TOWARDS ONLINE ADAPTATION IN ACTION GAMES: CASE STORAGE AND RETRIEVAL

Thomas Hartley and Quasim Mehdi
School of Computing and Information Technology
University Of Wolverhampton, UK, WV1 1EL
E-mail: T.Hartley2@wlv.ac.uk

KEYWORDS

Online adaptation, case storage and retrieval, k-d trees.

ABSTRACT

In this paper we present our work towards the development of an online learning and adaptation architecture for non-player characters (NPCs) (agents) in first person shooter (FPS) computer games. We will outline the development of our case storage and retrieval method, which uses an adaptive k-d tree based approach and discuss the issues related to employing this technique for online storage and retrieval of cases. We conclude by evaluating the performance of the developed data structures and discussing results.

INTRODUCTION

Action games traditionally take place in a 3D environment, in which the player interacts through the control of an avatar. These games usually involve running around and using deadly force against an enemy (Laird and Lent 2001). This typically means a player can shoot an opponent using a variety of weapons, which they pick up from the game world. Players can also pick up other items such as health packs, ammo and bonuses (e.g. shield, mega damage etc). The player observes the game world from a first person (the head position of the avatar) or third person perspective (behind the shoulder of avatar). Examples of action games include Half Life, Quake, Unreal and Tomb Raider.

Currently opponent behaviour in commercial action games tends to be achieved through pre-programmed scripts, finite state machines (FSMs) and the A* path finding algorithm. The use of these techniques has yielded impressive results. However by their nature scripts and FSMs are rigid, predictable and cannot adapt / generalise to circumstances that were not anticipated by the artificial intelligence (AI) programmer (Fairclough *et al.* 2003). In future, game developers may need to employ more sophisticated academic AI techniques, which enable them to make NPCs in their computer games more human-like and responsive to the game player. In Hartley *et al.* (2005) an online adaptation architecture for interactive simulations (specifically FPS computer games) has been proposed as one AI approach that can be used in games development. Online learning through interactions with a human user is one of the key research directions for intelligent agents. Traditionally computer game AI has not made use of unsupervised online learning (Spronck *et al*. 2003); however there has been a certain amount of research in the area (Dinerstein and Egbert 2005, Spronck *et al*. 2003). All these works produce more believable behaviour for NPCs and as Spronck *et al*. (2003) comments; they demonstrate the potential of this approach for improving the entertainment value of commercial computer games.

In this paper we concentrate on the storage and retrieval approach used in our online action prediction system (Hartley *et al.* 2005). We develop an adaptive k-d tree based technique, which employs a hybrid split strategy to build trees online as cases are observed. An optimisation technique is also developed, which rebuilds trees as they grow out of balance. The remainder of this paper is organised as follows: In the next section we discuss related work and the proposed action and behaviour prediction architecture. After that we outline the development of our k-d tree based storage and retrieval method. We introduce and discuss our hybrid split strategy and our optimisation approach. We conclude by discussing our initial results and future work.

ONLINE ADAPTATION SYSTEM OVERVIEW

The architecture outlined in Hartley *et al.* (2005) expands on existing work in the area of online behaviour adaptation. Work in this field has made use of a variety of approaches, such as prediction, user modelling, anticipation, reinforcement learning and plan recognition (Hartley *et al.* 2005). This work builds upon existing incremental case based approaches to modelling an observed entity in order to predict their behaviour (Dinerstein and Egbert, 2005, Fagan and Cunningham, 2003 and Kerkez and Cox, 2001), and makes a number of novel contributions to improve the capabilities of the system. Specifically our architecture offers a more comprehensive state representation, behaviour prediction, and a more robust case maintenance approach, including case storage and retrieval. One of the most interesting approaches to prediction / anticipation in computer games that has

