PATH-FINDING FOR LARGE SCALE MULTIPLAYER COMPUTER GAMES

Marc Lanctot Nicolas Ng Man Sun
Clark Verbrugge
School of Computer Science
McGill University, Montréal, Canada, H3A 2A7
marc.lanctot@mail.mcgill.ca nngman@cs.mcgill.ca clump@cs.mcgill.ca

KEYWORDS

Computer Games, Path-finding, Caching, Multiplayer

ABSTRACT

Path-finding consumes a significant amount of resources, especially in movement-intensive games such as (massively) multiplayer games. We investigate several path-finding techniques, and explore the impact on performance of workloads derived from real player movements in a multiplayer game. We find that a map-conforming, hierarchical path-finding strategy performs best, and in combination with caching optimizations can greatly reduce path-finding cost. Performance is dominated primarily by algorithm, and only to a lesser degree by workload variation. Understanding the real impact of path-finding techniques allows for refined testing and optimization of game design.

INTRODUCTION

Current, state-of-the-art approaches to path-finding in computer games incorporate multiple, hierarchical levels of searching and exploit a wide variety of searching heuristics. Our interest stems from a need to find good path-finding performance in the context of a research-based (massively) multiplayer environment. Large multiplayer games impose a general need to scale operations, whether computations are done on multiplayer servers or intelligent clients that manage multiple game entities, and so consideration of how well path-finding computations can be combined or reused is important. Other genre-specific properties may also affect performance of the system. These include the relatively large distances travelled (less dense points of interest), the presence of varying density of map obstacles, and the frequent use of teams, and thus the occasional use of strategized and collective movement. Understanding the influence of genre-specific properties of game workloads may also be important to efficient algorithm and design choices.

We investigate the performance of several path-finding implementation designs within the *Mammoth* multiplayer game research infrastructure. Using a selection of workloads from random data to player movement traces from a non-trivial multiplayer game we analyze the performance of different design and optimization choices in path-finding implementation. Our data shows the impressive effect of hierarchical approaches, particularly when underlying map data informs the hierarchy design, but also the great amount of opportunity for cache-based optimizations that exploit repetition in game player actions. Workload choice affects performance, but is generally overwhelmed by the algorithm performance.

Contributions of this work include:

- An experimental study of four path-finding approaches under different caching assumptions in a research multiplayer game framework.
- Our data and experimentation is based on analysis of real player movement data from a non-trivial multiplayer, team-based game.
- We provide an experimental comparison between three different forms of path-finding workload.

Below we present related work, followed by a description of our basic implementation environment and movement models. We then present our test methodology, the kinds of data we gathered under what scenarios. Discussion of the data is followed by conclusions and a description of further work.

RELATED WORK

Path-finding in computer games is commonly approached as a graph search problem. The world is decomposed, abstracted as a graph model, and searched, typically using some variant of IDA* (Korf 1985), based on the well-known A* (Hart et al. 1968) algorithm. Underlying world decompositions can have a significant impact on performance. Common approaches include those based on uniformly shaped grids, such as square or hexagonal tilings (Yap 2002), as well as the use of quadtrees (Chen et al. 1995; Davis 2000) or variable shaped tiles (Niederberger et al. 2004) for adaptivity to more arbitrary terrain boundaries. Properties of the decomposition, its regularity, convexity, or Voronoi guarantees, as well as geometric computations, such as visibility graphs, or even heuristic roadmap information (Kavraki et al. 1996) can then be used to improve search efficiency.

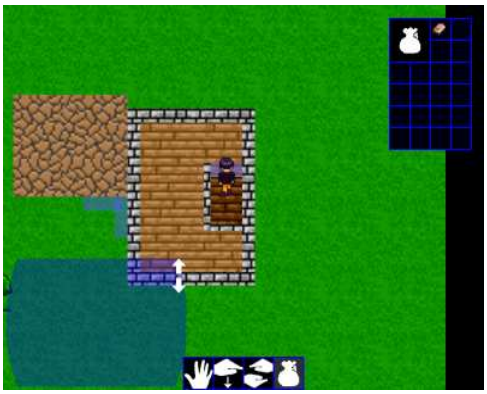


Figure 1: A screenshot of the Mammoth client.

