, and our software is freely available under the terms of the LGPL; the authors welcome feedback, bug reports, and source code contributions.

Contributions

Specific contributions of this work include:

- A new and formally-backed language for narrative specification. We give precise rules and structure for compilation of high level narrative source code to a corresponding low level representation that allows for narrative analysis.
- A Petri Net-based, interactive narrative interpreter and runtime system that integrates with the NuSMV model checker. This permits finding paths to winning and losing states, and verifying other game properties.
- Experimental data on the representation and analysis of small, medium-sized, and large actual interactive fiction narratives. Real data on non-trivial game narrative structure is of great benefit to further analysis.

In the next section, we discuss related work on narrative analysis. Subsequently, we provide a definition of our NFG formalism, slightly extended to allow for external input and output. We then give an overview of our software framework, and in the following two sections provide full details on the NFG interpreter and the intricacies of formal verification, and describe the PNFG compiler and its code generation strategies. Afterwards we present implementations of several narratives, and provide experimental results obtained using various size and complexity metrics and from our attempts at verification. Finally, we conclude and discuss future work.

RELATED WORK

Flaws in narrative construction and the corresponding need for better processes have been identified in all manner of commercial and non-commercial games [Adams, 2005]. Directed acyclic graph (DAG) representations of plotlines have been proposed as a solution to these problems several times, on `r.a.i-f` [Arnold et al., 1995], by an online IF magazine [Forman, 1997], and by the Oz group [Mateas, 1997]. IFM, the Interactive Fiction Mapper [Hutchings, 2004], is a tool that facilitates map generation and plot DAG creation by end users, and includes a solver that derives a walkthrough from task dependencies. However, DAGs most often cannot provide a complete representation as they cannot model arbitrary cycles or resource consumption.

Higher level narrative development frameworks have been explored [Brooks, 1996, Charles et al., 2002, Young, 2005], and there has also been considerable work on using logic for modelling and analysis. The language $\mathscr{E}$ provides a thorough logic-based approach to describing narratives using actions [Kakas and Miller, 1997], and narratives have also been studied as pure logic programs [Reiter, 2000]. Constraint logic programming can be used to analyse and detect flaws in story chronologies [Burg et al., 2000], and causal normalisation has been examined as a mechanism for ensuring consistency in games [Eladhari, 2002].

As a general rule of thumb, all of the interesting questions about the behaviour of 1-safe Petri Nets are PSPACE-hard [Esparza, 1998]. In order to limit the practical complexity of PN analysis we use the symbolic model verifier NuSMV [Cimatti et al., 2002], which supports a Binary Decision Diagram (BDD)-based backend [Bryant, 1992]. BDDs help to collapse the state space, making feasibile the determination of properties such as reachability for larger problem instances. NuSMV has been used to model PNs in the past [Bobbio and Horváth, 2001], techniques for encoding PNs efficiently using BDDs have been reported [Pastor et al., 2001], and recently it was suggested that clever application of brute force algorithms can be just as if not more efficient [Ciardo, 2004].

The PNFG language we will describe allows for high-level narrative descriptions to be compiled for use by our PN-based NFG interpreter. Previous work in other domains has also yielded methods for translation of languages to a PN model: a formal PN semantics has been defined for the Programmable Logic Controller (PLC) instruction list [Heiner and Menzel, 1998], and SynchNet compiles distributed object coordination specifications down to PNs [Ziaei and Agha, 2003].

This paper builds on our own previous theoretical work defining the Narrative Flow Graph (NFG) as a formal structure for computer narratives [Verbrugge, 2002]. Others have also sought to represent computer narratives using Petri Nets [Natkin and Vega, 2004], introducing several higher level control flow constructs. That work is extended in [Vega et al., 2004] to model spatiotemporal relationships in narratives using *connections* that replace edges dynamically based on transition firing patterns. Coloured PNs have also been used to model narratives in multi-agent interaction scenarios, as demonstrated through an implementation of the card trading game *Pit* [Purvis, 2004]. Finally, although not considered in the specific context of computer narratives, closely related work has seen PNs used to model relationships between tasks in workflow management [van der Aalst, 2002] and to provide a verifiable mechanism for browsing hypertext [Stotts and Furuta, 1989].

Interactive fiction authoring kits themselves have been a favourite of hobbyist programmers and many systems are available. Inform [Nelson, 2001] is one of the most popular, along with TADS [Roberts, 2005, Eve, 2005], Hugo [Tessman, 2004], ALAN [Nilsson and Forslund, 2005], ADRIFT [Wild, 2003], and Quest [Warren, 2004]. AIFT is a new Prolog-based toolkit [Merritt, 2004] inspired by previous work in using IF to teach Prolog [Merritt, 1996], and bears relation to our work in that its rule-based syntax is also amenable to formal verification.

FORMALISM

Narrative Flow Graphs (NFGs) are a class of 1-safe Peri Nets (PNs) that specify some simple abbreviations and additional markings to enforce the narrative flow, and backwards translation is straightforward. For the original Narrative Flow Graph (NFG) formalism and its derivation from 1-safe Petri Nets (PNs) and directed hypergraphs, refer to [Verbrugge, 2002]. Here we introduce a slightly revised but equivalent definition of an NFG in order to build our execution model.

Definition 1 *A* Narrative Flow Graph *(NFG) is a 6-tuple:* (S, T, M, a, w, l)*, where* S *is a set of unconnected places and* T *is a set of transitions such that each* $t = (S_s, S_c, S_d) \in T$ *is connected to* $S_s \subseteq S$ *source places,* $S_c \subseteq S$ *context places, and* $S_d \subseteq S$ *destination places.* M *is the set of markings or reachable states where each* $m \in M$ *is a unique distribution of tokens over* S*, one per place* s*.* a *is an identified axiom place that connects to transitions* $T_{initial} \subseteq T$ *via source edges* $a \rightarrow T_{initial}$ *only, and* w *and* l *are identified win and lose places that connect to transitions* $T_{final} \subseteq T$ *via destination edges* $T_{final} \rightarrow (w|l)$ *only. The graph is initialized to an axiom state or marking* m_a *by filling the axiom place with a token. Transitions are* enabled *when all connected* $s \in (S_s \cup S_c)$ *for a given* t *contain tokens, and can thus* fire*, emptying each* $s \in S_s$ *and filling each* $s \in S_d$*; tokens are* not *removed from any* $s \in S_c$*. Firing is mutually exclusive: although multiple transitions may be enabled, only one fires at a time. The narrative thus flows from* m_a *to either the winning state* m_w *or the losing state* m_l*, via the firing of transitions, and through some intermediate set of markings* $M_i \subseteq (M \setminus \{m_a, m_l, m_w\})$.

Although NFGs as defined allow for the full semantics of goal-oriented storytelling, the details of interactivity are unclear. We now extend the original definition to include input and output connections to transitions, for use in real computer narratives or games.

Definition 2 *An* Interactive NFG *(NFG') is an 8-tuple:* (S, T, M, a, w, l, I, O)*, where* S*,*T*,*M*,*a*,*w*,*l *are defined as before, and* I *and* O *are sets of input commands and output messages respectively, such that each* $i \in I$ *is attached to a single transition* $t \in T$ *and each* $o \in O$ *is attached to any number of transitions* $t \in T$*. An* internal transition $t_i \in T_{internal}$ *has zero input commands and can fire as soon as enabled, and an* action transition $t_a \in T_{actions}$ *has one or more equivalent input commands, and fires iff the system receives a matching input string and there is no enabled* t_i*. Both internal and action transitions may optionally have an output message o attached.*

Thus in an NFG', the narrative flows from axiom to win or lose states as before, but now alternates between waiting for input commands and firing series of transitions based on those commands. Output messages may be produced for the initial command or for any transition that fires as a result, as well as for the transitions that occur between m_a and the first idle state, i.e. during the narrative's prologue. We now redefine NFG to NFG'.

Previously, we also discussed several properties of narratives that can be analysed given the formal structure of an NFG. Among them are 1) *winnability* and *losability* at a given state, or the reachability of m_w and m_l; 2) the *distance* between two markings, or the shortest path between them; 3) the *separation* between two markings, or the longest acyclic path between them; 4) *pointlessness*, the separation between unwinnability and actually losing; and 5) *progress*, the distance between the current marking and m_w. In light of Definition 2, we now redefine these terms to exclude internal transitions. In this initial attempt at verification we concern ourselves only with winnability and losability and the paths to these goals.

SYSTEM OVERVIEW

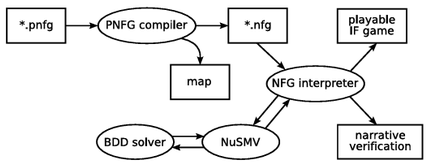


Figure 1: *System overview.*

In Figure 1, an overview of the (P)NFG system can seen. Source narratives in `.pnfg` format are fed to the PNFG compiler and can be used to produce various outputs, including a graphical map of game locations and their connectivity, and low level `.nfg` source files. Generated or handwritten `.nfg` files are then passed to the NFG interpreter which parses the `.nfg` file and creates a dynamic NFG initialized to the axiom state, and then accepts commands from the player until either the winning or losing state is reached, thus forming a playable IF game. The player can also query the interpreter as to the possibility of winning or losing, which causes the interpreter to construct a model of the NFG for the NuSMV model checker. NuSMV in turn depends on a Binary Decision Diagram (BDD) solver backend to find reachability, and ultimately a response is produced for the player.

NFG INTERPRETER

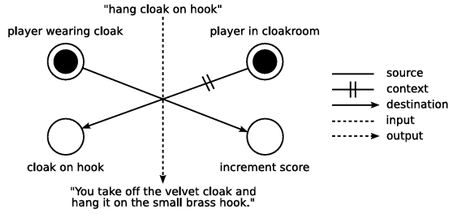


Figure 2: *Example NFG transition.*

An example NFG transition is shown in Figure 2. The player is wearing the cloak (source connection) and is in the cloakroom (context connection). The transition is enabled, since there are tokens in all source and context places. If the player inputs the command, "hang cloak on hook," the transition will fire. This removes the token from "player wearing cloak," keeps the token in "player in cloakroom," and creates new tokens for the "cloak on hook" and "increment score" destination places. Additionally, it causes the message, "You take off the velvet cloak and hang it on the small brass hook," to be printed. This is an *action transition*, as it requires user input to fire. The "increment score" place connects to a separate *internal transition* that fires as soon as enabled and without any user input.

```
transition {
   sources  = player_wearing_cloak;
   contexts = player_in_cloakroom;
   dests    = cloak_on_hook, increment_score;
   inputs   = "hang cloak on hook";
   output   = "You take off the velvet cloak and
               hang it on the small brass hook.";
}
```

Figure 3: *Example NFG source code.*

The `.nfg` source code for this transition is shown in Figure 3. The game specification is simple and consists of one axiom state, one win state, an optional lose state, and a series of transitions. Transitions are nameless, and new places are defined by using a symbol for the first time. A transition may or may not contain sources, contexts, destinations, inputs, or an output, and invalid combinations are weeded at runtime; for example, a transition with no sources or contexts must specify at least one input, for otherwise it would continually fire.

```
main() {

   build AST from .nfg input file;
   build NFG from AST;
   initialize NFG to axiom state;

   while (!won && !lost) {

      while (some t_i ∈ T_internal enabled) {
         fire t_i;
      }

      wait for user input;

      switch (input) {
         case "query win":
            ask NuSMV to find winning state;

         case "query lose":
            ask NuSMV to find losing state;

         case "query moves":
            print each enabled t_a ∈ T_actions;

         case (some enabled t_a ∈ T_actions):
            fire t_a;

         default:
            "Sorry, try something else.";
      }
   }
}
```

Figure 4: *NFG interpreter main().*

Pseudocode for the NFG interpreter `main()` is shown in Figure 4. The game input file is parsed and an abstract syntax tree (AST) constructed. A traversal over the AST is used to build the NFG, and it is initialized to its axiom state. Then an

event loop is entered, which iterates until either the game is won or lost. Inside the loop, first all enabled $t_i \in T_{internal}$ are fired, and this continues until an idle state is reached; the firing of one internal transition will commonly lead to the enabling of another. Once idle, a prompt is displayed, and the player can input a command. Entering "query win" will build a model for NuSMV with the invariant specification being that a winning state is not reachable; if NuSMV can find a counterexample it prints a trace, and a sequence of firing action transitions is extracted. The corresponding input commands are enumerated, thus presenting the player with a minimal winning solution. If no counterexample exists, the player is informed that winning is impossible. Similarly, "query lose" will either produce a losing solution, or inform the player that losing is impossible. Entering "query moves" does not invoke the verifier, and simply prints out input strings for each enabled $t_a \in T_{actions}$. If the player does not enter a query but an actual command, it is matched against an enabled t_a if possible and the transition is fired, otherwise a default "unrecognized command" error message is printed.

The key cost in creating the model for NuSMV, and indeed for any BDD-based verifier, is the number of boolean variables. Naïvely, places require one boolean each, but we use a token-based encoding in which we identify multiple disjoint $S_{mutex} \subseteq S$ where each S_{mutex} has a maximum of one token. Thus the cost for a S_{mutex} becomes $\lceil \log_2(|S_{mutex}| + 1) \rceil$ or $\lceil \log_2 |S_{mutex}| \rceil$ BDD booleans, depending on whether or not the set can have zero tokens. This token-based encoding becomes tedious and error-prone to do manually for large models, and we exploit the high level information available in the PNFG compiler to derive it automatically.

PNFG LANGUAGE AND COMPILER

For complex narratives, directly programming NFGs is impractical. The low level nature of NFGs can result in a large, intricate graph structure, and there is often significant code redundancy that becomes tiresome to manage, e.g. allowing multiple objects to be picked up and dropped in multiple locations. The size and unstructured complexity of the NFG graph can also be a challenge for efficient narrative analysis, and in general benefits from a higher level organisation.

The *Programmable NFG* (PNFG) format is a high level language designed to allow for easy narrative expression while maintaining a direct, efficient, and structured translation to an underlying NFG. Its syntax is close to those of standard IF toolkits, albeit with less features. A basic PNFG program structures the narrative into *object* and *room* declarations forming the core game data, and various *action* declarations implementing the game logic. Objects and rooms can contain state, counter, and timer variables, and action executions themselves are composed of sequences of commands that test, set, and act on game data. Below we discuss how data components are formed and mapped onto NFG structures, followed by the execution semantics and syntax.

Game Data

The PNFG language provides a simple, static structure for narrative game data. Concepts we now present, such as objects, rooms, state and counters are core to interactive fiction and, as we will show later, accommodate even quite complex game narratives.

Objects & Rooms

In a typical narrative game the player interacts with numerous game *objects*. An example PNFG declaration of a game object is given in Figure 5. This declaration results in a single in-game object referred to at runtime and compile-time by the name, "`cloak`."

```
object cloak { }
```

Figure 5: *A simple object declaration.*

A further basic IF design idiom is provided by *room* declarations, and an example is shown in Figure 6. In a typical narrative, rooms are the different, discrete locations where the player and other objects can be located. In practice, rooms are merely objects that also act as containers for other objects, and in fact a player with an inventory is also modeled by a room declaration.

A strict containment hierarchy is implied by the use of rooms. An object may only be in one room at a time, and must also must be contained in some room. A special, predefined `offscreen` room with no parent container operates as a base case and initial location for all game objects.

In order to model object containment, two NFG nodes are generated for each object in each possible location. For an object A and room B a node meaning "A is in B" and a node "A is not in B" are created. Use of these nodes is subject to the strict containment property, and all transformations guarantee that when the node for "A is in B" is active all other nodes "A in C", "A in D" and so on are inactive. This allows us to specify an S_{mutex} for each (object, room) pair containing the "object in room" and "object not in room" nodes.

