Clustering-Based Real-Time Player Modeling

Jason M. Bindewald, Gilbert L. Peterson, Michael E. Miller

***Abstract*—Being able to imitate individual players in a game can benefit game development by providing a means to create a variety of autonomous agents and in understanding how players play a game. This paper presents a clustering and locally weighted regression method for modeling and imitating individual players. A generic player cluster model is first generated that is then updated in real-time to capture an individual's game-play tendencies. The model can then be used to play the game as a specific individual or for analysis to identify how different players react to separate aspects of game states. The method is demonstrated on a tablet-based trajectory generation game called *Space Navigator*.**

***Index Terms*—Player modeling, human-like behavior, human-computer interaction, clustering.**

I. Introduction

Automating game-play in a human-like manner is the goal of a large area of intelligent gaming research, with applications from trying to succeed in a gaming version of the "Turing Test" [1] to creating human-like game avatars [2]. When we move from playing a game like a generic human to performing like a specific human, the dynamics of the problem change [3]. Generalized datasets can no longer be lumped into large groups of past game-play. In complex dynamic environments it can be difficult to differentiate individual players, because the insights exploited in imitating "human-like" game-play can become less useful in imitating the idiosyncrasies that differentiate specific individuals' game-play.

There are several benefits of individual player imitation. Individual player imitation provides insights into modeling more believable opponents [2]. Better understanding what sets individual players apart from others, allows a game designer to build more robust game personalization [4]. Learning the aspects of a game state that set individual players apart, allows for better understanding of how to adjust games according to skill level [5].

This paper contributes a real-time individual player modeling system that enables an automated agent to perform response actions in a game that are similar to those that an individual player would have performed in similar situations. This work improves upon past player modeling efforts, such as [6], emphasizing three things. First, the player modeling system automatically updates in real-time rather than requiring off-line computation to adjust to changing game-play over time. Secondly, the system takes advantage of insights gleaned from past game-play clustering to gain a statistically significant differentiation between players in a relatively short amount of game-play (five, five-minute games). Additionally, the clustering-based player modeling method allows the practitioner to glean insights into what differentiates the game-play of individual players.

This paper proceeds as follows. Section II reviews related work in the fields of player modeling and learning from past game-play. Section III presents a generic player model methodology and then uses the generic player model as a base for implementing a real-time individual player modeling system. This model is then demonstrated using the *Space Navigator* trajectory generation game as a test-bed. Section V gives experimental results showing the individual player modeling system's improvements over the generic modeling method. Section VI summarizes the findings presented and proposes potential future work.

II. Related Work

Player modeling research informs the methodology to create trajectories similar to those of an individual player. Methods involving learning from past experiences provide insight into how to generate new trajectories from past game-play instances. This section describes some over-arching areas of past work that influence the current research.

A. Player modeling

Three taxonomies for player modeling exist, each providing a different way of organizing the field. Each model is presented and explained. Interspersed with the model descriptions are examples of how the model would classify different player modeling research efforts.

1) Yannakakis Model: In the Yannakakis player model taxonomy [7], four input types are used to build player models of two types which provide four types of outputs. Inputs to a player model fall into four categories: game-play, objective, game context, and player profile. Game-play data (often called behavioral data) captures actions that a player takes in the given game environment. Objective data includes a player's measurable physiological responses to the game environment. Game context data denotes a representation of the real-time state of the game. A player profile is a static representation of the player outside of the context of the game (e.g. personality type). These four inputs are used in some combination to create a player model.

The resulting player model is either a model-based (top down) or model-free (bottom-up) player model. In a model-based player model the model is built on some form of a theoretical framework where player groupings are pre-defined according to some set of features. Examples of model-based player models include supervised neural networks [8], trait theory to pre-determine player types [9], strategy groupings

This work was supported in part through the Air Force Office of Scientific Research, Computational Cognition & Robust Decision Making Program (FA9550), James Lawton Program Manager.

based on game design features [10], and association rule mining to find player experience/activity relationships [11]. In model-free player models the goal is to find player types that naturally arise from the collected data. Clustering is a common method to find player types, some examples of which include hierarchical clustering [12], *k*-means [13], [14], neural networks [8], and self-organizing maps [12].

When utilized, player models produce an output when a given state or response is presented to it depending on its intended purpose. The outputs from player models can encompass scalar values, class membership, ordinal data (rankings), or no output (such as when learning a player model for clustering purposes).

2) Smith Model: The Smith player model taxonomy [15] classifies player models across four independent facets: domain, purpose, scope, and source. The domain of a player model is either game actions (similar to Yannakakis's game-play data input type) or human reactions (similar to the objective and player profile input types). The second facet, purpose, describes the end for which the player model is implemented: generative player models aim to generate actual data in the environment in place of a human or computer player, while descriptive player models aim to convey information about a player to a human. The scope of the player model describes the scope of players the model represents: individual (one), class (a group of more than one), universal (all), and hypothetical (some theoretical player or set of players that doesn't fit in the other categories). The source of a player model can be one of four categories: induced - objective measures of actions in a game; interpreted - subjective mappings of actions to a predefined category; analytic - theoretical mappings based on the game's design; and synthetic - based on some non-measurable influence outside of the game context (e.g. hunches).

For descriptive purposes, each player model is given a type for each of the four facets. For example, the player model created in [16] for race track generation models individual player tendencies and preferences (Individual), objectively measures actions in the game (Induced), creates tracks in the actual environment (Generative), and arises from game-play data (Game Action).

3) Bakkes Model: Bakkes *et al* [17] create a player behavior model that classifies player models that involve game-play data inputs in the Yannakakis model or fall in the game action domain in the Smith model into four categories:

- Player behavior models based on *player actions* map states encountered in the game to player actions. A good example of this type of model is the player models associated with research on poker player modeling [18].
- Player behavior models based on *player tactics* take multiple actions and/or the actions of multiple players into account to model different players, an example being the tactical offensive football play models created in [19].
- Player behavior models based on *player strategies* involve the use of different tactics in succession, and tend to account for "entire game" time frames. Examples of strategy level player behavior modeling exist in real-time strategy game research, such as systems designed to play *StarCraft* [20], [21].
- *Player profiling* involves the use of player behavior in games to establish psychological or sociological player profiles. Research efforts measuring entertainment in games [22], [23] tend to allow for this type of behavior modeling.

B. Learning from Previous Game-Play

Two areas of research that rely on past experience to inform future automated game-play include: Case-Based Reasoning (CBR) and Learning from Demonstration (LfD). Both CBR and LfD train an automated system to generate responses based on observations within an environment. In CBR, a "case-base" maintains a set of observed environment states and their associated responses (cases) [24]. When a new state is received by the CBR a previous case is retrieved, adapted to the current state, and a new response is fashioned. The new state and its associated response is then either added to the case-base or thrown away according to observed feedback. In LfD, a teacher demonstrates a skill that it would like the the automated system to learn [25]. The learner attempts to derive a policy based on the demonstration, and then attempts to execute the derived policy. The policy is then evaluated and updated with feedback in the environment.

The nearest neighbor principle maintains that instances of a problem that are a shorter distance apart more closely resemble each other than do instances that are a further distance apart [26]. This concept is applied in many locally weighted learning algorithms that learn how to perform regression or classification tasks by comparing an incoming instance to that of its nearest neighbors [27]. The nearest neighbor principle is used to find relevant past experiences in LfD tasks such as a robot intercepting a ball [28], CBR tasks such as a RoboCup soccer-playing agent [29], or tasks integrating both LfD and CBR such as in real time strategy games [30]. When searching through large databases of past experiences approximate nearest neighbors searches, such as Fast Library for Approximate Nearest Neighbors (FLANN [31]), have proven useful in approximating nearest neighbor searches while maintaining lower order computation times in large search spaces.

