Chapter 6
Believable Bot Navigation via Playback of Human Traces

Igor V. Karpov, Jacob Schrum and Risto Miikkulainen

Abstract Imitation is a powerful and pervasive primitive underlying examples of intelligent behaviour in nature. Can we use it as a tool to help build artificial agents that behave like humans do? This question is studied in the context of the BotPrize competition, a Turing-like test where computer game bots compete by attempting to fool human judges into thinking they are just another human player. One problem faced by such bots is that of human-like navigation within the virtual world. This chapter describes the Human Trace Controller, a component of the UT^2 bot which took second place in the BotPrize 2010 competition. The controller uses a database of recorded human games in order to quickly retrieve and play back relevant segments of human navigation behaviour. Empirical evidence suggests that the method of direct imitation allows the bot to effectively solve several navigation problems while moving in a human-like fashion.

6.1 Introduction

Building robots that act human requires solutions to many challenging problems, ranging from engineering to vision and natural language understanding. Imitation is a powerful and pervasive primitive in animals and humans, with recently discovered neurophysiological correlates [1]. Children observing adult behaviour are able to mimic and reuse it rationally even before they can talk [2]. As robotics platforms

I. V. Karpov (✉) · J. Schrum · R. Miikkulainen
The University of Texas at Austin, Austin, TX 78712, USA
e-mail: ikarpov@cs.utexas.edu

J. Schrum
e-mail: schrum2@cs.utexas.edu

R. Miikkulainen
e-mail: risto@cs.utexas.edu

P. Hingston (ed.), *Believable Bots*, DOI: 10.1007/978-3-642-32323-2_6,

continue to develop, it is becoming increasingly possible to use similar techniques in human-robot interaction [3].

How can we use imitation when building agents with the explicit goal of human-like behaviour in mind? We study this question in the setting of the BotPrize 2010 competition, which explicitly rewards agents for exhibiting believable human-like behaviour [4].

One problem faced by such bots is that of human-like navigation within the virtual world. Due to problems in level design and the interface used by the bots when acting in the environment, bots can get stuck on level geometry or fail to appear human when following the built in navigation graph.

As a way to address such challenges, this chapter introduces the *Human Trace Controller* (HTC) , a component of the UTˆ2 bot inspired by the idea of direct imitation of human behaviour. The controller draws upon a previously collected database of recorded human games, which is indexed and stored for efficient retrieval. The controller works by quickly retrieving relevant traces of human behaviour, translating them into the action space of the bot, and executing the resulting actions. This approach proves to be an effective way to recover from navigation artefacts in a human-like fashion.

This chapter is organized as follows. Related work is discussed in Sect. 6.2. The necessary background including a description of the BotPrize competition and domain used in it is discussed in Sect. 6.3. The main focus of this chapter, the Human Trace Controller, is described in detail in Sect. 6.4. Section 6.5 presents the results of qualitative and comparative evaluations of the controller. Sections 6.6 and 6.7 discuss future work and conclusions.

6.2 Related Work

An active and growing body work uses games as a domain to study Artificial Intelligence [5–8].

The use of human player data recorded from games in order to create realistic game characters is a promising direction of research because it can be applied both to games and to the wider field of autonomous agent behaviour. This approach is closely related to the concept of Imitation Learning or Learning from Demonstration, especially when expanded to generalize to unseen data [9–11].

Imitation of human traces has previously been used to synthesize and detect movement primitives in games [12], however this approach has not been evaluated in the framework of a human-like bot competition. The use of trajectory libraries was introduced for developing autonomous agent control policies and for transfer learning [13, 14]. Hladky et al., developed predictive models of player behaviour learned from large databases of human gameplay and used them for multi-agent opponent modeling [15]. Human game data is also collected in an attempt to design non-player characters capable of using natural language [16]. Imitation learning using supervised models of human drivers was used in order to train agent drivers in the

TORCS racing simulator [17]. In robotics, imitation learning approaches have been shown effective as well, for example in task learning and programming robosoccer players in simulation [18]. Statistical analysis of player trajectories was used in order to detect game bots in the Quake first person shooter game [19]. Sukthankar et al. use similar techniques in order to assign teams and recognize team player behaviour in multiagent settings [20]. Most recently, a competitor team, `ICE`, is using an interface for creating custom recordings of human behaviour in the BotPrize competition [21].

While human behaviour traces and learning from demonstration techniques are finding increasing use in both games and robotics applications, the BotPrize competition offers a unique opportunity to test such methods in creating human-like behaviour directly. The challenge of combining imitation and demonstration methods with other types of policy design methods remains to be met.

6.3 Background

This section describes the domain of Unreal Tournament, the BotPrize Competition, and some of the challenges to developing human-like game bot behaviour posed by the software interface used when developing the bot.

6.3.1 Unreal Tournament 2004

Unreal Tournament 2004 is a commercial sequel in a series of first person shooter computer games developed by Epic Games and Digital Extremes [22]. After it was published in 2004, the game received Multiplayer Game of the Year awards from IGN, Gamespy and Computer Gaming World.

The Unreal 2004 game engine consists of a server which runs the game simulation including 3D collision detection, physics, player score, statistics, inventories and events. Importantly, the Unreal 2004 game engine includes an embedded scripting system that uses Unreal Script, an interpreted programming language that provides an API to the game engine. This scripting interface is used by higher-level wrappers such as `GameBots2004` and `Pogamut` to allow external programs to control bot players and receive information about their state [23, 24]. Unreal Script also allows the recording of detailed game traces from games played by humans and bots.

