Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007)

Extracting NPC behavior from computer games using computer vision and machine learning techniques

Alex Fink
Department of Mathematics
University of California, Berkeley
finka@math.berkeley.edu

Jörg Denzinger
Department of Computer Science
University of Calgary
denzinge@cpsc.ucalgary.ca

John Aycock
Department of Computer Science
University of Calgary
aycock@cpsc.ucalgary.ca

***Abstract*— We present a first application of a general approach to learn the behavior of NPCs (and other entities) in a game from observing just the graphical output of the game during game play. This allows some understanding of what a human player might be able to learn during game play. The approach uses object tracking and situation-action pairs with the Nearest-Neighbor rule. For the game of Pong, we were able to predict the correct behavior of the computer controlled components approximately 9 out of 10 times, even if we keep the usage of knowledge about the game (beyond observing the images) at a minimum.**

Keywords: Computer Games, Object Tracking, Agent Modeling

I. Introduction

The behavior of non-player characters (NPCs) is often crucial for the success of a game. We broadly define NPCs as visible components of the game that are under the control of the computer, and that either work with the human player or against him or her. Designing NPCs is difficult, because human game players would like NPCs to be very human-like, challenging or supporting the player as another human would. This requires complex behaviors that usually are provided by the game designers as a script that uses the game state as visible to the human player, but often also internal information available only to the designers of the game. In fact, there is a temptation to use a lot of information not visible to the player to achieve intelligent behavior and to be able to beat the player (at least from time to time) to keep the game interesting. But the effects of using information not available to the human player can also produce the opposite effect, by making an NPC appear clairvoyant and not very human-like.

In this paper, we present a method to learn the behavior of an NPC based on observing the visual game display, which means that we use the same information as is available to the human player. The learned behavior can be used to see differences between the implemented behavior and what the human player will think the behavior is, but it can also be used to re-engineer a game into a non-script-based NPC version (if the observed behavior is the intended behavior). Another application of our method is to re-engineer the whole game in case the game code is not available – we treat the game as a black box and do not need the internal game state. Our architecture for NPCs, so-called situation-action pairs with the Nearest-Neighbor rule, then also allows for an easy modification of the NPC behavior (see our work in [1]). These intended applications are somewhat similar to what SegMan tries to accomplish for users of the Windows Graphical Interface (see [2]).

Our general approach is as follows: by observing sessions of game play, we capture the sequence of images that the game produces, together with any input from the human game player leading to the images. By using object-tracking techniques from computer vision, we identify and follow various pieces in the game display for which we want to learn the behavior; these are NPC candidates. Using these candidates, we abstract the images to what we call situations. For each pair of following images, we identify the change with respect to the situations represented by them for the different candidates, creating observed situation-action pairs for each candidate. We then compact the large set of observed situation-action pairs into a small set that can be used together with a distance measure on situations and the Nearest-Neighbor rule to represent the behavior of each candidate.

We applied our general approach to the game of Pong which, despite being a simple game, already offers some challenges with regard to determining the behavior of the computer player. Additionally, we tried to use as little knowledge about Pong as possible, so, for example, our method did not know the laws of physics regarding how a ball bounces when hitting an object. Our experiments showed that simple object tracking methods can identify and follow the game objects most of the time. The prediction accuracy possible using only the observed images is around 90 percent for the computer-controlled moving objects (slightly higher for the ball, slightly lower for the computer player) due to an accumulation of necessary approximations in the behavior extraction process. Repetitions of the game situations with different successor situations still presents some challenges, as experiments with several sessions show.

This paper is organized as follows: in Section II, we present our view of computer games, which is the basis for our approach. This general approach is presented in Section III. In Section IV, we present an instantiation of this general approach to Pong with the additional goal of using as little knowledge about game display and game play as possible. In this section we also present our experimental evaluation of the instantiation. Finally, in Section V, we conclude with some remarks on future improvements.

1-4244-0709-5/07/$20.00 ©2007 IEEE

II. Computer games: our view

In this section, we characterize the view of a computer game that we will be using in this paper. We also present an abstract view of the relevant properties of a computer game. While our view is valid for all computer games, it is clearly biased towards commercial games that are not based on search-heavy computer players (like chess or checkers).