Alternatively, at the expense of extra transitions, we can generate NFGs without these "not nodes", and then use the strict containment hierarchy to identify much larger S_{mutex}'s such that the cost of each game object is $\lceil \log_2 |R| \rceil$ instead of $|R|$ boolean variables, where R is the set of rooms.

States

Rooms and objects already provide for a simple form of interaction in moving objects from one place to another, and testing for containment. *State* declarations within rooms and objects enable the user to define other, observable binary properties. Figure 6 shows a declaration for a closet which may or may not be lit, and which may or may not be locked.

```
room closet {
   state {lit,locked}
}
```

Figure 6: *A room with 2 declared binary states.*

In the NFG output graph, each state variable (for each defining object) is translated to two nodes, one for each possible value (true or false) that each state variable can have. For Figure 6, four nodes would be generated, `-closet.lit`, `+closet.lit`, `-closet.locked`, and `+closet.locked`. Pairs of nodes for a particular object and state, like the containment relation, maintain a mutual exclusion property and guarantee that exactly one will be active at any one time.

Special state nodes are used to represent winning and losing a game. The built-in object `game` has states `win` and `lose`, with the true (`+`) value of each of those states corresponding to the required NFG win and lose nodes.

Counters and Timers

Counters are used to represent countable properties of rooms or objects, and an example is shown in Figure 7. Counters behave as state variables that can be set, incremented, and decremented to any value in a defined range, and which can also be tested against an arbitrary constant.

```
room you {
   counter {lives 0 3}
}
```

Figure 7: *A counter definition for the inclusive range 0..3.*

In principle, a finite bounded counter can be implemented using just object state declarations and operations. In our current NFG output, we eliminate the "not nodes" needed in such a solution by generating simple unary counters with a single state node for each potential counter value. More efficient binary counter representations and operations are intended for future work; however, as far as verification is concerned, here the cost of a counter is $\lceil \log_2 |C| \rceil$ rather than $|C|$ boolean variables, where C is the set of mutually exclusive counter places.

Counters require programmer code to modify their value. *Timers* are merely special counters which get automatically incremented after every user action is executed.

Execution and Actions

Execution of basic interactive fiction or turn-based adventure narratives consists of first an initialisation or prologue, and then a cycle of listening and responding to user commands, and executing any automatic, internal actions, followed by a finalisation or epilogue [Montfort, 2003]. An equivalent NFG structure is produced by the PNFG compiler, and is shown in Figure 8.

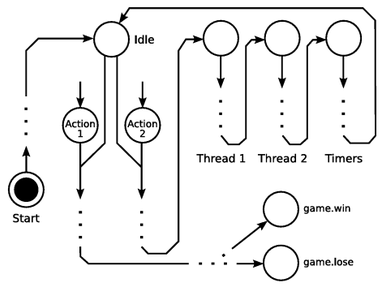


Figure 8: *The general NFG structure for a PNFG program.* The entry points for the main phases of execution are initialisation, user commands, user threads, timers, and finalisation.

A *start* node is the only initially active node; it leads to a series of initialisation activities, terminating at the main *idle* node. At this point user input is processed, activating one of the corresponding stream of actions. This will either terminate in a game win or loss, or eventually pass control to both internal and user-defined *threads,* or automatically executed sequences of instructions. Threads pass control from one to the other, and a special system thread is used to perform automatic timer updates. Finally, control returns to the idle node for another round of user input.

Actual actions consist of sequences of PNFG statements following a conventional procedural language structure. Figure 9 shows a code fragment for a "take all" command in one of the example narratives, *The Count*. Sets of rooms and game items are first defined; the room containing the player (`you`) is then determined, and all game items are considered. If an item is in the same room as `you`, then if you are not overloaded it is moved into `you` (your inventory), and the number of items you are carrying is increased by one. If you have reached full carrying capacity, a message to that effect is emitted instead of taking the object.

```
01  (you,take,all) {
02     places = {hall, kitchen, bedroom, ...}
03     stuff = {sheet, pillow, stake, ...}
04     places $p;
05     if ($p contains you) {
06        for (stuff $s) {
07           if ($p contains $s) {
08              if (you.overloaded) {
09                 "You are carrying too much.";
10              } else {
11                 move $s from $p to you;
12                 you.carried++;
13                 -?you.empty;
14                 if (you.carried==7) {
15                    +you.overloaded;
16  }  }  }  }  }  }
```

Figure 9: *A sequence of PNFG statements corresponding to a "take all" command.* Statements are referred to by number in the text.

Individual actions are sequenced in the NFG using a series of *context* nodes, schematically shown in Figure 10. Each action requires a unique context node as input, and must produce a unique context node on output. Context node activation defines and follows the runtime control flow, and is used to provide the PNFG execution semantics that is not otherwise guaranteed by the underlying NFGs or Petri Nets. Note that context *edges* between nodes and transitions are different: they indicate that the token is not to be removed when the transition fires.

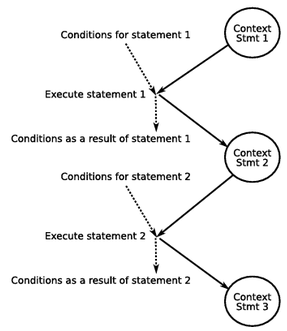


Figure 10: *NFG structure for a sequence of statements.*

Statements

Basic PNFG statements are designed to allow easy narrative game expression while ensuring a well-defined, feasible translation to NFGs. Figure 9 illustrates the most fundamental operations, which are surprisingly few. Below we briefly describe each along with its translation to NFGs.

- *Output.* Standard text output is performed by declaring a constant string, as shown in statement 09. The NFG formulation then consists of a single edge from input to output contexts, annotated to inform the NFG interpreter to emit the specified string. These strings are sent verbatim

to the console, although they could also provide canonical input to a more sophisticated output layer.

- *Move.* Basic object movement is shown in statement 11. Here one of the game items in the `stuff` set declaration, identified by the variable `$s` and found to be in the same room (`$p`) as `you`, is moved from `$p` to `you`. NFG code for a move operation is shown schematically in Figure 11, and consists of toggling the active object-location nodes appropriately.

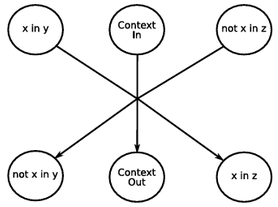


Figure 11: *NFG structure for statement, "*`move x from y to z`*".*

- *Set.* There are several ways to change state variables. In statement 15 the `you.overloaded` state is set to true; this is a blind operation that assumes the state is now false, and will cause control flow to stall if not. If the current state is uncertain, a *safe* set operation changes the state variable if it is not in the desired state already; an example safely setting `you.empty` to false is shown in statement 13. A *toggle* operation flips the state; these three variations are shown as NFG schemata in Figure 12.
- *Counters.* Operations on counters include incrementing and decrementing by a constant value; statement 12 shows a simple increment-by-1 of a counter, and statement 14 shows the use of counters in conditional tests. A schematic unary NFG representation for a counter update is shown in Figure 13. In general, addition or subtraction of a constant c from a counter that can assume r values generates r transitions, each trying to shift the active value node by c. End nodes must contend with potential over/underflow; here the decrement operator becomes the identity operator at the minimum counter value. More efficient math operations are left to future work.

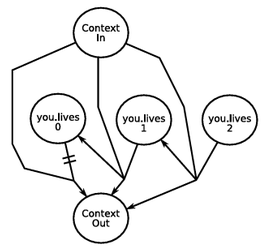


Figure 13: *NFG structure for a counter decrement statement, "*`you.lives--`*".*

- *If.* Conditional tests are allowed on containment, state and counter/timer values. Statement 05 for instance branches on whether the player is contained in any of the rooms in the `places` set. The NFG schema for simple if-statements is shown in Figure 14. Note that if-statements introduce distinct subsequences of statements on both the true and false branches; control flow (context activation) enters only one side, and merges with the other side to form a common output context.

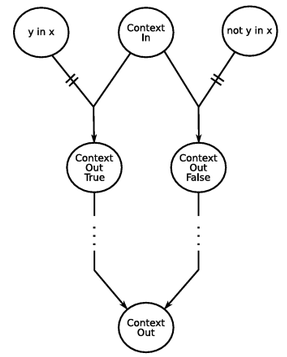


Figure 14: *NFG structure for a statement, "*`if (x contains y) {...} else {...}`*".* Negative containment tests ("`x !contains y`") and positive/negative state tests are structurally identical.

- *Variables & Sets.* Most operations accept object/room specifiers to be *sets* as well as specific objects; this reduces PNFG source code redundancy. For example, statement 02 declares a set called `places` consisting of the `hall`, `kitchen`, `bedroom` and so on. Statement 04 then declares an element of that set, abstractly represented as `$p`. This variable will induce replication of statements using `$p`; the if-statement of line 05, for instance, represents a collection of conditional tests and bodies, one for each object in the `places` set. The NFG schema for variable usage is shown in Figure 15. The code of lines 06–16 is replicated with the tests, each replica having `$p` bound to a distinct specific room. This can be optimised to eliminate redundancy by redirecting control flow to common subnet structures if replicas contain sequences of identical actions.

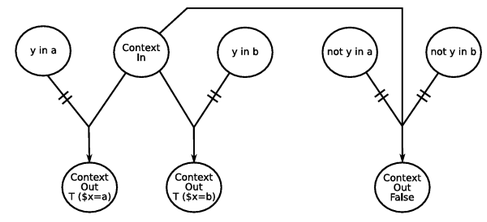


Figure 15: *Using variables.* NFG structure for a statement, "`if ($x contains y) ...`" where "`$x`" is an element of the set "`{a,b}`". Branch bodies and the following merge are not shown.

- *For.* A further use of sets is demonstrated on line 06. In contrast to an if-statement, which executes just one instance of its body even in the presence of set variables, a for-statement executes its body for all possible instantiations of the set variable. A for-statement is trivially expressed as a sequence of body executions, each with the set variable substituted by a different set element.

Actions and Threads

Commands and statements as described above can be executed during initialisation, as a response to user input, or due

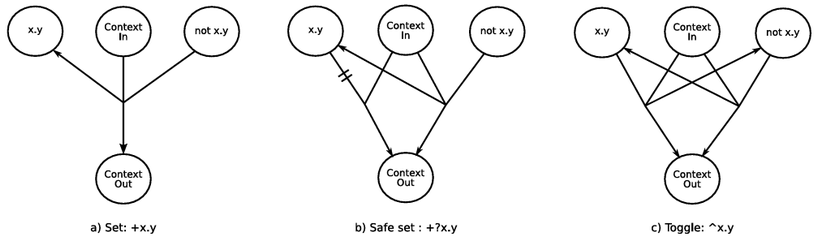


Figure 12: *NFG structure for the 3 main variations of the set statement.* Similar operations are defined for `-x.y` and `-?x,y`.

to automatic processes called *threads.* Figure 8 shows the general relationship between these three structures; here we describe how user actions and threads are defined.

User commands are specified assuming a simplified, canonical language as input. Actions are prefaced by either a (subject,verb) or (subject,verb,object) declaration; when input matching the user command declaration is received, the corresponding sequence of PNFG commands is executed. Currently the subject is always assumed to be "`you`" and ignored during NFG generation. A further syntactic convenience is provided by allowing action declarations to be defined within a room declaration as opposed to globally: this causes the action to be enabled only if the subject is in the enclosing room. Using this feature it is easy to encode the game map through room specific movement commands, as shown in Figure 16.

```
room lighthousefront {
   (you,go,north) {
      "You are now on the mountain pass.";
      move you from lighthousefront to
         mountainpass;
   }
   (you,go,east) {
      "You are now behind the lighthouse.";
      move you from lighthousefront to
         lighthouseback;
   }
   ...
}
```

Figure 16: *Room-specific actions.* These actions shadow global actions with the same user command specification, while the subject (`you`) is in the declared room.

```
thread (bomb.active) {
   if (bomb.ticksLeft==0) {
      "bang!";
      +game.lose;
   }
   bomb.ticksLeft--;
}
```

Figure 17: *A conditional thread declaration.* This thread only executes when the state `bomb.active` is true.

Threads are meant to automate actions that must be done each turn; these would otherwise have to be executed explicitly and redundantly at the end of each action. In the PNFG language threads can either execute unconditionally, or can be predicated on a conditional test, and thus act as "triggered" events. A thread modeling a timer countdown is shown in Figure 17.

In the following section we describe our experience in implementing and analysing several interactive fiction narratives expressed in our system.

EXPERIMENTAL RESULTS

Experimentation with our system at this point is largely focused on ensuring reasonable expressiveness, although we also present some preliminary work on automatic verification. We have selected one simple narrative, two medium-sized story chapters, and one complete and relatively large narrative for our investigations. Below we briefly present basic narrative properties and discuss relevant expression and verification concerns. Maps were generated by using the PNFG compiler to find actions that move `you` between rooms, and laid out using the tool `dot` [Gansner and North, 1999].

1) *Cloak of Darkness* (CoD), is a tiny game that was originally designed to demonstrate the syntax of various IF toolkits to new authors [Firth, 1999]. At the same time it provides a simple sanity check for the most ubiquitous features of IF. While there are only three rooms and three game objects, it includes basic object and room interaction, multiple commands, non-local effects, object state, and (small, finite) counting; an overview map of the game is shown in Figure 18. Expression in our system is straightforward, and data on three variations of it are given in columns 1–3 of Table 1.

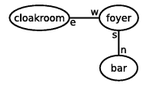


Figure 18: *Map for Cloak of Darkness.*

In *Cloak of Darkness*, the player moves between the foyer, cloakroom, and bar by issuing standard movement commands such as "go east" and "go south". The player starts off wearing a cloak, and as long as it is worn, the bar is obscured by darkness; the player can hang up his cloak on the brass hook in the cloakroom to bring light to the bar and score one point. If the player attempts any non-movement action or an invalid movement action whilst in the darkened bar, a warning message is printed and a counter is incremented. Once the player has lit the bar, it is possible to read a message in the dust on the floor. If the counter has been incremented past some predefined limit, it reads, "You have lost!", otherwise it reads, "You have won!" and the player's score goes up by another point. In either case the game ends immediately.

We first implemented *Cloak of Darkness* as an NFG, and in the absence of a parser mapped multiple variations of a command to a single action transition. We chained sequences of output messages and internal transitions together by having a destination of the first transition be a source of the second, much as context nodes order statements in NFGs generated by the PNFG compiler. We found it was necessary to duplicate a fair amount of code, and that accounting for all

possibilities was error-prone. Even after extensive playtesting, NuSMV was still able to detect a subtle flaw in our implementation: the player could win simply by entering "read message" twice in the foyer at the beginning of the game. The advantage of writing IF at such a low level is that the author has direct control over the evolution of the game state; however, it is somewhat like writing in an assembly language and this soons becomes tedious. These usability factors motivated us to develop the higher-level PNFG representation. In Table 1, "CoD (full)" is more robust and "CoD (tiny)" is very minimal, retaining only the essential semantics.

