Available online at www.sciencedirect.com
SCIENCE DIRECT®
Robotics and Autonomous Systems 47 (2004) 177–185
Robotics and Autonomous Systems
www.elsevier.com/locate/robot

Teaching Bayesian behaviours to video game characters

Ronan Le Hy*, Anthony Arrigoni, Pierre Bessière, Olivier Lebeltel

GRAVIR/IMAG, INRIA Rhône-Alpes, ZIRST, 38330 Montbonnot, France

Abstract

This article explores an application of Bayesian programming to behaviours for synthetic video games characters. We address the problem of real-time reactive selection of elementary behaviours for an agent playing a first person shooter game. We show how Bayesian programming can lead to condensed and easier formalisation of finite state machine-like behaviour selection, and lend itself to learning by imitation, in a fully transparent way for the player.

Keywords: Bayesian programming; Video games characters; Finite state machine; Learning by imitation

1. Introduction

Today's video games feature synthetic characters involved in complex interactions with human players. A synthetic character may have one of many different roles: tactical enemy, partner for the human, strategic opponent, simple unit amongst many, commenter, etc. In all of these cases, the game developer's ultimate objective is for the synthetic character to act like a human player.

We are interested in a particular type of synthetic character, which we call a *bot* in the rest of this article. It is a player for a first person shooter game named *Unreal Tournament* augmented with the Gamebots control framework [1] (see Fig. 1). This framework provides a tridimensional environment in which players have to fight each other, taking advantage of resources such as weapons and health bonuses available in the arena. We believe that this kind of computer game provides a challenging ground for the development of human-level AI.

After listing our practical objectives, we will present our Bayesian model. We will show how we use it to specify by hand a behaviour, and how we use it to learn a behaviour. We will tackle learning by example using a high-level interface, and then the natural controls of the game. We will show that it is possible to map the player's actions onto bot states, and use this reconstruction to learn our model. Finally, we will come back to our objectives as a conclusion.

1.1. Objectives

Our core objective is to propose an efficient way to specify a behaviour for our bot. This can be broken down into several criteria that hold either for the developers or for the player.

1.1.1. Development team's viewpoint

Programming efficiency. One crucial concern for the programmer is productivity: he needs both expressivity and simplicity of the behaviour programming system.

Limited computation requirements. The processing time allotted to AI in games is typically between 10

* Corresponding author.
E-mail address: ronan.lehy@inrialpes.fr (R. Le Hy).

0921-8890/$ – see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.robot.2004.03.012

Fig. 1. Unreal Tournament and the Gamebots environment.

and 20% of the total processing time [2]; therefore it is important for the behaviour system to be light in terms of computation time.

Design/development separation. The industrial development scheme often draws a separation between game designers and software developers. The system should allow the designers to describe behaviours at a high conceptual level, without any knowledge of the engine's internals.

Behaviour tunability. The ability to program a variety of different behaviours, and to adjust each of them without having to modify the system's back end is essential to the designer.

1.1.2. Player's viewpoint

'*Humanness*'. As defined by Laird [3], this implies the illusion of spatial reasoning, memory, common sense reasoning, using goals, tactics, planning, communication and coordination, adaptation, unpredictability, etc. One important criterion for the player is that the synthetic character does not cheat; its perceptions and actions should be as much as possible like a human player's.

Behaviour learning. This feature is gradually finding its place in modern games: the player can adjust its synthetic partners' behaviour. The behaviour system should therefore support learning.

The game industry mostly addresses these goals with specialised scripting systems, powerful but leading to behaviours hard to extend, maintain and learn [4]. More integrated systems are envisioned in the form of specialised inference engines or expert systems [5], but their actual use in the industry remains limited, as they seem hard to control or because of high computational costs. Flocking [6] is a popular way to yield an impression of complexity while being based on simple elementary behaviours; however it can hardly be generalised to any kind of behaviour.

Neural networks have also found their way to mainstream video games [7], and provide a promising alternative to scripted systems, well suited to learning, although the tuning of produced behaviours can be challenging. Decision trees have also been successfully used [8] as a way to implement fully supervised and reinforced learning.

Nevertheless, finite state machines remain, in various incarnations [9,10] the most common formalisation for reactive behaviours – they are easily mastered, and combined with other techniques such as planning; however they suffer from combinatorial explosion, and remain hard to learn.