Players connect to the server using Unreal Tournament clients (either locally or via a network using TCP and UDP protocols). Clients provide 3D graphics and audio rendering of the view of the Unreal level from the perspective of the player, and allow the player to control their character via keyboard and mouse commands. Commands are customizable, but basic keyboard controls include movement forward, back, left and right, jumping up, crouching, and selecting a weapon from the player's inventory, while the mouse allows the player to turn, aim, and fire primary and secondary weapons. Each player has some amount of health and armour and an

array of weapons from one of the predefined weapon types as well as some amount of ammunition for each.

Multiplayer games can include up to sixteen players per game simultaneously. These players can include both humans and bots. Normally players can choose from a number of *native bots* within Unreal—these are implemented as internal subroutines within the engine and can have a different skill level depending on a numerical parameter. Both bots and human players can choose from a wide variety of avatars or *skins* which represent them during the game. These are all humanoid in appearance and roughly similar in size—the engine uses the same underlying character animations during movement, using custom animations only in special situations such as taunts.

The goal of the normal Death Match mode is to be the first to eliminate a predetermined number of opponents by hitting them with weapon fire and reducing their health and armour to zero. Players who are killed quickly re-spawn at one of several predefined spawn locations on the level map. Games can optionally have a time limit, at the end of which the player with the highest number of frags (kills) wins.

6.3.2 The BotPrize 2010 Competition

The BotPrize competition aims to create a Turing-like interactive test for bot behaviour [4, 25]. During the 2010 competition, five teams competed during the course of three one-hour rounds consisting of three 20-minute games each. Each game took place on one of the three large levels pre-selected by the competition organizer (and known to the participants beforehand). These levels (*Goatswood*, *IceHenge* and *Colosseum*), were created by the Unreal Tournament player community after the release of the original game and are large relative to the character avatar and avatar motion speed, allowing a large number of participants to interact frequently but without too much interference from other players. The levels also contain complex 3D geometry and visual and physical obstacles such as trees, walls, columns, fences, bridges and water. Equal number of bot participants, human judges and native bots participate in each game. All the participants are anonymized by using a random in-game name and a random in-game skin. The rules of the game are modified from a regular death match as follows:

- One of the default weapons in the player arsenal is a "Judging gun" that allows players (both humans and bots) to record on-the-fly decisions about whether or not they think their opponent is a human.
- Damage by normal weapons is reduced to 40 % to allow players more time to interact.

Hitting a player with the primary fire of the judging gun for the first time designates the player as a human. Hitting a player with the secondary fire of the judging gun for the first time designates the player as a bot. If the designation is correct (the player

correctly guesses whether the opponent is a human) the player gets 10 frags and the opponent is instantly killed. If the designation is incorrect (the player fails to guess whether the opponent is a human) the player gets —10 frags and dies instantly. The judgement can only be made once per opponent—judging the same opponent twice has no effect.

The goal of the human players is to get the highest score—thus it is in their interest to guess whether or not their opponents are bots correctly. The goal of the bot players is to appear human, thus ideally their policy should be indistinguishable from what is expected from a human player in a given situation.

Additional details about the competition are available in previous publications by the organizer, Philip Hingston [4, 25], and in Chap. 5 of this book.

6.3.3 The GameBots2004 and Pogamut Interfaces

The `GameBots2004` interface allows a program to control a bot within Unreal Tournament using a network socket connection [23]. Synchronous and asynchronous messages are exchanged between the agent and the `GameBots2004` BotConnection running on the Unreal Tournament server, allowing the agent to receive updates about the game and the bot state and to send commands controlling its motion and its actions.

The `Pogamut` framework is a Java library that uses `GameBots2004` to provide the developer with a convenient API for accessing game and agent state and writing custom behaviours for Unreal Tournament 2004 bots [24]. In particular, the framework takes care of the details of updating agent's memory (state) information, wraps `GameBots2004` messaging protocol in a Java API, and provides a class hierarchy for representing useful data structures for building bot behaviour, including navigation graphs, inventory items, sensors, actions, and so on.

6.3.4 Navigation in Unreal Tournament 2004

The interfaces to the Unreal Tournament game engine support two main styles of navigation for bots. In one of these, the bot specifies the location (or locations) in its immediate vicinity where it wishes to move, and the game engine executes this motion, calculating the appropriate animations and adjusting the bot's location according to reachability and physical constrains. In the second mode of navigation, a *navigation graph* provided by the creators of the level maps can be used in order plan longer-term routes that take the bot from one location on the level to another.

In general, a navigation graph for a level consists of a relatively small (on the order of several hundred) number of named vertices, or *navpoints*, distributed throughout the level and connected by a network of *reachability edges* (Fig. 6.1). If two adjacent navpoints are connected by an edge, the engine should be able to successfully move the avatar between them.

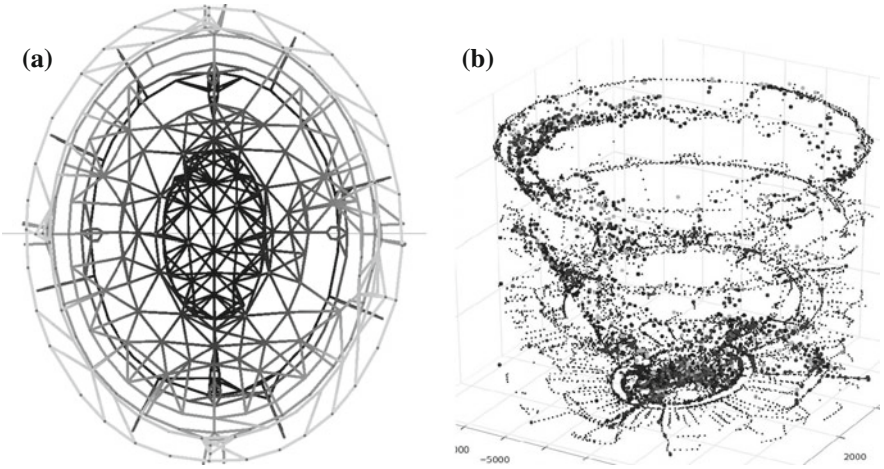

Fig. 6.1 A side-by-side comparison of the two-dimensional projections of the navigation graph (a) used by native Unreal bots and of three-dimensional view of the position and event samples of human behaviour (b) used by the Human Trace Controller and on one of the competition levels. **a** Navigation graph for Colosseum. **b** Human traces for Colosseum