III. Methodology

The real-time player modeling paradigm we present here improves on previous work in three ways. As the name suggests, the player model updates in real-time to adapt to changing player habits. Additionally, the paradigm pulls insight by clustering past game-play, differentiating between players quickly. Then, the resulting player models allow the practitioner to investigate specific individual game-play tendencies further. Figure 1 illustrates our real-time player modeling paradigm. This real-time player modeler creates responses to provided game states that are similar to those that an individual player would have given in response to similar states. This section explains the three main tasks of the real-time player modeler: creating a generic player model, generating similar response trajectories, real-time updating of the player model with individual player game-play.

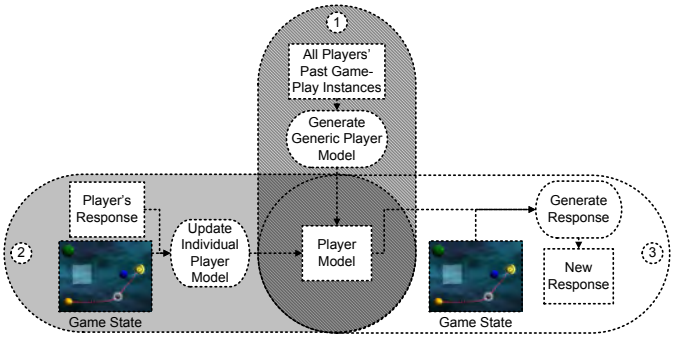

Fig. 1. A real-time updating individual player modeling paradigm.

A. Create Generic Player Model

This section outlines the generic player model creation process, shown in shaded area 1 of Figure 1. Clustering the past game-play instances according both by state and by response reveals general player tendencies. With information gained from clustering, pruning outlier instances creates a universally representative example game-play dataset. This dataset then forms the groundwork for a generic player model that maps state clusters to response clusters.

1) State and response clustering: Similar to the method used in [6], Ward agglomerative clustering [32] provides a baseline for player modeling. Clustering reduces the state-response pairs into a set of representative clusters, reducing the potential representation size of a player model. This method was proven effective for clustering in a trajectory creation game environment in [6], [33]. Agglomerative clustering starts with a set of game-play instances that contain a state and its associated response and assigns each instance to a state cluster and a response cluster. The number of clusters will depend on the environment and size of the underlying dataset. The mapping from a state cluster to a response cluster for state-response instance demonstrates a proclivity for a player to react with a given maneuver in a specific type of game situation. By determining the frequency of state cluster to response cluster mappings, common situational responses and outlier actions emerge.

2) Cluster outlier pruning: The frequency of state cluster to response cluster mappings reveal common and outlier situational responses, providing the basis for two types of pruning that generalize the generic player profile by removing infrequent interactions. First, instance frequency within clusters helps in pruning outliers from the set of all game instances. If a given state has only been seen in one instance by one player, that state is unlikely to provide much benefit in predicting future responses. Similarly, a response given by only one player in one instance is unlikely to be replicated in future player responses.

Clusters with outlier responses are removed first by removing all instances assigned to the least populated response clusters. The cutoff threshold for determining which instances to remove could be either a minimum response cluster size or just a percentage of response clusters to remove. For example, due to the distribution of cluster sizes in the *Space Navigator* database we removed instances falling in the bottom 25% of all response clusters according to cluster size. Setting cutoff thresholds relies on knowledge of the environment and underlying dataset distribution.

Next, outlier state clusters are removed in two ways. First, instances that fall in the bottom 25% of all state clusters according to cluster size are removed, removing all clusters that are rare in general. However, removing states not seen by many *different* players is also important. In addition to removal based on sheer cluster size, pruning also removes instances falling into a state cluster encountered by a minimal subset of players. This removes a subset of clusters not removed previously: state clusters with many instances reached by an extremely small subset of players. It is important to note that although the generic player model will be built on the pruned game-play database, the original state and response clusters are used. In this way, individual player modeling can still capture outlier states or responses individual players encounter that most players do not encounter.

3) Player model creation algorithm: Whereas [6] used an offline game-play database creation method, the current research uses a faster online system to model players. Algorithm 1 generates a generic player model that determines the likelihood each state cluster is to map to each response cluster.

Algorithm 1 Generic player model creation algorithm.

1: **inputs:**
2: $x =$ the number of state clusters
3: $y =$ the number of response clusters
4: $\mathbf{M} = \{\langle S_1, R_1\rangle, \langle S_1, R_2\rangle, \cdots, \langle S_x, R_y\rangle\}$, all state-response cluster mappings
5: $\mathbf{C} = \boldsymbol{O}_{x,y}$ ▷ $x \times y$ zero matrix
6: $\mathbf{P} = \boldsymbol{O}_{x,y}$ ▷ $x \times y$ zero matrix
7: **for** $i = 1 \rightarrow x$ **do**
8: **for** $j = 1 \rightarrow y$ **do**
9: $c_{i,j} =$ the number of instances assigned to cluster mapping $\langle S_i, R_j\rangle$
10: **end for**
11: **end for**
12: **for** $i = 1 \rightarrow x$ **do**
13: **for** $j = 1 \rightarrow y$ **do**
14: $\mathbf{P}_{i,j} = c_{i,j} / \sum_{k=1}^{y} c_{i,k}$
15: **end for**
16: **end for**
17: **return P**

The generic player model creation algorithm takes in the number of state and response clusters (x and y respectively), and the set of all state-response cluster mappings (**M**). Line 5 creates a matrix of counters (**C**) to help determine how many instances belong to each cluster in **M**. Line 5 creates an empty player model (**P**) that will hold likelihoods for each state-response cluster mapping in **M**. Both **C** and **P** are initialized to the $x \times y$ zero matrix. The loops beginning in Lines 7 and 8 process each of the state-response cluster mappings. For each cluster mapping, the number of instances provided by players that belong to cluster pairing $\langle S_i, R_j\rangle$ is recorded.

The loop beginning in Line 12 creates the player model **P** from the counter matrix **C**. The model contains a matrix of likelihoods that a given instance provided by a generic player chosen at random from the game-play database will belong to the indicated state and response cluster mappings. The likelihoods are determined by normalizing across the rows of **C**. For each state cluster, the count for each response cluster is divided by the total number of instances assigned to the state cluster. The matrix of likelihoods is returned as the generic player model **P**. This generic player model forms the baseline for individual player model creation. To model individual player gameplay habits, the individual player modeling techniques in the next section update the generic player model through observed game-play data.

B. Update individual player Model

For real-time individual player modeling, this research updates the generic player model created in Algorithm 1 as an individual plays the game. Over time, the updates shape a player model that represents an individual player's game-play tendencies, as illustrated in shaded area 2 of Figure 1. The individual player update process involves an algorithm to learn a player model with individual player tendencies over time. In order to train an individual player model quickly, the information gained from each state-response instance leads to an update of the state-response cluster scores.

1) individual player model real-time update algorithm: Algorithm 2 demonstrates the real-time updates that take place to learn an individual player's tendencies. The algorithm begins with the generic player model **P**. Once a player submits a response in the game environment, the current game state and the response are submitted. The algorithm finds the closest state (S_{close}) and response (R_{close}) clusters to the state and response passed in by the player. The player model is updated at the intersection of S_{close} and R_{close} by δ_{close}. Then the player model is normalized across all the R values for S_{close} so that the values sum to 1.

Algorithm 2 Individual player model real-time update algorithm.