Hierarchical path-finding incorporates multiple graph or search-space decompositions of different granularity as a way of reducing search cost, perhaps with some loss of optimality. Hierarchical information has been used to improve A* heuristics (Holte et al. 1996), and proposed in terms of using more abstract, meta-information already available in a map, such as doors, rooms, floors, departments (Maio and Rizzi 1994). Less domain-specific are graph reduction techniques based on recursively combining nodes into clusters to form a hierarchical structure (Sturtevant and Buro 2005). Our approach here is most closely based on the HPA* multi-level hierarchical design, where node clustering further considers the presence of collision-free paths between nodes (Botea et al. 2004).

Path-finding can also be based on the physics of dynamic character interaction. In strategies based on potential fields (Khatib 1985) or more complex steering behaviours (Reynolds 1999; Bayazit et al. 2002) a character's path is determined by its reaction to its environment. This reactive approach can be combined with search-based models to improve heuristic choices during searching (Pottinger 2000). There are many possible heuristics to exploit; in our implementations we use a "diagonal distance" metric to approximate the cost of unknown movement (Patel 2003).

IMPLEMENTATION ENVIRONMENT

Environment

The environment used for implementation was Mammoth (McGill University 2005): a Java-based 2D overhead-view multi-player game and research framework. The main goal of the ongoing Mammoth project is to provide a common platform to facilitate the implementation of research experiments. A screenshot is shown in Figure 1.

Mammoth allows human players to connect to and immerse themselves into a fairly large and intricate virtual environment (the world). Within the Mammoth world players can explore the physical setting (the map) and interact with world objects: static objects (walls, trees) and dynamic objects (items, other players). All objects are represented as polygons. The world is partitioned into irregularly-shaped regions called *zones*. Physically, each zone corresponds to a collection of world objects. Virtually, each zone determines the objects that clients must be aware of given the player's current location; a typical zone in Mammoth would be a room within a building. Zones are connected via *transition gates*: physical entities which act as the entry and exit points between zones.

Figure 2: A wireframe view of the Mammoth map. The color of each rectangle indicates its type: white for a simple texture/image, red for a static object, blue for a player, and green for an item.

The coordinate system used in the Mammoth map is $(x, y) \in [0, 30] \times [0, 30] \subset \Re^2$; the origin $(x, y) = (0, 0)$ is graphically placed at the bottom-left of the space. Each world object has a *position*: a pair of real values (x and y), a zone ID, and a *stairlevel* corresponding to its distance in discrete levels relative to the ground (stairlevel 0). Figure 2 gives a visualization of the map.

Movement and Pathfinding

Player movement in Mammoth can be effected in one of two main fashions: a basic movement scheme where the player has fine-grained control, and a more complex system based on algorithmic path-finding. The former allows for complete and highly-intentional control, and is the main method used for movement in our initial game implementation. This basic movement model is performed by the player selecting a destination point on the map through a mouse click. The player then

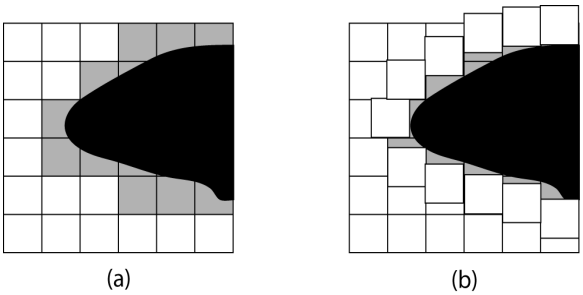


Figure 3: Variable grid level

moves in a straight-line path towards their destination, stopping when they reach their destination or become blocked by an obstacle. Collision detection is performed by checking if the player's new position leads to an overlap between the player's shape and an obstacle's shape. Maximal control is allowed by letting a player set a new destination at any time along its currently-planned path, in which case the player immediately starts moving towards the new destination instead.