Table 1: *Basic data on example narratives.* The number of BDD booleans is $\sum\lceil\log_2|S_{mutex}(i)|\rceil$, or the sum of the logarithms of disjoint place sets that maintain a mutual exclusion property, such that there is a maximum of one token in the set at any time. The cost of verification derives from the number of BDD booleans and transitions. All `.pnfg` narratives were compiled without "not nodes".

property	CoD (tiny)	CoD (full)	CoD (full)	RTZ-1 (tiny)	RTZ-1 (full)	RTZ-2 (full)	Count (tiny)	Count (full)
source	.nfg	.nfg	.pnfg	.pnfg	.pnfg	.pnfg	.pnfg	.pnfg
rooms	3	3	4	8	10	21	4	22
objects	3	3	1	7	19	36	3	29
PNFG lines	–	–	544	347	596	1133	244	2162
places	21	69	303	366	1275	1876	272	15378
transitions	45	167	462	850	3341	8030	459	82371
BDD booleans	21	69	27	42	98	117	30	212
verifiable	yes	yes	yes	yes	no	no	yes	no
steps to win	5	6	6	5	6	22	5	180

The PNFG version is structured somewhat differently from the hand coded NFG versions. Here only the cloak is defined as a PNFG object, and the other two immobile objects are encoded through state variables and messages. An extra room is also used to encode the player's inventory ("`you`"). The resulting NFG is about three to four times as large as the hand coded version, illustrating the relative cost of a high level structure and our current, quite naïve code generation strategies. However, due to high level knowledge about mutually exclusive places, we were actually able to generate a *more* efficient model for verification that used fewer boolean variables.

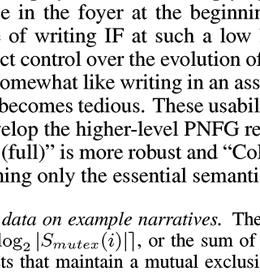


Figure 19: *Map for Return to Zork, chapter 1.*

2) *Return to Zork* (RTZ) [Barnett, 1993] is a large and complex graphical adventure, set in the same world as its pioneering IF ancestor *Zork* [Lebling et al., 1979]. We chose to translate this game to PNFG source code for two reasons. First, we wanted to demonstrate the ability of our system to model narratives that are not strictly text-based, and second, the representation of RTZ is greatly aided by the fact that it can be divided into specific chapters [Spear, 1994].

In columns 4–6 of Table 1 we show data on the first two chapters of RTZ, and in Figures 19 and 20 we show the corresponding maps. We also include a tiny variant of Chapter 1 that lacks many interactions, but that will verify due to the fewer BDD booleans and transitions. The full versions of the chapters still exceed the current capacity of our analyser; however, chapter sizes are in general within an order of magnitude of the size of a small, analysable narrative like CoD. A chapter-based division may thus be sufficient, as well as perhaps necessary for practical narrative analysis.

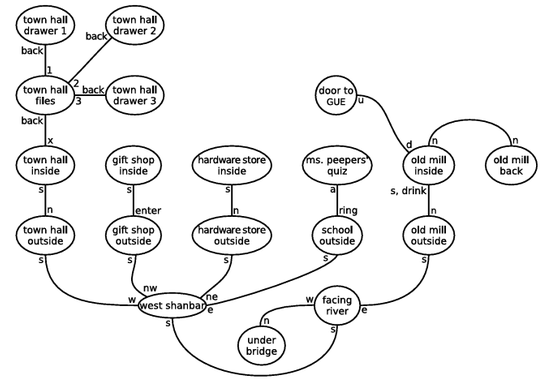


Figure 20: *Map for Return to Zork, chapter 2.*

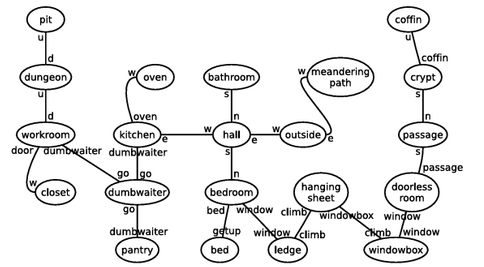


Figure 21: *Map for the full version of The Count.* Not shown is that the `sleep` command, as well as an automatic timeout at nightfall will return `you` to the bed from any room.

3) *The Count* [Adams, 1981] is one of the original Scott Adams adventure games, a great source of classic IF narratives. This game is smaller than the sum of the RTZ chapters, but at least as complex in terms of narrative structure. Here we have implemented both a partial test of the initial 4 game rooms and 3 objects (column 7), and the full version with 22 rooms and 29 objects, which includes multiple timers, counters, and user threads (column 8). The map for the full version is shown in Figure 21. The minimal solution depth for the full version is 1–2 orders of magnitude longer than those of our other narratives, giving a further indication of relative complexity.

The Count possesses a number of properties of interest to verification. At several points progress can become *pointless,* owing to loss of an essential item (e.g., the stake, cigarette) or expiration of different time limits. There are also a number of subtle story logic flaws, such as a locked closet that can be used to prevents the antagonist's access to some objects (the stake) but not others (the sheet). The latter problem is highly game specific, but in general, custom verification queries could be used within our system to check important aspects of narrative consistency.

As a complete and non-trivial game *The Count* provides a good benchmark goal for our system. At this stage it is much too large to analyse formally; however it gives a good indication as to the scale of a realistic problem space. Segmenting the narrative into separately analyseable portions may reduce the complexity, and an automatic system for doing so is part of our future work.

CONCLUSIONS & FUTURE WORK

The complexity of the structures generated for our larger narratives implies a need to significantly improve the process of verification. Use of chapters and other narrative decompositions can certainly help, as seen for *Return to Zork*, and exploring different verification strategies and Petri Net encodings can greatly affect the analysis cost. For example, the elimination of "not nodes" from our original design reduced the number of BDD booleans required for *The Count* from 890 to 212, although the number of transitions roughly doubled. Another strategy that appears quite promising is identification of individual puzzles, or groups of strongly related tasks and state variables, followed by construction of more modular, hierarchical Petri Net models.

An advantage of our system is that we can exploit the high-level PNFG structure during verification, and in general the complete PNFG→NFG→NuSMV path provides many opportunities for optimisation. It will also be interesting to analyse other properties besides winnability and losability, such as pointlessness, and to be able to provide the player with answers to questions such as, "How do I get this door unlocked?"

As far as usability of the system is concerned, we have not conducted any kind of user study outside of our own narrative development. Our system is a prototype design, and does not support many advanced programming features provided by other IF toolkits, such as object orientation, inline VM assembly, and animated graphics, or their robust standard libraries that enable parser customization, multiple world models, and NPC interaction. However, in terms of the features that (P)NFG does currently support, we find it to be comparably usable.

Furthermore, interoperability with other IF toolkits is an important goal. In this study we have translated narratives by hand to get them into our input format; a direct, automatic translation of game specifications, however, would allow us to more efficiently examine a much larger body of works [Kinder et al., 2005]. Of course, the complex syntax and details of different language specifications make this a non-trivial technical challenge.

Our system is playable as interactive fiction, but quite minimal. Adding natural language processing, a staple of most IF environments, would certainly make game play more true to the genre. This would be conceptually straightforward to add as a component that generates and reacts to our canonical input and output. Arbitrary forms of I/O could be connected similarly, allowing many multimedia effects, and expansion into other turn-based genres.

Narratives are a basic and ubiquitous component of computer games, and writing complex and error-free computer narratives is a difficult task that affects many genres, not only interactive fiction. Modern games are large, intricate software programs, and formal approaches and analyses stand to benefit both developers and players. The (P)NFG language, compiler, and runtime system provides a formal structure for narrative analysis, and helps move the burden of narrative debugging away from the play tester and into software tools.

ACKNOWLEDGEMENTS

We would like to thank Alessandro Cimatti for his help with NuSMV. This research was funded by NSERC and FQRNT.

REFERENCES

[Adams, 2005] Adams, E. (1998–2005). The designer's notebook: Bad game designer, no twinkie! parts I–VI. `http://www.gamasutra.com`.

[Adams, 1981] Adams, S. (1981). The Count. Adventure International. `http://www.msadams.com`.

[Arnold et al., 1995] Arnold, J., Baggett, D., Clements, M., Russoto, M. T., Newland, J., Plotkin, A. C., and Shiovitz, D. (1995). Game design in general. Discussion thread in `rec.arts.int-fiction` archives.

[Barnett, 1993] Barnett, D. (1993). Return to Zork. Activision Publishing, Inc.

[Bobbio and Horváth, 2001] Bobbio, A. and Horváth, A. (2001). Model checking time petri nets using NuSMV. In *Proceedings of the 5th International Workshop on Performability Modeling of Computer and Communication System (PMCC'05)*, Erlangen, Germany. Extended abstract.

[Brooks, 1996] Brooks, K. M. (1996). Do story agents use rocking chairs? The theory and implementation of one model for computational narrative. In *Proceedings of the fourth ACM International Conference on Multimedia*, pages 317–328, Boston, Massachusetts.

[Bryant, 1992] Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318.

[Burg et al., 2000] Burg, J., Boyle, A., and Lang, S.-D. (2000). Using constraint logic programming to analyze the chronology in "A rose for Emily". *Computers and the Humanities*, 34(4):377–392.

[Charles et al., 2002] Charles, F., Mead, S. J., and Cavazza, M. (2002). Generating dynamic storylines through characters' interactions. *International Journal of Intelligent Games & Simulation*, 1(1):5–11.

[Ciardo, 2004] Ciardo, G. (2004). Reachability set generation for petri nets: Can brute force be smart? In *Proceedings of Applications and Theory of Petri Nets 2004: 25th International Conference (ICATPN'04)*, volume 3099 of *LNCS*, pages 17–34, Bologna, Italy. Springer-Verlag.

[Cimatti et al., 2002] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV version 2: An opensource tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 359–364, Copenhagen, Denmark. Springer-Verlag.

[Eladhari, 2002] Eladhari, M. (2002). Object oriented story construction in story driven computer games. Master's thesis, Stockholm University.

[Esparza, 1998] Esparza, J. (1998). Decidability and complexity of petri net problems—an introduction. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 374–428. Springer-Verlag.

[Eve, 2005] Eve, E. (2005). *Getting Started in TADS 3: A Beginner's Guide, version 3.0.8*. `http://tads.org`.

[Firth, 1999] Firth, R. (1999). Cloak of darkness. `http://www.firthworks.com/roger/cloak/`.

[Forman, 1997] Forman, C. E. (1997). Game design at the drawing board. *XYZZY News*, 4:5–11.

[Gansner and North, 1999] Gansner, E. R. and North, S. C. (1999). An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233.

[Heiner and Menzel, 1998] Heiner, M. and Menzel, T. (1998). A petri net semantics for the PLC language instruction list. In *Proceedings of the Fourth Workshop on Discrete Event Systems (WoDES '98)*, pages 161–172, Cagliari, Italy.

[Hutchings, 2004] Hutchings, G. (2004). *IFM: Interactive Fiction Mapper, version 5.1 manual*. `http://www.sentex.net/~dchapes/ifm/`.

[Kakas and Miller, 1997] Kakas, A. C. and Miller, R. (1997). A simple declarative language for describing narratives with actions. *Journal of Logic Programming*, 31(1-3):157–200.

[Kinder et al., 2005] Kinder, D., Granade, S., Blasius, V., and Baggett, D. M. (2005). The interactive fiction archive. `http://www.ifarchive.org`.

[Lebling et al., 1979] Lebling, P. D., Blank, M. S., and Anderson, T. A. (1979). Zork: A computerized fantasy simulation game. *IEEE Computer*, 12(4):51–59.

[Mateas, 1997] Mateas, M. (1997). An Oz-centric review of interactive drama and believable agents. Technical Report CMU-CS-97-156, School of Computer Science, Carnegie Mellon University.

[Merritt, 1996] Merritt, D. (1996). *Adventure in Prolog*. Amzi! Inc., 5861 Greentree Road, Lebanon, Ohio 45036 USA.

[Merritt, 2004] Merritt, D. (2004). AIFT: Amzi! Interactive Fiction Toolkit. `http://www.ainewsletter.com/downloads/if_docs/`.

[Montfort, 2003] Montfort, N. (2003). *Twisty Little Passages*. The MIT Press.

[Natkin and Vega, 2004] Natkin, S. and Vega, L. (2004). A petri net model for computer games analysis. *International Journal of Intelligent Games & Simulation*, 3(1):37–44.

[Nelson, 2001] Nelson, G. (2001). *The Inform Designer's Manual*. The Interactive Fiction Library, PO Box 3304, St Charles, Illinois 60174, USA, 4th edition.

[Nilsson and Forslund, 2005] Nilsson, T. and Forslund, G. (2005). *The ALAN Adventure Language, version 3.0dev36 manual*. `http://www.alanif.se`.

[Pastor et al., 2001] Pastor, E., Cortadella, J., and Roig, O. (2001). Symbolic analysis of bounded petri nets. *IEEE Transactions on Computers*, 50(5):432–448.

[Purvis, 2004] Purvis, M. K. (2004). Narrative structures for multi-agent interaction. In *2004 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2004)*, pages 232–238, Beijing, China. IEEE Computer Society.

[Reiter, 2000] Reiter, R. (2000). Narratives as programs. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 99–108, San Francisco. Morgan Kaufmann.

[Roberts, 2005] Roberts, M. J. (1987–2005). TADS: The Text Adventure Development System. `http://tads.org`.

[Spear, 1994] Spear, P. (1994). *Return to Zork - The Official Guide to the Great Underground Empire*. BradyGames.

[Stotts and Furuta, 1989] Stotts, P. D. and Furuta, R. (1989). Petri-net-based hypertext: document structure with browsing semantics. *ACM Transactions on Information Systems (TOIS)*, 7(1):3–29.

[Tessman, 2004] Tessman, K. (2004). *The Hugo Book—Hugo: An Interactive Fiction Design System*. The General Coffee Company Film Productions, Toronto, Canada, 1st edition. `http://www.generalcoffee.com`.

[van der Aalst, 2002] van der Aalst, W. (2002). *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press.

[Vega et al., 2004] Vega, L., Grünvogel, S. M., and Natkin, S. (2004). A new methodology for spatiotemporal game design. In *Proceedings of the CGAIDE'2004, Fifth Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 109–113.

[Verbrugge, 2002] Verbrugge, C. (2002). A structure for modern computer narratives. In *CG'2002: International Conference on Computers and Games*, volume 2883 of *LNCS*, pages 308–325.

[Warren, 2004] Warren, A. (2004). *Quest Documentation, version 3.51*. Axe Software. `http://www.axeuk.com/quest/`.

[Wild, 2003] Wild, C. (2003). *ADRIFT: Adventure Development & Runner—Interactive Fiction Toolkit, version 4.0 manual*. `http://www.adrift.org.uk`.

[Young, 2005] Young, R. M. (2005). Cognitive and computational models in interactive narrative. In Forsythe, C., Bernard, M. L., and Goldsmith, T. E., editors, *Cognitive Systems: Human Cognitive Models in Systems Design*. Lawrence Erlbaum. To appear.

[Ziaei and Agha, 2003] Ziaei, R. and Agha, G. (2003). SynchNet: A petri net based coordination language for distributed objects. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, volume 2830 of *LNCS*, pages 324–343.

A PATTERN CATALOG FOR COMPUTER ROLE PLAYING GAMES

C. Onuczko, M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy,
K. Waugh, M. Carbonaro* and J. Siegel
Department of Computing Science, *Department of Educational Psychology
University of Alberta, Edmonton, AB, Canada T6G 2E8
{onuczko, meric, duane, jonathan, mcnaught, troy, siegel }@cs.ualberta.ca, *mike.carbonaro@ualberta.ca

KEYWORDS
Generative design patterns, scripting languages, code generation, computer games.