We conceive of a game as possessing a set of *states*, and being in exactly one of these states at any given time while it is in progress. The game interacts with a human player by receiving a stream of inputs caused by the player's *actions* which influence the state, and presenting a stream of outputs which are each computed as a function of the state and the received actions. These streams of states, actions, and outputs can be taken to be discrete; even if a game is modeling a continuous change, this change will be discrete at a sufficiently low level of implementation. In this initial work, we assume that the game being analyzed is a single-player game.

Since our perspective is that of modeling a game (and especially the NPCs in that game) based only on observations, we let the timing of the output stream determine the timing of the other two streams. With each output we associate one state. This may be the same as the state of the previous output, if nothing has happened in the game; of course, some states of the game may not be reflected in our sequence if the outputs are few enough. We associate one action with the interval between each pair of states; again, this may require postulating single actions corresponding to the player doing nothing, or doing multiple things, in this time.

More formally, let S, Act, and Out be the sets of states, actions, and outputs of a game. With the passage of time the game takes on a sequence $s_0, s_1, s_2, \ldots$ ($s_i \in S$) of states. Each state is obtained from the preceding one by a function $f_{\text{tr}} : S \times Act \to S$ embodying the behavior of the game: for each index $i \in \mathbb{N}$, $s_{i+1} = f_{\text{tr}}(s_i, a_i)$, where $a_i \in Act$ is the action taken by the player between s_i and s_{i+1}.[1] The game's output is computed by another function $f_{\text{out}} : S \to Out$; while the game is in state s_i, it produces the output $o_i = f_{\text{out}}(s_i)$, yielding the observable sequence $o_0, o_1, o_2, \ldots$ of outputs.

Naturally, a player's external perspective on a game means that the player has no access to the game state s_i except via the history of outputs o_i. If the function f_{out} is one-to-one, s_i is recoverable, and this causes no difficulties. However, in general, there can be *hidden* aspects of the game's state not reflected in the output. If these exist, the state transitions which the player has the capacity to observe can be characterized by a relation $g_{\text{tr}} \subseteq Out \times Act \times Out$ such that for each index i, $(o_i, a_i, o_{i+1}) \in g_{\text{tr}}$. The presence of these hidden components of states complicates approaches which attempt to learn the game's behavior as a function like f_{tr}, because g_{tr} is not necessarily well-defined as a function. This is a problem familiar from agent modeling (see, for example, [3]). Nevertheless, human players expect a strong correspondence between f_{tr} and g_{tr}, because a total inability to predict how the game behaves does not yield a fun game!

The set S generally has considerable structure. Often it can be decomposed as a subset of a Cartesian product $S_1 \times \cdots \times S_k$, where each S_i corresponds to the state of one particular entity within the environment simulated by the game. When such a decomposition is meaningful, the function f_{tr} can be looked at in terms of its component functions $f^i_{\text{tr}} : S \times Act \to S_i$, so that f^i_{tr} describes the behavior of the i-th entity alone.

We are particularly interested in understanding the behavior of those entities that can be considered non-player characters, but we cannot consider these entities in isolation. The actions of NPCs will affect other entities in the environment, which in turn will have an impact on the entities controlled by the player, so in fact it is necessary to model the behavior of all of these components. Even the player's control over their own entities might not occur in an obvious fashion, so it is worthwhile to model the behavior of these entities in response to the player's input as well. Thus, from the point of view of our system, our problem is simply to model an arbitrary entity in a game.

Obviously, learning the behavior of a game and all its entities is not easy. It might even be necessary to get help from a human being in suggesting hidden states, or hidden components of states. But, as already stated, we see the modeling of the behavior of NPCs based on the observations alone as a help for a game developer to see if the game is really doing what it is supposed to do. Therefore an additional goal for us was to create an architecture for describing game entity behavior that can be easily understood and extended. This is the concept of situation-action pairs, as described in the next section.

III. Extracting NPC behavior

In this section, we present our general approach for extracting (or *learning*) NPC behavior. We first present the approach on a high level and then we concentrate on the two main steps, namely creating situations out of images and approximating f_{tr}.

