Probabilistic Models of Quake II Player Movement

Stefan Schoenmackers and **Seth Cooper**

Abstract

Probabilistic models have proven an effective means for reasoning in the face of uncertainty. We are interested in applying these probabilistic methods to computer games. We build Markov models of Quake II maps and use them to analyze and predict player movements. We show these models provide decent prediction in the short term, but grow worse when trying to predict further into the future. We further analyze the difficulty of predicting behavior over longer time periods, and use the KL-divergence metric to indicate the differences in play styles.

Introduction

Probabilistic Models

Many models exist that attempt to represent the world in such a way that computers can reason about it and make rational choices. One family of models involves using logic to represent the world. In this case, things are either true or false, and the rules of logic can be used to reason about the world. However, in a world where we do not have full information, it can be difficult to determine without a doubt whether something is actually true or not. This is where probability enters. In a probabilistic model, things are still either true or false, but there is some amount of certainty associated with the value, called its probability.

Probabilistic models work well for a game world because there is such limited information. Because of that, it is difficult, if not impossible, for a player to deduce the state of the world accurately. The best that can be hoped for is to have some high confidence about the state of the world. Further, the world is rapidly changing and there may be other entities in the world competing against the player. The probabilistic model we choose to represent our world is a Markov model. We will discuss how we use it to represent the world later.

Quake II

Quake II is a popular First Person Shooter (FPS) game that was released several years ago. Screenshots from the game can be seen in figures 1, 2, and 3. Although at this point the graphics appear quite dated, Quake II is an excellent platform for testing new gaming ideas because the source is available under GPL. Quake II has a single player mode, in which a human player is presented with a map full of obstacles (generally taking the form of computer-controlled opponents) which the player must overcome to reach the exit. Quake II has another mode of play, known as "deathmatch", in which several human players can compete against each other in free-for-all combat. Computer-controlled players known as "bots" can also compete, but we consider only human players because their behavior is more interesting. Deathmatch is the mode of play to which we turn our attention. In a deathmatch, players appear on the map at one of a set of predetermined "spawn" locations. They begin with a minimal set of equipment, but

Figure 1: Quake II: Main menu

Figure 2: Quake II: A staircase

Figure 3: Quake II: About to get pWnz0r'd

can build up their arsenal by collecting items that are scattered throughout the map. When one player defeats another player in combat, it is known as a "frag". Once a player has been fragged, he will lose the equipment he has gained and respawn. After a certain amount of time is up, the player with the most frags wins.

Motivation

We are curious as to whether or not the probabilistic models that have been applied to real-world movement prediction will fare well in a game world. We pose the following questions:

- How well does a player's previous movement predict his future movement? Presumably, players are moving erratically to make it difficult for other players to predict their location. At what point is gaining more data a diminishing return? We refer to this as the "Self Prediction" problem.
- How well can we predict where a player will go next? Knowing this could be useful in giving a player advice of safer paths, or reminding them if they are passing by a commonly visited location. Can we figure out which item a player will attempt to get next? We refer to this as the "Interest Target Prediction" problem.
- Can we use one player's movement data to predict another player's movement? In a fair environment, the only movement data we will have access to is our own; if we had access to the other player's location we could just use that to know where they are. Are the high-level movement patterns of different players similar enough for these estimates to be useful? Do players generally navigate the world in a similar fasion? We refer to this as the "Opponent Prediction" problem.

Game and Model

We began by modifying the freely available Quake II source (`ftp://ftp.idsoftware.com/idstuff/source/quake2.zip`) to record a log file of each player's movement. We also logged other information that could prove useful in movement analysis, such as when the player fired, gained health, got a frag, and so forth. We then proceeded to collect data by playing a deathmatch for an extended period of time on the first map of Quake II demo.

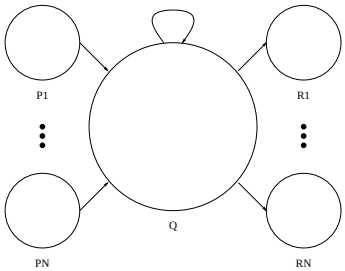

Figure 4: One state in a first order Markov model. Edges are weighted with probabilities.

Models

We represent maps as a directed graph. Nodes in the graph represent locations on the map and edges represent possible paths between the locations.

To use these graphs to capture the movement patterns of a player, the first model we tried was a first order Markov model (FOMM). States in the FOMM represent locations on the map, and the edges are weighted by the probabilities that the player will exit that location to a different location. Self-referential edges represent the player staying at that location. Making a FOMM essentially amounted to weighting the edges of our graph of the map with probabilities; one is shown in figure 4.