1: **inputs:**
2: $\mathbf{P}$ = an $x \times y$ generic player model created by Algorithm 1
3: $\langle s_{in}, r_{in} \rangle$ = a player-provided state-response pair
4: $\mathbf{M} = \{\langle S_1, R_1 \rangle, \langle S_1, R_2 \rangle, \cdots, \langle S_x, R_y \rangle\}$, all state-response cluster mappings
5: S_{close} = the closest state cluster to state s_{in}
6: $\delta_{close} = q \cdot (\delta_{cp} + \delta_{cmv} + \delta_{pma})$, S_{close}'s update increment weight
7: R_{close} = the closest response cluster to response r_{in}
8: $\mathbf{P}(S_{close}, R_{close}) = \mathbf{P}(S_{close}, R_{close}) + \delta_{close}$
9: **for** $\mathbf{P}(S_{close}, i)$ where $i = 1 \rightarrow y$ **do**
10: $\quad \mathbf{P}(S_{close}, i) = \mathbf{P}(S_{close}, i) \,/\, (1 + \delta_{close})$
11: **end for**

2) Update increment weighting: The player model update algorithm is useful in modeling player behavior, but there are certain states from which more can be gleaned than others. Weighting the increment values for a given state-trajectory pair can be useful in quickly learning idiosyncrasies that set a player apart from the generic player. Specifically, knowing which state clusters contain the most information for future player modeling is useful. Traits gleaned from the clustered data provide ways to help determine which state clusters should create larger learning increments, and which states provide minimal information to extend beyond the generic player game-play model. Three binary traits contribute to the update increment, δ, in Line 6 of Algorithm 2. The three traits calculated to help weight δ include cluster population, cluster mapping variance, and previous modeling utility.

Cluster Population: When attempting to learn game-play habits quickly, knowing the expected responses of player to common game states is important. Weighting δ according to the size of a state cluster in comparison to that of the other state clusters across the entire game-play dataset emphasizes increased learning from common states for an individual player model. States that fall into larger clusters can provide better information for quickly learning how to differentiate individual player game-play habits. To calculate the cluster population trait, all state cluster sizes are calculated and a population threshold is selected. Any state cluster with a population above the population threshold is given a cluster population trait weight of $\delta_{cp} = 1$ and all other state clusters receive a weight of $\delta_{cp} = 0$.

Cluster Mapping Variance: When mapping state clusters to response clusters, some state clusters will consistently map to a specific response cluster across all players. Other state clusters will consistently map to several response clusters across all players. Very little about a player's game-play tendencies is learned from these two types of state clusters. However, state clusters that map to relatively few clusters per player (intra-player cluster variance), while still varying largely across all players (inter-player cluster variance) can help quickly differentiate players. The state cluster mapping variance ratio is the total number of response clusters to which a state cluster maps across all players divided by the number of response clusters to which the average player maps, essentially the ratio of inter-player cluster variance to the intra-player cluster variance. The cluster mapping variance trait weight, δ_{cmv}, is set according to a cluster variance ratio threshold. All state clusters with a variance ratio above the threshold receive a weight of $\delta_{cmv} = 1$ and all others receive a weight of $\delta_{cmv} = 0$

Previous Modeling Utility: The last trait involves running Algorithm 2 on the existing game-play data. Running the individual player update model on previous game-play data provides insights into how the model works in the actual game environment. This trait requires the use of a system that automatically generates responses to presented states using a player model is already established.

First, Algorithm 2 runs with $\delta = 1$ for all state clusters, training the player model on some subset of a player's game-play data (training set). Then it iterates through the remaining game-play instances (test set) and generate a response to each presented state, using both the individual player model and the

generic player model. This repeats for each individual player in the game-play dataset. For each test set state, we then determine which response was most similar to the player's actual response. Each time the individual player model is closer than the generic player model to the actual player response, tally a 'win' for the given state cluster and a 'loss' otherwise. The ratio of wins to losses for each state cluster makes up the previous modeling utility trait. The previous modeling utility trait weight, δ_{pma}, is set according to a previous modeling utility threshold. All state clusters with a previous modeling utility above the threshold receive a weight of $\delta_{pma} = 1$ and all others receive a weight of $\delta_{pma} = 0$.

Calculating δ: When Algorithm 2 runs, δ is set to the sum of all trait weights for the given state cluster multiplied by some value q which is an experimental update increment set by the player. Line 6 shows how δ is calculated as a sum of the previously discussed trait weights.

IV. Case Study: *Space Navigator*

This section demonstrates the player modeling paradigm, focusing specifically on the response generation section of the player modeling paradigm (unshaded area 3 of Figure 1), with a specific application in generating trajectory responses in *Space Navigator*. First, the application environment is introduced along with the outline of an initial data capture experiment within it. Then, solutions are presented to three challenges specific to the game environment: developing a state representation, comparing disparate trajectories, and finding a trajectory distance measure that is meaningful to humans. These solutions are then used to develop a trajectory response generation algorithm that utilizes a player model to generate trajectories similar to those that would have been provided by an individual player in the same situation.

A. Application Environment

This research uses a trajectory-based tablet game, *Space Navigator* [6], [34]. In designing the test-bed, the end product needs to allow for games that could begin with complete manual control, allowing for the capture of human game-play, and then support an automated system that controls a portion of the overall human task within the game environment. It also must be easily understood to facilitate data gathering from as wide a range of players and skill levels as possible.

1) Space Navigator: Space Navigator is a tablet computer-based trajectory generation game similar to games such as *Flight Control* [35], *Harbor Master* [36], and *Contrails* [33]. Figure 2 shows a screen capture from the game and identifies several key objects within the game. Spaceships appear at set intervals from the screen edges. The player directs each spaceship to its destination planet (designated by similar color) by drawing a line on the game screen using his or her finger. The spaceship then follows the entire drawn trajectory.

Points accumulate when a ship encounters its destination planet or one of a number of small bonuses that randomly appear throughout the play area. Points decrement when spaceships collide, and each spaceship involved in the collision is lost. Points are also lost when a spaceship traverses one of several "no-fly zones" (NFZs) that move throughout the play area at a set time interval. For every second a spaceship traverses a NFZ, the player loses points. The game ends after five minutes.

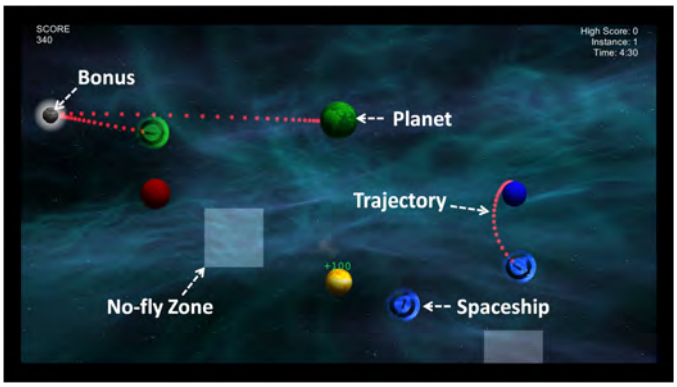

Fig. 2. Screen capture from a game of *Space Navigator*, pointing out spaceships, planets, trajectories, bonuses and no-fly zones.

Space Navigator contains only one player action (drawing trajectories), while containing enough dynamism that an automation can not generate a "best" input. This is assured through the moving NFZs, random location of appearing spaceships, and random appearances of bonuses. The single player input allows focus on the trajectories drawn as the only possible response to a given state.

2) Initial Data Capture Experiment: An initial experiment captured a corpus of game-play data for further comparison and benchmarking of human *Space Navigator* game-play. Player data collection used a set of Samsung ATIV Smart PC tablet computers running the Windows 8 operating system. Data was collected from 32 participants playing 16 five-minute instances of *Space Navigator*. The instances represented four difficulty combinations, with two specific settings changing: (1) the number of NFZs and (2) the rate at which new ships appear.

The environment captures data associated with the game state whenever the player draws a trajectory. The data includes: time stamp, current score, ship spawn rate, NFZ move rate, bonus spawn interval, bonus info (number of bonuses, location, and lifespan of each), NFZ info (number of NFZs, location, and lifespan of each), other ship info (number of other ships, ship ID number, location, orientation, trajectory points, and lifespan of each), destination planet location, selected ship info (current ship's location, ship ID number, orientation, lifespan, and time to draw the trajectory), and selected ship's trajectory points. The final collected dataset consists of 63,030 instances, with each player's dataset including an average of 1,950 state-trajectory instances.