Bots can use standard A* pathfinding such as the Floyd-Warshall method [26] to get from one location on the map to another. A* and Floyd-Warshall are pre-packaged in the Pogamut framework, and they work well, but they are different (and seemingly not quite as good as) what is built into UT2004 and used by the native bots. Part of the reason for this difference may be that path following sometimes involves well-timed jumps. The navigation and jumping capability is dramatically improved in the Pogamut 3.1 release, however, the BotPrize 2010 and the evaluations presented in this chapter were done using the competition version.

Even with the best of interfaces, error-free navigation is not usually available for domains with high complexity. This is certainly the case with mobile robot navigation where sensor and motor errors, physical slips and other inaccuracies can combine to both stochastic and systemic errors over time. Even in simulated environments such as Unreal Tournament, the well-timed execution of new destination commands along the path could be adversely affected by the additional network latency or computational overhead of the interfaces.

Because navigation primitives used by the bot are not error-free, it often gets "stuck", where the actions that it chooses to execute as part of the path following (or combat) behaviour do not cause any progress because an obstacle is in the way. This can be caused by navigation command execution errors, by collisions with other players, by imprecisions in the navigation graph, or by simulated physics interactions with weapon fire. Humans do not play in this manner, and judges often exploit this behaviour to make negative decisions . The UTˆ2 bot uses the *Human Trace Controller* (HTC) in order to quickly detect and recover from such navigation problems.

6.4 The Human Trace Controller

The UT^2 bot is implemented using a method similar to behaviour trees. The overall architecture of the bot is described in detail in Chap. 5. In the BotPrize 2010 competition, the bot uses the HTC in order to improve its navigation behaviour: when the bot gets stuck while moving along a path or during combat (Sect. 6.4.3), it executes actions selected by the controller.

The HTC uses a database of previously recorded human games to execute navigation behaviour similar to that of human players. This section describes how the data is recorded (Sect. 6.4.1) and indexed (Sect. 6.4.2) to select what actions to execute (Sect. 6.4.4).

6.4.1 Recording Human Games

Player volunteers were selected from graduate and undergraduate students at UT Austin. All players had considerable previous experience with video games in general and first person shooter games in particular. Familiarity with the Unreal Tournament 2004 game varied between the players.

An early version of the competition mod that allowed the judging gun to be used was instrumented with the ability to log each player's pose and event information into a text file, as follows. The standard BotPrize mod was decompiled into its original UnrealScript using the standard tools that come with the Unreal Tournament 2004. The script was then modified to write out a detailed log of the events and commands used by the players into a text file. The text file was then processed in order to extract traces of human behaviour, and these traces were stored in a SQLite database [27]. The frequency of the samples in the human database thus roughly coincided with the average logic cycle of the Pogamut/GameBots configuration, around ten times a second.

Two types of data points were recorded in the database for each human player: pose data and event data. The player's *pose* includes the current position, orientation, velocity and acceleration of the player. This data was recorded every logic cycle, or around ten times a second. The player's *events* were recorded as they happened together with their time stamp, and included actions taken by the player or various kinds of interactions with other players or the environment. Example event types include picking up inventory items, switching weapons, firing weapons, taking damage, jumping, falling of edges, and so on. Taken together, all the pose and event samples for a particular player in a particular game form a *sequence*, and are stored in such a way as to allow the controller to recover both preceding and succeeding event and pose samples from any given pose or event.

Games were conducted on the three levels selected for the competition (Fig. 6.1). Knowledge of the levels in this competition allowed the possibility of using a method that does not generalize to previously unseen levels.

Three types of games were recorded: standard games, judging games, and synthetic coverage games. Standard games were recorded in order to capture what human players do when playing a standard first person shooter death match variant of Unreal Tournament 2004. The judging games were recorded in order to overcome the potential differences in behaviour that manifest themselves when human players are also judging. Finally, because the competition levels were relatively large and because some problematic parts of the maps were visited infrequently by human players in the other data sets, synthetic data sets were collected where human players intentionally spent time navigating around areas with low data coverage.

In the first type of recorded games, two human volunteers were separated so that they could not see or hear each other outside of the game. They were joined by an equal number of native Unreal bot players. The rules were standard death match rules (first player to get 15 frags wins) and the human data was recorded and used as part of the database of human behaviour.

In the second type of recorded game, two human volunteers unfamiliar with the details of the bot were separated in two different rooms such that they could not see or hear each other outside the game. One of the authors (alternating) and two instances of an early version of the bot constituted the other participants of the games. The volunteers were asked to be vocal about what their thought process is and how their judgements were made. Recorded human traces of all participants in these games were stored in the database and used as part of the dataset for the Human Trace Controller.

In the third type of game recording, the authors participated in games designed specifically to fill in the areas of the dataset where the performance of the Human Trace Controller was found to be weaker due to lack of human data. These games involved alternating normal combat with movement localized to areas of the map where most of the stuck events were seen during testing, such as under the bridge in *Goatswood* or among the columns in the *Colosseum*.

The results presented in this chapter were obtained using a relatively modest number of human traces (Table 6.1), however, even for this dataset an efficient storage scheme had to be developed to support timely retrieval and playback. The next section discusses three data indexing schemes that were used over the course of development with properties that make it possible to scale this method to much larger data sets.