One weakness of a FOMM is that is assumes that the next state is only a function of the current state; it has no memory of past events. This assumption does not work well with the way that a player moves through a level, because which location a player was at previously has a large effect on which location the player will visit next. Consider a player running down a long hallway: the fact that the player moved to the right previously means that the player will probably move to the right next. Remembering a player's last action is a useful tool when predicting a player's next one. We can extend the FOMM to a model which takes into account a player's movement history. Instead of the next state only being a function of the current state, we can make it a function of the current state and some number of previous states. This is known as an nth order Markov model.

Building Models from Data

We obtained many of the ideas we used for building our model from (Ashbrook & Starner 2003). The first step of analyzing the data was to turn the list of points logged during play into a set of locations. We used a k-means clustering algorithm to find locations. The basic k-means algorithm we employed is as follows:

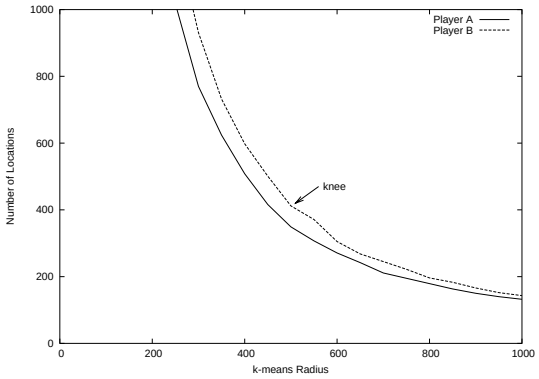

Figure 5: Effect of k-means radius on number of locations

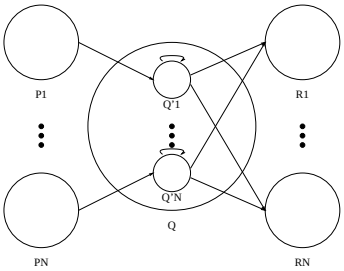

Figure 6: One state in our second order Markov model

1. Select a point from the data; this point is the starting mean. We selected the first point in the list.
2. Take the mean of all points within some radius r of the current mean; that becomes the new mean.
3. Repeat step 2 until the mean moves less than some threshold value.
4. Add the mean to the set of locations and remove all points within r of the mean from the data set.
5. Repeat steps 1-4 until there are no points in the data set.

This clusters spatially similar data and effectively discretizes the map into a more manageable model. As the radius of the mean goes up, the number of locations goes down. In order to find the optimum radius for k-means, we try several radii and plot radii against number of locations, as in figure 5. We then attempt to find the "knee" of the graph where the number of locations converges on the number of points. We find this point simply by looking for where the slope of the line goes from being less that -1 to being greater than -1. For our data we concluded the optimal k-means radius was 500.

Once we have a set of locations, we must figure out the connectivity of the locations. To do this, we first assign each point in the data set to a location. This is simply the closest location to the point. Then we traverse the list of points in order and increase the probability of moving from a point's location to the following point's location by 1 for each pair of points. Note that this often corresponds to an edge referring back to the same location. At the end, we go back and renormalize all probabilities.

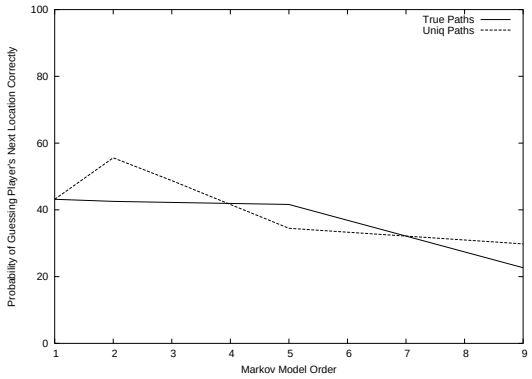

Figure 7: Effect of Markov model order on prediction accuracy

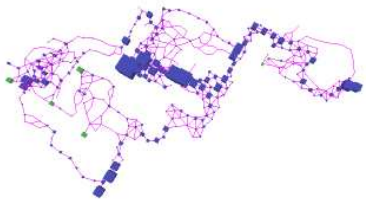

Figure 8: Model visualization. A location's size is proportional to the number of points assigned to it. Green locations are spawn points.