B. State Representation

Space Navigator states are dynamic both in number and location of objects. Bonuses and spaceships appear and disappear throughout the game and spaceships and NFZs move throughout the scene over time. The resulting infinite number of configurations makes individual state identification difficult.

To shrink the large feature vectors obtained in the data capture, the state representation contains only the elements of a state that directly affect a player's score (other ships, bonuses, and NFZs) scaled to a uniform size along with a feature indicating the relative length of the spaceship's original distance from its destination. Algorithm 3 describes the state-space feature vector creation process.

Algorithm 3 State-space feature vector creation algorithm.

1: **input:**
2: $L =$ the straight-line trajectory from the spaceship to its destination planet.

3: **initialize:**
4: $\eta \in [0.0 \cdots 1.0) =$ a weighting variable
5: $s =$ an empty array of length 19
6: $zoneCount = 1$

7: Translate all objects equally s.t. the selected spaceship is located at the origin.
8: Rotate all objects in state-space s.t. L lies along the X-axis.
9: Scale state-space s.t. L lies along the line segment from $(0, 0)$ to $(1, 0)$.
10: **for** each object type $\vartheta \in (OtherShip, Bonus, NFZ)$ **do**
11: **for** each zone $z = 1 \rightarrow 6$ **do**
12: $zoneCount = zoneCount + 1$
13: **for** each object o of type ϑ in zone z **do**
14: $d_o =$ the shortest distance of o from L
15: $w_o = e^{-(\eta \cdot d_o)^2}$ ▷ Gaussian weight function
16: $s\,[zoneCount] = s\,[zoneCount] + w_o$
17: **end for**
18: **end for**
19: **end for**
20: $s\,[19] =$ the non-transformed straight-line trajectory length
21: Normalize values of s between $[0, 1]$
22: **return** s

The algorithm first transforms the state-space features to a straight-line trajectory frame in Line 2. Line 7 translates the state space so the selected ship is at the origin. Line 8 rotates all the objects in state-space so that the straight-line trajectory between the ship and the destination planet is located on the X-axis. Then, Line 9 scales the state-space such that all straight-line trajectories are of equal length. These transformations allow disparate trajectories to be compared in the state-space.

The loop beginning on Line 10 accounts for the different element types and the loop beginning on Line 11 divides the state-space into six zones as shown in Figure 3. The first dividing line creates two zones along the straight-line trajectory. The second and third dividing lines occur perpendicular to the straight-line trajectory at the location of the spaceship and destination planet respectively. This effectively divides the state-space into three zones with relation to the spaceship's straight-line path: behind the spaceship, along the path, and beyond the destination.

To compare disparate numbers of objects, the loop beginning in Line 13 uses a method similar to that used in [29]. Each zone collects a weight score (s) for each object within the zone. This weight score is calculated using a Gaussian weighting function based on the minimum distance an object is from the straight-line trajectory. For objects beyond the destination planet or behind the spaceship, the minimum distance will not be perpendicular to the straight-line trajectory.

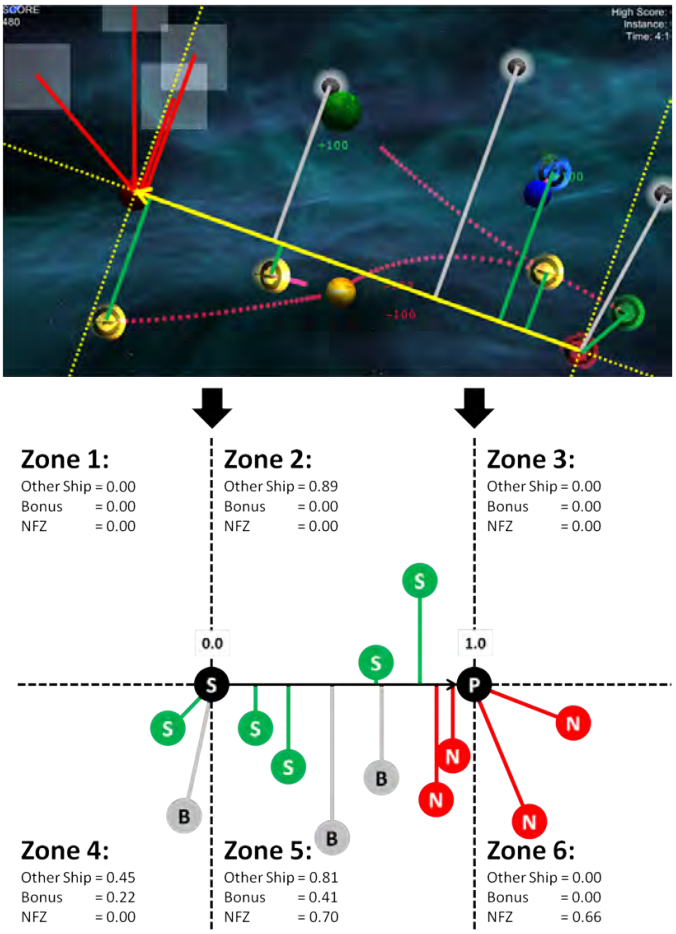

Fig. 3. The six zones surrounding the straight line trajectory in a *Space Navigator* state representation and the state representation calculated with Algorithm 3.

Figure 3 shows the transformation of the state into a feature vector using Algorithm 3. The state-space is transformed in relation to the straight-line trajectory, and a value is assigned to each "entity type + zone" pair accordingly. For example, Zone 1 has a bonus value of 0.11 and other ship and NFZ values of 0.00, since it only contains one bonus. The weighting function is evident in the fact that closer entities (Zone 6 - NFZ) have a higher score than entities that are farther away from the straight-line trajectory (Zone 1 - bonus).

Lastly, the straight-line trajectory distance is captured. This accounts for the different tactics used when ships are at different distances from their destination. Ships that are very close to their destination are more likely to result in responses close to a straight-line trajectory, while those that must traverse nearly the entire screen will see a wider variance from the straight-line trajectory. The resulting state representation values are normalized between zero and one.

C. Trajectory Comparison

Trajectory generation requires a method to compare disparate trajectories. This is crucial to being able to determine the similarity or dissimilarity of two response trajectories [37]. However, trajectories generated within *Space Navigator* can vary in composition, containing differing numbers of points and point locations. This section describes how the trajectory generator permits trajectory comparison. Trajectory comparison requires both re-sampling and transformation. Trajectory re-sampling, based on linear interpolation [38], ensures all trajectories consist of the same number of points. The trajectory generator can then compare trajectories using a simpler distance measure.

Algorithm 4 performs trajectory re-sampling. The algorithm begins by keeping the same start and end points, then iterates through until the re-sampled trajectory is filled. The process first finds, in Line 10, the proportional relative position (p_m) of a point. The proportional relative position indicates where the i-th point would have fallen in the original trajectory and may fall somewhere between two points. Calculated in Line 16, the proportional distance (d_m) that p_m falls from the previous point in the old trajectory (p_0) is the relative distance that the i-th re-sampled point falls from the previous point. To compare trajectories, the target number of points is set to 50 for re-sampling all the trajectories (i.e. n_{avg}). Fifty is approximately the mean number of points found in all the trajectories during the initial data capture.

Algorithm 4 Trajectory re-sampling algorithm.