1.2. Technical framework

As mentioned earlier, we used the Gamebots framework to conduct our experiments. This implies that our bot communicates with Unreal Tournament via a text protocol on a Unix socket. It receives messages covering its perceptions: its position and speed, health level, ammunition, visible opponents and objects, etc. In return, it sends actions: move to a given point, rotate, change weapon, etc.

The environment is perceived by the bot as a graph, of which nodes are characteristic points of the topology and various objects. The bot perceives only what is in its field of vision.

As our objectives and framework have been exposed, we shall now proceed to explicit our model of behaviour selection, and discuss its interest for the specification and learning of behaviours.

2. Bayesian model

Before examining our particular bot model, we review in the next section the principles of Bayesian programming [11].

2.1. Bayesian programming

Rational reasoning with incomplete and uncertain information is quite a challenge. Bayesian program-

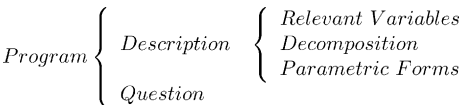

Fig. 2. Structure of a Bayesian program.

ming addresses this challenge, and relies upon a well established formal theory: the probability theory [12]. As a modelling tool, it encompasses the framework of Bayesian networks [13].

In our framework, a Bayesian program is made of two parts: a *description* and a *question* (Fig. 2).

The description can be viewed as a knowledge base containing a priori information available about the problem at hand. It is essentially a joint probability distribution. The description is made up of three components: (1) a set of *relevant variables* on which the joint distribution is defined. Typically, variables are motor, sensory or internal. (2) A *decomposition* of the joint distribution as a product of simpler terms. It is obtained by applying Bayes theorem and taking advantage of the conditional independencies that may exist between variables. (3) The *parametric forms* assigned to each of the terms appearing in the decomposition (they are required to compute the joint distribution). These are called *parametric* because they can include parameters that may change (i.e. be learned or adjusted) during the life of the model.

Given a distribution, it is possible to ask *questions*. Questions are obtained first by partitioning the set of variables into three sets: (1) *Searched*: the searched variables, (2) *Known*: the known variables, and (3) *Free*: the free variables (variables neither searched nor known for the particular question, but participating in the model). A question is then defined as the distribution:

$$P(\text{Searched}|\text{Known}). \tag{1}$$

Given the description, it is always possible to answer a question, i.e. to compute the probability distribution $P(\text{Searched}|\text{Known})$. To do so, the following general inference rule is used:

$$\begin{aligned} &P(\text{Searched}|\text{Known}) \\ &\quad = \frac{\sum_{\text{Free}} P(\text{Searched Free Known})}{P(\text{Known})} \\ &\quad = \frac{1}{z} \times \sum_{\text{Free}} P(\text{Searched Free Known}), \end{aligned} \tag{2}$$

where Z is a normalisation term.

As such, the inference is computationally expensive (Bayesian inference in general has been shown to be NP-Hard). A symbolic simplification phase can reduce drastically the number of sums necessary to compute a given distribution. However the decomposition of the preliminary knowledge, which expresses the conditional independencies of variables, still plays a crucial role in keeping the computation tractable.

2.2. Modelling our bot

2.2.1. Bayesian program

Our particular bot behaviour uses the following Bayesian program.

2.2.1.1. Relevant variables.

S_t	the bot's state at time t. One of *Attack*, *SearchWeapon*, *SearchHealth*, *Explore*, *Flee*, *DetectDanger*. These states correspond to elementary behaviours, in our example programmed in a classic procedural fashion.
S_{t+1}	the bot's state at time $t+1$.
H	the bot's health level at t.
W	the bot's weapon at t.
OW	the opponent's weapon at t.
HN	indicates whether a noise has been heard recently at t.
NE	the number of close enemies at t.
PW	indicates whether a weapon is close at t.
PH	indicates whether a health pack is close at t.

The elementary motor commands of the bot are the values of variables S_{t+1} and S_t. They include an attack behaviour, in which the bot shoots at an opponent while maintaining a constant distance to him and strafing; a fleeing behaviour, which consists in trying to escape (locally) an opponent; behaviours to fetch a weapon or a health bonus the bot noticed in its environment; a behaviour to detect possible opponents outside the current field of view of the bot; and a behaviour to navigate around the environment and discover unexplored parts of it.