The implementation of the path-finding algorithm is based on the classic A* algorithm (Hart et al. 1968). Our design operates on 2 levels, following the general ideas of a hierarchical path-finding system as outlined by Botea et al. (Botea et al. 2004).

At the lower or "grid" level, the game world is effectively discretized into a small grid. This is performed using the concept of a "ghost player." The ghost player is a special entity maintained by Mammoth's physics engine: it abides by the same physical rules as any regular player except that it is invisible and only exists temporarily. The A* algorithm searches the world by moving the ghost player around in discrete steps. This method has a few advantages over a regular grid partitioning approach: 1) extra memory is only required for the region that is currently being explored and not for the entire map, 2) dynamic map changes are easily accommodated, and 3) the granularity of the steps with which the ghost player moves can be adjusted to accommodate the density of objects within a zone or to more closely mesh with the boundaries of non-axis-aligned obstacles, as shown in Figure 3.

Since our map is set in a urban environment, the presence of rooms with doorways or staircases provides a natural way to decompose the world at a higher level. We analyze the transition gates already present in the Mammoth map and use that information to derive a connectivity graph where each node is a zone and transition gates establish edges between them. This approach is particular well-suited for our work since our lower grid-based level is never explicitly generated or kept for the entire map, and thus cannot be the basis for a higher level abstraction as is done in other hierarchical pathfinding approaches.

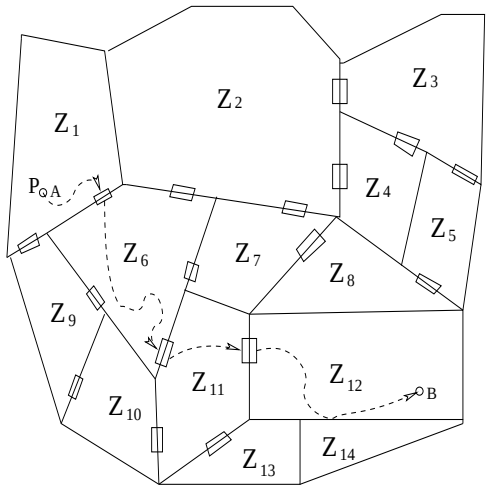


Figure 4: An example path calculation. Dotted curves represent grid-level paths.

Pathfinding at the zone-level occurs by first connecting the start position *waypoints* or transition gates to the zone-level graph, searching the graph using the A* algorithm, and finally connecting the path to the final destination point.

The following example is illustrated in Figure 4. Suppose player P at position A, in zone Z_1, wants to travel to position B, in zone Z_{12}. The zone-level path is found to be $(Z_1, Z_6, Z_{11}, Z_{12})$. Then, individual grid-level paths are gradually resolved between the source, transition gates, and destination.

As an acceleration technique, once a player leaves a zone, that zone is marked as blocked and will not be searched while refining the path to the next waypoint. This helps to further trim down the number of unnecessary nodes that A* has to explore before finding a path.

In most games, and especially in a multiplayer environment, players tend to traverse paths that have been previously traversed. Caching is thus expected to have a large impact on performance. We used 2 main types of data caching:

- Path caching at the grid level. Our approach works in the following way: for each point on a computed path, we only store the next point on the way to the destination point. For example, to go from point P_1 to P_5, we only need to know at P_1 that our next move should be P_2, then at P_2 move to P_3, and so on. A further improvement is done by abstracting every position at the grid-level to a larger grid so that several grid-level positions will actually map to the same cache node. The combination of these 2 techniques can help to reduce memory requirement and at the same time increase cache hit ratio compared to a more brute-force approach of storing complete paths at each point. Figure 5 illustrates how the grid path cache helps A* in practice.