A. The high-level approach

The problem of modeling NPCs in a game as formalized above can be framed as learning the appropriate components of the function f_{tr}, given the input action sequence $(a_i)_{i\in\mathbb{N}}$, the output sequence $(o_i)_{i\in\mathbb{N}}$, and perhaps some additional knowledge about the game. In theory, this can be achieved solving two subproblems: the first is to invert f_{out} to recover the state sequence $(s_i)_{i\in\mathbb{N}}$, the second is to learn f_{tr} given $(a_i)_{i\in\mathbb{N}}$ and $(s_i)_{i\in\mathbb{N}}$. Unfortunately, as stated in the last section, states might have components that are not reflected in the output, making our task a little more difficult and requiring the introduction of some additional concepts.

While an element of S contains all the information needed to compute the follow-up state (given an action) to a state s, the output o produced by this state s may not provide

[1] Note that even the use of a random number generator does not destroy this view of game play if states are modeled appropriately, i.e., including the seed of the random number generator.

all the information that we need. So, it will not be possible in most cases to invert f_{out}. But in most cases it should be possible to create out of the history $(o_i)_{i<k}$ leading to a particular output o_k together with the history of actions $(a_i)_{i<k}$ an approximation of the inverted f_{out} into a structure that we call a *situation* sit, which is an element of a set Sit. Even more, we would like to come up with a function $\overline{f_{\text{out}}} : Sit \times Act \times Out \rightarrow Sit$ that allows us to concentrate only on a single output (and the previous situation or parts of it) to create the next situation.

Working with situations naturally means that we can only approximate the function f_{tr} used by the game, namely a function $\overline{f_{\text{tr}}} : Sit \times Act \rightarrow Sit$. Even more, using situations instead of the states is not the only reason why our function $\overline{f_{\text{tr}}}$ is an approximation of what is really happening in the game. On the one hand, we will probably not observe all possible situations within the output sequence that we use. This means that we need to come up with some way to deal with situations that were not observed while learning the game behavior, since these situations might arise later. For our problem, the use of situation-action pairs as suggested in [6] is an appropriate way for realizing $\overline{f_{\text{tr}}}$. The basic idea of this approach is to define a similarity measure on situations and actions and to determine the value of $\overline{f_{\text{tr}}}$ for a given situation sit and an action a by computing the similarity of (sit,a) to all observed pairs, then returning the follow-up situation of the pair that is most similar to (sit,a).

Given this realization of $\overline{f_{\text{tr}}}$, we face the problem of a large computational effort if we have to compute the similarity of (sit,a) to a large number of observed pairs. Therefore we need to reduce the amount of observed pairs, which is unfortunately another reason why the result of our approach is only an approximation of the game's f_{tr}-function.

Our general approach consists of the following steps: Given the two sequences $\{o_i\}_{i\in\mathbb{N}}$ and $\{a_i\}_{i\in\mathbb{N}}$ and the start output o_0 of the game, we

- determine a suitable set Sit using the output sequence (and perhaps some human help),
- determine sit_0 using o_0,
- for each i, we compute sit_i using o_i, a_i and sit_{i-1}, resulting in the set $SPP = \{((sit_{i-1}, a_i), sit_i) | i \in \mathbb{N}\}$
- compact SPP.

Since determining a Sit and computing elements of it from the output are highly related, we will look at these two problems together in the next subsection. Then we will look at one possibility for creating and compacting a set SPP.

B. From outputs to situations

The output of a computer game, with the exception of "games" played in large simulators, consists of graphics on the computer screen and audio signals. While some games use audio to convey additional information, most games use images as the primary information-bearing mode of output, and the generated sound serves either as background or to highlight some significant event (state change) in the game that is also graphically indicated. Therefore we concentrate on determining a suitable set Sit and the situations corresponding to the observed output from graphical output.[2]

When constructing a game situation from a frame of graphical output, we have to include in a situation specifications of any moving object we discover to be visible in that frame. Additionally, in many games there are frame areas that contain changing graphical symbols, usually out of a small set of possible symbols. While the frame areas with changing symbols can be rather easily identified and the set of symbols approximated by simply collecting the symbols in an area that are observed in an output sequence $\{o_i\}_{i\in\mathbb{N}}$, the detection and tracking of moving objects requires help from the field of *object tracking* (see [5]), a well-studied area within computer vision.