ABSTRACT

The current state-of-the-art in computer games is to manually script individual game objects to provide desired interactions for each game adventure. Our research has shown that a small set of parameterized patterns (commonly occurring scenarios) characterize most of the interactions used in game adventures. They can be used to specify and even generate the necessary scripts. A game adventure can be created at a higher level of abstraction so that team communication and coding errors are reduced. The cost of creating a pattern can be amortized over all of the times the pattern is used, within a single adventure, across a series of game adventures and across games of the same genre. We use the computer role-playing game (CRPG) genre as an exemplar and present a pattern catalog that supports most scenarios that arise in this genre. This pattern catalog has been used to generate ALL of the scripts for three classes of objects (placeables, doors and triggers) in BioWare Corp.'s popular Neverwinter Nights CRPG campaign adventure.

MANUAL SCRIPTING

A computer game typically contains a game engine that renders the graphical objects and characters, and manages sound and motion. A programming team writes a game engine that can be re-used across multiple game adventures and enhanced for future games. They also produce a set of computer aided design (CAD) tools that are used by a team of writers, artists, musicians, voice actors and other skilled craftspeople to create content such as backgrounds, models, textures, creatures, props, sounds, and music that are shared across game adventures. Adventure (story) designers also use these tools to create individual adventures. For example, Figure 1 shows BioWare's (http://www.bioware.com/) Aurora Toolset that is used to create game adventures for Neverwinter Nights (NWN) (http://nwn.bioware.com/).

A game engine typically dispatches game events to scripts that support interactions between the player character (PC) and game objects. These interactions vary for each game adventure and programmers must write the scripts that control them. For example, an adventure designer may want the PC to agree to complete a quest before allowing the PC to enter a castle. To ensure that the quest is accepted, a heat source is placed close to the castle door that prevents the PC from getting close enough to the door to use it. The designer provides a non-player character (NPC) with a cloak of fire resistance that will be given to the PC after the PC has agreed to undertake the quest

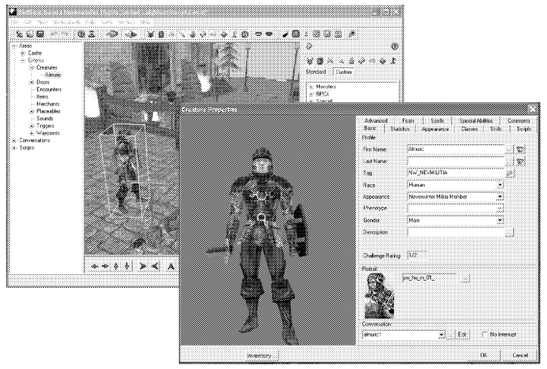


Figure 1: The Aurora Toolset CAD Tool for NWN Adventure Designers

```
void main() {
  object Enterer;
  object Cloak;
  object FireCenter;
  location jumperLocation;
  vector jumperPosition;
  float jumperFacing;
  object jumperArea;
  vector targetPosition;
  vector heading;

  Enterer = GetEnteringObject();
  Cloak = GetObjectByTag("CloakofFireResistance");
  if( ! GetItemInSlot(INVENTORY_SLOT_CLOAK, Enterer)
    == Cloak) {
    ApplyEffectToObject(DURATION_TYPE_INSTANT,
      EffectVisualEffect(VFX_IMP_FLAME_S), Enterer);
    FireCenter = GetObjectByTag("firecenter");
    jumperLocation = GetLocation(Enterer);
    jumperPosition = GetPositionFromLocation
      (jumperLocation);
    jumperArea =
      GetAreaFromLocation(jumperLocation);
    jumperFacing = GetFacingFromLocation
      (jumperLocation);
    targetPosition = GetPositionFromLocation
      (GetLocation(FireCenter));
    heading = Vector(targetPosition.x -
      jumperPosition.x, targetPosition.y -
      jumperPosition.y, targetPosition.z -
      jumperPosition.z);
    heading = VectorNormalize(heading);
    jumperPosition = jumperPosition - 2.5*heading;
    jumperLocation = Location(jumperArea,
      jumperPosition, jumperFacing);
    AssignCommand(Enterer, JumpToLocation(
      jumperLocation));
    FloatingTextStringOnCreature
      ("The heat is too strong.", Enterer, FALSE);
  }
}
```

Figure 2: An NWScript Script for a Barrier

The adventure designer may ask a programmer to implement this scenario as scripts attached to game objects. One of these scripts prevents the PC from getting to the door unless the PC is wearing the cloak. The other script gives the cloak

to the PC, after the PC agrees to undertake the quest. Figure 2 shows the script that prevents the PC from getting near the door, written in NWScript, the NWN scripting language. This script is long and complex, containing local variables, literal tags (representing objects created by the designer using the Aurora Toolset), conditional expressions and many function calls to the NWScript library. A professional game designer without extensive programming experience could not write this script.

Computer games typically have thousands of game objects that must be scripted. There are four serious disadvantages to scripting these objects manually: 1) poor script tracking, 2) simplistic scripts that take too long to write, 3) scripting errors and 4) team communication problems.

Manually written scripts are *hard to track*. The large number of game objects makes it difficult to organize and track objects during adventure development. Organizing and tracking scripts is even more difficult since most scripts involve interactions between several objects. A change to an object or a script often results in unexpected negative consequences.

Most scripts provide only *simplistic game behaviors*. Since a large number of objects require scripts, all but the most important objects must have very simple scripts. Unless an object is on the critical path of the main plot line, it usually has a single trivial script. This makes the game repetitive and predictable, and therefore boring. More interesting interactions are desirable, but are not cost effective to write because of the large time investment needed. Even scripting simple behaviors consumes *too much programmer time*, which could be better spent on developing better game engines and additional tools for the game designers.

Many common *scripting errors* are difficult to detect without manually playing through all of the game scenarios and trying all of the different combinations of user choices. For example, scripts are often created using cut-and-paste techniques and it is not uncommon for the programmer to cut-and-paste scripts without making all the changes needed for the new context. There are so many game objects and scripts that it has become standard practice to use object numbers or script numbers as part of their names. An off-by-one error in a name often results in a legal script that performs incorrectly.

Many designers are unable to write scripts themselves and must rely on programmers. Delegating the scripting to a programmer can lead to inconsistencies between the game designer's intent and the programmer's scripts due to *communication errors*.

At least professional adventure designers have access to programmers who can write scripts for them. Recently, there has been a trend to create computer games where amateur designers can create adventures and post them online for others to play. For example, NWN has an active adventure designer community. Thousands of players contribute adventure modules of their own creation. Contributed modules can be freely downloaded from the Neverwinter Nights Vault web site (http://nwvault.ign.com). The most popular of the 4,100 community-created modules at this site has been downloaded over 252,000 times (as of May 2005), and the tenth-most, 88,000 times. Most of these community designers are non-programmers, so they cannot write the scripts themselves. Instead, they try to copy existing scripts from other adventures and paste them into their own adventures. This does not work very well, since the copied scripts usually have many adventure-dependent literal tags and other context dependent code. Therefore, the adventure designers turn to the community for help by posting to one of the NWN scripting forums. There have been about 150,000 scripting-related postings to the BioWare forums (http://nwn.bioware.com/forums/viewforum.html?forum=47) (as of May 2005). Unfortunately, the help they receive from these forums is often not very useful (Carbonaro et al 2005).

FROM SCRIPTING TO PATTERNS

Our approach to solving these problems is to provide an alternative to manual scripting, based on a higher-level abstraction than scripting code. Our first goal was to identify a set of patterns that describe all of the object interactions that commonly occur in CRPGs. A pattern is a familiar commonly occurring scenario or idiom in an adventure of the appropriate genre. An example of a pattern in the CRPG genre is a secret door that appears when a protagonist gets close enough to it and notices it. Our second goal was to build a tool that uses this set of patterns to generate the scripting code automatically.

Our work is inspired by the use of design patterns to describe object collaborations (Gamma et al. 1994) in general purpose programs. A design pattern specifies the solution to a software design problem at a higher level of abstraction than a program that implements the design. For example, recall the scenario from the previous section. It required two scripts. One script was used to prevent the PC from getting close to the door. The other script allowed an NPC to give a protective cloak to the PC after a conversation had taken place. We provide the game designer with a set of re-usable patterns that can be used to specify such scripts at a higher level of abstraction. The adventure designer would still use two scenes, the conversation and the encounter near the door. We provide two general patterns that can be adapted for these scenes and for many other scenes as well.

In this example, the game designer uses two patterns, *Trigger enter/exit – barrier* and *Conversation when/what*. The first pattern prevents a PC from entering or exiting a trigger region and the second pattern controls whether a particular conversation point can be reached and makes something happen if it is reached. Patterns support adventure design at a higher level of abstraction. For example, the first pattern applies whenever the designer wants to prevent the PC from getting into or out of a region. It is not restricted to being near a door or due to a heat source. Patterns reduce and clarify design team communication, since low-level details do not obscure the designer intent. Designers and programmers can quickly grasp the meaning of a *barrier* pattern. Patterns foster re-use across scenarios in the design of a particular adventure, across the design of multiple adventures for a single game and even across games of the same genre.

A traditional software design pattern is *generic*. It provides a set of solutions to a general design problem (Gamma et al. 1994). The pattern must be adapted to a specific context during application design. The designer refines the design solution family to a single solution in the context of the application.

Our CRPG patterns are also generic. Each pattern must be adapted to a specific scenario by the adventure designer. For example, the *Trigger enter/exit – barrier* has several options, including a *trigger*, a *distance* and a *visual effect*. The *trigger* is a polygonal region that the designer paints into the adventure using a computer-aided design tool. When a character in the game steps into or out of the region, the game engine generates an *onEnter* or *onExit* trigger event respectively and the appropriate script attached to the trigger object is executed. The *trigger* option of the barrier defines the region that a creature cannot enter or exit. The *distance* option defines the distance that the creature trying to cross the barrier will bounce back from the barrier. The *visual effect* option defines the visual effect that will occur when the creature tries to cross the barrier. These options are "hard-wired" into the script that appears in Figure 2, but not into the pattern. Using a pattern is much more generic and safer than cutting and pasting and changing these "hard-wired" values. Setting options is only the simplest form of seven kinds of adaptation that can be used to adapt CRPG patterns. These other forms of adaptation differentiate a pattern from a simple function call and are discussed in the next section. The important idea is that each pattern defines a *generic* family of solutions that must be adapted to the particular context of a game adventure, analogously to the way traditional software design patterns must be specialized to the context of an application.

A traditional software design pattern is *descriptive* (Gamma et al. 1994). Each pattern provides a design lexicon, describes a set of solution structures and describes the reasoning behind the solutions. A programmer selects an appropriate descriptive design pattern, adapts it to the application program and manually *translates* it to code. Experienced programmers who have implemented the same design pattern in other contexts can usually perform adaptation and coding more quickly than novice programmers, where unfamiliar or ambiguous natural language pattern documentation can lead to slow progress and coding errors. Our CRPG patterns can be used descriptively so that the adventure designer can communicate adventure designs more concisely and accurately to a programmer who can implement these descriptive patterns by writing scripting code. However, there is another kind of design pattern.

A *generative design pattern* (GDP) (Budinsky et al. 1996) (Florijn et al. 1997) (MacDonald et al. 2002) has all of the characteristics of a descriptive pattern. In addition, it generates application code automatically so that a programmer does not have to manually translate the pattern to code. Novice and experienced designers can use GDPs to speedup code production and reduce coding errors. Recently, we have used GDPs in computer games for the first time (McNaughton et al. 2003). If GDPs are used, then the adventure designer gains one more advantage from using patterns rather than manual scripting. GDPs allow a game designer to generate scripting code automatically without any need for programmer intervention. This eliminates any chance of communication error between the adventure designer and the programmer.

The pattern catalog presented in this paper is supported as a generative pattern catalog in a tool called ScriptEase (http://www.cs.ualberta.ca/~script/scripteasenwn.html) that automatically generates scripting code for NWN. An adventure designer can also use this pattern catalog to create descriptive design patterns for any CRPG. In this case, each pattern serves as a template for the explicit specification of a scenario that can be implemented by a script programmer. A pattern catalog is not a static entity. It is meant to evolve by expanding (and contracting) as appropriate to satisfy the needs of adventure designers. Therefore, ScriptEase also includes a pattern design tool, which allows adventure designers to modify existing patterns and to create new ones.

PATTERN ADAPTATION

To understand pattern adaptation, it is necessary to understand the component parts of a pattern (McNaughton et al. 2004b). Each *pattern* contains one or more event-driven scenarios called situations. Each *situation* (icon *S*) contains the *event* (icon *V*) that activates it and a set of *definitions* (icon *D*), *conditions* (icon *C*) and *actions* (icon *A*). For example, Figure 3 shows the components of the *Trigger enter/exit – barrier* pattern.

The first situation has been opened to show its components, but the other three are closed (for brevity). There are no conditions in this pattern. The first action *(jumps towards with effect*) is an example of an *action encounter*, which is a re-usable action that contains other actions. It has four options, *Jumper* – bound to *Enterer*, *Target* – bound *to The Center*, *Distance* – bound to *Negative Bounce Distance*, and *Impact Effect* – bound to *Touch Effect*.

To use a pattern, a designer creates an instance of the pattern and adapts it for a specific scenario. The simplest form of adaptation is to set the pattern options as described previously. However, setting options provides only limited abstraction, equivalent to setting function parameters and is not sufficient for the kinds of adaptation needed to support CRPG patterns. Other forms of adaptation include adding or removing components. Table 1 lists the various kinds of adaptation in increasing order of complexity.

Table 1: Kinds of Pattern Adaptation in Increasing Order of Complexity

1.	Setting options
2.	Removing situations
3.	Removing actions and definitions
4.	Removing conditions
5.	Adding actions and definitions
6.	Adding conditions
7.	Adding situations

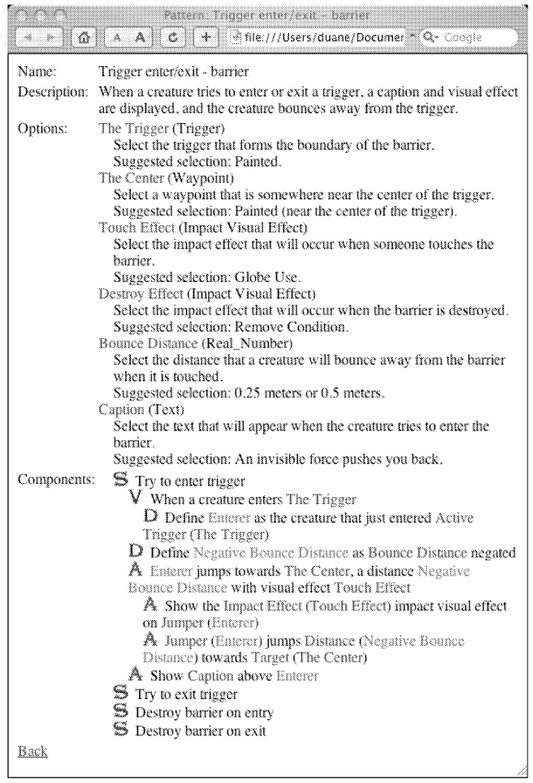


Figure 3: The *Trigger enter/exit - barrier* Pattern

To use this pattern for the scenario described in the previous section, the designer adapts the instance by:

1. setting the options to the appropriate objects and values: *The Trigger* – a region near the door ("Firetrigger"), *The Center* – a waypoint near the center of the trigger ("firecenter"), *Touch Effect* – A flame visual effect (VFX_IMP_FLAME_S), *Destroy Effect* – not used, *Bounce Distance* – 2.5, *Caption* – "The heat is too strong.",

2. removing the unwanted scenarios: *Try to exit trigger*, *Destroy barrier on entry* and *Destroy Barrier on exit*, and

3. adding a definition and a condition so that the barrier will not work on a creature that is wearing the cloak.