1: **inputs:**
2: n_{old} = Number of points raw trajectory we are re-sampling contains
3: n_r = Number of points to which we are re-sampling
4: t_{old} = Array of (x, y) points representing the raw trajectory we are re-sampling

5: **initialize:**
6: t_r = Empty array of (x, y) points of length n_r to hold the re-sampled trajectory

7: $t_r[1] = t_{old}[1]$
8: $t_r[n_r] = t_{old}[n_{old}]$
9: **for** $i = 2 \rightarrow n_r - 1$ **do**
10: $\quad p_m = \left(\frac{n_{old}}{n_r}\right) \cdot i$
11: $\quad p_0 = \lfloor p_m \rfloor$ ▷ The position directly before p_m
12: $\quad p_1 = \lceil p_m \rceil$ ▷ The position directly after p_m
13: $\quad$ **if** $p_m = p_0$ **then**
14: $\quad\quad t_r[i] = t_{old}[p_m]$
15: $\quad$ **else**
16: $\quad\quad d_m = p_m - p_0$
17: $\quad\quad (x_0, y_0) = t_{old}[p_0]$
18: $\quad\quad (x_1, y_1) = t_{old}[p_1]$
19: $\quad\quad t_r[i] = (x_0 + d_m (x_1 - x_0), y_0 + d_m (y_1 - y_0))$
20: $\quad$ **end if**
21: **end for**
22: **return** t_r

Re-sampling the points in this manner has two advantages. First, the re-sampling process remains the same for both trajectories that are too long and too short. Secondly, the re-sampling process maintains the distribution of points along the trajectory. A long or short distance between two consecutive points, relative to other consecutive point distances within the trajectory, remains in the re-sampled trajectory. This ensures that trajectories drawn quickly or slowly maintain those sampling characteristics to some extent.

Once re-sampled, trajectories are translated, rotated, and resized in relation to the straight-line trajectory. Since *Space Navigator* state-space feature vector creation geometrically transforms a state, the trajectories generated in response to the state must be transformed in the same manner. This transformation ensures the state-space and trajectory response are positioned in the same state space.

D. Distance Measure

To ensure the trajectories generated in *Space Navigator* are similar to those of an individual player, a distance measure must capture the objective elements of trajectory similarity such as comparing specific points. Additionally, an ideal similarity measure will also be meaningful to human players, in that the distance measure will be small when a human would think two trajectories are similar and large when two trajectories are dissimilar. A human-subject study confirmed that Euclidean trajectory distance not only distinguished between trajectories computationally, but also according to human conceptions of trajectory similarity. The experiment was conducted as follows in the *Space Navigator* environment.

Each of the 35 participants played two five-minute games of *Space Navigator* for familiarization purposes. Then each player completed 60 pre-scripted instances taken from previously captured games of *Space Navigator*. Each scenario starts from a paused *Space Navigator* instance and the spaceship upon which the player is expected to act blinks. The player responds to the scenario by drawing a trajectory for the blinking ship. The game is paused and the trajectory response is recorded. The scenario is then shown to the player again, with their trajectory replaced by three new trajectories superimposed onto the state. The player is asked to choose the trajectory that is "most similar" to the one they drew.

The three trajectories shown to the player include a straight line from the spaceship to its destination planet, the trajectory in the game-play database that is closest to the provided trajectory according to Euclidean trajectory distance, and the response trajectory to a random state from the same state cluster as the current state. The trajectories are presented as A, B, and C in randomized order. The trajectory selected by the player is recorded as the player's choice as the most similar trajectory. The final collected dataset consists of 35 players completing 60 instances each, for a total of 2,100 instances.

Average intra-trajectory distance between all three presented trajectories and trajectory length show Euclidean trajectory distance's effectiveness. If Euclidean trajectory distance properly captures human conception of trajectory similarity, small intra-trajectory distances should mean that all three trajectories

are similarly indistinguishable to humans. Very small intra-trajectory distances should indicate an almost random choice of "most similar" trajectory for the player, while the smallest Euclidean trajectory distance from the trajectory the player drew to one of the presented trajectories should be chosen with regularity at high average intra-trajectory distances. Additionally, smaller straight-line trajectory lengths allow for less distinguishability due to the constrained nature of possible actions at shorter distances. Therefore, small trajectory lengths should induce less certainty in the choice of "most similar" trajectories.

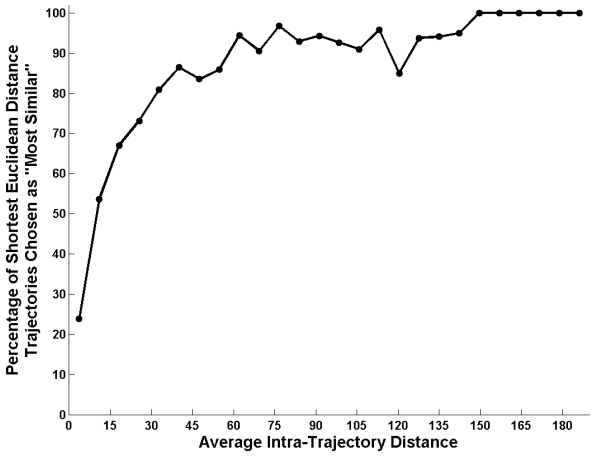

Fig. 4. The percentage of times human conception of "most similar" trajectory agreed with the trajectory deemed most similar according to Euclidean trajectory distance as a function of the average intra-trajectory distance.

The results in Figure 4 show that Euclidean trajectory distance captures human conception of trajectory similarity well. All histograms were compiled in MATLAB, the number of bins (k) set according to Rices rule [39] ($k = 2n^{1/3}$, $n =$ the number of observations), and the k bins equally sized between the minimum and maximum trajectory lengths. As expected, those trajectories presented with extremely small average intra-trajectory distances are chosen at an essentially random rate (23.8%). As the average intra-trajectory distance grows, the shortest Euclidean trajectory distance aligns with human conceptions of "most similar" at rates approaching 100%.

Euclidean trajectory distance also accounts for humans being less able to distinguish between shorter trajectories. Since players are more constrained in possible trajectory choices at short straight-line trajectory lengths, the average intra-trajectory distance correlates well with length as demonstrated in Figure 5. This shows a strong positive correlation ($r = 0.5635$, $p = 0$).

Combining these insights, Figure 6 shows as trajectory length increases, the percentage of trajectories classified as most similar by humans more regularly matches with Euclidean trajectory distance. Euclidean trajectory distance, therefore, serves as an adequate measure of trajectory similarity in the *Space Navigator* game.

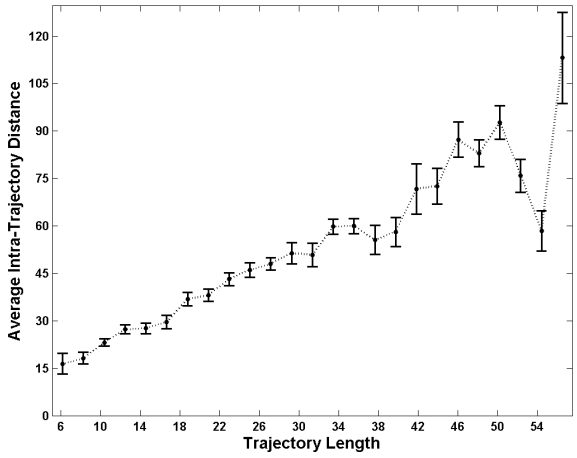

Fig. 5. Average intra-trajectory distance as a function of trajectory length.

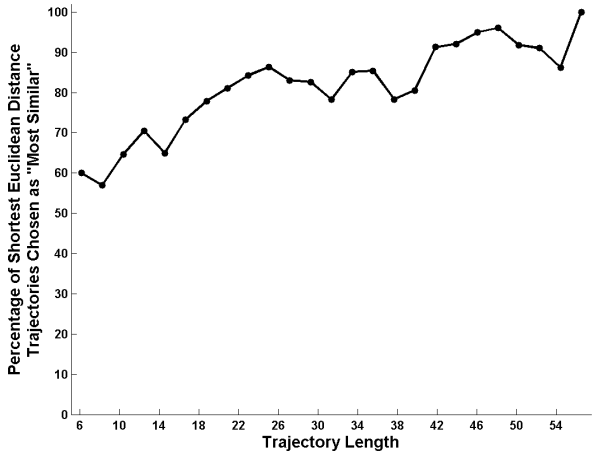

Fig. 6. The percentage of times human conception of "most similar" trajectory agreed with the trajectory deemed most similar according to Euclidean trajectory distance as a function of trajectory length.