We take each entity in a game to be an object which moves about within a two-dimensional space, having a position and (instantaneous) velocity, and perhaps being absent altogether at certain times. If the moving object is allowed to change its displayed representation, then the mode of representation also needs to be represented in the game situation, since obviously it is also part of the state of this object that we want to approximate. Our motivation for including velocity in this setup of a situation instead of position alone is to provide a compact representation of the history of an object. While there might be additional history information that should be included in a situation, there are also games where the velocity of a moving object is not of consequence at all. However, velocity is an feature which we expect from our physical world and therefore an obvious part of a situation description.[3]

So, an element of the Sit we need has to at least contain the following parts:

- for each moving object ob_i:
  x_i, y_i, v_i, r_i
  with x_i,y_i the coordinates in the two-dimensional picture (including the value $\perp$ indicating that the object is not visible now), v_i the velocity of ob_i and r_i its current display representation, i.e., the symbol out of a set R_i used to display ob_i in the current situation
- for each non-moving, but changing object cob_j:
  r_j
  with r_j being the current display representation for cob_j, again out of a set R_j of possible symbols.

As already stated, the elements of each R_i are determined by going through all observed outputs and collecting all symbols

[2]The games industry is constantly looking for additional and/or alternative output devices that improve the gaming experience for a human player. But we believe that visual output will always play an important role, simply because our visual sense is so central in our life. We nevertheless want to point out that there are already techniques to learn any kind of output language that might be used in addition to graphical output, as, for example, suggested in [4], and such techniques can be integrated into our approach to create additional parts of a situation.

[3]We are aware that some game designers play interesting games with physics engines, like manipulating gravity, to achieve certain effects and for such games there will be the need for a human to assist our approach. But our approach will at least show that there was some kind of manipulation going on, as explained later.

appearing in a particular place. We represent each R_i as a list and then an r_i as above is simply a reference to the list (i.e., the number indicating the position in this list).

This leaves us with the problem of finding out what the moving objects in the game are. A naïve way to solve this problem in any given image would be to find connected components of non-background-color pixels. We have taken a slightly more sophisticated approach. Acknowledging that there may be, for instance, scenery "objects" in the background or features not rendered in a single solid color, we instead take differences of successive pairs of frames to detect all changes between these frames, which we take to be indicative of moving objects. This was sufficient for our application in Section IV, but for some games there might be the need to use more complex (and more expensive) techniques for object tracking. For implementational simplicity we immediately convert the output images to gray-level images; this tends to preserve recognizability of scenes (see [5]).

Let I_1 and I_2 be two consecutive frames of output. Their *difference image* $I_2 - I_1$ is the image with

$$(I_2 - I_1)_{ij} = (I_2)_{ij} - (I_1)_{ij},$$

where I_{ij} is the (i, j) pixel of image I. Every object which moved (or changed representation) between I_1 and I_2 generates a region of nonzero pixels in $I_2 - I_1$. In most cases a *connected* component of nonzero pixels will be the result, and so we generally identify one object with each such connected component. However, this is not always the case: one particular counterexample is illustrated in Figure 1, which our system heuristically identifies and treats as a single object (see Section IV-B for more detail).

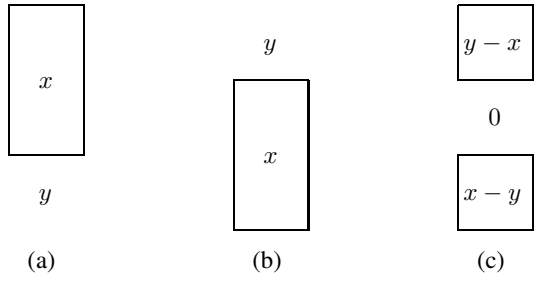

Fig. 1. A moving object whose motion from (a) to (b) generates the disconnected difference seen in (c). Labels of regions denote uniform colors which fill the region.

Of course, simply finding the individual appearances of each moving object in each pair of frames is not enough. In order to define Sit it is also necessary to know which of the objects discovered in various output frames are actually the same object, and how many objects actually appear. Our approach assigns to each difference image component a label, indicating which object it represents. We find the total number of objects, k, and then for each of these ob_is ($1 \leq i \leq k$) construct their contribution to the situation by finding the two nearest attestations of objects with label i, and interpolating between their attested positions to find a position and velocity.