KL-Divergence

KL-divergence (or relative entropy) is a useful measure of the distance between two probability distributions (Zhai 2003). We use KL-divergence to measure the distance between the probability distributions of two Markov models over time. The models each start with some distribution of probabilities across their locations, and at each timestep the probabilities are updated by moving the probability distributions at the locations along the edges. Thus the KL-divergence of two distributions P and Q at timestep t can be calculated as:

$$D_t(P \parallel Q) = \sum_{l \in L} P_t(l) \log \frac{P_t(l)}{Q_t(l)}$$

In practice, we assign each location some small probability when calculating the divergence to prevent the singularities in the equations.

Self Prediction

Upon finding the connectivity of the locations, we experimented with various orders of Markov models to determine which would give us the highest accuracy when predicting player movements. We also experimented with the way in which the model remembered a player's location history. One model remembered the player's previous locations as the path of locations the player went through to get to the current location, including transitioning to the same state multiple times in a row. We call this the "true path". The other model remembered the path of the player's previous locations, but removed any locations visited twice in a row from the path. We call this the "uniq path". Using the uniq path allows us to capture more of a player's previous history in a lower order model. To test the accuracy of a model, we built them using the first 90% of the data. Then we looked at each time the player made a transition from one state to another the remaining 10% of the data. The accuracy of a model was the number of transitions made which corresponded to the most likely transition in the model over the total number of transitions made. The results are shown in figure 7. We decided to use a second order Markov model (SOMM) using the uniq path, since it had the highest prediction accuracy. To implement a SOMM, we augment each state of our initial FOMM with a number of substates, one for each incoming edge, as in figure 6. Each substate has its own probability distribution over exiting states. In this way, the next state depends on not only the current state but also the previous one. Notice that transitions back to the same state transition back to the same substate. This ensures uniqness of paths. A visualization of one of our model can be seen in figure 8. We repeated the previous experiment, removing blocks of 10% and testing for next-location accuracy. On average, we were able to predict the player's next location accurately about 55% of the time. A random guess was correct only 41% of the time.

A simple initial test of predictability in the Quake II environment is to build the SOMM as described above, and examine how well it is able to predict that same player's actions over longer amounts of time. We had initially thought this would yield a relatively high accuracy, similar to the behavior tracking in (Liao, Fox, & Kautz 2004), since players tend to repeat actions. However, our initial experiments showed that this was not the case. We realized that with no opponent, a user's actions would probably tend to be predictable (and uninteresting), but once there is an adversary in the environment, a player changes his style, due to the complex hunter-hunted nature of the game. Namely, players spend very little time picking up items and much more of their time searching for or dueling their opponent. The pattern of a player's search for his opponent is often highly random, based only on the player's guess of where his opponent is, and the details of how a player moves during a fight are further altered and biased by the location of his opponent. This leads to small intervals of highly deterministic behavior (e.g. grabbing a weapon when the opponent isn't around) followed by large intervals of unpredictable behavior.

Since we weren't able to accurately predict this behavior, we wanted to examine to what extent it was at least self-similar. Namely, how much data is needed to generate something similar to a player's model of the world built using the full data set.

To examine the similarity of two models, we compare the KL-divergence of the two models as their belief states vary over time, given that the player starts at one of the spawn points. We use a common set of locations/states, and compute the transition probabilities for each model based on the data for that model.

To account for the randomness in a player's movement, and to ensure that we don't reach singularities in the KL-divergence equation, we assume that each state has a base probability of some very small epsilon. We then initialize each of the spawn points' probabilities to 1/#spawn points. At each step, we update the probabilities for each state as each SOMM probability model specifies, and normalize so the total probability is 1. We then compute the KL-divergence of the two models. We do this each step for 1000 steps, since this is more than enough time to reach steady-state, and would be more than enough time for a player to get anywhere on the map.

We run this over many time-steps and examine how the models diverge over time. If the two models are identical, then the divergence will be 0, whereas if they are different, the divergence will be a very large number. Consider the following example: the world is a long hallway with a box in the middle. If player A always goes left around the box, and player B always goes right around the box, but otherwise they move identically from one end of the hall to the other, the divergence over time will slowly increase up to the time when the players would reach the box, and then it would decrease until it reached a steady state. The divergence in this case would be a nonzero, but small number, related to their slight differences in path. In computing the divergence over time, it provides information both on how similar the models are at a given point in time, and how similar they are at the steady state.