E. Generate Response

The response generator utilizes a player model **P** to generate player responses. This section describes a method to generate new trajectory responses using the cluster weights in **P** that derive from either a generic or learned player model.

Existing trajectory generation research has tended to gravitate toward methods creating trajectories one point at a time. Using methods like trajectory libraries [40] or Gaussian mixture models [41], the trajectory generator predicts only the next point on the trajectory. Then it recursively continues the process of creating further points until it reaches the desired en state and returns the entire created trajectory. However, humans tend to think in terms of "full maneuvers" when generating trajectories, specifically for very quick trajectory generation tasks such as trajectory creation games [33]. Therefore, the *Space Navigator* trajectory response generator creates "full maneuver" trajectories.

The trajectory response generation algorithm takes as input:

the number of trajectories to weight and combine for each response (k), the number of state and trajectory clusters (x and y respectively), the re-sampled trajectory size (μ), a new state (s_{new}), a player model (**P**), the set of all state-trajectory cluster mappings (**M**).

Algorithm 5 Trajectory response generation algorithm.

1: **inputs:**
2: $k =$ the number of trajectories to combine
3: $x =$ the number of state clusters
4: $y =$ the number of trajectory clusters
5: $\mu =$ the re-sampled trajectory size
6: $s_{new} =$ a state we have not seen before
7: $\mathbf{P} =$ an $x \times y$ player model
8: $\mathbf{M} = \{\langle S_1, T_1\rangle, \langle S_1, T_2\rangle, \cdots, \langle S_x, T_y\rangle\}$, all state-trajectory cluster mappings
9: **initialize:**
10: $t_{new}(\mu) \leftarrow$ an empty trajectory of μ points
11: $S_{close} =$ the closest state cluster to state s_{new}
12: $\mathbf{P}_{close} = \max_{k} \left[\mathbf{P}_{S_{close},(z|\forall z \in 1,\ldots,y)}\right]$
13: **for** each $P_{close,i} \in \mathbf{P}_{close}$ **do**
14: $T_i =$ the trajectory cluster associated with $P_{close,i}$
15: $s_{close,i} \leftarrow$ state closest to s_{new} in $\langle S_{close}, T_i\rangle$
16: $t_{close,i} \leftarrow$ the response trajectory to $s_{close,i}$
17: **for** $\nu = 1 \rightarrow \mu$ **do**
18: $t_{new}(\nu) = t_{new}(\nu) + t_{close,i}(\nu) \cdot P_{close,i}$
19: **end for**
20: **end for**
21: $t_{new} = t_{new} / \sum_{i=1}^{k} P_{close,i}$
22: **return** t_{new}

Line 10 begins by creating an empty trajectory of length μ which will hold the trajectory generator's response to s_{new}. Line 11 then finds the state cluster (S_{close}) to which s_{new} maps. $\mathbf{P}_{close}$, created in Line 12, contains a set of likelihoods. $\mathbf{P}_{close}$ holds the likelihoods of the k most likely trajectory clusters to which state cluster S_{close} maps.

The loop beginning in Line 13 then builds the trajectory response to s_{new}. Line 15 finds the instance assigned to both state cluster S_{close} and trajectory cluster T_i with the state closest to s_{new}. The response to this state is then weighted according to the likelihoods in **P**. The loop in Line 17, then combines the k trajectories using a weighted average for each of the μ points of the trajectory. The weighted average trajectory points are then normalized across the k weights used for the trajectory combination in Line 21. The trajectory returned by Line 22 is the trajectory response generation algorithms response to state s_{new} according to the player model **P**

V. Experiment and Results

This section describes an experiment to test the real-time individual player modeling trajectory generator and presents insights gained from the experiment. The results show that the individual player modeling trajectory generator is able to create trajectories more similar to those of a given player than a generic player-modeling trajectory generator, with a limited amount of training data. Additionally, the results show how the model provides insights for a better understanding of what separates different players' game-play through an analysis of the individual player models in comparison to the generic player model.

A. Experiment Settings

The experiment compares trajectories created with the generic player model and the individual player model, they are further compared with a trajectory generator that always draws a straight line between the spaceship and its destination planet. The first five games worth of state-trajectory pairs are set aside as a training dataset and eleven games of state-trajectory pairs are set aside as a testing dataset, with each game containing on average 123 state-trajectory pairs. Five training games was chosen as a benchmark for learning an individual player model to force the system to quickly pull insights that would manifest in later game-play. For each of 32 players, the individual player model is trained on the five-game training dataset using Algorithm 2 with the trait score weights. The generic player model and straight-line methods do not require training.

Next, each state in the given player's testing set is presented to all three trajectory generators. The difference between the generated trajectory and the actual trajectory provided by the given player is recorded. The experiment presents states in the order recorded in the original games. The individual player model does not train on the testing data. The experiment also saves the individual player models for later comparison and evaluation. Table I shows specific experimental values for the individual player model.

TABLE I
Experimental variable settings for individual player modeling using Algorithm 2

Variable	Value
Update Increment (q)	0.01
Cluster Population Threshold	240
Cluster Mapping Variance Threshold	17.0
Previous Modeling Utility Threshold	3.0

The three learning thresholds were set specifically for *Space Navigator* as follows. The state cluster population threshold is set at a value of one standard deviation over the mean cluster size, specifically 240. Forty of 500 state clusters received a cluster population weight of $\delta_{cp} = 1$ and 460 received a population weight of $\delta_{cp} = 0$. The cluster variance ratio threshold is 17, with 461 of 500 state clusters receiving a cluster variance weight of $\delta_{cmv} = 1$. For the previous modeling utility, a player model was trained for each of the 32 players with five games worth of data. Then each of the remaining 11 games were predicted using both the trained player model and the generic player model. For each state

across all 32 players, a Euclidean trajectory distance from the generic and individual player models predicted trajectories was calculated from the actual trajectory responses. The cutoff is a learning value of 3, with 442 of 500 clusters receiving a previous modeling utility score of $\delta_{pma} = 1$.

To account for the indistinguishability of shorter trajectories described in Section IV-D, results were removed for state-trajectory pairs with straight-line trajectory length less than length 10.12 meters in the *Space Navigator* environment (approximately 3.5 centimeters on the tablets with 29.5 centimeter screens used for experiments). This distance was chosen as it represents the intersection in Figure 6 at which trajectory lengths reach an accuracy one standard deviation below the mean of trajectory similarity classification accuracy.

B. Individual Player Modeling Results

Testing of the game-play databases shows that the trajectories generated using the individual player model significantly improved individual player imitation results when compared to those generated by the generic player model and the straight line trajectory generator. Table II and Figure 7 show results comparing trajectories generated using each database with the actual trajectory provided by the player, showing the mean Euclidean trajectory distance and standard error of the mean across all 32 players and instances.

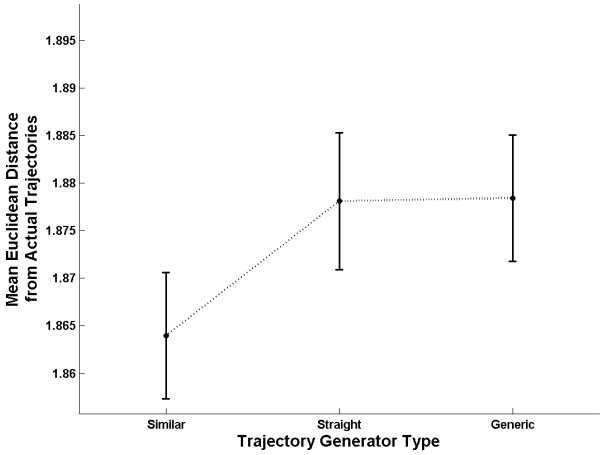

Fig. 7. Euclidean trajectory distance between generated trajectories and actual trajectory responses across three trajectory generation methods.

TABLE II
MEAN AND STANDARD ERROR OF THE EUCLIDEAN TRAJECTORY DISTANCES (IN $SpaceNavigator$ ENVIRONMENT METERS) ACROSS ALL STATE-TRAJECTORY PAIRS.