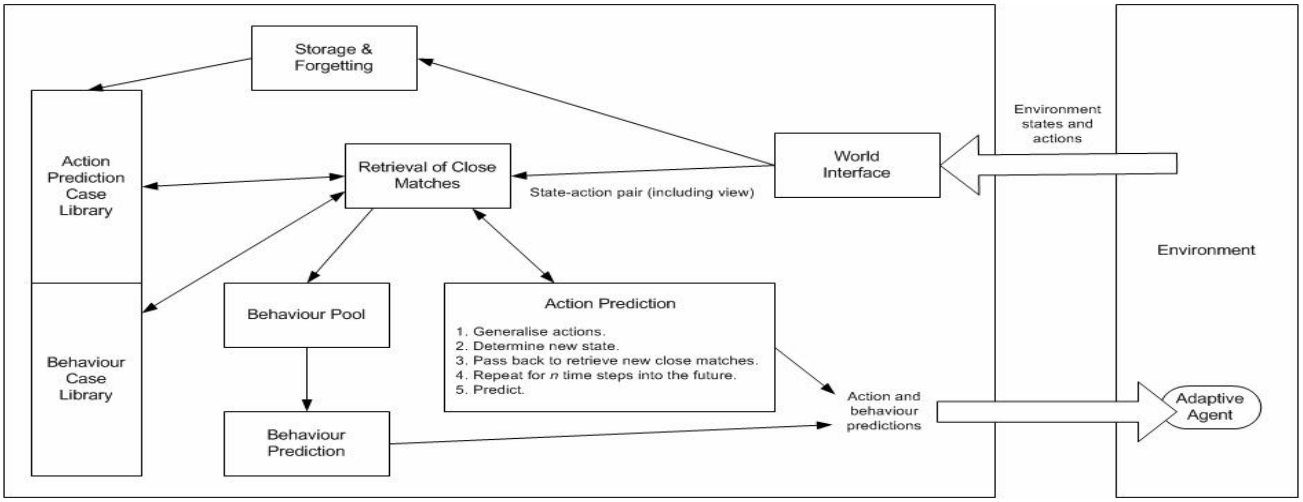

Figure 1: High-level overview of the proposed system architecture

inspired our work can be found in Dinerstein and Egbert (2005). Their system has been designed specifically for use in real-time interactive simulations, it provides the ability to model an observed entity and predict their actions.

Figure 1 illustrates a high-level overview of the proposed action and behaviour prediction layer of our adaptation architecture. The input for the system comes from observed environment events and the output is action and behaviour predictions. The world interface module converts environment input into state-action pairs. Once the input has been properly formatted the system retrieves close matches to the input from the case libraries. The action prediction library returns cases, which are close to the query state-action pair. The behaviour case library returns cases, which contain similar states to the query states. These are then passed to the behaviour pool and actions prediction modules, which are independent processes and will run in parallel (i.e. different threads) or sequentially. The action prediction module generalises actions and determines a new predicted state based on the generalised action. The predicted state is then passed back to the retrieval module in order to determine close matches to it. The process, between the retrieval and action prediction modules, is repeated for *n* time steps into the future, the predicted state is then passed to an agent upon the completion of this process. Once predictions have been made the storage and forgetting module updates the case libraries using a case maintenance policy.

Case Maintenance

In order to guarantee the performance of our action prediction case library, a case maintenance policy is required. Too many cases in the library can significantly slow down the system and lead to poor prediction and generalisation (Kolodner 1993). In addition deleting cases (or "forgetting") is very important when learning something as non-stationary as human behaviour (Dinerstein *et al*. 2005). In Dinersteins' work the number of cases in a region of the state space is fixed and cases are selected for replacement based on their age and unimportance (i.e. their similarity to a new case being added). This approach is satisfactory, however limiting the number of cases in a region of the state space can prevent the system from learning behaviours and as stated above having too many cases will reduce the prediction quality and speed of the system.

In order to keep the action prediction case library at optimal performance we will use the following procedure:

- If the closest library case to a query case exceeds a predefined upper threshold the query case will be added.
- If the closest library case to a query case is below a lower threshold, the query case's view (see next section) will be compared to the library cases view(s). If their views are dissimilar the associated view and action of the query case is attached to the library case, if they are similar the query case is discarded.
- If the closest library case to a query case is in between the lower and upper threshold (as illustrated in Figure 2), all cases in this area will be evaluated using a metric. The library case with the lowest returned metric value will be removed and the query case added. We formally define the case replacement policy as follows:

$$\text{if } M(\mathrm{s},\mathrm{a}) < \mathrm{r} \text{ then replace all } (\mathrm{s},\mathrm{a}) \text{ with } (\mathrm{s}_t,\mathrm{a}_t).$$

Where (s,a) is a case in the case library, $(\mathrm{s}_t,\mathrm{a}_t)$ is the query case and r is a predefined threshold value. The metric *M* is defined as follows:

$$M(\mathrm{s},\mathrm{a}) = -\alpha \times \mathrm{age} + \beta \times ||(\mathrm{s},\mathrm{a}),(\mathrm{s}_t,\mathrm{a}_t)|| + \delta \times \mathrm{useful}$$

This replacement metric is based on Dinerstein *et al*. (2005) approach; however we have also incorporated "usefulness", which defines how often the case has been used in prediction. A case which is used more often will more likely be retained in the case library. For example the simple usefulness metric: $\text{useful} = \text{age} \div \text{times used}$ could be implemented. The time since the case was last used in prediction could also be taken into account.