We wanted to examine how much data is needed to create an accurate model of the player's behavior. To do this, we computed the KL-divergence of a model built using only the first X% of the data, and the model built using the entire data set. Figure 9 shows the divergences over time of models using varying amounts of data with respect to the model using all of the data. The graphs for player A's partial models were similar. As is expected, only using small amounts of data leads to large divergences, since the first 10% of the data is missing a large amount of the (current state, previous state) pairs needed to fully create the SOMM.

In the early steps there are large divergences, due to slightly different models of how the player moves away from the spawn points (some of those points were only

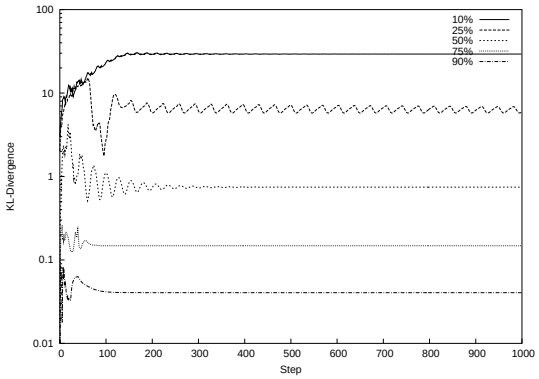

Figure 9: KL-Divergence of the models based on using just the first X% of player B's the data, with respect to their full model. As expected, the more data causes the model diverge less. The relatively large values for models up through 75% shows that a player's behaviors have great variation over time.

appeared a few times for a player, and thus the divergence is probably due to missing data.) The oscillations in the 50% models are most likely due to small repeating patterns (e.g. a player running in a loop), and the model is missing some critical information to break that loop. We suspect the 75% model has similar oscillations to begin with, but contains enough information to escape from those loops.

Finally, eventhough the 90% line looks relatively low, reaching a steady-state of 0.035 divergence, it's still far from ideal, since it is probably due to the inability of the final 10% of the data to change the probability model much, and not due to the increased ability of model to accurately predict a player's behavior.

Interest Target Prediction

Given the difficulties of predicting a player's exact path, we also examined the problem of predicting which "interesting location" the player would go to next. There has been recent work (Liao, Fox, & Kautz 2004) in automatically finding "interesting locations" in people's daily habits. This work has taken the approach of defining an interesting location as a location where a person spends a great deal of time (e.g. their work, home, friend's house, etc.) While this assumption holds for predicting everyday behavior, we realized that this approach would not be suitable in the Quake II environment. In the real world, a person has quite a bit to gain by staying in one place (e.g. making progress in their work, sleeping, etc.), but if a player stays still in the Quake II environment, all they have to look forward to is being an easy target [1]. In Quake II it is advantageous for a player to be constantly moving, which makes them more difficult to frag, but also makes the method of (Liao, Fox, & Kautz 2004) more difficult.

We can instead define an "interesting location" as a location containing a beneficial item in the map. Namely, we define it as any location containing a health pack, weapon, or ammunition. We believe this is fair, since they are unique locations that players tend to visit repeatedly. We would like to predict where the player is going at a high level (i.e. to which item), so that we can suggest safer routes to that target.

We wanted to examine if our model could predict what item the player would pick up next, given their current location. Since the SOMM performed better than the FOMM one in the previous prediction steps, we used it to predict the next interesting location the player will move to. In this model, the clustered locations represent the states, and each state predicts what item the player will pick up next, given his current and previous locations.

To determine the probabilities, we first trace the player as they walk through the world. If they enter a location with an item, then we propagate this information in the model by incrementing a counter for each (location, previous location) pair the player went through from either their last time of being fragged, or the last item they picked up. Finally, we normalize the counters to turn them into appropriate probabilities.

To measure the accuracy, we used a "leave-one-out" rule. Namely, we build the model and probabilities on X% of a player's data, and then test the model's in prediction accuracy on the remainder of their data. To account for noise, we do this starting at different positions in the data, and average the results. To calculate the accuracy of the model, each time a player enters a location, we use the model to find their most likely next target. If this ends up being the actual next target, we increment a counter for the number of correct predictions. The overall accuracy is defined as the number of correct predictions/the total number of predictions. We ignore predictions made from the last item a player picks up until they are fragged, since these predictions would unfairly skew the results, as the player is not given the chance to get to that item. When the player respawns they will most likely head for other items, since they will be in an entirely new location.