2.2.1.2. Decomposition.

The joint distribution is decomposed as

$$\begin{aligned} &P(S_t\, S_{t+1}\, H\, W\, \text{OW HN NE PW PH}) \\ &\quad = P(S_t)\, P(S_{t+1}|S_t)\, P(H|S_{t+1})\, P(W|S_{t+1}) \end{aligned}$$

$$P(\mathrm{OW}|S_{t+1})\,P(\mathrm{HN}|S_{t+1})\,P(\mathrm{NE}|S_{t+1})$$
$$P(\mathrm{PW}|S_{t+1})\,P(\mathrm{PH}|S_{t+1}).$$

To write the above, we make the hypothesis that knowing S_{t+1}, any sensory variable is independent to each other sensory variable (which makes our model a Hidden Markov Model where observations are independent knowing the state). Although it may seem to reduce the expressivity of our model, it allows to specify it in a very condensed way; this point will be emphasised upon in Section 2.2.2.

2.2.1.3. Parametric forms.

- $P(S_t)$: uniform;
- $P(S_{t+1}|S_t)$: table (this table will be defined in Section 2.2.2);
- $P(Sensor|S_{t+1})$ with Sensor each of the sensory variables: tables.

2.2.1.4. Identification. The tables we use are probability distribution tables describing Laplace laws, whose parameters can be adjusted by hand, or using experimental data. We describe these two processes in Sections 3 (*Specifying a Behaviour*) and 4 (*Learning a Behaviour*).

2.2.1.5. Question. Every time our bot has to take a decision, the question we ask to our model is[1]

$$\begin{aligned}
&P(S_{t+1}|S_t\ H\ W\ \mathrm{OW\ HN\ NE\ PW\ PH})\\
&\quad= \frac{P(S_{t+1}\ S_t\ H\ W\ \mathrm{OW\ HN\ NE\ PW\ PH})}{P(S_t\ H\ W\ \mathrm{OW\ HN\ NE\ PW\ PH})}\\
&\quad= \frac{P(S_t)\,P(S_{t+1}|S_t)\prod_i P(Sv_i|S_{t+1})}{\sum_{S_{t+1}}\left(P(S_t)\,P(S_{t+1}|S_t)\prod_i P(Sv_i|S_{t+1})\right)}\\
&\quad= \frac{P(S_{t+1}|S_t)\prod_i P(Sv_i|S_{t+1})}{\sum_{S_{t+1}}\left(P(S_{t+1}|S_t)\prod_i P(Sv_i|S_{t+1})\right)}.
\end{aligned}$$

Knowing the current state and the values of the sensors, we want to know the new state the bot should switch into. This question leads a probability distribution, on which we draw a value to decide the actual new state. This state translates directly into an elementary behaviour which is applied to the bot.

[1] In this equation, $(Sv_i)i$ denotes variables such as H, W, etc.

2.2.2. Inverse programming

We shall now emphasise the peculiarities of our method to specify behaviours, compared to one using simple finite state machines (FSMs). The problem we address is, knowing the current state and the sensors' values, to determine the next state: this is actually naturally accomplished using an FSM.

Let us consider the case where each of our n sensory variables has m_i $(1 \leq i \leq n)$ possible values.

In an FSM modelling a behaviour [14,15], we would have to specify, for each state, a transition to each state, in the form of a logical condition on the sensory variables.

It means that the programmer has to discriminate amongst the $\prod_i m_i$ possible sensory combinations to describe the state transitions. Not only does this pose the difficult problem of determining the appropriate transitions, but it raises the question of convenient formalised representation. This approach could actually lead to several implementations, but will possibly [10] result in a script resembling the following:

if $S_t = A$ **and** $W = None$ **and** $OW = None$ **then**
  if $HN = False$ **and** $NE! = None$
  or $NE = TwoOrMore$ then
    $S_{t+1} \leftarrow F$
else if $HN = True$ **or** $NE = One$
  and $PW = True$ **then**
    $S_{t+1} \leftarrow A$
else...

This kind of script is hard to write and hard to maintain.

In contrast, our approach consists in giving, for each sensory variable, for each possible state, a distribution (i.e. m_i numbers summing to 1). In practice, we write tables like Table 1, which represents $P(H|S_{t+1})$. Values of H are enumerated in the first column, those of S_{t+1} in the first line; cells marked x are computed so that each column sums to 1.