Table 6.1 Size of the recorded human game dataset used in the competition

Level	Unique players	Events	Pose samples
Colosseum	6	4,318	40,474
GoatswoodPlay	7	6,085	40,961
IceHenge	4	8,927	29,736
Total	17	19,330	1,11,171

The total dataset represents about ten hours of play time

6.4.2 Indexing Databases of Human Behaviour

In order to be able to quickly retrieve the relevant human traces, an efficient indexing scheme of the data is needed. In particular, it is necessary to quickly find all segments of the recorded games database that pass within the vicinity of the bot's current location. Throughout the course of development of the UT^2 bot, several such schemes were tested. Two of the most effective indexing schemes are described below.

6.4.2.1 Octree Based Indexing

An *octree* is a data structure routinely used in computer graphics and vision to index spatial information [28]. It is constructed by finding the geometric middle of a set of points, and subdividing the points into eight subsets defined by the three axis-aligned hyperplanes passing through the point. The process continues recursively until a termination condition, such as depth, smallest leaf dimension, or smallest number of points per leaf, is reached.

In order to index the pose data in the database of human game traces, an octree was constructed over the set of all points with the termination condition defined to be the first of (a) reaching the smallest leaf radius (set to twice the average distance moved per logic cycle), or (b) reaching the minimum number of points allowed in a leaf (set to 20).

Each point in the pose database was then labeled with an octree leaf. Given the bot's location, it is possible to traverse the octree index structure and find the smallest octree node that encloses it, and quickly retrieve the set of points within this octree node. Only part of the octree is stored in memory, while the rest is backed by SQL queries which dynamically retrieve the points needed. The Java/SQLite implementation allows the entire database to be stored in memory if it is small or to scale to other storage if it exceeds available memory.

6.4.2.2 Navigation Graph Based Indexing

As described in Sect. 6.3.4, levels in Unreal Tournament and many other similar games come with *navigation graphs*, which connect a number of named vertices distributed throughout the level with reachability edges.

It turns out that the nodes of the navigation graph can be used as labels for the points in the trace database, allowing the controller to quickly retrieve those points which are closest to a particular node. While this indexing process does take some time, it can be done efficiently by forming a KD-tree [29] over the navigation graph and iterating over the points in the database, performing a nearest neighbour search over the (relatively few) vertices in the navigation graph.

Once the nearest navpoints to the bot's location is found, the database yields the set of points in the pose database that are closest to the navpoint (belong to the navpoint's

Voronoi cell [30, 31]). Like the octree implementation above, the database-backed data structure can scale to a very large number of points because the tradeoff between memory and speed is adjustable.

Both of these schemes yield a subset of the pose database which is considered during *playback* (Sect. 6.4.4), however, the navigation graph scheme relies more on the availability of good level meta-data, while the octree-based indexing relies more on the quality of the human games database.

6.4.3 Detecting When the Bot is Stuck

Several different heuristics were employed in order to quickly determine when the bot is stuck. These were formulated after observing the behaviour of several earlier versions of the bot and characterizing the times when it was stuck.

- **SAME_NAV**—the bot keeps track of the number of logic cycles it finds itself next to the same navigation point as the previous cycle. The SAME_NAV condition is triggered when this number increases past a threshold.
- **STILL**—the bot keeps track of the number of logic cycles it finds itself within a short distance of the previous position. The STILL condition is triggered when this number increases past a threshold.
- **COLLIDING**—the Unreal Tournament game engine and the `GameBots2004` API report when the bot is colliding with level geometry, and this information is incorporated into the bot's senses by the Pogamut framework. The COLLIDING condition is set to true whenever the corresponding sense is true.
- **BUMPING**—the Unreal Tournament game engine and the `GameBots2004` API report when the bot is bumping into movable objects such as other players and this information is incorporated into the bot's senses by the Pogamut framework. The BUMPING condition is set to true whenever the corresponding sense is true.
- **OFF_GRID**—this condition is triggered when the bot finds its distance from the nearest navpoint reach a threshold.

If one of these conditions is set to true, the bot is considered stuck and the Human Trace Controller is executed as the controller for that logic cycle.

6.4.4 Retrieval and Playback of Human Behaviour Traces

When the bot finds itself stuck, if calls upon the Human Trace Controller in order to get unstuck. The controller keeps track of when it was last called, and either follows a previous path or creates a new one.

Several conditions have to be met in order for the bot to follow an existing path.

These are:

- The path was started recently enough.
- The bot has not strayed from the selected path. The "current" point along the path is within a set distance of the bot's current position.
- The path is not interrupted or terminated by recorded events such as falls, death, or large gaps between sampled positions.

If one or more of these conditions is not met, the Human Trace Controller attempts to start a new path. Otherwise, it continues along the previously started one.

The bot selects a set of relevant points from the pose database using one of the indexing techniques described in Sect. 6.4.2. Once the points are selected, one of them is picked as the starting point. Two methods for doing so that were tested during the development of the UTˆ2 bot—the random point selection and the nearest point selection. Based on the results of these tests, the competition version of the bot picks the nearest point unless this point is picked twice in a row, in which case a random point is selected.

Selecting a point from the subset specifies the sequence the agent will follow. The agent can then estimate the parameters of the movement action by using an estimate of the logic cycle length and the time stamps of the pose records.

In order to continue along the path, the agent uses the time passed since the starting point was selected in order to pick the next point from the database. The time delay between the pose samples along the stored paths in the database and the logic frames executed by the agent are both variable and different from each other. In order to address this problem, the agent interpolates between two database points in order to get an estimate for where on the path it should move to. The controller keeps an estimate (arithmetic mean) of the logic frame cycle length from its experience in order to do so.