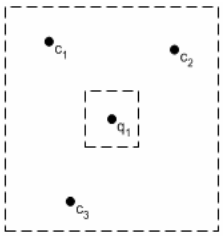

Figure 2: A lower and upper threshold for a 2 dimensional case base. q_1 is the query case and c_1 to c_3 are library cases

CASE STORAGE AND RETRIEVAL

Incremental case-based techniques are used in our proposed system (Hartley *et al.* 2005) to model an observed entity. State-action pairs are treated as cases and stored in a case library. States and actions are made up of *n*-dimensional feature vectors. We have proposed a dual state representation, consisting of primary and secondary state definitions. The primary state space is relatively compact and is used for the main searching and indexing of the system. Additional (or secondary) state information is stored in environment views that correspond to specific primary states and is used to provide a more comprehensive match of states stored in the case library to query states. As cases are observed a case maintenance policy (See previous section) is used to determine if a case will be stored in the library.

The case library is used in the action adaptation layer to predict an observed entity. Prediction is achieved through the nearest neighbour (Mitchell 1997) algorithm, which determines close matching library cases to a query case. A states associated action is then used to predict an observed entity's action. A naive nearest neighbour algorithm could be used for prediction, where every instance in the case-base is examined. However this approach is only suitable for small case bases as it has a time complexity of *O(N)*, where *N* is the size of the case base. Spatial access methods are used to structure the case base more efficiently, in order to avoid examining every case in the library and improve retrieval times.

ADAPTIVE K-D TREES

One of the most well-known main memory *d*-dimensional data structures is the k-d tree (Bentley 1975). The idea behind k-d trees is to partition the state space into disjoint regions by means of iso-oriented hyperplanes that pass through at least one data point. The direction of the hyperplane alternates between dimensions (attributes) at each level of the hierarchy and acts as a discriminator. The data points represent nodes in the tree (Gaede and G¨unther 1998).

k-d trees have a number of disadvantages in that they are unbalanced, sensitive to the order in which points are added and data is scattered throughout the tree (Gaede and G¨unther 1998). The adaptive k-d tree (Bentley and Friedman 1979) overcomes these deficiencies by choosing a split point that divides data into about equally populated partitions. The splitting hyperplanes are still parallel to axes, but they need not contain a data point and their directions do not have to strictly alternate (Gaede and G¨unther 1998). Interior nodes of the tree contain the dimension and the position of the split. Data is stored in terminal / leaf nodes. A leaf can contain a list of points, usually up to a fixed number. Figure 3 illustrates an example adaptive k-d tree for a 2-dimensional state space, with a maximum leaf node size of 1.

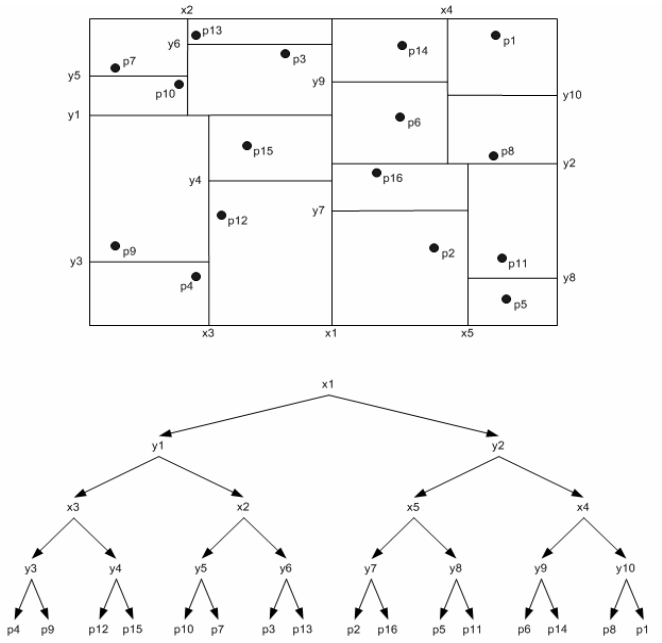

Figure 3: Adaptive k-d Tree

The adaptive k-d tree is a rather static structure, which is difficult to keep balanced when faced with regular insertions and deletions (Gaede and G¨unther 1998). However there are a number of techniques, which can be used to mitigate / overcome these limitations.