Our approach to this problem is based on the assumptions that objects generally move continuously, so that the positions of the same object in two successive frames will overlap; and that objects retain roughly the same *color* as they move. By 'color' we mean the average absolute value of all pixels in the connected component in the difference image; this value will be a function of the color of the original object if the background has a solid color.

The algorithm of Figure 2 attempts to perform this object classification making only a single pass through the output data. This algorithm, however, has problems with tracking objects when they move discontinuously or disappear for a time, or when one object overlaps another (because the two will be merged into one connected component). Accordingly, the algorithm which we implemented is based on this one, but includes a number of ad hoc refinements using properties of the particular game in our application to handle these cases. We will present these refinements in Section IV-B.

```
1  mark all image components unused
2  for each frame F
3  do  for each image component C in F
4      do  if there exists any image component C'
           in an earlier frame F' such that C' and C
           overlap and have similar colors
5          then let D be the most recent such
               component
6               classify C as the same object as D
7               mark D used
8          else classify C to belong to a new object
```

Fig. 2. Approximate pseudocode for classifying objects.

C. $\overline{f_{tr}}$ and creating a compactified SPP

The last subsection described how we can create a sequence $(sit_i)_{i\in\mathbb{N}}$ out of an observed output sequence $(o_i)_{i\in\mathbb{N}}$ and a sequence of user actions $(a_i)_{i\in\mathbb{N}}$. In this subsection, we present how our approximation function $\overline{f_{\text{tr}}}$ is defined using these sequences and how we create functions $\overline{f^i_{\text{tr}}}$ for the entities of the game.

The intention of $\overline{f_{\text{tr}}}$ is to predict what the next game situation will be, given the current game situation and the action that the user takes, i.e., $\overline{f_{\text{tr}}}$: $Sit \times Act \to Sit$. For a game entity, which obviously includes all NPCs, we are only interested in the changes to a particular entity. For a moving object ob_i, this means we want to predict the new position of the object, its velocity and its new representation (and all other parts of a situation that refer to the object). For a non-moving, but changing object, we want to know its new representation. In general, this means that $\overline{f^i_{\text{tr}}}$: $Sit \times Act \to \overline{Sit^i}$, where Sit^i is the set of possible value-combinations of the situation components describing entity i and $\overline{Sit^i}$ is essentially the same set but uses relative positions for the position part instead of absolute positions. So, we describe

only the change of position (relative to the old position in x and y coordinates). This is necessary, since we might not have observed the game output for each possible situation, and our approximation should predict the same movement for similar situations.

To construct the $\overline{f^i_{\text{tr}}}$s, we use the idea of situation-action pairs of [6], except that in our case we have a situation together with a user action as the first element of a pair and a prediction of what will happen, as described above, as the second element of the pair. The general idea is to have a set $SPP^i = \{(prot_1, pred_1),...,(prot_n, pred_n)\}$, where $prot_j \in Sit \times Act$ and $pred_j \in \overline{Sit^i}$ for all j. We then also define a *similarity* (or *distance*) *function* $dist^i$: $(Sit \times Act)^2 \rightarrow \mathbb{R}$ that measures how similar two pairs of situations and actions are. If we want to predict the value of $\overline{f^i_{\text{tr}}}$ for (sit,a), we compute $dist^i(prot_j,(sit,a))$ for all $j \in \{1, ..., n\}$ and return $pred_m$ for the element m in SPP^i for which the distance is minimal. If there are several $m_1,...,m_l \in \{1, ..., n\}$ for which the distance is minimal, we return the prediction that occurs most often among $pred_{m_1},...,pred_{m_l}$. Ties in this respect are broken randomly.

This leaves us with how to get an appropriate distance function. While there are many possible candidates for such a function (see [7] for some possibilities), we have to take into account what we want to achieve with our approach, namely to create a model of NPC behavior reflecting what an observer of the game most probably will assume as being the model. And with this regard, situations where all moving entities are at approximately the same places in the display usually are considered very similar, whereas situations that only differ quite a bit in the position of one entity are less similar and situations that differ a lot in the positions of several entities are not considered as similar at all. This suggests the use of a Euclidean distance measure for the position and velocity parts of situations.