Executing the planned motions outside of the navigation mesh poses its own challenges. If a point is not specified far enough in advance as the target of motion, the bot will appear to stall after every logic cycle. If a point too far ahead is specified, however, the bot will appear to change directions suddenly. The navigation interface provided by the GameBots API includes several different kinds of location-based movement primitives. Because the logic frame rate can be variable and latency can influence when an action actually reaches the engine and begins to execute, the `MoveAlong` primitive is used. This primitive takes two positions, $\mathbf{p_1}$ and $\mathbf{p_2}$, in order to make the movement appear smooth in the face of unpredictable latency. In effect, the `MoveAlong` action *schedules* the motion to $\mathbf{p_1}$ and then to $\mathbf{p_2}$, making the assumption that the next `MoveAlong` command will arrive when the agent is somewhere between p_1 and p_2. If the actions are interpolated carefully as part of a continuous path such as a trace of a human game, the resulting path appears smooth and purposeful.

The recorded and indexed human data, the firing conditions, and the retrieval and playback of the human trajectories together constitute the entirety of the Human Trace Controller component of the UTˆ2 bot. The next section discusses how this component was evaluated.

6.5 Results and Discussion

The performance of the human trace module was evaluated in several ways. First, it was used in the competition version of the UT^2 bot, which placed second out of five competing systems in the BotPrize 2010 competition (Sect. 6.5.1). The results of this evaluation were limited to high-level performance metrics based on human judgements and to qualitative insights extracted after the competitions based on video recordings of agent performance. The second type of evaluation was performed after the competition and was aimed at empirically comparing how the different variants of the unstuck controller contributed to the bot's ability to get unstuck and to the overall performance (Sect. 6.5.2).

6.5.1 BotPrize 2010 Competition Results

A summary of the overall bot humanness rating results is given in Table 6.2. In addition to these results, the detailed records of the judgements as well as game demo files from the games were made available after the competition.[1] This section summarizes the qualitative evaluation of the Human Trace Controller part of the bot based on these records.

The game records are provided in the Unreal demo format, which allows playback of the entire game from the perspective of a free camera (spectator mode) or from the perspective of any human player (follow mode). Several qualitative observations can be made based on reviewing these records. These observations are described below.

6.5.1.1 Data Sparseness

One problem that the bot frequently encountered on one of the levels, Colosseum, was getting stuck in the narrow hallways radiating outward from the centre of the level. Because the navigation graph does not extend into these areas, the bot will often bump into a wall if it tries to run to a node or an item after finding itself there.

Table 6.2 BotPrize 2010 results, including the average humanness rating of the native bots. The humanness rating is the percentage of judgements of the bot (by humans) that identified it as a human player

Bot	Humanness
Native UT2004 Bot	35.3982
Conscious-Robots	31.8182
UT^2	27.2727
ICE-2010	23.3333
Discordia	17.7778
w00t	9.3023

[1] http://www.botprize.org/2010.html

The Human Trace Controller can solve this problem, but only when a record exists for the particular area where the bot is stuck. Observations of the competition records from the UT^2 bot confirm this, because the first version of the bot's database used on the Colosseum level resulted in the bot being often stuck in the columns area, and this situation improved dramatically with the addition of traces specifically in the problem areas.

However, as noted below, one future direction for the controller is to use machine learning techniques in conjunction with egocentric sensors to generalize between environments that look similar. The hallways are a great example of where such an approach would be particularly useful.

6.5.1.2 Correspondence Problem

One fundamental problem faced by all designers of human-like behaviour is the *correspondence problem*, or the difference between the actions and observations available to humans and those available to artificial agents [11]. This problem can include differences in decision frequency, the kind and amount of information expressed in the sensors, differences in body morphology or capability and so on.

This problem sometimes leads to the bot's inability to reproduce a human reaction either because its observations are insufficient or because its actions are limiting.

For example, in the BotPrize competition, the human players control the bot via keystrokes and mouse movements that are processed by the game engine at a very high frequency. They receive information about the environment from a two-dimensional rendering of the three-dimensional world, which includes rich information such as texture, colour, shadow, effects of explosions, sounds, and so on. Further, they can rely on the powerful ability of the human mind to interpret and synthesize these events into higher-level stimuli, using highly parallel and complex "wet-ware", the workings of which we are only beginning to understand.

Bots, in contrast, send commands controlling their player about ten times a second, and the command repertoire does not include direct equivalents to keystroke and mouse based control of a human player. The observations of a bot include a lot of information about the environment that the human is not given immediately (such as reachability grids, navigation graphs, exact locations of items and event notifications) but also exclude important features, such as an effective way to react to sound, the ability to see the nature and extent of different environments such as lava or water, and an effective ability to deal with level geometry.

As a concrete example of this problem, in the *Goatswood* level, the bot would sometimes get stuck next to short obstacles in its path, and trying to follow a human trace would not lead to a successful navigation because in order to do so the bot would have needed to send a (well timed) jump command in addition to a `MoveAlong` action. Humans, in contrast, were able to "push through" such obstacles because such small jumps are built in to the pawn behaviour.

6.5.1.3 Environmental Features

Another class of problems has to do with special features of the environment. For example, both the *Goatswood* and the IceHenge level contain areas with water flowing through them. These special areas change the way the bot responds to commands, making movement slower in some directions and faster in others. This in turn causes human path execution to fail, because the bot tries to execute the same actions in two very different environments, as it does not operate in the same action space as the human did.

Because the water hazards are particularly difficult for the human trace controller to navigate, during the competition the UT^2 bot used a specially designed *Water Controller* when it was stuck in water areas. The controller uses a *goto* primitive along with additional hand-crafted navigation nodes in order to get out of the water as quickly and smoothly as possible. Additionally, proximity to important items located on the side of the water hazard in IceHenge can cause the bot to go to that location instead.

While this partial solution improved the performance of the bot on levels with water hazards, it does not result in a particularly human-like behaviour. For example, people occasionally used water as cover to sneak up on opponents or to escape pursuit.