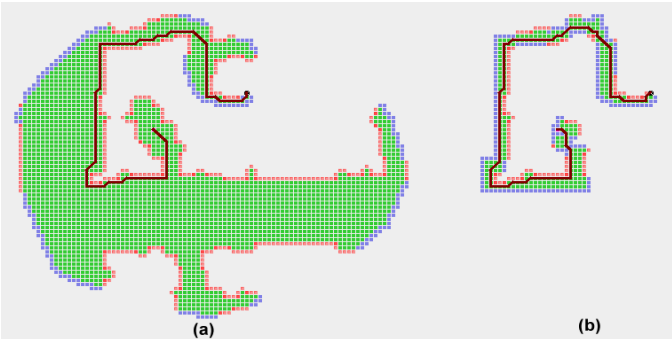


Figure 5: a) Exhaustive A* search without path cache b) Once cached, searching for a path between the same points is much more focused.

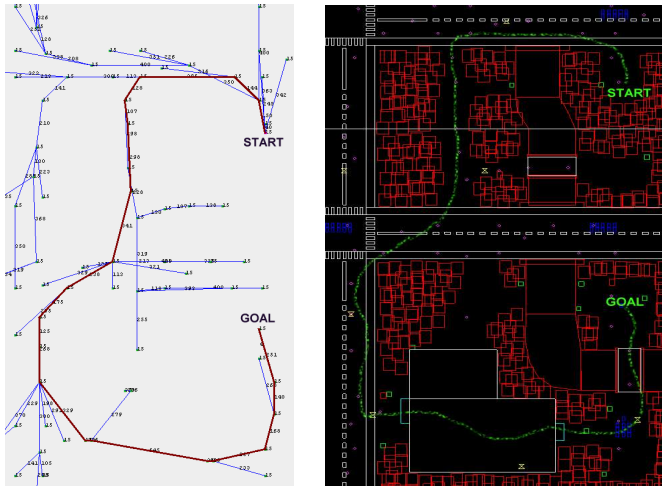
- Collision caching. Every move made at the grid level requires checking for collision. Querying the Mammoth architecture for this is expensive. By caching this information for the static environment, we expect to see a significant saving in time.

We used existing, pre-defined zones to build an abstract, high-level connectivity graph. This has the disadvantage that since the outside area in our map was mainly represented as one large zone only the lower-level, grid-based pathfinding is used for a large part of the map. We have thus also investigated the use of a roadmap planner based on actual player movement, as sampled from real gameplay in our environment. Roadmap connectivity is built by performing visibility checks between the most commonly-occurring sample points. The roadmap is attached to exit waypoints of all buildings, integrating with the existing hierarchial approach and serving as a middle layer between the zone-level and grid-level. Figure 6 shows the hierarchy at work.

A final variation in hierarchy is to build a more balanced upper-level decomposition that has some relation to expected player movement. Based on the *interest points* gathered during the course of our workload generation (see next Section) we thus construct a Voronoi diagram, and use the computed regions for our "Voronoi zoning" scheme. Actual waypoints for these zones are computed from the midpoints of the edges in the corresponding Delaunay triangulation, and are intended to represent locations where players may enter/exit popular regions.

METHODOLOGY

We conducted several experiments under different movement model assumptions, and investigated the performance as it varied due to different workloads. Three basic workloads are used during our simulations, two of which are based on points automatically recorded from real player movements. In the first set, interest points are chosen from a uniform distribution of map coordinates; this represents data easy to acquire/generate, but



(a) (b)

Figure 6: An example path found in our simulations showing a) the zone-level path generated and b) the actual path taken by the player

quite artificial, and thus potentially inaccurate. The other two sets of interest points are from actual gameplay: *Operation: Orbius* was organized to collect data from players during several multi-player gaming sessions, and is discussed in detail below. One data set is from an abstract model of Orbius game-play, where interest points are mainly defined from a relatively small list of the most travelled locations over a series of games. The third set represents the actual paths of players in the game, constructed from the Orbius game movements.

Measurements

We use several metrics to evaluate the quality of the different pathfinding techniques.