BASIC CONCEPTS OF OUR APPROACH

The idea behind our solution for fast case retrieval is a tree structure that partitions the state space into disjoint cells with each leaf encompassing a region of the

space. Leafs (or buckets) are of fixed size and store cases that are contained within its region. Decomposition will be achieved through point based adaptive k-d trees, which are built online as cases are stored. The split dimension of the tree will be rotated (i.e. x, y, z, x…) and the split position can be chosen locally optimal (i.e. optimal to the cases in the bucket to be split) or independently of the data.

This type of data structure is called semi-dynamic as it allows weak insertions and/or deletions. Weak updates can be viewed as updates that do disturb the balance of the tree, but not too drastically (Overmars 1981). Semi-dynamic data structures can be transformed into dynamic ones by building a new structure as the old one degrades. This process is referred to as dynamization (Overmars and Leeuwen 1981). The idea is that neither insertions nor deletions actually need to restore the "balance" of a data structure immediately, as long as the structure remains "in reasonable shape" (Overmars and Leeuwen 1981). Since our case maintenance policy replaces cases within a region of the state space, rather than making arbitrary deletions our system fits well with this idea.

CONSTRUCTION OF THE K-D TREE

In this section we discuss our tree approach in more detail and explain the process involved in inserting a new case. Since our adaptive k-d tree is built online and faces regular updates we need a fast approach that keeps the tree relatively balanced. We opted to build the tree as cases are added, using a construction technique that is akin to LSD-Trees (Henrich *et al*. 1989). Inserting a case involves traversing the tree to find the bucket, which corresponds to the region of the state space that encompasses the new case. The case is then added to that bucket. After a number of insertions a bucket reaches its limit and a new insertion results in it being split, which divides its region into two sub regions. Initially one bucket corresponds to the entire state space. When a split occurs a bucket changes into an internal node and generates two child buckets that store its cases, according to a split strategy. The new internal node contains the dimension and position of the split.

The split strategy involves selecting a split position and location. In general two categories of split strategies can be distinguished (Henrich *et al*. 1989). Data dependant split strategies depend on the case stored in the structure, for example using the median cut along a dimension with the widest distribution of cases. Distribution dependant split strategies choose the split position independently of data. For example a uniform case distribution could be assumed and a buckets region could be split into equal sizes (Henrich *et al*. 1989).

To reduce processing requirements we use the traditional approach of rotating around the split dimension (i.e. x, y, z, x, …), however selecting a split location is more complicated. Data dependant strategies can result in a skewed tree if data is inserted in pre-sorted order (Henrich 1996). Whereas distribution dependant split strategies require a hypothesis of the data distribution to be formulated prior to the tree being built (Henrich 1996). Since cases in our proposed system are observed online, based on an agents relative position to an observed entity, it is difficult to accurately determine data distribution before hand. In addition as cases in our proposed system are sequences of relative positions they will appear in a geometric order, rather than randomly. As a result we have used a hybrid split strategy, which attempts to combine the best of both approaches by adapting the choice of split location to avoid degradation of the data structure.

Limited Redistribution

We implemented a hybrid split strategy called limited redistribution (Henrich 1996), which works as follows. If a bucket cannot accept another case we attempt to redistribute a case from the bucket to its sibling (I.e. the