Database	Mean Euclidean Traj Dist	Std Err
Individual Player Model	1.8640	±0.0063
Straight Line Generator	1.8781	±0.0069
Generic Player Model	1.8784	±0.0063

The individual player model generator provides an improvement over the other models. The mean Euclidean trajectory distance of 1.8640 provides a statistically significant improvement over the straight line and generic player models, as standard error across all instances from all 32 players does not overlap with the latter two player models. The similar player model improves the generic databases accuracy by learning more from a selected subset of presented states to ensure that the player model more accurately generates similar trajectories.

C. Individual Player Model Insight Generation

The individual player models provide insight into general and specific game-play. Comparing the player model learning value changes with the aspects of a state representation allows us to understand what aspects of a state influence game-play and to what degree. How player model changes correlate with the state representation enables game designers a better understanding of what distinguishes individual game-play within the game environment. In turn, this understanding allows for game design improvements.

Table III shows the results of a Pearson's linear correlation between the mean learning value change of each state cluster across all 32 players and the state representation values of the associated state cluster centroids. The results show that there is a statistically significant negative correlation between the mean learning value changes and all of the zones, but some changes are much larger than others.

TABLE III
CORRELATION OF EACH STATE REPRESENTATION VALUE WITH THE MEAN CHANGE IN ASSOCIATED STATE CLUSTER LEARNING VALUES IN PLAYER MODELS

Value	Pearson's r	p-value
Zone 1 - Other Ships	−0.1227	0.0060
Zone 2 - Other Ships	−0.3911	0.0000
Zone 3 - Other Ships	−0.1616	0.0003
Zone 4 - Other Ships	−0.1465	0.0010
Zone 5 - Other Ships	−0.4244	0.0000
Zone 6 - Other Ships	−0.1903	0.0000
Zone 1 - Bonuses	−0.1569	0.0004
Zone 2 - Bonuses	−0.3552	0.0000
Zone 3 - Bonuses	−0.2212	0.0000
Zone 4 - Bonuses	−0.1662	0.0002
Zone 5 - Bonuses	−0.3693	0.0000
Zone 6 - Bonuses	−0.2056	0.0000
Zone 1 - NFZs	−0.1002	0.0251
Zone 2 - NFZs	−0.2749	0.0000
Zone 3 - NFZs	−0.1184	0.0080
Zone 4 - NFZs	−0.1159	0.0095
Zone 5 - NFZs	−0.2398	0.0000
Zone 6 - NFZs	−0.1040	0.0200
Ship to Planet Distance	−0.6434	0.0000

The overall negative correlation arises among object/zone pairs intuitively. High object/zone pair score imply a large or

close presence of an object of the given type, constraining the possible trajectories available to all players. For example, a large presence of other ships in a given zone influences all players to avoid sending trajectories near that area. Therefore, there is more differentiability of player actions available when more freedom of trajectory movement is available.

With the "Ship to Planet Distance" feature, longer distances correlate to less learning value change among player models, with the strongest correlation of all features: r of -0.6434 and p-value < 0.0001. There are several possible explanations for this behavior, including: (1) players are more constrained over long distances and therefore differentiate their actions less, (2) as distances get longer, the variance in the way an individual player draws trajectories in similar situations increases, therefore allowing for no learning of individual tendencies, (3) shorter distances better capture consistent tendencies that a player will carry along to distinguish his game-play over time.

Another aspect that Table III begins to show is the importance of the middle zones in comparison to the "before" and "after" zones. Figures 8, 9 and, 10 illustrate this point graphically. The r values show that the middle two zones

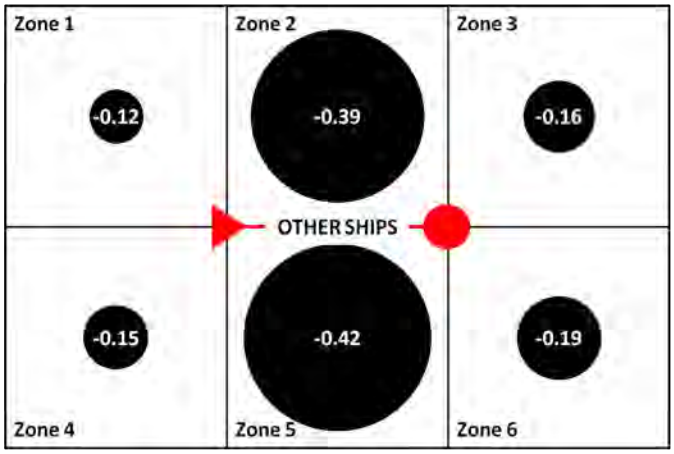

Fig. 8. Graphical representation of the correlation coefficient for each Other Ship/Zone score with the mean change in learning values in player models.

provide an larger influence on the amount of change in the learning values. For example, in Figure 8 the r values for zones two and five are more than double those of any other zone. This idea is somewhat intuitive as this is the area that the ship will traverse, providing the most likely cause for interaction with objects of any given type.

Figures 8, 9, and 10 and Table III provide insight into the relative value that players place on certain types of objects. For example, determining the correlation coefficients of different Object/Zone Pairs can show that No Fly Zones in the middle two zones provide a significantly smaller influence on learning value changes than other ships do in the same zones. Since there is such a large difference, we can infer that players reactions to other ships are more valuable in determining how a person will play the game than No Fly Zones.

Three examples of how player modeling insights can be used in game applications involve training, game design, and player automation. The player models can be used to find

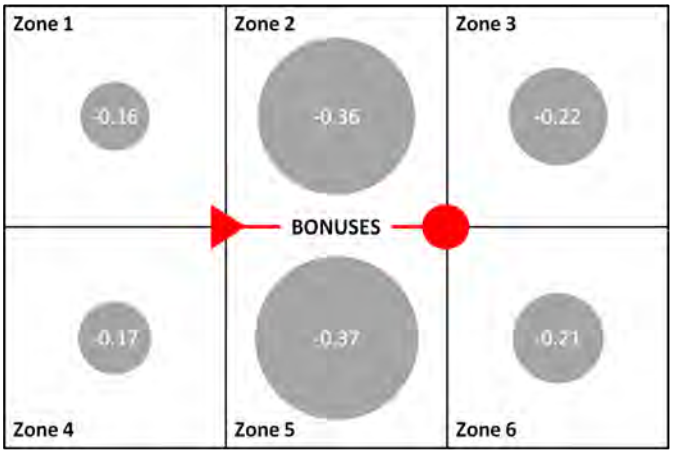

Fig. 9. Graphical representation of the correlation coefficient for each Bonus/Zone score with the mean change in learning values in player models.

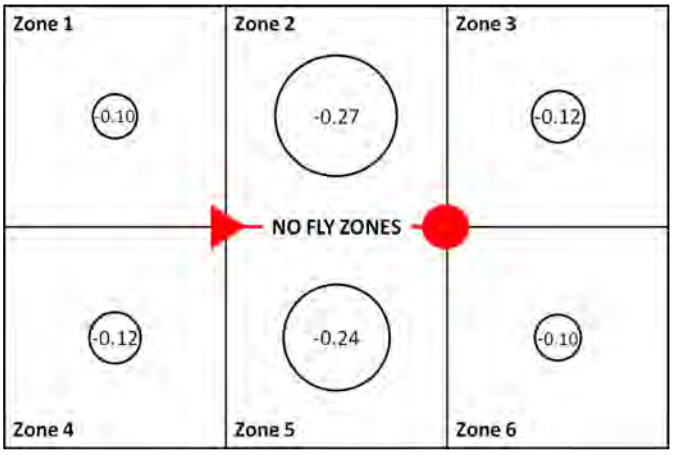

Fig. 10. Graphical representation of the correlation coefficient for each No Fly Zone/Zone score with the mean change in learning values in player models.

places where specific users who are doing really well are properly valuing certain actions (e.g. avoiding other ships) according to the incentives. Proper valuations can then be communicated to players during training within the environment. Another example is that, we can use the player modeling insights to design point structures to more closely align with the way players perceive the value of different object types. In *Space Navigator*, increasing the point magnitudes of No-Fly Zones and Bonuses makes the game more difficult by equally balancing the incentive structure, encouraging less focus on a single objective over the others. Lastly, modeling a specific player enables the designer to incorporate an automated player to play like a specific expert or current user within the game.