After adapting this instance, it looks like the pattern in Figure 4. This instance can serve as a specification for a programmer. Alternately, if the adventure designer is designing for NWN, then ScriptEase can be used to generate the scripting code automatically.

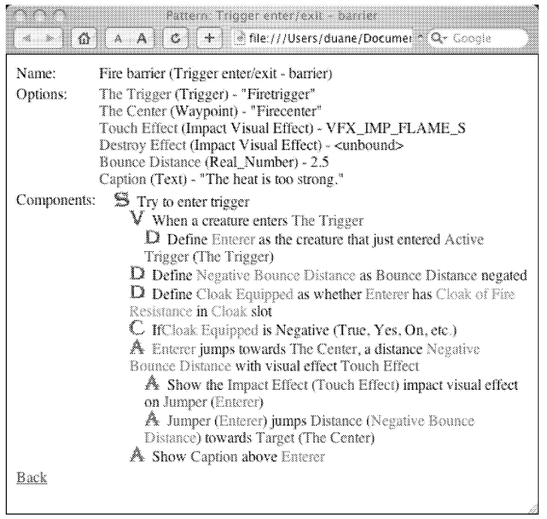


Figure 4: An Adapted Instance of the *Trigger enter/exit - barrier* Pattern

THE PATTERN CATALOG

We have identified four kinds of patterns that are necessary to generate all of the scripts found in CRPGs: *encounter*, *dialogue*, *behavior*, and *plot*. In total our pattern catalog has 60 patterns, consisting of 56 encounter patterns, 1 dialogue pattern and 3 behavior patterns. We are actively engaged in adding more patterns, especially dialogue, behavior and plot patterns. Our pattern catalog is available online at http://www.cs.ualberta.ca/~script/patterncatalog/.

An *encounter pattern* is used to script an interaction between the PC and an inanimate game object. It is useful to divide inanimate objects into groups that can be interacted with in different ways. Three examples of inanimate object groups are: *placeables*, *doors* and *triggers*. A placeable is an inanimate object that can be placed anywhere in the story world. Examples include chests, statues, chairs, tables, levers, and piles of rubble. A placeable is considered a *container* if it can hold items. A *door* can only be placed at the entrance to a structure or between two rooms in a structure. A *trigger* is a region of space that generates an event when a character enters or exits its perimeter. The *Trigger enter/exit – barrier* pattern described earlier is an example of an encounter pattern. The pattern catalog contains 28 placeable, 15 door and 13 trigger encounter patterns, for a total of 56 encounter patterns.

A *dialogue* pattern is used to control conversations. A tree is a common model for conversations in an interactive adventure. At alternate levels in the tree, either the game player selects a conversation node from those available for the PC, or a script selects a conversation node for the NPC. Figure 5 shows an example NWN conversation tree in ScriptEase, for the scenario described previously. Nodes marked [OWNER] (red) are for the NPC and the other nodes (blue) are for the PC.

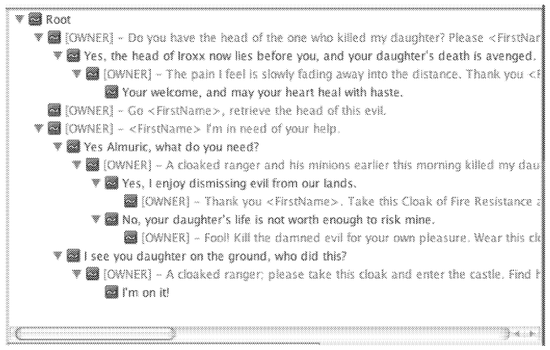


Figure 5: A NWN Conversation Tree in ScriptEase

There are actually two kinds of scripts that can be attached to a conversation node. A *when* script evaluates a *Boolean* that indicates whether the node should appear in the conversation or not. A *what* script provides actions that are taken if the conversation node is reached. Our pattern catalog currently contains a single generic dialogue pattern. The *Conversation when/what* pattern allows the adventure designer to generate *when* and *what* scripts for a conversation node. The sample scenario described earlier can be created using an instance of this pattern. Figure 6 shows the instance of this pattern that is attached to the conversation node "[OWNER] – Thank you <FirstName>. Take this Cloak …". This pattern instance transfers the cloak from the NPC to the PC and fires a visual effect. In general, this pattern has two situations: *When displayed* and *What actions*. The designer has deleted the first situation during adaptation, since this node should always be displayed if its parent node in the tree is displayed.

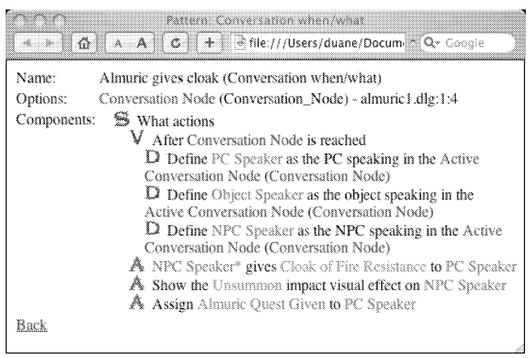


Figure 6 An Adapted Instance of the *Conversation when/what* Dialogue Pattern that uses the *What actions* Situation

Figure 7 shows an example of using this pattern to control whether a conversation node appears in a conversation or not. The adventure designer would like the first [OWNER] node in Figure 5 to appear only if the PC has completed the quest, and the second [OWNER] node to appear only if the PC has accepted the quest, but not yet completed it. Notice from Figure 6 that the PC is given a plot token called *Almuric Quest Given* after agreeing to complete the quest. This plot token can be used in a *Conversation what/when* pattern to hide the second [OWNER] node until the PC has obtained the plot token. Figure 7 shows an adapted instance of the *Conversation what/when* pattern that achieves this objective.

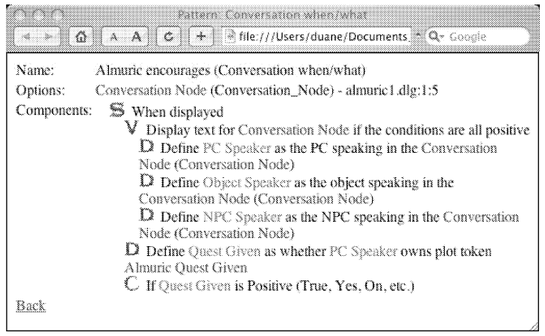


Figure 7: An Adapted Instance of the *Conversation what/when* pattern Dialogue Pattern that uses the *When displayed* Situation

A different instance of this pattern is used to hide the first [OWNER] node until the PC returns with the head of the evil-doer. This pattern is not split into two separate patterns since there are often times when a conversation node is both guarded by a "when" and requires "what" actions. This single dialogue pattern in our pattern catalog can be used to control all conversations on a node-by-node basis. We are currently developing other dialogue patterns that can be used at a higher level of abstraction to model frequently occurring conversation patterns consisting of many nodes.

A story designer can use a *behavior pattern* to specify the actions of an NPC. For example, the adventure designer may want an NPC to stay near an object and to start a dialogue whenever the PC gets close to that object. Our pattern catalog has a pattern called *Creature heartbeat – (PC near object) show dialogue* that supports this behavior. There are three behavior patterns in our catalog at the current time and we are actively adding more behavior patterns.

A plot pattern guides the player character (PC) through the story. For example, in CRPGs it is common to give the player quests. The player advances through the quest in a series of states: *unassigned*, *assigned*, *resolved* and *closed*. A common way to have the player participate in a quest is through a dialogue with a non-player character (NPC), which consists of a series of conversations. The dialogue pattern *Simple verbal quest* specifies which conversation is used for each of the various states of the quest. This pattern depends on other patterns to set a plot token which causes the quest state to change. External patterns are used to provide flexibility since a quest can involve solving a riddle posed by a different NPC, defeating a creature, opening a door, obtaining a specific item, etc. We are currently building a basic set of plot patterns to add to our catalog. Although there are currently no plot patterns in our catalog, many are under development. In the meantime, we have introduced the concept of a *plot token* as illustrated in the previous example.

EVALUATION OF THE PATTERN CATALOG

In a previous paper (McNaughton et al. 2004a), we described how we used encounter patterns to generate all of the scripting code attached to placeable objects in the NWN official campaign story. In that experiment, we replaced 497 calls to 182 different scripts comprising 1925 non-comment lines of hand-written code by pattern-generated code using 431 instances of 23 different encounter patterns and our 1 dialogue pattern.

To ensure that our pattern catalog could be used by non-programmers, we invited a high school English class to use the Aurora Toolset, our pattern catalog and ScriptEase, to write short stories as adventures in NWN. The students succeeded in using our patterns to generate interesting stories (Szafron et al. 2005) that play as NWN adventures.

Besides NWN, we have identified patterns in two other CRPGs, Fable (http://www.fablegame.com) by Lionhead Studios and The Elder Scrolls III: Morrowind (http://www.morrowind.com) by Bethesda Softworks. For example, *Placeable use – toggle door* can be found in Fable where there are four rocks and a door. The player must hit the rocks in the correct order to open the door. Currently, the PC must attack the rocks, but it makes more sense to restructure the puzzle so that the PC is required to touch the rocks rather than hit them. In Morrowind, this pattern is used to open a door when the PC uses a lever. The pattern *Door click – show monologue* can be observed in several areas of Fable that involve the use of riddles. Throughout the game there are several doors called Demon Doors, which require the player to solve a riddle to open them. When the user clicks on the door, the door speaks a monologue giving the user the riddle that must be solved. This pattern is used in Morrowind near the beginning of the game. When the user clicks on a door, the PC is told to look in a nearby barrel for a ring. The pattern *Trigger enter – spawn creature near object* is used in Fable for an ambush. The player at one point in the game is asked to escort a person to a nearby farm. When the person being escorted enters a trigger, an enemy is spawned nearby to attack the person. In Morrowind this pattern is used to spawn a person high above the player that falls to his death, due to the misuse of a jumping potion.

CONCLUSION

In this paper, we have presented a pattern catalog for CRPGs. This catalog contains 60 patterns that can be used by adventure designers to effectively communicate their stories to programmers who must write the scripts to make these adventures come alive. These patterns can also be used to automatically generate scripts for adventure designers working with the NWN system.

ACKNOWLEDGEMENT

This research was supported by grants from the (Canadian) Institute for Robotics and Intelligent Systems (IRIS), the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta's Informatics Circle of Research Excellence (iCORE), BioWare Corp. and Electronic Arts (Canada) Ltd. We thank former ScriptEase team members James Redford (M.Sc.), Dominique Parker (M.Sc.), Stephanie Gillis (High School Teacher) and Sabrina Kratchmer (WISEST summer student) for their efforts on ScriptEase. We especially thank our many friends at BioWare for their feedback, support and encouragement, with special thanks to Mark Brockington.

REFERENCES

Budinsky, F., Finnie, M., Vlissides, J. and Yu, P. 1996. "Automatic code generation from design patterns". *IBM Systems Journal*, 35, 2, 151-171.

Carbonaro, M., Cutumisu, M., McNaughton, M., Onuczko, C., Roy, T., Schaeffer, J., Szafron, D., Gillis, S., Kratchmer, S. 2005. "Interactive Story Writing in the Classroom: Using Computer Games" In *Proceedings of DiGRA 2005 Conference: Changing Views – Worlds in Play*, (Vancouver, Canada, June), 323-338.

Florijn, G., Meijers, M. and van Winsen, P. 1997. "Tool support for object-oriented patterns". In *Proceedings of the 11th European Conference on Object-Oriented Programming*, Vol. 1241 of Lecture Notes in Computer Science, Springer, 472-495.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S. and Tan, K. 2002. "Generative Design Patterns", In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, (Edinburgh, UK, September), 23-34.

McNaughton, M., Cutumisu, M., Szafron, D., Schaeffer, J., Redford, J., and Parker, D. 2004a. ScriptEase: "Generative Design Patterns for Computer Role-Playing Games", In *Proceedings of the 19th Interntational Conference on Automated Software Engineering*, (Linz, Austria, September), 88-99.

McNaughton, M., Redford, J., Schaeffer, J., and Szafron, D. 2003. "Pattern-based AI Scripting using ScriptEase", *In Proceedings of the 16th Canadian Conference of Artificial Intelligence*, (Halifax, Canada, June), 35-49.

McNaughton, M., Schaeffer, J., Szafron, D., Parker, D. and Redford, J. 2004b. "Code Generation for AI Scripting in Computer Role-Playing Games", In *Proceedings of the Challenges in Game AI Workshop at AAAI-04*, (San Jose, USA, July), 129-133.

Szafron, D., Carbonaro, M., Cutumisu, M., Gillis, S., McNaughton, M., Onuczko, C., Roy T. and Schaeffer, J. 2005. "Writing Interactive Stories in the Classroom", *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning (IMEJ)*, Volume 7, Number 1, May.

ON THE DEVELOPMENT OF A FREE RTS GAME ENGINE

Michael Buro and Timothy Furtak
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
email: {mburo,furtak}@cs.ualberta.ca

KEYWORDS
Real-time strategy game, server-client architecture, scripting

ABSTRACT

The genre of real-time strategy (RTS) video games is very popular and poses numerous challenges to AI researchers who want to create systems that play autonomously or aid human players. One obstacle for AI progress in this area is closed commercial software which restricts game access to inflexible graphical user interfaces. In this article we describe the current state of the free RTS game engine ORTS which allows users to define RTS games in form of scripts and to connect arbitrary game client software — ranging from 3d GUIs to distributed AI systems. This flexibility opens up new avenues for RTS game competitions and AI research, of which some are discussed here as well.

BACKGROUND

Real-time strategy (RTS) games such as Starcraft[tm] and Age of Empires[tm] are fast-paced war simulations which have become quite popular in recent years. Constructing AI systems that play these games well is challenging because of incomplete information, real-time aspects, and the requirement of long-range planning. Many commercial RTS games feature AI scripts that can win against novice players by being favored in various ways. Examples range from giving AI components access to normally hidden information (such as opponents' unit locations), over executing actions faster, to increasing the influx of resources. While this approach may result in challenging single-player missions for beginners, it is not applicable in fair competitions. Furthermore, it does not tackle the real AI issues such as reasoning, abstract planning, learning, and opponent-modeling. Machines are still inferior to humans in these areas, which is obvious when watching machines play each other repeatedly.

To improve the performance of RTS game AI we made the case for studying real-time AI problems in the context of RTS games in (Buro 2002; Buro & Furtak 2003; Buro 2004; Buro & Furtak 2004). There we also described the design rationales and components of the free RTS game engine ORTS (Open Real-Time Strategy). Table 1 summarizes the major differences between ORTS and current commercial RTS games.

Feature	Commercial RTS Games	ORTS
Cost	≈US$ 55	US$ 0
License	closed software	free software (GPL)
Game Specification	fixed	user-definable
Network Mode	peer-to-peer	server-client
Prone to Map-Revealing Hacks	yes	no
Communication Protocol	veiled	open
Network Data Rate	low	low to medium
Unit Control	high-level, sequential	low-level, parallel
Game Interface	fixed GUI	user-definable

Table 1: How ORTS relates to commercial RTS games

Commercial RTS games software is closed and not expandable. This prevents researchers and hobbyists from tailoring RTS games to their needs and from connecting remote AI modules in order to gauge their playing strength. ORTS, by contrast, is a free software RTS game *engine* which means that its source code and artwork are available free of charge and users can specify their own RTS games.