Table 1
$P(H|S_{t+1})$

$H\|S_{t+1}$	A	SW	SH	Ex	F	DD
Low	0.001	0.1	x_3	0.1	0.7	0.1
Medium	0.1	x_2	0.01	x_4	0.2	x_5
High	x_1	x_2	0.001	x_4	0.1	x_5

Moreover, instead of specifying the conditions that make the bot switch from one state to another, we specify the (probability distribution of the) sensors' values when the bot goes into a given state. This way of specifying a sensor under the hypothesis that we know the state is what makes us call our method 'inverse programming'.

Although somewhat confusing at first, this is the core advantage of our way to specify a behaviour. As a matter of fact, we have to describe separately the influence of each sensor on the bot's state, thereby reducing drastically the quantity of needed information. Furthermore, it becomes very easy to incorporate a new sensory variable into our model: it just requires to write an additional table, without modifying the existing ones.

Finally, the number of values we need in order to specify a behaviour is $S^2 + snm$, where s is the number of states, n the number of sensory variables, and m the average number of possible values for the sensory variables. It is therefore linear in the number of variables (assuming m constant).

3. Specifying a behaviour

3.1. Basic specification

A behaviour can be specified by writing the tables corresponding to $P(S_{t+1}|S_t)$ and $P(\text{Sensor}|S_{t+1})$ (for each sensory variable). Let us consider for instance Table 1, which gives the probability distribution for H knowing S_{t+1}. We read the first column this way: given the fact that the bot is going to be in state *Attack*, we know that it has a very low probability (0.001) to have a low health level, a medium probability (0.1) to have a medium health level, and a strong chance ($x = 1 - 0.001 - 0.1$) to have a high health level.

This form of specification allows us to formalise conveniently the constraints we want to impose on the behaviour, in a condensed format, and separately on each sensory variable. For instance, Table 1 formalises the relation of the bot's health level to its state: if it starts attacking, then its health is rather high; if it starts searching for a health pack, then its health is very probably low; if it starts fleeing, then its health is probably rather low, but with a high degree of uncertainty.

Table 2
$P(H|S_{t+1})$

$S_{t+1}\|S_t$	A	SW	SH	Ex	F	DD
A	x_1	x_2	x_3	x_4	x_5	x_6
SW	10^{-5}	x_2	10^{-5}	10^{-5}	10^{-5}	10^{-5}
Ex	10^{-5}	10^{-5}	10^{-5}	x_4	10^{-5}	10^{-5}
F	10^{-5}	10^{-5}	10^{-5}	10^{-5}	x_5	10^{-5}
DD	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	x_6

All tables on the sensory variables are built on the same pattern; the one giving $P(S_{t+1}|S_t)$ (see Table 2) is special. It gives some sort of basic transition table; i.e. it answers in a probabilistic way the question: knowing nothing but the current state, what will be the next state?

The answer our sample table gives is: tend to stay in your current state (notice the xs on the diagonal) or switch to attack (notice the xs on the first line) with the same high probability; switch to other states with a very low probability (10^{-5} – which in our example we found to be representative of 'very low').

Again, this makes a parallel with an FSM with probabilistic transitions: with our transition table $P(S_{t+1}|S_t)$, we give a basic automaton upon which we build our behaviour by fusing the tendencies given separately on each sensory variable.

3.2. Tuning the behaviour

Tuning our behaviour amounts to tuning our probability distributions. For instance, to create a *berserk* character that is insensible to its health level, we put only uniform distributions (i.e. in our notation, only xs) in table $P(H|S_{t+1})$. A *berserk* is also very aggressive, so the transition table we proposed in Table 2 is quite adapted. A transition table for a more prudent character would not have those xs on the first line, so that the state A would not be particular.

To create a unique behaviour, we therefore have to review all our tables, i.e. the influence of each sensory variable on the character according to the said behaviour.