VI. Conclusions and Future Work

The real-time individual player modeling paradigm presented in this paper is able to generate trajectories similar to those of a specific *Space Navigator* player. The system is able to operate in real-time without needing to perform time-consuming offline calculations to update player models. Additionally, the gains in individual player imitation are found

in a relatively small amount of game-play (five games/25 minutes). The player models developed to imitate players also allow for a better understanding of what traits of a given state provide understanding of player differences which occur for different states.

This work provides opportunities for several areas of future work. Further studies will research the effects of using the trajectory generator to act as an automated aid for players interacting with the *Space Navigator* game. Additionally, further analysis of the player modeling methods could yield further insights into how much differentiation of individual players can be gained over different amounts of time. Moreover, imitating individual players could provide helpful insights in determining how experts play *Space Navigator* to aid in experiments to learn how to improve player training.

REFERENCES

[1] J. Schrum, I. V. Karpov, and R. Miikkulainen, "Human-like combat behaviour via multiobjective neuroevolution," in *Believable bots*. Springer, 2012, pp. 119–150.

[2] D. Gamez, Z. Fountas, and A. K. Fidjeland, "A neurally controlled computer game avatar with humanlike behavior," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 5, no. 1, pp. 1–14, 2013.

[3] M. Kemmerling, N. Ackermann, and M. Preuss, "Making diplomacy bots individual," in *Believable Bots*. Springer, 2012, pp. 265–288.

[4] H. Yu and M. O. Riedl, "Personalized interactive narratives via sequential recommendation of plot points," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, no. 2, pp. 174–187, 2014.

[5] R. Lopes and R. Bidarra, "Adaptivity challenges in games and simulations: a survey," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 2, pp. 85–99, 2011.

[6] J. M. Bindewald, G. L. Peterson, and M. E. Miller, "Trajectory generation with player modeling," in *Advances in Artificial Intelligence*. Springer, 2015, pp. 42–49.

[7] G. N. Yannakakis, P. Spronck, D. Loiacono, and E. André, "Player modeling," *Artificial and Computational Intelligence in Games*, vol. 6, pp. 45–59, 2013.

[8] D. Charles and M. Black, "Dynamic player modeling: A framework for player-centered digital games," in *Proc. of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004, pp. 29–35.

[9] C. Bateman, R. Lowenhaupt, and L. E. Nacke, "Player typology in theory and practice," in *Proceedings of DiGRA*, 2011.

[10] B. Weber and M. Mateas, "A data mining approach to strategy prediction," in *IEEE CIG 2009*, Sept 2009, pp. 140–147.

[11] J. Gow, S. Colton, P. A. Cairns, and P. Miller, "Mining rules from player experience and activity data." in *AIIDE*, 2012.

[12] A. Drachen, A. Canossa, and G. Yannakakis, "Player modeling using self-organization in tomb raider: Underworld," in *IEEE CIG 2009*, Sept 2009, pp. 1–8.

[13] J. Gow, R. Baumgarten, P. Cairns, S. Colton, and P. Miller, "Unsupervised modeling of player style with lda," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 3, pp. 152–166, Sept 2012.

[14] C.-X. Zhang, Z.-K. Zhang, L. Yu, C. Liu, H. Liu, and X.-Y. Yan, "Information filtering via collaborative user clustering modeling," *Physica A: Statistical Mechanics and its Applications*, vol. 396, no. 0, pp. 195 – 203, 2014.

[15] A. M. Smith, C. Lewis, K. Hullet, and A. Sullivan, "An inclusive view of player modeling," in *Proceedings of the 6th International Conference on Foundations of Digital Games*. ACM, 2011, pp. 301–303.

[16] J. Togelius, R. De Nardi, and S. M. Lucas, "Making racing fun through player modeling and track evolution," *Optimizing Player Satisfaction in Computer and Physical Games*, p. 61, 2006.

[17] S. C. Bakkes, P. H. Spronck, and G. van Lankveld, "Player behavioural modelling for video games," *Entertainment Computing*, vol. 3, no. 3, pp. 71–79, 2012.

[18] J. Rubin and I. Watson, "Computer poker: A review," *Artificial Intelligence*, vol. 175, no. 56, pp. 958 – 987, 2011, special Review Issue. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370211000191

[19] K. Laviers, G. Sukthankar, D. W. Aha, M. Molineaux, C. Darken *et al.*, "Improving offensive performance through opponent modeling." in *AIIDE*, 2009.

[20] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Case-Based Reasoning Research and Development*. Springer, 2007, pp. 164–178.

[21] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in starcraft," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 5, no. 4, pp. 293–311, 2013.

[22] G. N. Yannakakis and J. Hallam, "Real-time game adaptation for optimizing player satisfaction," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 1, no. 2, pp. 121–133, 2009.

[23] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *Affective Computing, IEEE Transactions on*, vol. 2, no. 3, pp. 147–161, 2011.

[24] S. Begum, M. U. Ahmed, P. Funk, N. Xiong, and M. Folke, "Case-based reasoning systems in the health sciences: A survey of recent trends and developments," *Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews*, vol. 41, no. 4, pp. 421–434, July 2011.

[25] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009.

[26] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[27] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning," *Artificial Intelligence Review*, 1999.

[28] B. Argall, B. Browning, and M. Veloso, "Learning by demonstration with critique from a human teacher," in *Proceedings of the ACM/IEEE international conference on Human-robot interaction*. ACM, 2007, pp. 57–64.

[29] M. W. Floyd, B. Esfandiari, and K. Lam, "A case-based reasoning approach to imitating robocup players," in *FLAIRS Conference*, 2008, pp. 251–256.

[30] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Learning from demonstration and case-based planning for real-time strategy games," in *Soft Computing Applications in Industry*. Springer, 2008, pp. 293–310.

[31] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, 2009.

[32] J. H. Ward Jr, "Hierarchical grouping to optimize an objective function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963.

[33] H. H. S. T. Huang, Victor and C. J. Tomlin, "Contrails: Crowd-sourced learning of human models in an aircraft landing game," in *Proceedings of the AIAA GNC Conference*, 2013.

[34] J. M. Bindewald, M. E. Miller, and G. L. Peterson, "A function-to-task process model for adaptive automation system design," *International Journal of Human-Computer Studies*, vol. 72, no. 12, pp. 822–834, 2014.

[35] L. Firemint Party, "Flight control," https://itunes.apple.com/us/app/flight-control/id306220440?mt=8, December 2011.

[36] L. Imangi Studios, "Harbor master," https://itunes.apple.com/us/app/harbor-master/id313014213?mt=8, April 2011.

[37] B. Morris and M. Trivedi, "Learning trajectory patterns by clustering: Experimental studies and comparative evaluation," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 312–319.

[38] X. Li, W. Hu, and W. Hu, "A coarse-to-fine strategy for vehicle motion trajectory clustering," in *ICPR 2006*, vol. 1. IEEE, 2006, pp. 591–594.

[39] D. M. Lane, Ed., *Introduction to Statistics: An interactive eBook*. Rice University, 2013.

[40] M. Stolle and C. G. Atkeson, "Policies based on trajectory libraries," in *ICRA 2006*. IEEE, 2006, pp. 3344–3349.

[41] J. Wiest, M. Hoffken, U. Kresel, and K. Dietmayer, "Probabilistic trajectory prediction with gaussian mixture models," in *IEEE Intelligent Vehicles Symposium*. IEEE, 2012, pp. 141–146.