Furthermore, commercial RTS games as well as the free RTS game engine (Stratagus 2005) utilize peer-to-peer as opposed to server-client technology to reduce network traffic. In peer-to-peer mode the complete game state is maintained on each player's computer – by means of broadcasting all player actions – and the software just hides the invisible part of the game state from the players. By tampering with the client software it is possible to reveal the entire state and thereby gain an unfair advantage. So-called map-revealing hacks are wide-spread and pose a serious problem for online tournaments. We feel that this is unacceptable for playing fair games on the internet. Therefore, we implemented a server-client architecture in ORTS. The entire game state is maintained in the server which repeatedly sends out individual player views, receives player actions, and executes them. (Buro 2002) claims that the resulting system is "client-hack-free" in the sense that client software changes will not benefit attackers. Of course, a truly fair setup also requires trusted servers and trusted communication.

Another advantage of open server-client game architectures is that users can connect whatever client software they like. This openness leads to new and interesting possibilities ranging from fair on-line tournaments of autonomous

AI players to gauge their playing strength to hybrid systems in which human players use sophisticated GUIs which let them delegate laborious or repetitive tasks to AI helper modules. Examples include smart group pathfinding, computing efficient build orders, and small-group combat tactics.

One downside of the server-client operation compared to peer-to-peer implementations is increased network data rates, especially for the server which uploads views to the clients. In ORTS the data requirements are lowered by sending out compressed incremental view updates (Buro 2002), which is sufficient to play games with 1000 visible moving objects at a data rate of 2.5 KB per game tick.

The ORTS source code is mainly written in C++ with the exception of game specifications and GUI customization for which we developed a simple scripting language. Scripting allows us 1) to change settings without triggering compilation and 2) to use the same executables for different game types. The C++ code uses the following libraries which are available for many systems: SDL, SDL_net, Qt, OpenGL, GLUT, and GLEW. ORTS is being developed under Linux and Cygwin using gcc, but it now also natively builds under Windows and Mac OS X. In addition to the C++ source code, a sample game is provided in the distribution including game specification scripts, a set of 3d models, and user interface customization scripts for the GUI. ORTS software, artwork, and documentation can be downloaded from (ORTS 2005)

In the following sections we give a high-level overview of the major ORTS components with emphasis on the latest developments and scripting. We conclude the paper with a brief discussion of the project's future.

SERVER

The ORTS server is responsible for simulating unit actions and determining what each player is allowed to know about the current state of the world.

Every cycle players can send an action for each unit they control. The server applies these actions in a random order, removing any units that have died. Then the positions of moving objects are updated and colliding objects are stopped. Finally, the region visible to each player is computed and any changed or newly visible tiles are sent along with visible units.

To simplify the description of the world, the terrain is tile-based. Each corner of a tile may be set to an integer height, allowing tiles to be sloped in various ways. Boundaries are automatically generated where two adjacent tiles do not line up or are different types e.g. a land tile next to a water tile. To help make the terrain less blocky we support half-tiles, where a tile is split along the diagonal into two different types and/or the heights on one side of the diagonal do not line up with the heights on the other. The two sides of half-tiles are independent of each other with regard to computing vision; a unit on the lower half may not be able to see a unit on the higher half of the same tile. The default terrain generation produces cliff tiles to ease the transition between different height levels, but this is not required.

Motion

Objects are simple geometric shapes – mobile units are usually circles, buildings are rectangles, and boundaries are line segments. Although object positions are restricted to a fixed grid, collisions for moving objects are computed exactly at a higher resolution, so fast-moving objects won't pass through each other.

Which units can collide with each other is determined by a collision bitmask for each object, set by default to the object's z-category (on land, flying, underwater, etc.). Exceptions to this may be specified in another bitmask, so that special objects can pass through each other without needlessly complicating the default collision rules.

Vision

Visibility is computed in terms of which tiles can be seen from the center of the tile an object is on. If the center of the tile can be seen then that tile is entirely visible and any objects that intersect the tile can be seen. If only a portion of the tile is visible, say a corner or a side, then the type of tile is known but not any units on that tile.

Local visibility for each tile and for the entire map is stored as a bitmap. At the expense of caching the bitmap for each tile after the initial computation, determining visible tiles is quickly done via boolean operations on the bitmaps. A separate visibility computation is performed for cloaked units and the detector units that can see them.

SCRIPTING

The scripting engine performs the interesting game-specific logic and allows for flexible game definitions and client interfaces. High performance tasks common across a large number of possible RTS games such as accurate unit motion and unit vision in the presence of terrain are handled separately by the server. Everything else, such as weapons and special abilities, is scripted as part of the game definition.

The scripting language was designed to provide a convenient way to define unit types and actions. Unit definitions are given in the form of blueprints which list named (usually) integer attributes and actions. The blueprints use a loose multiple inheritance system, allowing them to be combined and nested. New unit types can easily be constructed from functional components. In the client the object creation system is used to create GUI widgets such as buttons and status windows.

When the client receives the game description, which includes unit blueprints, it can locally extend those blueprints by adding extra attributes, sub-objects, or actions. The client can use this functionality to write wrappers for complex actions, add simple background AI, or add event handlers for when an attribute changes. By adding a 3d model sub-object the client specifies how an object will be represented in the world and allows for context sensitive animations.

The client extends the scripting language functionality by registering special functions that allow access to OpenGL commands for drawing bitmaps and then simply calling

```
blueprint marine
  # include a set of common attributes and default values
  is generic_unit

  # create a sub-object of type "kevlar" named "armor"
  class kevlar armor

  # the rifle sub-object has already been defined and
  # has a "shoot" action defined
  class rifle  weapon

  # make zcat constant and assign it the enum ON_LAND
  setf zcat  ON_LAND
  setf max_hp    100
  set  hp        100
  setf sight     6
  setf radius    5
  set  max_speed 3
end
```

Figure 1: Marine blueprint

those functions within the script. Mouse and keyboard events received by the client are transferred to the script by calling the actions of a special root GUI object, passing the event information as parameters. This object recursively calls the interface actions of its children until it is handled.

Since the scripting language was designed to be able to perform reasonably complicated game logic, eventually errors will occur that cannot be simply debugged by inspection. At this point it becomes invaluable to have some way for the script to write information to the console or to inspect the current state. As a compiler option the interpreter can maintain a stack trace of the current execution with a printout of line numbers and the statement being evaluated at each step. This trace is automatically printed when a trappable error occurs in the script, and can be printed manually from inside a debugger such as gdb.

Because it is relatively trivial to extend the scripting language by adding external C functions it is tempting to do so whenever additional functionality is needed. This can quickly lead to numerous special purpose functions and bloated syntax. Consider the problem of implementing an STL-like vector container. One option is to try to force the language to do something it was never intended to, perhaps by implementing a complicated linked list. Another is to add a C function that returns a pointer to an actual STL vector, with additional functions for adding to it, sorting, etc.

To help make the scripting language extensible, objects in the script are all derived from a common base class, with game objects being only one possible option. To address the previous concern, wrappers have been written for STL vectors and sets, allowing them to be created in the same manner as classes described by blueprints. By modifying the new objects' incremental update functions the container can be used as a sub-object within game units. The graphical client uses derived classes for 3d models and particle systems to attach these things to objects in the game.

Script actions take generic script variables as parameters, which may be object pointers, integers, or something else.

```
blueprint missile
  has  core_attr
  has  movement
  setf shape       CIRCLE
  setf radius      3
  setf max_speed   20
  set  speed       0
  setf zcat        IN_AIR
  setf targetable  0
  setf invincible  1

  var hidden det_range    3
  var hidden blast_range  20
  var hidden min_dmg      200
  var hidden max_dmg      350

  # set the collision mask to ignore all other objects
  var collides 0

  # this action takes one object as a parameter, no
  # integer variables, and no hidden variables.

  action track_obj(targ;;) {
    gob e;
    int dmg, damage_type;

    damage_type = this.damage_type;

    if (targ.targetable < 1) break;
    if (distance(this,targ) <= this.det_range) {
      # "-1" -> not owned by any player
      e = create("explosion", -1);
      e.x = this.x;
      e.y = this.y;
      e.zcat = targ.zcat;
      e.radius = this.blast_range;
      e.damage_type = EXPLOSIVE;
      # add the "boom" action to the action queue and
      # execute it sometime in the current tick
      e.boom(;this.min_dmg, this.max_dmg, 0;) in 0;

      # mark the missile as dead - it can still act,
      # but cannot queue any more actions, and will
      # be deleted at the end of the current tick
      kill(this);
    } else {
      # move events are handled after script actions.
      # the object isn't teleported, it walks/flies to
      # the target location at its speed
      move(this; targ.x, targ.y);

      # accelerate the missile - applies to above command
      this.speed += 4;
      if (this.speed > this.max_speed)
        this.speed = this.max_speed;

      # execute this action again in 1 tick
      # without "in 1" action would be called immediately
      this.track_obj(targ;;) in 1;
    }
  }
end
```

Figure 2: Missile blueprint

When evaluating scripts in the client, actions that are part of the original game description are not evaluated locally, but are automatically placed in the outgoing action list to be sent to the server.

The game simulation is tick-based, and a large number of object actions naturally depend on time constraints e.g. weapon cool-down, construction times. To better support time in the scripts the language is able to specify that actions are to occur some number of ticks in the future. These actions are stored in a priority queue until they need to be evaluated. A small amount of bookkeeping is required to ensure that dead objects still referenced by a pending action are not deleted until they are no longer pointed to. A dead object can no longer perform actions, but functions can check if it is still alive.

CLIENT SOFTWARE

Unlike commercial RTS games, ORTS players can connect *whatever* client software they like and can issue commands to *all* of their units in each game tick (usually more than 8 times a second). Consequently, ORTS clients have much more control over game objects which greatly impacts game design. Consider default unit-behavior. In Starcraft[tm] for example, tanks automatically fire on enemy units within range. But very powerful spells like "lock-down" and "psionic storm" have to be cast manually by the player, thus limiting their effectiveness. In ORTS, all units can become so-called auto-casters by letting client AI modules decide when and where an object acts without having to wait for slow-paced player instructions. Thus, the cost of ORTS game objects has to be balanced in light of ubiquitous auto-casting.

The ORTS software currently provides basic client functionality such as communication with the server, maintaining the game state, a GUI, and some low-level AI modules, which are discussed below. The main focus of future client software additions will be on making AI components smarter to allow players to concentrate more on high-level strategic decisions, and eventually let the AI play games autonomously.

Maintaining the Game State

Because the ORTS server sends out incremental and compressed view updates and receives compressed action sequences, it is helpful to encapsulate the game state and communication in classes for everybody to use. Another advantage is that the communication protocol and compression can be changed without breaking client code. Server and client share the same Game class. In clients, this class represents the current game state in view of the player, and provides access to tiles and visible game objects. The Game class is part of GameStateModule, which communicates with the server, updates the state, and informs registered users about server messages by invoking event handlers. Each game object has an action member which can be set either by AI modules or the GUI as a result of user actions. In each game tick, actions for all objects under player control are sent to the server by invoking a function in the GameStateModule class.

Graphical User Interface

For interacting with human players and AI demonstration purposes a graphical user interface is essential. We have implemented a client component (class GfxModule) that uses OpenGL to render arbitrary 3d views of the current game state in a window together with a minimap, an information panel, and action button panel (Fig. 3). Moreover, rectangular overlays can be created to display additional information such as pathfinding results and influence maps. The widget layout, keyboard command shortcuts, and actions attached to buttons are scriptable. The graphics module communicates with the server through GameStateModule.

Low-Level AI Components

The server does not provide any default high-level functionality, so any tasks involving multiple low-level actions must be coordinated by the client. Basic gameplay tasks such as pathfinding, gathering resources, and automated defenses are implemented client-side via pluggable C++ modules. These components communicate with the GUI and with each other via a simple message passing system. When a user sends a unit to a location a pathfinding event is generated. The pathfinding module then plans a route to the target and babysits the unit, sending move commands for each leg of the path. As the world is explored the pathfinding module receives messages notifying it of new obstacles and units, letting it update its map of the world. The resource gathering module, once initiated by the client, works with the pathfinding module. It broadcasts a pathfinding message to send a unit to a given resource, receives confirmation of arrival, and then orders the unit to start mining. Similarly for returning resources to the base once collected.

Figure 3: GUI screenshot

ORTS.NET

A recent addition to ORTS is the ORTS.net internet game service where players can meet and initiate ORTS games by communicating through a generic game server (GGS). ORTS.net is comprised of three programs:

netservice: ORTS.net game manager. Stores player data such as buddy lists and ratings. Also maintains a list of networkers, sets up games, and assigns networkers to host them.

netclient: Graphical (Qt) front-end of the ORTS.net service featuring log-on and chat dialogs and more. Communicates with netservice and players via GGS.

networker: ORTS server controlled by netservice. It hosts ORTS.net games and clients, such as ortsg, connect to it directly.

Figure 4 shows how these programs are connected. Central to ORTS.net is GGS, a message passing server which can be downloaded from `www.cs.ualberta.ca/~mburo`. GGS allows connected parties to exchange messages using a simple text-based protocol. Before an ORTS game can be initiated, netservice and one or more networkers have to be connected to GGS. Networkers register themselves with netservice to indicate that they are available for hosting ORTS games. After players connect to GGS using netclient, they can chat with each other and arrange ORTS games by sending messages to netservice. When netservice creates a game it selects an available networker and sends its IP address to the netclients along with a one-time password. The netclients then launch ortsg which connects to the networker to start the game. Finally, when the game is over, the networker sends the result back to netservice, disconnects the clients, and becomes available for hosting another game.

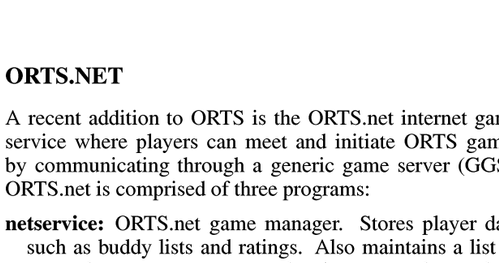


Figure 4: ORTS.net network topology

OUTLOOK

With all major components now functional, the ORTS software has reached the point where it can be used as platform for real-time AI research, the development of new RTS games, and on-line competitions.

ORTS can still be improved in various ways. For instance, the game state currently cannot be saved, GUI customization is incomplete, the graphics performance needs to be improved. Moreover, work on RTS game AI that is executed in ORTS clients has just begun.

Currently our research group is looking at pathfinding, small scale combat, optimizing build orders, and high-level planning based on Monte Carlo simulations. We hope that the availability of a (hack-) free RTS game engine sparks more interest in RTS game AI and competition among researchers, students, and hobbyists. (Molineaux 2005) reports that work already has begun to interface ORTS with (TIELT 2005), a testbed for integrating and evaluating learning techniques in real-time games.

ACKNOWLEDGMENTS

We thank Keith Yerex and Sami Wagia-alla for their numerous contributions to the ORTS project. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

Buro, M., and Furtak, T. 2003. RTS games as test-bed for real-time research. In *Proceedings of the JCIS Workshop on Game AI, extended version at `www.cs.ualberta.ca/~mburo/orts`*, 481–484.

Buro, M., and Furtak, T. 2004. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)*, 63–70.

Buro, M. 2002. ORTS: A hack-free RTS game environment. In *Proceedings of the Third International Conference on Computers and Games*, 156–161.

Buro, M. 2004. Call for AI research in RTS games. In *Proceedings of the AAAI Workshop on AI in Games*, 139–141.