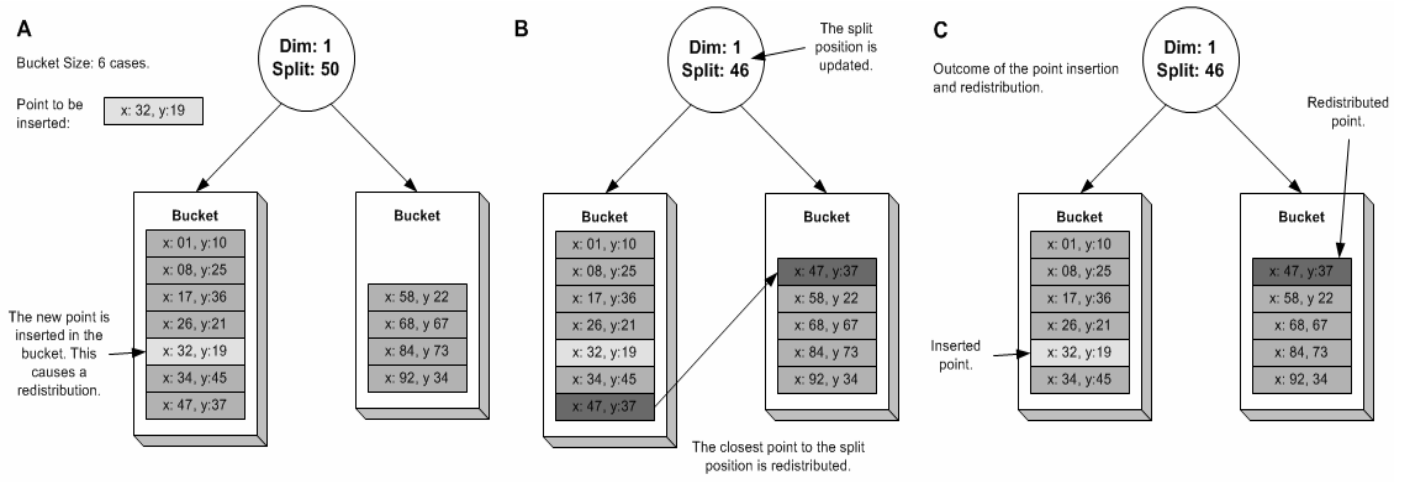

Figure 4: Case Redistribution

other child of the bucket's parent node), rather than splitting the bucket. If the sibling bucket can accommodate the new case without splitting, a local redistribution is performed, which shifts the case that is closest to the split line from the bucket into its sibling and adjusts the split line in its parent node (Henrich 1996). Figure 4 illustrates this process.

By using this approach the originally set split line can only be considered preliminary, since it is adjusted as cases are added. A consequence of this is that the data structure becomes sensitive to the insertion of presorted cases (Henrich 1996). Therefore Henrich (1996) suggests stopping redistribution for those paths that are in danger of degrading due to presorted insertions. He formally defines this process as follows. Let β denote the number of buckets in the k-d tree and let l denote the length of the path from the root node to the bucket. Limited redistribution is only performed if $3 < l - \log_2 \beta$. Redistribution is therefore only performed when the path from the root node to the bucket being split is short enough (i.e. considered to be balanced, based on the number of buckets in the structure).

OPTIMAL TREES: A FULLY DYNAMIC K-D TREE

As discussed above the drawback to our construction method is that it only supports weak updates. In order to stop the tree degrading drastically we developed an optimisation routine, which rebuilds a balanced tree online. To this end, we based our design on dynamization work by Overmars (1981) and Overmars and Leeuwen (1981). Overmars (1981) developed a dynamization scheme that transformed semi-dynamic data structures into dynamic ones by maintaining multiple static structures. They demonstrate that a new structure can be constructed and take over from the old one in a set number of updates, if enough processing time is available.

Our k-d tree dynamization process works as follows. The case base at most consists of 2 structures, OLD-MAIN and MAIN, however normally only MAIN exists (Overmars 1981). As updates occur the tree slowly becomes imbalanced. When the number of insertions and deletions become half the structures initial size we check if $l - \log_2 \beta > 3$, where l is the longest path length in the tree. When this becomes true, MAIN is made into OLD-MAIN and a new MAIN is started. At this point we assume MAIN had n_0 cases and that the new MAIN can take over from OLD-MAIN in λn_0 updates, where λ is some factor, which in this case should be less than half the size of the initial case-base ($\lambda < ½\ n_0$). Otherwise the structure may need to be rebalanced before it is rebuilt. In addition we have added the check $l - \log_2 \beta > 3$, so that the tree is only rebuilt when the current structure is actually out of balance. Overmars (1981) assumes the tree will need rebuilding when the number of updates reaches half its initial size, however this may not always be the case.