Figure 10 shows the average "next item" prediction accuracy when training on varying amounts of data. The results are averaged between the two players. As you can see, we are able to correctly predict the next item 1.5-3.5 times better than random. While the overall accuracy is not that high (7% at most), this is not too bad considering how erradic a player's motions are. This also doesn't include information about when an item was present at that location (since items have a respawn time), so a player would probably skip visiting a location if the item were not present at the time. If we had this information, then we should be able to increase this prediction accuracy to a much larger number. However, due to time constraints, we weren't able to modify

[1] This excludes waiting in a strategic area for your opponent to show up, but this behavior, known as "camping", is generally frowned by the community, and there was only one such point in the map for the test data.

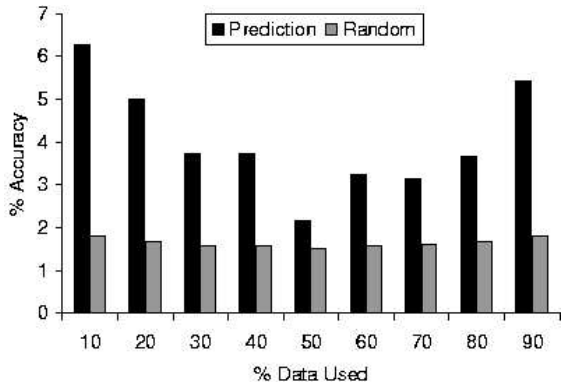

Figure 10: Interesting Target prediction accuracy, using X% of the data as a model, averaged over both players. The model performs 1.5-3.5 times better than random at predicting what the next item the player will head for is. This is artificially low due to a player skipping a location when the item hasn't spawned yet.

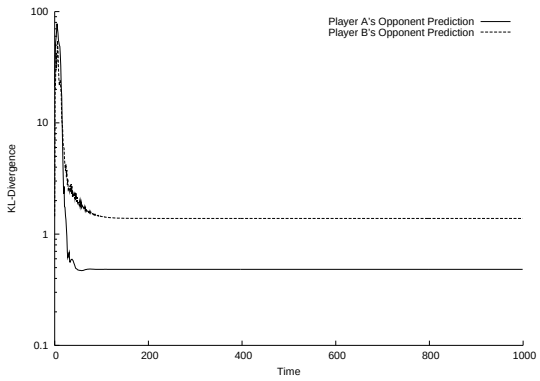

Figure 11: KL-divergence of each player with respect to the other. The large divergences in the beginning indicate that the players have different strategies when first spawning, and the steady-state divergences indicate that there is a large discrepancy between where each player moves in the long run.

the model to account for these mispredictions, so the model is unfairly penalized because of this.

Opponent Prediction

Previous sections focused primarily on predicting a player's behavior, based on their past behavior. In this section we examine the possibility of using one player's behavior to predict their opponent's behavior. Initially we had hoped to predict with some accuracy, and without cheating, where your opponent is. This would be useful in everything from recommending safe paths to items to allowing the player to set traps for their opponent. However, as we observed in the section on Self Prediction, even predicting your own patterns over time is difficult, due to the erratic way players move through the environment. While we do not expect a high accuracy in predicting where your opponent is, even a relatively low accuracy would give some knowledge to the system. Furthermore, a comparison of one player's model with their opponent would give some indication of the playing style. Namely, if a player has a similar style as their opponent, their models should be similar (to the extent a player's model is self-similar). However, if they have drastically different styles, then their models should differ significantly.

We examined two methods of Opponent Prediction. The first method we examined was the KL-divergence as in the Self Prediction method above. Namely, we create the probability distributions of each state in an SOMM, and then compute the KL-divergence of the two models over time.

Figure 11 shows the KL-divergences of the model for each player with respect to their opponent. The large divergences in the beginning, which are much larger than any of the Self Prediction models in 9, indicate that players have different strategies when they spawn. Furthermore, the relatively large steady-state divergences indicate that, even with the full model, the players would end up in significantly different distributions of states. Since the models are so significantly different, even from the Self Prediction models that had any accuracy, we are able to see that the two players have significantly different styles of play. When we traced what each player did, we found this to be true, since one player spent a fair amount of time "camping", while the other player moved around the map much more.

In addition to comparing the KL-divergences of the two models, in Figure12 we also examined the accuracy of using a particle filter to predict the opponent's location. The use of a particle filter has several advantages over the method of comparing KL-divergences. For one thing, it is fast enough to do online. Additionally, we can use other signals a player receives during play.

The KL-divergence measure looks at statically predicting where a player will move, given that they start at a spawn location. With a particle-filter, we can use information about whether the player fired, was hit, was fragged, or fragged their opponent to give a better indication of where that player is. This doesn't violate the "non-cheating" rule, since all observations are made on properties of the player, and not on any information obtained from the opponent.