Molineaux, M. 2005. NRL (Washington D.C.) personal communication.

ORTS. 2005. Free Server-Client RTS Game Engine at `http://www.cs.ualberta.ca/~mburo/orts`.

Stratagus. 2005. Free Peer-to-Peer RTS Game Engine at `http://stratagus.sourceforge.net`.

TIELT. 2005. Test-bed for integrating and evaluating machine learning techniques in real-time games at `http://nrlsat.ittid.com`.

MICHAEL BURO is an associate professor for computer science at the University of Alberta. After receiving his Ph.D. in Germany he worked as a scientist at the NEC Research Institute in Princeton for seven years before he moved to Edmonton in 2002. His main research interests are heuristic search and planning in AI and machine learning applied to games. He is the author of LOGISTELLO — a learning Othello program that in 1997 defeated the human World-champion 6–0.

TIMOTHY FURTAK is entering the masters program at the University of Alberta's computing science department. He has spent the last two years developing the ORTS game engine at the University of Alberta and is interested in applying machine learning to games.

STUDENT PAPERS

VALIDATING VITRUAL TRAINING: Study of a Forward Observer Simulator

J.P. McDonough and Mark W. Strom
Modeling, Virtual Environments and Simulation (MOVES) Institute
Naval Postgraduate School
1 University Drive
Monterey, California
E-mail: jpmcdono@nps.edu
E-mail: mwstrom@nps.edu

KEYWORDS
Virtual Environments, Virtual Training, Open Source.

ABSTRACT

Due to declining budgets and decreases in ammunition allowances, the opportunity for military units to conduct live fire artillery training has been greatly reduced. The available artillery simulations are either outdated, require specialize operator support, or are not deployable. Forward Observer PC Simulator (FOPCSim) was developed with open source software, as a game based, call for fire virtual environment that runs on a laptop. The system provides users with real-time performance feedback based on the Marine Corps Training and Readiness standards and was designed according to a cognitive task analysis of the call for fire procedures. Too many times simulators are fielded without investigating how well they train a particular task. To evaluate how well FOPCSim trains the call for fire procedures, an experiment was conducted at The Basic School in Quantico, Virginia. FOPCSim was used in place of the current training simulator: Training Set, Fire Observation (TSFO) to evaluate its training effectiveness.

INTRODUCTION

Before accepting a virtual environment trainer (VET) as a suitable replacement or augmentation to live training, it should be evaluated for its effectiveness. Major Walt Yates, a recent graduate of the MOVES Institute at the Naval Postgraduate School (NPS), looked at this question for a marksmanship trainer that has been in use in the Marine Corps for over ten years without being evaluated. Yates states that, "Despite how commonly VETs are used there are many fielded VETs for which there has been no detailed study conducted to validate the effectiveness of a VET. Such studies are referred to as verification of skills acquisition (or training transfer). A positive verification of skill acquisition requires a quantified measure of improvement at task performance (Fredriksen and White 1989). To justify the expense of developing and fielding VETs they must be verified to accomplish skill acquisition as well as conventional methods of training or a reduced level of effectiveness must be accepted as a trade-off for reduced cost or increased safety."(Yates 2004)

In September 2002, Lieutenant Colonel Dave Brannon and Major Mike Villandre, graduates of the MOVES Institute at the NPS, pursued the above questions concerning the task of calling for fire. They chose to develop a PC based call for fire trainer called Forward Observer PC Simulator (FOPCSim). The goal of their research was "focused on development of a virtual environment in which a trained forward observer could conduct a basic call for fire (CFF) having to execute the same procedures as he would in the real world." (Brannon and Vilandre 2002). For our research we chose to take the idea for the application they started and continue to build on it. We rewrote the software using the open source game engine Delta3D, added a more intuitive interface based on the MCRP 3-11.1A Platoon Commander's Tactical Notebook CFF worksheet, and a tutoring system that provides real-time performance feedback.

To test its effectiveness, we chose to look at students initially learning the forward observer skill of calling for indirect fire. We conducted an experiment at The Basic School (TBS) in Quantico, VA, where newly commissioned lieutenants receive training in the skill set needed to be platoon commanders in the Marine Corps. Our pre-test hypothesis was that students who used FOPCSim under instructor supervision for two hours and then were free to use the simulator on their personal computers to practice "Calling for Fire" would have more individual practical-application time than the group which just received two hours of group instruction using the traditional simulator TSFO. This change would lead to better scores on the supporting arms exam and better performance during the live call for fire exercise.

METHOD

Participants

The participants were 250 predominantly male, Marine Corps Lieutenants assigned to B Company attending the Officer Basic Course, at TBS. The selection criteria for the control group which consisted of 2/3 (166) of the company and the experimental group 1/3 (84) of the company was randomly selected based on which day of the week the participants were scheduled for training. All students were given a copy of FOPCSim: participants in the experimental group were given the simulator after their classroom training but before the final exam, and the control group did not get a copy until they completed their final exam.

Design and Materials

A posttest-only design was used to explore the effectiveness of a freely available, game based trainer with a performance feedback system versus a commercial, classroom-oriented, instructor-driven simulator. The independent variable was the type of simulation used: TSFO or FOPCSim. Because the participants were formally tested on the call for fire procedures, it was important that the control group and experimental group received the same level of training.

The normal artillery training package consists of three phases: 2.5 hours of classroom training, 5 hours of training utilizing simulators, and a live fire artillery shoot. The normal 5 hour block of simulation training includes a 1 hour review of the call for fire procedures followed by 2 hours of TSFO simulator and 2 hours of "lawn darts" pneumatic mortar physical simulation. The experimental group received the exact same training as the control group except that the 2 hour TSFO block was replaced by 2 hours of FOPCSim training and the 1 hour review session was conducted utilizing FOPCSim vice an overhead projector. The 1 hour review session was conducted using FOPCSim to help familiarize the students with the user interface while reviewing the basic procedures. For the 2 hour TSFO block, students are required to work up call for fire missions based on what they see on a large screen in the front of the classroom and one student per mission is called upon to read back the mission. The instructor critiques the call for fire as the participant conducts the mission. The benefit of this method is the whole class hears the mission and instructor's critique of that mission; they therefore have the opportunity to learn from the mistakes of others in the class. Because in a 2 hour session only 5-7 students have the chance to be in the "hot-seat", it is hard to identify individual students that are having problems. For the first hour of the 2 hour FOPCSim block, the instructor picked students to perform missions on the large screen similar to the TSFO block. For the remaining hour each student had a chance to be in the "hot-seat" and work up and input multiple missions into their own virtual environment. The simulator scored each mission and a feedback system critiqued the student's performance and gave individual feedback based on the Marine Corps Training and Readiness Standards. This allowed the participants that were proficient at the call for fire procedures to complete many missions, while those participants that were having problems with the procedure could get individual help without hindering the rest of the class.

RESULTS

The true test of how well someone can call for fire can only be evaluated by measuring their performance on a live fire mission. Unfortunately, based on time and logistical constraints the live fire missions for the students are not graded events. Instead the students each get the opportunity to conduct a live mission as part of a two person team. Until the live fire mission becomes an evaluated event, there will be no true test of the ability of students to call for fire or of the training events that prepare them for that.

Since the live fire test is not evaluated, the next best thing is the written Supporting Arms Exam that all students take. This test to evaluate their supporting arms exam knowledge covers several areas not specifically associated with the Call for Fire procedure such as the controlling of Close Air Support (CAS). It includes both a multiple choice portion that is computer-graded as well as a short answer/ fill in the blank portion that is hand graded by the instructors. The portions of the test that specifically covers the call for fire procedures are the multiple choice section and a portion of the hand-graded section.

Table 1 shows the comparison between the TSFO group (Group A) and the FOPCSim group on the written Supporting Arms Exam. For the statistical results we only included those students for whom we had complete scoring data and survey results. The group who used FOPCSim scored significantly higher ($p < .05$) than the TSFO group. This group was split evenly between those that did not use FOPCSim after the two hour class (Group B: 30) and those that did (Group C: 31). When looking at this data, some would argue that of course the group who got to use FOPCSim for more than two hours would do better because "more is always better." But in this case, we did not see this result.

Table 1: Supporting Arms Exam Results

	Group	N	Mean	Std. Deviation	Std. Error Mean
Overall Score	Group B & C FOPCSim	61	85.3484	10.03880	1.28534
	Group A TSFO	166	82.0959	9.96684	.77358

When we further broke up the FOPCSim group into two groups, we found that those who chose to use the simulation more did not perform statistically better than the other two groups, in fact there overall scores were lower than the group that just used it in class (see Table 2 Overall Scores). One explanation for this is those who felt comfortable with the material felt that two hours was enough, whereas, those individuals who did not, chose to use the simulation on their own. If these, indeed, corresponded to those who were less likely to do well on exam if they stopped after only 2 hours use in class, then we would get the kind of results seen for Overall Score in Table 2. Indeed, using the FOPCSim after class didn't cause low scores, but a fear of low scores caused some to use FOPCSim after class. Group C's scores might indeed be higher than if they'd taken the test after only the 2 hour class, but if they started from a lower base (on average) then they'd appear to do worse.

As part of the classroom training, the FOPCSim group worked individually (or as a team depending on computer availability) on four graded missions that were scored, and the results were stored as a text file on the computer. We took the results of these four missions and produced an Average SimScore for each individual or team. We then compared these scores to their results on the Supporting Arms Exam. We did not expect to get a high correlation

Table 2: Overall Score by Group

Subgroup		Overall
A	Mean	82.0959
	N	166
	Std. Dev.	9.96684
B	Mean	86.9111
	N	30
	Std. Dev.	9.69603
C	Mean	83.8360
	N	31
	Std. Dev.	10.28932
Total	Mean	82.9699
	N	227
	Std. Dev.	10.06820

between these Simscores (r^2 = 0.245) and their Supporting Arms Exam score since there were several weeks between this training session and the actual exam. However what we did see was how these SimScores could be used a predictor for those who would not perform as well on the Supporting Arms Exam. If we look at those who scored above an 85% on their SimScore Average, we see they are very likely to pass the Supporting Arms Exam (35 of 35). Whereas, if we look at those who scored below an 85%, they have nearly a 20% (4 of 19) chance of failing. If an instructor is given this knowledge prior to the exam he can focus his efforts on those identified by their SimScore Average to prevent them from failing exam.

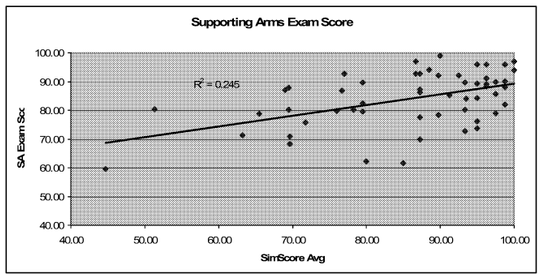


Figure 1: SimScore Average vs. Overall Score

DISCUSSION

The results of our experiment show that FOPCSim performs at least as well if not better than the existing system to train entry level students to call for and adjust indirect fire based on the results of the written Supporting Arms exam. Unfortunately, until the live fire CFF is evaluated for students at TBS, there can be no true test of how well simulation helps users learn this specific task.

The largest difference between the group that used the TSFO and the group who used FOPCSim is the number of missions that each got to perform in the virtual environment. For the TSFO group, only 5-7 of the group of twenty performed a mission in front of the group with instructor critique. In the FOPCSim group each student got the opportunity to perform several missions at their own pace during the two hour period. Since their missions were evaluated by the built in scoring system, they received feedback on each mission. In addition, if they did not understand their errors, the instructor was now free to give individuals help without interrupting the whole class.

CONCLUSION

FOPCSim was not designed to be an all-in-one simulator to train call for fire. There are several high-dollar simulators in use throughout DOD that are more technologically advanced, that cost much more to acquire and operate, and are not deployable. These systems can be excellent assets to the schoolhouses and units that can afford to acquire and maintain them. Our goal is to complement these systems with a low-cost, freely distributable, deployable system that can be used by observers to practice the call for fire when the real training is not possible. Based on the experiment we conducted, we can say that FOPCSim performs at least as well if not better than the current system used at TBS based on the testing metric used at the school.

REFERENCES

Fredriksen, J. R., White, B.Y. (1989). An approach to training based on principle task decomposition." *Acta Psychologica.* 71. 89-146.

Brannon, D. & Villandre, M. (2002). *The Forward Observer Personal Computer Simulator (FOPCSIM).* Unpublished masters thesis, Naval Postgraduate School, Monterey, CA.

Yates, W. (2004). *A Training Transfer Study of the Indoor Simulated Marksmanship Trainer.* Unpublished masters thesis, Naval Postgraduate School, Monterey, CA.

BIOGRAPHY

JAMES P. MCDONOUGH graduated from the U.S. Naval Academy in 1994 with a B.S. in Systems Engineering. He is currently a Major in the United States Marine Corps with the military specialty of Field Artillery. He is currently pursuing his master's degree in Modeling, Virtual Environemnts, and Simulation at the Naval Postgraduate School, Monterey, CA.
Email: <jpmcdono@nps.edu>

MARK STROM graduated from the U.S. Naval Academy in 1999 with a B.S. in Computer Science. He is currently a Captain in the United States Marine Corps with the military specialty of Aviation Supply. He is currently pursuing his master's degree in Computer Science at the Naval Postgraduate School, Monterey, CA.
Email: <mwstrom@nps.edu>

DARK WATERS: SPOTLIGHT ON IMMERSION

Dominic Arsenault
Département d'histoire de l'art et d'études cinématographiques
Université de Montréal
P.O. Box 6128, succ. Centre-ville, Montréal (QC) H3C 3J7
Canada
E-mail: dominic.arsenault@umontreal.ca

KEYWORDS
Immersion, Engagement, Design, Storytelling, Presence.

ABSTRACT

This paper combines several empirical studies and some theoretical research to shed some light on the dark, undefined waters in which we plunge when we are "immersed". Immersion, across all media, comes in three different types and in three different degrees, and can be hindered by barriers, such as inaccessibility, or favored by fuel, such as using one's imagination. The resulting model of immersion can be applied to experiences formed by any type of media object, but is particularly relevant to video games.

INTRODUCTION

Immersion is one of those words we keep hearing when we talk about video games. A problematic aspect of the term, however, is that it can take on a variety of senses depending on the author, text, and context. Consider, for example, the following usages of the term:
- "The sound and lighting effects actually made me feel like I was part of the scene."
- "I am totally immersed in the story, I can't wait to see what is going to happen next."
- "I have been playing for so long that I don't even see the game pieces anymore, only the patterns of play."

Those three examples have something in common that we can call immersion, using Janet Murray's general definition of "the sensation of being surrounded by a completely other reality […]that takes over all of our attention" [8]. Yet the three examples cited above are all very unique experiences that rely on different mechanisms. I will therefore build upon the work done so far by other scholars and propose a refinement and combination of multiple theories on immersion in order to better understand this "excessively vague, all-inclusive concept"[7].

THREE TYPES OF IMMERSION

Laura Ermi and Frans Mäyrä have built a gameplay experience model which they call the SCI-model[6]. This model establishes three types of immersion: Sensory, Challenge-based, and Imaginative. Their model seems adequate enough to describe the experience of a player going through a video game-playing session, so I will use it as the main framework for studying the phenomenon.
Sensory immersion, as its name implies, attempts to focus the senses: "Large screens close to [the] player's face and powerful sounds easily overpower the sensory information coming from the real world, and the player becomes entirely focused on the game world and its stimuli. Challenge-based immersion occurs "when one is able to achieve a satisfying balance of challenges and abilities." Finally, Imaginative immersion is described as occurring when "one becomes absorbed with the stories and the world, or begins to feel for or identify with a game character."