Taken together, these observations provide important insights into the kinds of problems that need to be addressed when designing human-like behaviour. In Sect. 6.6, we discuss some future work that may help address these issues.

6.5.2 Comparative Evaluation

In order to evaluate the contribution of the Human Trace Controller to the overall UT^2 bot, comparisons were made between three versions of the bot, where the only difference was the controller used to get the bot unstuck. The following three controllers were used in the comparison:

- The *Null Controller* simply ignores the stuck condition and continues to whatever action would fire in the bot otherwise.
- The *Scripted Unstuck Controller* is a scripted controller designed to get the bot unstuck during evolution (Sect. 6.5.2.1).
- The *Human Trace Controller* as used during the BotPrize 2010 competition.

In all three versions the overall bot is modified from the competition version by removing level-specific special cases and the `WaterController` in order to make sure that only one controller is used to get unstuck during the comparison.

6.5.2.1 The Scripted Unstuck Controller Baseline

The Scripted Unstuck Controller is a scripted controller designed to get the bot unstuck reliably and quickly. Because it is relatively simple and does not rely on changing external human databases, the Scripted Unstuck Controller was used whenever the bot got stuck while evolving the battle controller for the UTˆ2 bot (see Chap. 5). It also provides a strong benchmark for comparisons with the performance of the Human Trace Controller.

In our experiments, the controller picks one of the following actions depending on the current state of the bot:

- If the bot is currently at location $\mathbf{x_1}$ and detected a collision at $\mathbf{x_2}$, the controller requests a move of $5 \cdot (\mathbf{x_2} - \mathbf{x_1})$.
- If the bot is currently at location $\mathbf{x_1}$ and detects a bump at location $\mathbf{x_2}$, the controller requests a move of $5 \cdot (\mathbf{x_2} - \mathbf{x_1})$.
- Otherwise,
  - with probability 0.5, the controller performs a `DodgeShotAction`, which results in a single jump in a random direction.
  - with probability 0.25, the controller requests the bot to run forward continuously until another command is selected.
  - with probability 0.25, the controller requests the bot to go to the nearest item.

6.5.2.2 Comparison Results

The results of these comparisons for the three levels are given in Figs. 6.2, 6.3 and 6.4. The three bars represent UTˆ2 bot using no unstuck controller (*NONE*), the Human Retrace Controller (*HTC*), and the scripted controller (*SCRIPTED*). Each bar is an average of thirty runs, where each run represents ten minutes of game time in a game with two Hunter bots and the modified UTˆ2 bot. Standard error is indicated by the error bar.

Overall, both the Scripted Unstuck Controller and the human trace controller perform similarly in terms of the number of cycles stuck. However, the Human Trace Controller still has two advantages: it generates qualitatively smoother paths when executed in isolation with random restarts, and the average length of a stuck segment for the Human Trace Controller is shorter than that of the scripted controller. Because the evaluations are very noisy, further evaluations are needed in order to determine statistical significance of these findings. However, the need for an unstuck controller is significant in that both controllers outperform the Null Controller.

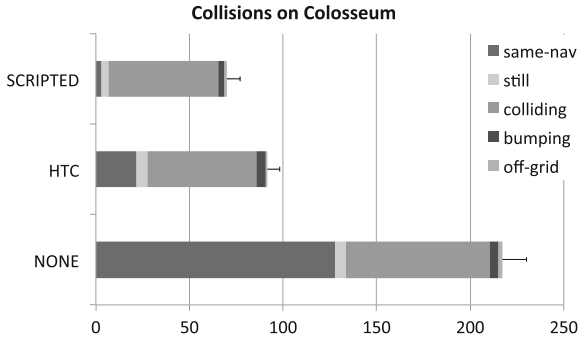

Fig. 6.2 Number of logic cycles stuck by condition on the *Colosseum* level. Averages of thirty ten-minute runs and standard error are shown

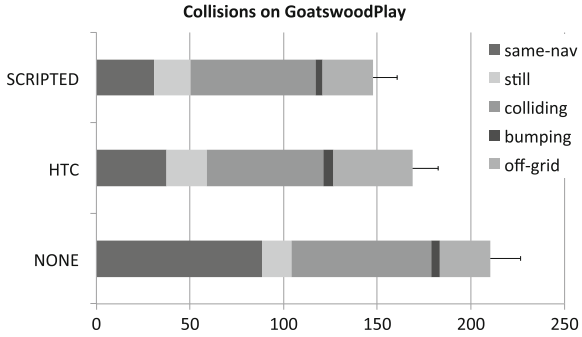

Fig. 6.3 Number of logic cycles stuck by condition on the *Goatswood* level. Averages of thirty ten-minute runs and standard error are shown

6.5.2.3 Post-competition Improvements

After the 2010 BotPrize competition, the UTˆ2 bot was modified to improve several aspects of using human traces. First, the scripted unstuck controller was integrated with the Human Trace Controller to allow the scripted controller to take over when traces are unavailable or failing to replay correctly. Second, the HTC was also used by an additional top-level UTˆ2 controller that allowed the bot to explore the level in a human-like fashion in the absence of any other goals. Third, the database was filtered to include only smooth segments, not interrupted by jumps or other artefacts. Finally, HTC playback was modified to ensure that positive path progress was made by keeping an estimate of bot speed and selecting points along the path by distance traveled according to this estimate.