3.3. Results

Several observations can be made when out bots are playing the game. The first is that their behaviour corresponds to what we want: the behaviour switch-

ing occurs reasonably, given the evolution of the sensory variables. The second is that they cannot compete with humans playing the game. Noting this allows to pinpoint the fact that our method's interest mostly resides in the gain of ease and power in the design of behaviours. It does not pretend to overcome the limitations of the elementary behaviours we are switching between, nor can it do more than what the Gamebots framework allow, in terms of perception and action. Therefore, what we aimed for, and finally obtained, is a reliable, practical and efficient way to specify the real-time selection of elementary behaviours.

Our attempt to tune the behaviour shows that the differences between our 'reasonable' bot and our 'aggressive' bot are visible, and correspond to what we tried to specify in the tables. For instance, the aggressive bot is excited by the presence of several opponents, whereas this situation repels the reasonable bot; and the aggressive bot is not discouraged to attack when its health level goes low.

4. Learning a behaviour

Our goal now is to teach the bot a behaviour, instead of specifying all the probability distributions by hand. It requires to be able to measure at each instant sensory and motor variables of the controlled bot. In particular, it is necessary to determine the state S_t at each instant. It can be done by letting the player specify it directly in real time, or by inferring it from his natural actions in the game.

4.1. Selecting behaviours

This form of learning by example presents a simple interface to the player, shown on Fig. 3.

The player controls in real time the elementary behaviour that the bot executes, by using buttons that allow switching to any state with a mouse click. In addition to the ordinary Unreal Tournament window on the right, part of the internal state of the bot is summed up in the learning interface on the left.

4.2. Recognising behaviours

While the previous method of teaching a behaviour works, it deprives the player of the interface he is used to; his perceptions and motor capabilities are mostly adjusted to the bot's. In order to solve this problem, it is possible to give the player the natural interface of the game, and try to recognise in real time the behaviour he is following.

Fig. 3. Interface used to teach the bot: on the right is the normal Unreal Tournament window showing our bot; on the left is our interface to control the bot.

To recognise the human's behaviour from his low-level actions, we use a heuristic programmed in a classical imperative fashion. It involves identifying each behaviour's critical variables (for instance, attack is characterised by distance and speed of the bot to characters in the centre of his field of view), and triggering recognition at several timescales.

Recognition is done by examining a series of criteria in sequence; the first that matches is chosen. The first criterion is a characteristic event which is back-propagated to states in the past not yet recognised (for instance picking a health bonus indicates that the character has been looking for health). The second examines critical variables over a fixed period (for instance, danger checking is characterised by a complete rotation with little translation, in a short amount of time). Finally, some short-term variations of critical variables are examined (like attacking and fleeing, which consist in moving in a particular direction in the presence of opponents). Exploration is a default state, when a state does not match any of the criteria.

We do this recognition off-line, on data representing 10–15 min of game-play; processing these data and producing the tables that represent our behaviour takes 5–10 s.

4.3. Results

Table 3 shows a comparison between different specification methods. Numbers are the average point

Table 3
Performance comparison on learned, hand-specified, and native Unreal bots (lower is better)

Recognition learned, aggressive	4.4
Recognition learned, cautious	13.9
Selection learned, aggressive	45.7
Manual specification, aggressive	8.0
Manual specification, cautious	12.2
Manual specification, uniform	43.2
Native (level 3/8) UT bot	11.0

difference to the winning bot, over 10 games won by the first bot reaching 100 kills (for example, a bot with 76 points at the end of a game has a point difference of 24 to the winning bot, since the game ends as soon as a bot reaches 100 points). Therefore, a bot winning all games would have a score of zero. Our bots compare well to the native *Unreal Tournament* bot, whose skill corresponds to an average human player. Aggressive bots (grey lines) perform significantly better, and learning by recognition does much better than learning by selection, along with hand specification.

Lessons from these results can be summed up in the following way (we will refer here to Table 4, which is the same as Table 1, but learnt by recognition):