For the particle filter, we update particles at each step based on the player's model of the world. Each particle has some likelihood of staying in the same state, or transitioning to a new state. When a player has one of these indications, we resample the particles to be within a small radius of the player's current position (since if the player is firing, that means their opponent is probably nearby). Similarly, if the player got a frag, the particles are resampled from the static spawn-points with equal likelihood (since at that point the opponent's location is known to be at one of the spawn points).

To measure the accuracy of the particle filter at each

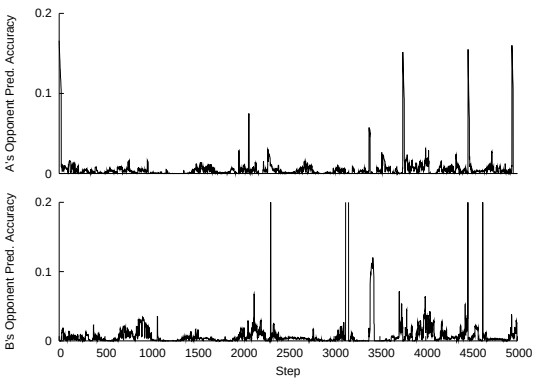

Figure 12: Accuracy of particle filters on predicting your opponent's location, over a 7.5 minute span of gameplay. The accuracies increase and decrease at the same time, indicating when the two players are dueling. Large spikes represent that player getting a frag (i.e. opponent is now at a spawn point).

step, we look at the number of particles in the opponent's state/total number of particles. The results of this are shown in Figure12 As was expected, the prediction accuracies were quite low (<1%, on average). However, it is interesting to observe that the accuracies of the two players increase at the same times, such as the pattern at about 3700 samples. These increases in probability are during a duel, where the two players are both firing at each other. The accuracies increase, because we are more sure that the opponent is in the immediate area during these times. Finally, the large spikes in accuracy generally occur just after a player gets a frag, since immediately following that they know their opponent's location must be one of the spawn points. These results just confirm the difficulty of using one player's behavior model to predict their opponents behavior, especially given the differences in style of the two players.

Conclusions

We conclude that at best, it is difficult to predict where a player will move in Quake II. There are several reasons for this.

One is that many players are making a conscious decision to produce unpredictable movement patterns. If a player moves in a very predictable way, then the other players will most likely pick up on it and use it to their advantage.

Also, players are for the most part constantly moving. This distributes the density of datapoints evenly and makes it difficult to find important locations in the data. Unlike real life, where people may spend around half their day at work and the other half at home, in our analysis there was no one location where a player spent the bulk of his time. Again, if a player spends most of his time in one place, other players will know to expect him there.

Figure 13: Quake II: Prototype user guidance system. The light blue squares direct the user to a point of interest.

The relative density of spawn points and size of the maps present a large problem with predicting where an opponent will be. Since he could start at any spawn point, that already diffuses the probabilities of where he might be. Since the maps are fairly small, it doesn't take that long for a player to run all the way across it. This causes the probability density to diffuse very quickly from the spawn points, until it is hard to say with any high probability where an opponent is.

Future Work

The next step seems to be using the models we have built to present the user with useful information. This could take the form of an on-screen display that the user can turn on or off. It could inform the user of the probability that an opponent is nearby, or inform the user of safe paths to nearby points of interest. To this end, we developed a simple prototype of a user guidance system, shown in figure 13. It consists simply of a directed pointer that can guide the user toward a particular point. It would be interesting to see if guidance from the predictions made by our model could improve the performance of human players.

Another interesting idea would be to try to pick out different movement patterns within a player's overall movement data. A player will most likely have different movement patterns if he is low on health, in combat with another player, or trying to reach a specific location. Currently, we look at all points in the data set as equal. We could split the input data sets into different sets based on the extra information in the log files. For example, we could take the points where a player has less than half health and presume that they represent the player's movement when he is on guard, and take the points around where a player fires or is damaged and presume that they represent the player's movement during combat. We could then do a more fine-grained movement analysis that would allow us to better predict how a player would move in a given situation.

References

[1] Ashbrook, D., and Starner, T. 2003. Using gps to learn significant locations and predict movement

across multiple users.

[2] Liao, L.; Fox, D.; and Kautz, H. 2004. Learning and inferring transportation routines.

[3] Zhai, C. 2003. Notes on the kl-divergence retreival formula and dirichlet prior smoothing.