But what about actions, representations, and possible other parts in a situation description? Here, there are obviously different ways that these components (and their differences) can be integrated into the distance function; different human players may vary in how they use the information represented by these components in their model of what an entity will do. In general, this means that the definition of the distance function has the following structure:

$$dist((sit,a),(sit',a')) = w_a \cdot d_a(a,a') + \sum_{j=1}^{k} w_r^j \cdot d_r^j(r_j, r'_j) + \sum_{j=1}^{k} (w_p^j \cdot (\|x'_j - x_j\| + \|y'_j - y_j\|) + w_v^j \cdot \|v_j - v'_j\|)$$

w_a, w_r^j, w_p^j and w_v^j are weight parameters that can be chosen by the user of our system and d_a and d_r^j are distance functions on Act and R_j, respectively, that need to be defined by the user (if actions and representation should play a role in measuring the similarity of situations).

The final problem we have to address is how to get the sets SPP^i for the different entities. An obvious way is to just create them out of all $((sit_j,a_j),sit_{j+1})$ tuples out of $(sit_i)_{i\in\mathbb{N}}$, $(a_i)_{i\in\mathbb{N}}$ by computing for each entity the necessary components, which boils down to computing the position and velocity changes. But this can result in very large sets SPP^i which requires a substantial computational effort to evaluate $\overline{f^i_{\text{tr}}}$ for a given situation and action (remember, we have to compute the distance to each element in SPP^i). Therefore we have to reduce this initial candidate set for SPP^i to an appropriate and manageable size. There is some work that addresses the problem of *instance reduction* in relation to using a set of examples and the Nearest-Neighbor rule (see, for example, [8], [9], [10], [11]), which essentially is the problem of selecting a small subset out of a given set of instances while retaining without (too much) loss the predictive accuracy of the approach.

The general idea of most of these approaches is to use the distance function used by the Nearest-Neighbor rule to cluster the set of examples into the appropriate number of classes and then to select out of each cluster an instance to represent the cluster. Naturally, the instances in a cluster might differ in what their prediction is, and how this is handled is where the difference between the different approaches lies. There is obviously a trade-off between the targeted size of the end set and the accuracy that this set will achieve (compared to the unreduced set).

Potentially, all of these methods can be used for our purpose. We used the source provided in [11] for our experiments of the next section, modified to use our instance structure (i.e., prototype-prediction pairs) and our distance function.

IV. Experimental evaluation with Pong

Examining our general approach described in the last section, there are several points in it that offer the possibility or even the need to integrate knowledge about the game, or an NPC entity, into the process of creating a behavior model. The consequence of having such knowledge is that it allows deriving some deep information, for example distance measures for the representations an object can have, which is otherwise a rather subjective task.

For our experimental evaluation, we wanted to use a game where such subjective tasks are not necessary, so that we get a baseline for what our approach can achieve without having a user provide help to it. And naturally this means that we should use a game that at least offers the possibility of success for such a fully-automated approach, since games with hidden information in their states obviously would result in failure without using additional knowledge provided by the user. "Classic" video games such as Pong, Breakout, Space Invaders, Pacman, and so forth, are examples of games that expected the human player to learn the game behavior to beat the game in the end. And therefore we have chosen one of these games, Pong, for our experimental evaluation. In the remainder of this section, we will first describe the Pong variant we use, then present how we instantiate our

Fig. 3. A Pong screen-shot

general approach to Pong and finally our experiments with this instantiation.

A. Pong

Pong, as the name suggests, is the first attempt of computer games to imitate human sports, namely ping pong (i.e., table tennis). There are many versions of Pong and we have chosen the one in the 1977 Atari game *Video Olympics* in the human vs. computer mode (Figure 3 is a screen-shot of the game). We ran this game using Stella (see [12]), an open source multi-platform emulator for the Atari 2600 that already has screen-shot capabilities and that we modified to do a screen-shot for each frame and write a log of input events (i.e., actions).

The game itself is easily explained. Each player controls one of the paddles, i.e., the large rectangles in the playing field (the computer controls the left one), and putting a paddle in the path of the third rectangle, the ball, will cause the ball to bounce off (according to the laws of physics as implemented in the game) towards the side of the opponent. The human player steers his/her paddle by pointing the mouse where it is supposed to go. A player scores if the ball is not intercepted by the opponent before it leaves the screen on the opponent's side. The current score is indicated by the symbols (numbers) at the top of the screen.