1. learnt tables share common aspects with hand-written tables (as for the transition table $P(S_{t+1}|S_t)$; for instance, in the fleeing state F, health level is much more probably low or medium than high;
2. differences in behaviour of the teacher influence the learnt behaviour: aggressivity (or the lack of it) is found in the learnt behaviour, and translates into performance variations (in our set-up, aggressive behaviours seem to be more successful);
3. nevertheless, differences between hand-specified and learnt models are noticeable; they can be explained by
   (a) player-specific behaviours: humans almost always attack and do not retreat; another example is the low probability of $P(H = \text{High}|S_{t+1} = \text{SW})$ in the learnt table (dark grey cell on Table 4): it can be explained by the fact that human players give a much higher priority to searching a good weapon over searching for health bonuses;
   (b) additional information: some parts of the hand-written tables are specified as uniform (as a result from a refusal or impossibility to specify theoretically a link between two events, like the value of the opponent's weapon knowing that the bot is exploring), whereas their learnt counterparts include information;
   (c) perceptive differences: a human player and a bot have a different perception of sound (the human perceives direction combined with the origin of sound, like an impact on a wall or the sound of the shooting itself, whereas the bot senses only direction);
   (d) bias induced by data quantity: a human player has almost always a medium health level (which is due to a poor choice of discretization for the health level variable), which explains higher values in the learnt Table 4 (line of light grey cells). The discretization is subject to the following constraints: its being too fine makes specifying behaviours by hand harder; it also slows down inference, and increase the amount of data necessary to learn the model; on the other hand, a coarse discretization impedes the expressiveness of the model, by coagulating sensory states too different from one another. A rule to choose discretization is therefore to take a (possibly non-linear) scale where real values from two successive steps are deemed qualitatively different by a human player. For instance, the health level could be better split in the following way: *very low* beneath 70, *low* between 71 and 95, *medium* between 96 and 110, *high* between 111 and 130, *very high* above 131.
4. our learning methods lead to functioning behaviours; learning using behaviour recognition scores best, and allows to reach the level of an average native UT bot.

Table 4
Learnt $P(H|S_{t+1})$

$H\|S_{t+1}$	A	SW	SH	Ex	F	DD
Low	0.179	0.342	0.307	0.191	0.457	0.033
Medium	0.478	0.647	0.508	0.486	0.395	0.933
High	0.343	0.011	0.185	0.323	0.148	0.033

5. Discussion

5.1. Evaluation

We shall now come back to the objectives we listed at the beginning, to try and assess our method in practical terms.

5.1.1. Development team's viewpoint

Programming efficiency. Our method of behaviour design relies upon a clear theoretical ground. Programming the basic model can use a generic Bayesian programming library, and needs afterwards little more than the translation into C++ (for instance) of the mathematical model. Design is really expressed in terms of practical questions to the expertise of the designer, like 'if the bot is attacking, how high is his health level?'; it does not require a preliminary formalisation of the expected behaviour to program. Moreover, in our model behaviours are data (our tables). It means that they can easily be loaded and saved while the behaviour is running, or exchanged amongst a community of players or developers.

Limited computation requirements. The computation time needed for a decision under our model can be shown to be linear in both the number of sensory variables and the number of states.

Design/Development separation. Development amounts to incorporating the Bayesian framework into the control architecture, and establishing the Bayesian model; design consists in establishing relations between the variables in the form of probability distributions. A designer really has to know about what the bot should do, but does not need any knowledge of the implementation details; he needs but a light background on probabilities, and no scripting or programming at all.

Behaviour tunability. We have seen that our way of specifying behaviours gives a natural way to formalise human expertise about behaviours, and that it implies that tuning a behaviour is possible, as they are expressed in natural terms and not in artificial logical or scripting terms. Moreover, the quantity of data needed to describe a behaviour is kept small compared to an FSM, and this helps keeping the analysis and control of a behaviour tractable for the designer.

5.1.2. Player's viewpoint

'*Humanness*'. This criterion is hard to assess, although it can be done [16] in ways comparable to the Turing test [17]. Our method of specifying a behaviour helps the designer translate his expertise easily, and therefore gives him a chance to build a believable bot.

Behaviour learning. We have seen that learning under our model is natural: it amounts to measuring frequencies. This is a chance for the player to teach its teammate bots how to play. Recognising high-level states on the basis of low-level commands is possible, and allows a player to adjust a behaviour completely transparently, with the original controls of the game.

5.2. Perspectives

We have shown a way to specify FSM-like action selection models for virtual robots, and to learn these models by example. The recognition involved in learning from the natural actions of a player in the game remains a classically programmed heuristic; an obvious perspective is to formalise it within the Bayesian framework, in order to perform probabilistic behaviour recognition. This would grant more adaptability to variations in the behaviour model.