Building the new MAIN consists of two steps, building the tree and inserting buffered updates. For this process we use the following notation, where S is a structure containing n points (Overmars 1981). $I_S(n)$ is the time it takes to perform a weak insertion, $D_S(n)$ is the time it takes to perform a weak deletion and $P_S(n)$ = the time it takes to build a tree.

Step 1. For some time we will do the following:

- Continue to update OLD-MAIN, so we have an up to data structure.
- Spend $I_S(n_0 + \lambda n_0) + P_S(n_0) / \lambda n_0$ time building the new tree with every insertion and $D_S(n_0 + \lambda n_0) + P_S(n_0) / \lambda n_0$ time building the new tree with every deletion.
- Store each update in a buffer BUF. The updates are inserted into the new tree when it is built.

Step 2. For some more time, until BUF is empty we shall do the following:

- Continue to update OLD-MAIN, so we have an up to data structure.
- Spend $I_S(n_0 + \lambda n_0) + P_S(n_0) / \lambda n_0$ time processing updates in BUF with every insertion and $D_S(n_0 + \lambda n_0) + P_S(n_0) / \lambda n_0$ time processing updates in BUF with every deletion.
- Store each update in BUF if it is not empty, otherwise the update can be inserted in the new MAIN.

Once step 2 is complete the new main can take over from the old main, which is removed from memory. This process shows us how much time we need to spend building the data structure when an update occurs. However additional time can be spent building the tree when no updates are occurring. For example when there are no observable entities in range.

Basic Procedure for Rebuilding the k-d Tree

The tree rebuilding process is similar to the incremental building scheme outlined above. However rather than using a hybrid split policy the median split position of cases is found. The tree is rebuilt to a specified maximum bucket utilisation. For example an 80% bucket utilisation, where each bucket can store at most 10 cases would result in a data structure where each bucket had a maximum of 8 cases. Specifying a less than 100% bucket utilisations would reduce the impact of buffered and new cases on the tree, however it would increase the trees size and memory usage.

The basic recursive procedure for rebuilding the k-d tree is described below. Initially the method, BUILDKDTREE, is called to begin the process. This function creates a root node containing all the cases in the case base and calls a recursive algorithm, OPTIMISEKDTREE, which performs the rebuild. Every node within the tree represents a subset of the case base and the root node represents the whole case base.

Algorithm BUILDKDTREE (*C*)
Input: *C*, a set of cases.
Output: The root node of a k-d tree storing *C*.
1. Create new root node *V*.
2. *V*.ADD(*C*)
3. *V*.OPTIMISEKDTREE
4. **return** *V*

Algorithm OPTIMISEKDTREE()
Input: None.
Output: None.
Preconditions: *C*, a set of cases.
splitDimension, the current split dimension.
1. **if** *C* contains less than the maximum threshold
2. **then return**
3. /* Determine the new split dimension from the current split dimension. */
newDimension ← NEXTDIMENSION(*splitDimension*)
4. Split *C* into two subsets with a line *l* through the median coordinate on the dimension *splitDimension*, of the points in *C*. Let C_1 be the set of points to the left of *l* or on *l*, and let C_2 be the set of points to the right of *l*.
5. Compute the size *S* of the two subset regions created in step 4. Let S_1 be the region equal to and to the left of *l* and let S_2 be the region to the right of *l*.
6. left ← CREATENEWCHILD(R_1, C_1, *newDimension*, *l*)
7. right ← CREATENEWCHILD (R_2, C_2, *newDimension*, *l*)
8. left.OPTIMISEKDTREE()
9. right.OPTIMISEKDTREE()
10. **return**

The algorithm described above uses two sub-procedures NEXTDIMENSION and CREATENEWCHILD. The NEXTDIMENSION procedure determines which dimension should be used for the next split. In our system the dimensions are rotated however, other approaches (such as, the dimension with the widest distribution of points (Henrich *et al.* 1989)) can be used. The CREATENEWCHILD procedure generates a new instance of a node according to the passed values.

The most time consuming part of the algorithm is finding the median split position as it requires linear time. The algorithm built time T(*n*) for a set of *n* points is therefore:

$$T(n) = \begin{cases} O(1), \text{if } n = 1 \\ O(n) + 2T([n/2]), \text{if } n > 1 \end{cases}$$

Hence a tree for a set of *n* points can be constructed in O(*n* log *n*) time.