B. Instantiating our method

The first step of our approach is to use object tracking on the screen-captured frames to determine the objects and to create the structure of an element of Sit. Figure 3 is already a gray-scale image of the normal color image of the Pong game we used. All entities within this game (even background and boundaries) have distinct colors, except for the scores that are in the color of the player that has the score. But our system cannot be sure about that. In fact, for Pong the problem depicted in Figure 1 occurs very frequently. The Pong-specific heuristic we used to deal with this situation is to compute the difference in grey-scale average value of neighboring difference image components and to declare two components as representing the same object if the difference is equal (or near) to zero. Obviously, this workaround does not work if the representation of a moving object goes beyond being a connected component of the same color. Because of the boundary between playing field and the scores, we had no problems due to the score symbols using the same color as the paddles.

Using this workaround, our system was able to detect the ball, the computer paddle, and the player paddle and to keep track of them over most of the captured frames. A problem occurred every time after either player scored, since after scoring, the ball, that had vanished when the score happened, reappears in the center of the playing area and the paddle of the opponent player jumps to a position in front of the ball. If we want to learn the behavior of a single entity, then this "teleportation" of entities is not very straightforward, and it is no surprise that this causes problems for our system in finding the entity in the next frame. But if we commit to having only two paddles and the ball in the playing area (which we did), then our system can use this additional information to identify the new position of the paddle and the ball when preparing the situation sequence out of the output sequence.

Out of the results of the object tracking we get as structure of an element of Sit for Pong:

$(x_{co},y_{co},v_{co},x_{pl},y_{pl},v_{pl},x_{ba},y_{ba},v_{ba},r_{cosc},r_{plsc})$

where "co" identifies the components belonging to the computer player, "pl" the ones belonging to the human player and "ba" to the ball. "sc" indicates that it is a component belonging to a non-moving score object. We do not include representation components for the moving objects, since they do not change and do also not contribute to the distance measure, as is explained in the next paragraph.

Given this structure, and that an action in Pong is characterized by the coordinate in the playing field that the mouse points to, the only instantiations that we need for the general distance measure are how to treat the representations of objects and actions, and what weights to choose for the components of the function. Since we want to minimize the additional knowledge we put into our instantiation, we deal with the representations of objects by declaring $d_r(r,r') = 0$ if $r = r'$, and 1 otherwise. This allows us to get rid of the representation component of the moving objects, while still making the fact that a non-moving object changed available to our function $\overline{f_{\text{tr}}}$. Since actions are positions, we employ the Euclidean distance again, as in the case of positions of objects. We set all weights to 1.

C. Results

Since we already presented the results of the object tracking (at least with regard to the detection of objects) in the last subsection, the main question remaining is if the behavior models for the NPCs (or moving objects) are approaching the real thing. As stated in Section III-A, there are many aspects that conspire against the perfect accuracy of our method and many of these aspects are true even for Pong.

TABLE I
EXPERIMENTAL RESULTS: ACCURACY PERCENTAGE

training sequence	accuracy		
	ball	computer player	human player
1	93.6	88.6	77.9
2	93.6	89.0	80.9
3	91.2	88.2	88.2
1 + 2 + 3	88.7	85.6	77.1

We do not have the source code for the Video Olympics' version of Pong, so that a direct comparison between the real game state and a situation and the prediction from this situation was not possible for us. So, what we had to do was to compare the accuracy of the predictions for the entities with what really happened in the game. We set our evaluation experiments up as follows. Each training sequence was obtained by having a human play against the computer until 4561 frames were produced. Our results are reported in Table I and each accuracy entry represents the best result among the methods from [11]. As can be seen by the results for the individual games (i.e., the rows for training sequences 1, 2 and 3), the prediction accuracy of the models generated for the three moving objects is indeed not perfect. The best results are obtained (in all three cases) for predicting what the ball does, which is a good result, since this is really the first information that a human player needs to get his/her own paddle into the right place. The results for predicting where the computer goes are also very similar in all cases and often much better than predicting what the human did.