The results of these comparisons are shown in Fig. 6.5. The amount of time the bot spends stuck decreases both when using Human Traces to get unstuck and when using them to explore the environment. Additionally, the human trace replay used

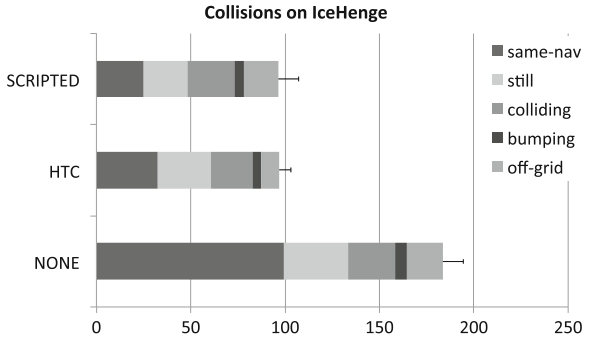

Fig. 6.4 Number of logic cycles stuck by condition on the *IceHenge* level. Averages of thirty ten-minute runs and standard error are shown

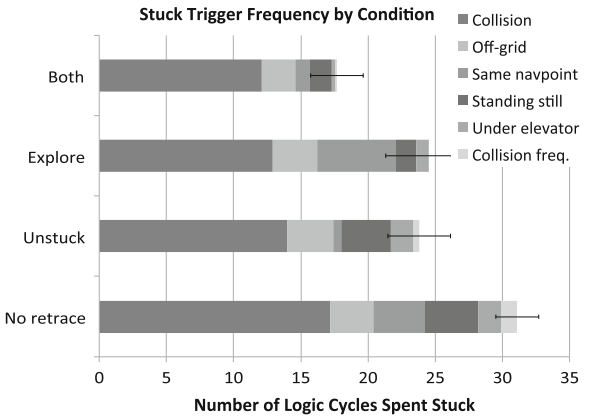

Fig. 6.5 A comparison of several versions of the 2011 version of the UTˆ2 bot. Average unstuck counts are shown when the bot is using no human traces (*No traces*), only to get unstuck (*Unstuck*), only to explore the level (*Explore*), or both to get unstuck and to explore (*Both*). Average over 10 runs with standard error bars are shown for the Osiris2 level

for exploration allows the bot to avoid traveling along the navigation graph and looks smooth and human-like when observing.

6.6 Discussion and Future Work

The Human Trace Controller presented in this chapter is a simple way to utilize recorded human behaviour to improve parts of the agent's navigation policy. However, the technique is much more generally applicable and can be extended to further

improve the navigation system, to generalize to previously unseen environments, and to support higher-level decision making such as opponent modeling.

One further area where human trace data can be useful is to improve other components of the navigation subsystem. Because the levels for Unreal Tournament 2004 are designed by hand, some areas of the levels are missing navpoints or edges in places where they would be quite useful to bots. In such places, it is possible to use the human data to induce a more complete version of the navpoint graph. Such a graph could then be used without modification by graph navigation and path planning modules.

The Human Trace Controller as described in this chapter suffers from one major weakness—it does not generalize to new environments (or even to unseen parts of existing environments). Enabling the bot to generalize to previously unseen environments would require (1) recasting of the problem in an egocentric state space and (2) selecting an appropriate machine learning technique to support better generalization. Some work has already been done towards achieving (1)-the combat module of the UT^2 bot uses a set of ego-centric and opponent-centric sensors and actions which could be reused when building a human player model. In order to achieve (2), the indexing techniques described in Sect. 6.4.2 can be naturally replaced with an instance-based machine learning algorithm such as a decision tree or a nearest-neighbour algorithm. This instance-based method can be compared and contrasted with other machine learning methods capable of compressing the data such as neural networks or probabilistic techniques. Whatever solution is used, one property that would be useful to retain from the current implementation is the ability to select what parts of what kinds of environments should be added to the database for maximum gain in the model's accuracy.

In order to address the correspondence problem, or the difference between the human and the bot's observation and action spaces, and to support the use of machine learning for effective generalization based on previously seen human behaviour, a translation scheme between human and bot observations and actions needs to be developed. This could be done automatically by learning the bot actions and parameters that most accurately recreate small sections of human behaviour, and using those actions as primitives when replaying new traces or outputs of a learned human behaviour model.

In addition to using a model of human behaviour to mimic it as part of the agent's policy, it is possible to use this information in other ways. For example, it may be possible to use the human behaviour database to design a "humanness fitness function" for evolving human-like control policies. Such a function can be used for example as part of the multi-objective neuroevolution technique discussed in Chap. 5 to compare the behaviour generated by a candidate bot with that available in the human database. As another promising future use of human trace data, the bot can use its model of human behaviour to predict and reacquire a human opponent it is chasing if it loses track of him or her. Such behaviour can be seen as purposeful and cognitively complex, and thus very human.

6.7 Conclusion

The Human Trace Controller component of the UTˆ2 bot takes a step towards building human-like behaviour in a complex virtual environment by directly replaying segments of recorded human behaviour.

Evaluation of the resulting controller as part of the BotPrize competition and via comparative experiments suggests that the replay of human traces is an effective way to address the navigation problems faced by the UTˆ2 bot. Additionally, human traces can be used for exploration of the level in the absence of other goals. The resulting behaviour appears smooth and human-like on observation, while also allowing the bot to navigate the environment with a minimal number of failures.

Finally, the work demonstrates the feasibility of using large databases of human behaviour to support online decision making. The approach can scale and improve with experience gained naturally from domain experts; it is applicable to the explicit goal of building human-like behaviour; and it supports imitation, a primitive that is ubiquitous in examples of intelligent behaviour in nature.

Acknowledgments The authors would like to thank Philip Hingston for organizing the BotPrize competitions and 2K Australia for sponsoring it. The authors would also like to thank students in the Freshman Research Initiative's Computational Intelligence in Game Design stream and members of the Neural Networks Research Group at the University of Texas and to Christopher Tanguay and Peter Djeu for participating in recordings of human game traces and for volunteering to critique and evaluate versions of UTˆ2. This research was supported in part by the NSF under grants DBI-0939454 and IIS-0915038 and by the Texas Higher Education Coordinating Board grant 003658-0036-2007.