IMPLEMENTATION AND RESULTS

Our system is implemented in Java and follows closely the design outline here and in Hartley *et al.* (2005). Games agents are visualised in Quake 2 through the QASE API (Gorman *et al.* 2005), as illustrated in Figure 5. A detailed discussion of the full system implementation is reserved for future work. Here our experimentation focuses on evaluating the data structure and its performance to determine its capability for use in an online case based system such as ours.

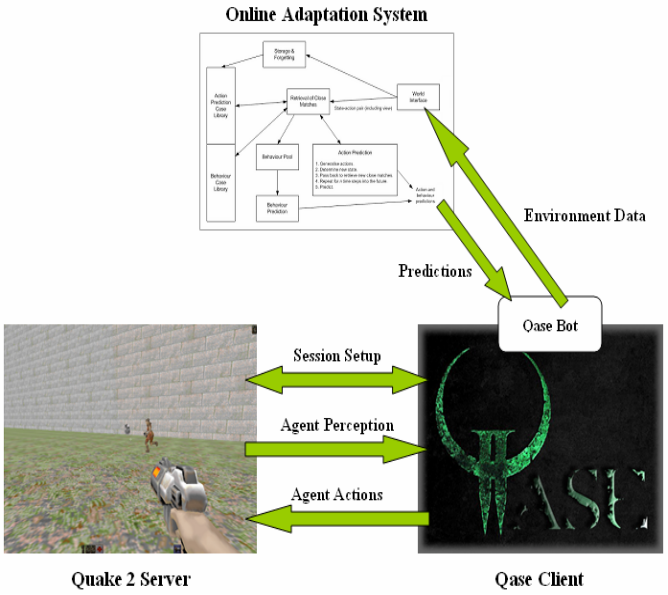

Figure 5: The proposed online adaptation system connected to Quake 2 through the QASE API (Gorman *et al.* 2005)

Since the development of the data structure was motivated by its ability to perform well online and adapt to cases, it is essential to analyze its performance in this area. Initial experiments focused on how our algorithms scale and optimisation times.

Table 1 outlines the relationship between the number of cases in a case base and the build time that is required to create a balance tree with the optimisation routine. A visual representation of our tree structure is pictured in Figure 6. Uniformly distributed 4-dimensional points were for our experiments. The points were presorted with respect to their distance to the point 0, 0. For all experiments the average time of 3 builds was taken, the maximum bucket capacity was set to 7 and the rebuild bucket capacity was set to 100% (i.e. 7 cases). A 3.0 GHz Pentium 4 processor was used, with 512MB of RAM.

Number of Cases	Average Build Time (ms)
500	41
1000	88
1500	156
2000	260
3000	542
5000	1526

Table 1: Build time to create a balanced tree. Where ms = milliseconds.

The results in table 1 indicate that the build time for 5000 cases is relatively large, about 1.5 seconds. By using the dynamization approach outlined in the previous section this time can be spread over multiple updates to the structure, thereby reducing its impact on the system. However dynamization would not be worthwhile when only a small number of cases exist (i.e. less than 500) as the build time is very fast. In addition our O(n log n) time construction analysis in the pervious section matches the results in table 1 quite closely.

The QASE API requires an agent to make a decision every 100 milliseconds (i.e. 1 frame), however a case would not need to be stored this often. Storing a case every 500 milliseconds seems an appropriate approach. If dynamization is used our initial results would appear to fit well within this time scale, however further implementation and analysis is required. For example prediction time and its CPU usage also needs to be taken into account.

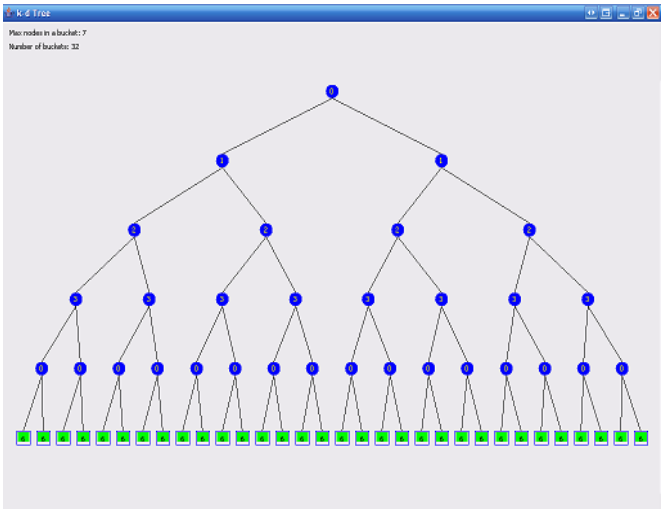

Figure 6: A visual representation of our k-d tree.