> "For example, multi-sensory virtual reality environments, [..] or just a simple screensaver, could provide the purest form of sensory immersion, while the experience of imaginative immersion would be most prominent when one becomes absorbed into a good novel. Movies would combine both of these. But challenge-based immersion has an essential role in digital games since the gameplay requires active participation: players are constantly faced with both mental and physical challenges that keep them playing." [6]

Fictional Immersion

The first of two amendments I would like to suggest is to name Imaginative immersion "Fictional Immersion" instead. The reason for this change is that we can be immersed in a story without exercising our imagination. Cognitive psychology and the reader-response school in film studies and literary theory have shown that the consumption of a media object is never completely passive; in fact, readers and spectators are constantly mapping mental schemas and building sense from what is presented to them, forming hypotheses on the outcome of the plot, attributing motives and backstories to characters, piecing together the physical setting of the action, and likewise exercising their "active creation of belief"[8] in order to enjoy immersion. [1, 2, 3]

However, to say that this is making usage of one's imagination is to render the concept of Imaginative immersion "excessively vague and all-inclusive", in McMahan's words, since we are constantly evaluating things and situations according to mental schemas. By not taking the criterion of fictionality into account, the concept of immersion suddenly becomes so broad that it loses relevance. To avoid this sort of theoretical dead-end, we need to distinguish between "using one's imagination" and "immersion". We can see the act of using our imagination as a measure taken among others in order to accomplish Fictional immersion. Besides, Ermi and Mäyrä's definition of Imaginative immersion ("one becomes absorbed with the stories and the world, or begins to feel for or identify with a game character.") implicitly relies on the concept of

fictionality. The best way to describe Fictional immersion is to take the illusionist conception of realism that Marie-Laure Ryan presents: it strives to make us feel that "there is more to this [the fictional, represented] world than what the text displays of it: a backside to objects, a mind to characters, and time and space extending beyond the display." [10, p.158]. The term "Fictional immersion" is narrow enough to prevent the pits of Imaginative immersion, yet broad enough to include all forms of storytelling, like narration and representation, found in video games.

Systemic Immersion

The second modification I propose to make to the SCI-model concerns Challenge-based immersion. The argument for this type of immersion is that video games require active participation, and henceforth, are challenging. There are, however, many ways to experience a form of challenge in traditional, non-participatory media. The viewer of a whodunit TV show, for instance, constantly forms hypotheses and tries to interpret the clues so as to find the culprit before the show gives it away. The learned cinephile who watches a movie and notices the intricacies of lighting, camera angles, and similar details of construction, is in a state that is very similar to the chess master that sees the patterns of pieces on the chess board. But where does it tie in with the concept of immersion?

Taking Murray's metaphorical definition and extracting its fundamental idea, I believe we can define immersion as a phenomenon that occurs when a layer of mediated data is pasted upon the layer of unmediated data with such vividness and extensiveness that it blocks the perception of the latter. Immersion occurs when one gazes at a painting, listens to music, is lost in a book or absorbed in a game of chess, so much that he ceases to perceive the museum or the sounds of the street, forgets the events happening in the real world, and suspends his knowledge of its laws.

Systemic immersion occurs when one accepts that a system (of rules, laws, etc.) governing a mediated object replaces the system governing a similar facet of unmediated reality. To think about the player's avatar's chances of survival in a typical RPG in terms of Hit Points, Attack values and such rather than torso and arm size, weight of the weapon, etc., is to adopt the game's system and reject the laws of real-world physics (unless, of course, the game system does take into account the arm size and weight of the weapon rather than Hit Points and Attack values, in which case the reasoning is reversed). Similarly, the learned cinephile that examines the shots of a movie is attempting to schematize and decipher how the mind of the director works. Learning a language is a similar effort, as the expression "linguistic immersion" asserts. As a non-native English speaker, for instance, I need to immerse myself in the proper "English" mindset before writing this article; however, once I am thinking in English, I do not find it hard to write. Since one can be immersed in a system without necessarily being challenged by it, the term "Systemic immersion" seems more adapted to design this experience.

So far I have suggested a classification of different types of immersion, which would transform the SCI-model to a SSF-theory. Another issue has been raised by Elena Gorfinkel in a conversation on immersion: "Immersion is not a property of a game or media text but is an effect that a text produces."[11] It is crucial to remember that for a media object to qualify as immersive, it does not have to be so at all times and for everyone experiencing it. Indeed, as Ryan notes in the form of the water metaphor, most objects alternate between immersive and reflexive stances throughout their course: "The ocean is an environment in which we cannot breathe; to survive immersion, we must take oxygen from the surface, stay in touch with reality." [10, p.97] These opinions join the body of work done by multiple scholars who argue that immersion is also a matter of degrees, and a matter of individual experiences. I will now integrate this notion into the SSF-theory of immersion.

THREE DEGREES OF IMMERSION

Emily Brown and Paul Cairns' study of immersion using Grounded Theory[4] provides us with three degrees of immersion: engagement, engrossment, and total immersion. Each level can only be reached if certain barriers are removed:

Engagement necessitates investment from the player (in time, effort, and concentration) and accessibility (the game is not of a type that the player avoids like the plague, features responsive controls, etc.). It makes the player want to keep playing.

Engrossment follows engagement, provided the game does not suffer from bad construction (visuals, interesting tasks, and plot are given as factors of construction). Once they reach this stage, players become emotionally invested: "The game becomes the most important part of the gamers' attention and their emotions are directly affected by the game."

Total immersion is, according to Brown and Cairns, a synonym of "presence", and occurs when the player can empathise with the game characters and feel the atmosphere of the game. For an adequate atmosphere to exist, "The game features must be relevant to the actions and location of the game characters." When players enter this stage, they are cut off from reality and the game becomes the only thing that affects them.

While good, this classification suffers from a confusion among the types of immersion I have presented above. One barrier given that prevents total immersion is an impossibility for the player to identify with the game characters. It is, however, entirely possible to experience total sensory or systemic immersion while playing *Doom*, or even Atari's *Battlezone*, two games notorious for their absence of plot and characters. Hence, total immersion is not exclusive to story-oriented games. (see Ermi and Mäyrä's measure of all three different types of immersion in a single game session. [6]) The barriers to the last degree of immersion need to be reconceptualized so as to apply to each of the three types of immersion. This is, however, an

enterprise that far exceeds the scope of this paper, and is best left for future work; furthermore, the barriers system is a negatory tool: it tells us what can prevent immersion, but not how to achieve it. I would like to pursue this exercise of integration and study the elements that actively contribute to immersion.

"GIMME FUEL, GIMME FIRE, GIMME THAT WHICH I DESIRE" – Metallica, *Fuel*

Essentially, one can always reach the deepest level of immersion as long as no barriers stand in the way. The process is, however, much easier and quicker when fuel is on hand. Fuel is any activity, or the positive qualities, both in the player and the game, that contributes to make the player advance through the degrees of immersion. These can be either specific to a type of immersion or general. For instance, Murray's "active creation of belief" is fuel for fictional immersion, and so is "using one's imagination". "General fuel", on the other hand, favors multiple types of immersion.

Information Load, Expectation, and Coherence

David Nunez studied the question of whether or not the data provided by an object needs to be sensory in order to contribute to immersion. [9] He found that immersion is hindered or favored by two things: expectation and information load. The former's link to immersion is that "realism" is a recurring term among many scholars who seek to understand how immersion can take place, and Nunez, citing cognitive psychology studies, argues that expectation is a better term to use: "we will perceive of something as realistic if it is in line with our expectations of what one will find in that particular setting." As for information load, "Whether a virtual environment is capable of matching the user's expectations seems to be a function of the amount of information presented to the user." The framerate or amount of visual detail in *Doom 3*, the amount of diegetic information such as books and dialogs in *Morrowind*, and the high number of statistics, character classes and different possible strategies in *Final Fantasy Tactics*, all are specific fuel for the three types of immersion.

The information load, however, needs to be handled or assimilated correctly by the user, whose capacity to do so depends on his level of mastery of the channels through which the information is transmitted. Just as the casual movie-goer will probably lose interest if he watches a 3-hour long characterless and plotless film on the aesthetics of Deleuze versus Metz, the average 70-year old female would likely not be able to digest *Halo 2*. Information load is fuel as long as the user has a tank large enough to hold it.

A curious aspect of immersion has also been found by Kevin Cheng and Paul Cairns[5]. Their study was based on the idea that "One particular barrier to immersion was thought to be caused when the different aspects of the game did not cohere across different modalities.", a result of Brown and Cairns' 2004 study cited above. They examined the experience of a group of players that were playing a game programmed by them to have the laws of physics of its world change and become incoherent at some point. The surprising result is that "immersion overcame the deleterious usability elements. Due to immersion, participants completely failed to notice what had been determined to be modal incoherence – a mismatch between graphical and behavioural realism compared to what the participant expected." This suggests that as players become more immersed, their spectrum of expectations also broadens, which means that immersion could be viewed as a feedback loop: the more immersed one is, the easier it is to become even more immersed.

CONCLUSION

This model of immersion, which started from the SCI-model and evolved to integrate three types of immersion, three degrees, and factors that positively and negatively influence it, may not seem to accomplish much, but it is a solid foundation. Future works on it can take on a variety of forms, such as redefining the barriers to total immersion. I am personally interested in expanding the types of immersion to include subdivisions. Marie-Laure Ryan's temporal, spatial and emotional immersions seem like good candidates for subcategories of Fictional immersion. In the meantime, this paper has shed some light on the dark waters of immersion.

REFERENCES

[1] Bordwell, D. 1985. *Narration in the Fiction Film*, University of Wisconsin Press, Madison, WI.

[2] Bordwell, D. 1989. "Case for Cognitivism". *Cinema and Cognitive Psychology*, No.9 (Spring), 11-40.

[3] Branigan, E. 1992. *Narrative Comprehension and Film*, Routledge, London & New York.

[4] Brown, E. and P. Cairns. 2004. "A Grounded Investigation of Game Immersion." *CHI 2004 Proceedings*, ACM Press, 1297-1300.

[5] Cheng, K. and P. Cairns. 2005. "Behaviour, Realism and Immersion in Games." *CHI 2005 Proceedings*, ACM Press, 1272-1275.

[6] Ermi, L. and F. Mäyrä. 2005. "Fundamental Components of the Gameplay Experience: Analysing Immersion." *Proceedings of DiGRA 2005 Conference: Changing Views – Worlds in Play*. Available online at http://www.gamesconference.org/digra2005/viewabstract.php?id=267

[7] McMahan, A. 2003. "Immersion, Engagement, and Presence". In *The Video Game Theory Reader*, M. J.P. Wolf and B. Perron (Eds.). Routledge, London & New York, 67-86.

[8] Murray, J. 1997. "Immersion". In *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*, The MIT Press, Cambridge, MA.

[9] Nunez, D. 2004. "How is presence in non-immersive, non-realistic environments possible?"**.** *Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, Stellenbosch, South Africa

[10]Ryan, M.-L. 2001. ***Nar****rative as Virtual Reality: Immersion and Interactivity in Literature and Electronic Media.* The Johns Hopkins University Press, Baltimore, MD.

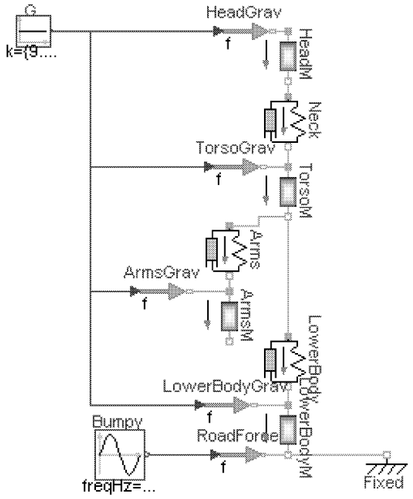
[11]Conversation cited in Salen, K. and E. Zimmerman. 2004. *Rules of Play: Game Design Fundamentals*. The MIT Press, Cambridge, MA, 452.

LATE PAPER

Modelica for the Generation of Physically Realistic Game Code

Hans Vangheluwe
Weigao Xu
Jörg Kienzle
School of Computer Science, McGill University
Montréal, Canada, H3A 2A7
email: {hv, wxu15}@cs.mcgill.ca, Joerg.Kienzle@mcgill.ca

The demand for physically realistic computer games increases as current hardware allows for real-time execution. Currently, writing physically correct and efficient code is not straightforward. We demonstrate how the object-oriented modelling language Modelica (www.modelica.org) can be used for component-based modelling of complex physical systems. The aggresive use of computer algebra (in our μModelica compiler for example) guarantees efficient, real-time code. The simple example we use concers the cervical syndrome which some people suffer of. Their neck is not sufficiently stiff to connect their heads solidly with their upper torso. Therefore, if their upper torso is exposed to vibrations, as when riding in a car, these people may experience severe headaches. The figure below shows a Modelica model of a sitting human body.



To model the mechanical domain, the Modelica Standard Library (MSL) defines the following basic types.

```
type Length = Real(final quantity="Length",
                   final unit="m");
type Position = Length;
```

These are used to describe the dynamics of a rigid body. Note how a Flange interface (not shown) groups both force and position.

```
partial model Rigid
 "Rigid connection of two translational 1D flanges"
  Flange_a flange_a "(left) driving flange";
  Flange_b flange_b "(right) driven flange";
  SIunits.Position s
   "absolute position of flange of component";
  parameter SIunits.Length L=1 "length of component";
equation
  flange_a.s = s - L/2;
  flange_b.s = s + L/2;
end Rigid;
```

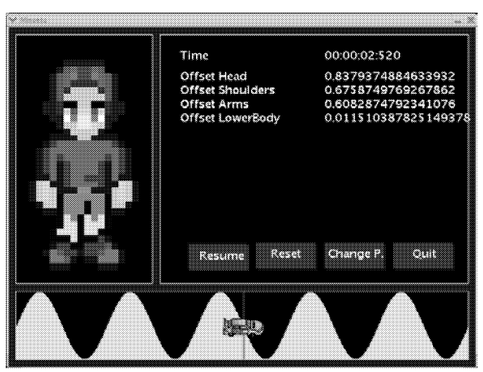
The generic rigid body re-uses the rigid super-class to form a sliding mass. Note that the SlidingMass equations are declarative (computationally non-causal) ! Causality assignment is performed by the Modelica compiler followed by numerical code generation.

```
model SlidingMass "Sliding mass with inertia"
  extends Interfaces.Rigid;
  parameter SI.Mass m=1 "mass of the sliding mass";
  SI.Velocity v "absolute velocity of component";
  SI.Acceleration a "absolute acceleration";
equation
  v = der(s);
  a = der(v);
  m*a = flange_a.f + flange_b.f;
end SlidingMass;
```

The diagram shown above is translated into the following Modelica model by a visual modelling tool.

```
model CervicalSyndrome
  Modelica.Mechanics.Translational.SpringDamper
   Neck(s_rel0=0.1, d=0.8, c=0.3);
  Modelica.Mechanics.Translational.SlidingMass
   HeadM(m=1.2, L=0.2, s(start=2.4));
  ...
equation
  connect(HeadGrav.flange_b, HeadM.flange_a);
  connect(HeadM.flange_b, Neck.flange_a);
  ...
end CervicalSyndrome;
```

We modified the backend of our μModelica compiler to generate Java code to be plugged into the Minueta framework developed by Alexandre Denault. The resulting application is shown below.



AUTHOR LISTING

AUTHOR LISTING