References

1. Rizzolatti, G., Fogassi, L., Gallese, V.: Neurophysiological mechanisms underlying the understanding and imitation of action. Nat. Rev. Neurosci. **2**(9), 661–670 (2001)
2. Gergely, G., Bekkering, H., Király, I.: Developmental psychology: rational imitation in preverbal infants. Nature **415**(6873), 755 (2002)
3. Nicolescu, M., Matarić, M.J.: Task learning through imitation and human-robot interaction. In: Dautenhahn, K., Nehaniv, C. (eds.) Models and Mechanisms of Imitation and Social Learning in Robots, Humans and Animals: Behavioural, Social and Communicative Dimensions, Cambridge University Press, Cambridge (2005)
4. Hingston, P.: A Turing Test for computer game bots. IEEE Trans. Comput. Intell. AI Games **1**(3), 169–186 (2009)
5. Laird, J.E., van Lent, M.: Interactive computer games: human-level AI's killer application. AI Mag. **22**(2), 15–25 (2001)
6. Aha, D.W., Molineaux, M.: Integrating learning in interactive gaming simulators. In: Proceedings of the AAAI'04 Workshop on Challenges of Game AI, AAAI Press (2004)
7. Bowling, M., Fürnkranz, J., Graepel, T., Musick, R.: Machine learning and games. Mach. Learn. **63**, 211–215 (2006)
8. Molineaux, M., Aha, D.W.: TIELT: a testbed for gaming environments. In: Proceedings of the Twentieth National Conference on Artificial Intelligence (Intelligent Systems Demonstrations), AAAI Press (2005)

9. Schaal, S.: Learning from demonstration. In: Advances in Neural Information and Processing Systems, pp. 1040–1046. Citeseer (1997)
10. Atkeson, C., Schaal, S.: Robot Learning from Demonstration. In: Proceedings of the Fourteenth International Conference on Machine Learning, ICML'97, pp. 12–20. Citeseer (1997)
11. Argall, B.D., Chernova, S., Veloso, M., Browning, B.: A survey of robot learning from demonstration. Robot. Auton. Syst. **57**(5), 469–483 (2009)
12. Thurau, C., Bauckhage, C., Sagerer, G.: Synthesizing movements for computer game characters. Lect. Notes Comput. Sci. **3175**, 179–186 (2004)
13. Stolle, M., Atkeson, C.G.: Policies Based on Trajectory Libraries. In: Proceedings of the International Conference on Robotics and Automation (ICRA 2006) (2006)
14. Stolle, M., Tappeiner, H., Chestnutt, J., Atkeson, C.G.: Transfer of policies based on trajectory libraries. In: Proceedings of the International Conference on Intelligent Robots and Systems (2007)
15. Hladky, S., Bulitko, V.: An evaluation of models for predicting opponent positions in first-person shooter video games. In: Proceedings of the IEEE 2008 Symposium on Computational Intelligence and Games (CIG'08). Perth, Australia (2008)
16. Orkin, J.D., Roy, D.: Automatic learning and generation of social behavior from collective human gameplay. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09), vol. 1, pp. 385–392 (2009)
17. Cardamone, L., Loiacono, D., Lanzi, P.L.: Learning drivers for TORCS through imitation using supervised methods. In: Proceedings of the IEEE 2009 Symposium on Computational Intelligence and Games (CIG'09) (2009)
18. Aler, R., Valls, J.M., Camacho, D., Lopez, A.: Programming robosoccer agents by modeling human behavior. Expert Syst. Appl. **36**(2), Part 1, 1850–1859 (2009)
19. Pao, H.-K., Chen, K.-T., Chang, H.-C.: Game bot detection via avatar trajectory analysis. IEEE Trans. Comput. Intell. AI Games **2**(3), 162–175 (2010)
20. Sukthankar, G., Sycara, K.: Simultaneous team assignment and behavior recognition from spatio-temporal agent traces. In: Proceedings of Twenty-First National Conference on Artificial Intelligence (AAAI-06) (2006)
21. Murakami, S., Sato, T., Kojima, A., Hirono, D., Kusumoto, N., Thawonmas, R.: Outline of ICE-CEC 2011 and its mechanism for learning FPS tactics. In: Extended Abstract for the Human-like Bot Workshop at the IEEE Congress on Evolutionary Computation (CEC 2011) (2011)
22. Epic Games, Inc. and Digital Extremes, Inc.: Unreal Tournament 2004. Atari, Inc., Mar 2004
23. Kaminka, G.A., Veloso, M.M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A.N., Scholer, A., Tejada, S.: GameBots: a flexible test bed for multiagent team research. Commun. ACM **45**(1), 43–45 (2002)
24. Gemrot, J., Kadlec, R., Bída, M., Burkert, O., Píbil, R., Havlíček, J., Zemčák, L., Šimlovič, J., Vansa, R., Štolba, M. et al.: Pogamut 3 can assist developers in building AI (not only) for their videogame agents. In: Agents for Games and Simulations, pp. 1–15 (2009)
25. Hingston, P.: A new design for a Turing Test for bots. In: IEEE Transactions on Computational Intelligence and AI in Games (2010)
26. Floyd, R.: Algorithm 97: shortest path. Commun. ACM **5**(6), 345 (1962)
27. "SQLite". http://www.sqlite.org/
28. Jackins, C., Tanimoto, S.: Oct-trees and their use in representing three-dimensional objects. Comput. Graph. Image Process. **14**(3), 249–270 (1980)
29. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
30. Dirichlet, G.L.: Über die Reduktion der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen. J. für die Reine und Angewandte Mathematik **40**, 209–227 (1850)
31. Voronoi, G.: Nouvelles applications des paramètres continus à la théorie des formes quadratiques. J. für die Reine und Angewandte Mathematik **133**, 97–178 (1907)