A little bit puzzling at first is the last row, that sees clearly worse results when using all sequences together to create the prediction models. But we have to be aware that with different game sessions the chance for having the same situation-action pairs occurring with different predictions gets higher (than it already is in a single session, as indicated by the accuracy there) due to having more frames per score (to give just one reason) and therefore there will be more chances for making the wrong prediction. Also, the tactics of a human player do not usually change much during a game, while with some time in between usually changes occur (reflecting the learning of the player) which also accounts for more cases where the same prototype is associated with different predictions. And we also have to take into account that the compaction process amplifies this problem.

V. CONCLUSION AND FUTURE WORK

We presented the idea to use techniques from computer vision and machine learning to extract the behavior of NPCs in games out of the game by observing the interactions between the game and the human player. Our general approach is to use object tracking through the frames of graphical output produced by the game during a session of play to identify and follow the objects of the game (including the NPCs). This way, we can transfer the frames into situations (possibly using additional knowledge about the game), that, together with the user interactions, are used to model the behavior of each object using situation-prediction pairs and the Nearest-Neighbor rule.

Our evaluation of this approach using the game of Pong and trying to include as little knowledge about the game as possible shows that predictions of the behavior of computer controlled entities are possible with an accuracy of around 90 percent. While this already can help a game designer to comprehend what a game player will be able to learn about NPC behavior while playing the game, obviously some improvement will be necessary if we want to tackle more complex games.

In fact, our general approach allows for a lot of improvements if using additional knowledge about the game is allowed; exploring these possibilities is our future work. On the object tracking side of things, we employed a rather primitive version that did not require knowledge about icons and graphical presentations for a particular entity. Adding the knowledge of how an entity represents itself on the screen allows for improvements, although we will have to use techniques that deal with representations overlapping each other that go beyond what we did so far. The use of situation-prediction pairs allows for various ways that we can use to include knowledge: in the distance measure, by adding certain known behaviors (as in [1]) and by adding components to the description of a situation that represent this additional knowledge. We also expect to need more knowledge about previous situations for more complex games, like, for example, first person shooter games with their potentially rapid change of what the player sees from frame to frame. Finally, any approach to learn a computer's strategy in board games like chess or checkers will need deep knowledge about the game's rules and quite a number of experiences by the learner.

VI. ACKNOWLEDGMENTS

The authors would like to thank Jim Parker for helpful suggestions, especially regarding object tracking. The first author did this work while at the University of Calgary. The third author's research is supported in part by grants from the Natural Sciences and Engineering Council of Canada.

REFERENCES

[1] J. Denzinger and C. Winder. Combining coaching and learning to create cooperative character behavior, Proc. CIG-05, Colchester, 2005, pp. 78–85.
[2] M.O. Riedl, R. St. Amant. SegMan technical notes, http://www.csc.ncsu.edu/faculty/stamant/segman-introduction.html.
[3] P. Gmytrasiewicz and E. Durfee. Reasoning about Other Agents: Philosophy, Theory and Implementation, Proc. 12th WS on DAI, 1993, pp. 143–153.
[4] L. Steels. The Origins of Ontologies and Communication Conventions in Multi-Agent Systems, *Autonomous Agents and Multi-Agent Systems* 1(2), 1998, pp. 169–194.
[5] J.R. Parker. Algorithms for image processing and computer vision, Wiley, 1998.
[6] J. Denzinger and M. Fuchs. Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, Kyoto, 1996, pp. 48–55.
[7] J. Denzinger and A. Schur. On Customizing Evolutionary Learning of Agent Behavior, Proc. AI-04, London, ON, Springer LNAI 3060, 2004, pp. 146–160.

[8] D.W. Aha, D. Kibler, M.K. Albert. Instance-Based Learning Algorithms, *Machine Learning* 6, 1991, pp. 37–66.

[9] B.V. Dasarthy, J.S. Sánchez, S. Townsend. Nearest Neighbor Editing and Condensing Tools-Synergy Exploitation, Pattern Analysis and Applications, vol. 3, Springer, 2000, pp. 19–30.

[10] J. Denzinger and J. Hamdan. Improving Modeling of other Agents using Tentative Stereotypes and Compactification of Observations, Proc. IAT 2004, Beijing, 2004, pp. 106–112.

[11] D.R. Wilson, T.R. Martinez. Reduction Techniques for Instance-Based Learning Algorithms, *Machine Learning*, 2000, pp. 257–286.

[12] B.W. Mott et al. Stella: a multiplatform Atari 2600 VCS emulator, http://stella.sourceforge.net.