Acknowledgements

This work was partially funded by a grant from the French Ministry of Research, the BIBA project (funded by the European Commission), and the CNRS ROBEA project *Modéles Bayésiens pour la Génération de Mouvement*.

References

[1] G. Kaminka, M.M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A.N. Marshall, A. Scholer, S. Tejada, Gamebots: the ever-challenging multi-agent research test-bed, Communications of the ACM, January 2002.

[2] S. Woodcock, Game AI: the state of the industry 2000–2001, Game Developer, July 2002.

[3] J. Laird, Design goals for autonomous synthetic characters, draft (2000), http://www.ai.eecs.umich.edu/people/laird/papers/AAAI-SS00.pdf.

[4] [Baldur's Gate] Complete Scripting Guide, SimDing0, http://www.tutorials.teambg.net/scripting/index.htm

[5] J.E. Laird, It knows what you're going to do: adding anticipation to a quakebot, in: Proceedings of the AAAI Spring Symposium Technical Report, 2000.
[6] S. Woodcock, Flocking with teeth: predators and prey, in: M. Deloura (Ed.), Game Programming Gems 2, Charles River Media, 2001, pp. 330–336.
[7] Neural Network AI for Colin McRae Rally 2.0, Website (Generation5), http://www.generation5.org/content/2001/hannan.asp.
[8] Artificial intelligence: Black and white, Website (Gamespot), http://www.gamespot.com/gamespot/features/pc/hitech/p2_01.html.
[9] E. Dysband, A generic fuzzy state machine in C++, in: M. Deloura (Ed.), Game Programming Gems 2, Charles River Media, 2001, pp. 337–341.
[10] Unrealscript language reference, http://unreal.epicgames.com/UnrealScript.htm.
[11] O. Lebeltel, P. Bessière, J. Diard, E. Mazer, Bayesian Robot Programming, Autonomous Robots, Vol. 16, Kluwer, Dordrecht, January 2004, pp. 49–79.
[12] E.T. Jaynes, Probability Theory: The Logic of Science, unpublished, http://bayes.wustl.edu/(1995).
[13] M. Jordan (Ed.), Learning in Graphical Models, MIT Press, 1998.
[14] E. Dysband, A finite-state machine class, in: M. Dekiyra (Ed.), Game Programming Gems, Charles River Media, 2000, pp. 237–248.
[15] M. Zarozinski, Imploding combinatorial explosion in a fuzzy system, in: M. Deloura (Ed.), Game Programming Gems 2, Charles River Media, 2001, pp. 342–350.
[16] J.E. Laird, J.C. Duchi, Creating human-like synthetic characters with multiple skill-levels: a case study using the Soar quakebot, in: Proceedings of the AAAI Fall Symposium Technical Report, 2000.
[17] A.M. Turing, Computing machinery and intelligence, Mind 59 (236) (1950) 433–460.

Ronan Le Hy received the Diplôme d'Ingénieur from the Ecole Centralede Nantes, France, in 2001, and the Diplôme d'Etudes Approfondies in cognitive science from the Institut National Polytechnique de Grenoble (INPG), France, in 2002. He is currently pursuing Ph.D. degree in cognitive science from the INPG. His current research interests includes programming and learning behaviours for synthetic characters.

Anthony Arrigoni received the Diplôme d'Ingénieur from the Département Télécom of the Institut National Polytechnique de Grenoble (INPG), France, in 2003, and the Diplôme d'Etudes Approfondies in robotics from the INPG the same year. He has explored the learning of behaviours for video game characters.

Pierre Bessière is a senior researcher at CNRS (Centre National de la Recherche Scientifique) since 1992. He took his Ph.D. in artificial intelligence in 1983 from the Institut National Polytechnique of Grenoble, France. He did a post-doctorate at the Stanford Research Institute and then worked for several years in the computer science industry. He has been working for the last 15 years on evolutionary algorithms and Bayesian inference. He leads, with Emmanuel Mazer, The 'LAPLACE Research Group: Stochastic models for perception, inference and action' (http://www.laplace.imag.fr).

Olivier Lebeltel received his Ph.D. in cognitive sciences from the Institut National Polytechnique de Grenoble, France, in 1999. Currently, he is a research associate at the Institut Nationalde Recherche en Informatique et Automatique of Grenoble. He works on modelling, inference, and learning with Bayesian approaches applied to bio-inspired robotics and virtual reality.