CONCLUSIONS

In this paper we have proposed a case storage and retrieval technique for our online prediction system. An adaptive k-d tree based approach was developed, which employs a hybrid split strategy to build trees online as cases are observed. An optimisation technique was also developed to rebuilds trees as they grow out of balance. The approach offers better adaptation to the distribution of cases than fixed partitioning, such as the technique used in Dinerstein and Egbert (2005), as the data structure built based on cases in the case base.

The initial experimental results show that the build time of trees is relatively slow and cannot be done every frame of a game loop. However when incorporated with the dynamization technique outline above this limitation is mitigated. Future work includes fully implementing the online prediction system and dynamization approach, including a more in-depth evaluation.

REFERENCES

Bentley, J. (1975) Multidimensional Binary Search Trees Used for Associative Searching, Communications of the ACM, 18 (9), 509-517.

Bentley, J. and Friedman, J. (1979) Data Structures for Range Searching, Computing Surveys, 11 (4), 397-409.

Dinerstein, J. and Egbert, P. (2005) Fast multi-level adaptation for interactive autonomous characters. ACM Trans. Graph. 24, 2 (Apr. 2005), 262-288.

Fagan, M. and Cunningham, P. (2003) Case-based recognition in computer games. Technical Report TCD-CS-2003-01 Trinity College Dublin Computer Science Department.

Fairclough C., Fagan, M., MacNamee, B. and Cunningham, P. (2001) Research Directions for AI in Computer Games. Technical report, Trinity College Dublin, 2001

Gaede, V. and G¨unther, O. (1998) Multidimensional access methods. ACM Computing Surveys, 30(2):170–231, 1998.

Gorman, B., Fredriksson, M., and Humphrys, M. (2005) QASE - An integrated API for imitation and general AI research in commercial computer games. In Proceedings of 7th *International conference on Computer games: Artificial intelligence, animation, mobile, educational, and serious games* (CGAMES), Angoulême, France.

Hartley, T., Mehdi, Q. and Gough N. (2005) *"Online Learning from Observation for Interactive Computer Games"*, 6th International Conference on Computer Games: AI and Mobile Systems CGAIMS 2005, July, Louisville, Kentucky, USA.

Henrich (1996) "A Hybrid Split Strategy For k-d-Tree Based Access Structures." 4th ACM Workshop on Advances in GIS. New York: ACM Press, 1996. 1-8.

Henrich, A., Six, H. and Widmayer, P. (1989) The LSD tree: Spatial access to multidimensional point and non-point data. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 45–53, Amsterdam, The Netherlands, August 1989.

Kerkez, B. and Cox, M. (2001) Incremental Case-Based Plan Recognition Using State Indices, Cased-based Reasoning Research and Development: Proceedings of 4th International Conference on Case-Based Reasoning (ICCBR 2001), Aha, D.W., Watson, I., Yang, Q. eds., pp. 291-305, Springer-Verlag, 2001.

Lent, M. and Laird, J (2001) Learning procedural knowledge through observation. In: International conference on Knowledge capture, Victoria, British Columbia, Canada, ACM Press (2001) 179–186.

Mao, W. and Gratch, J. (2004) Decision-Theoretic Approach to Plan Recognition, ICT Technical Report ICT-TR-01-2004.

Mitchell, T. (1997) Machine Learning. McGraw-Hill.

Overmars, M. (1981) Transforming semi-dynamic data structures into dynamic structures. Technical Report RUU-CS-81-10 Institute of Information and Computing Sciences, Utrecht University.

Overmars M. and Leeuwen J. (1981) Worst-case optimal insertion and deletion methods for decomposable searching problems. Inform. Process. Lett., 12(4):168-173, 1981.

Spronck P., Sprinkhuizen-Kuyper I. and Postma E. (2003). Online Adaptation of Game Opponent AI in Simulation and in Practice. *GAME-ON 2003 4th International Conference on Intelligent Games and Simulation* (eds. Quasim Medhi, Norman Gough and Stephane Natkin), pp. 93-100.