- total time taken: a quick measure of performance.
- total distance traveled: a measure of path optimality. The grid based approach should always return the shortest path because it computes globally optimal solutions, while greater use of the hierarchy should imply less optimal paths. It is interesting to compare the penalty in path optimality against the gain in time.
- number of nodes explored: a measure of the efficiency of the algorithm. Fewer nodes usually implies smaller total time and smaller memory usage as well. We expect collision caching to improve the total time without actually changing number of nodes explored, while path caching should do the same but by reducing the number of nodes explored. A more accurate heuristic should further reduce the number of explored nodes; a high-level abstraction that accurately captures the topology of the map should help

Figure 7: A screenshot of two adversaries tickling each other, several orbs lying on the ground, and a red team's base during *Operation: Orbius*

make the low-level search more efficient.

- average delays before the player starts moving: measures how responsive the game is after the player issues a pathfinding query. This can help evaluate player satisfaction.

Orbius

Orbius is a Mammoth sub-game: a goal-oriented game played within the Mammoth world with other participating players. Orbius is a team-based subgame in which players collude in attempt to win before every other team. The game is designed to reflect the generally understood behaviour of larger-scale multiplayer games: players are grouped (teams), map exploration is critical, both constrained (city, interior) and unconstrained (outdoor) movement areas are present, and different map locations have different levels of interest to players. A screenshot of Orbius is presented in Figure 7. Twenty-four (24) game players participated in the *Operation: Orbius* event. In total, 5 games sessions were played, each having 6 teams of 4 players. Teams were not changed between games to encourage natural strategy development between players. Two types of data were logged during the game-playing: a player's set destination action, and a player's actual move update at each time step. For each action, the game server logged the current time, the action type, the player ID, the player's new position in the case of a move update or the player's current position and selected destination otherwise.

Orbius-Based Data

Interest points chosen randomly on the Mammoth map may or may not correspond with map locations actually visited by real players. Actual game behaviour may in general bias movement, and thus performance. To consider this bias in our workload we not only analyze random-generated path data, but also data derived from a model of player movement, and data from a set of actual player paths.

Our movement model is intended to reflect common player movements, and so we determined which map areas were most frequently traversed. The game space is discretized (as for grid-level pathfinding), and an *interest grade* representing the total number of times that each grid cell was occupied during gameplay was calculated. To further generalize the data interest maps are then passed through a series of transformations: *blurring*, *localization*, and *average composition*. After each transformation, the interest grade values are normalized.

- Blurring simultaneously sets each interest grade to the average of its immediate cell neighborhood, including diagonal neighbours.

- Localization simultaneously sets each interest grade to itself plus the sum of "neighboring influences". Here, a "neighboring influence" is the interest grade of a nearby grid cell weighted inversely by the distance between the two cells, up to a maximum considered distance.

- Averaging allows us to combine experimental data from different runs. For each grid cell in our final output grid its interest grade is the average of the interest grades from the corresponding grid cells of the original game runs.

The highest-valued interest points indicate the most often travelled areas, and form the basis of our modeling workload (and our Voronoi zoning) approach. The top points are selected considering a minimum spread distance between interest points; these then form the set of possible start/end path destinations during random path generation.

Paths actually derived from the Orbius data are our most closely representative movement data. For this the paths were derived in the following manner. For each player we find the mean (μ) and standard deviation (σ) of the intervals between that player's successive move updates. Intervals (Δt) which are sufficiently above the expected value ($\Delta t > T$) are considered "stop points"; two successive start/end position pairs represent a path taken by the player. The threshold value is arbitraily chosen to be $T = \mu + k\sigma$, with $k = 1$ in our experiments. The connected update segments are then used to build our third workload path data set.

EXPERIMENTAL ANALYSIS

To measure performance, we used a modified (isolated) version of the Mammoth stand-alone client. A single player was setup and had to move to 100 destination

Run	Δt (sec)	MV	SD
1	899	98619	10970
2	903	202563	32526
3	740	146142	34878
4	894	124613	54414
5	798	175435	60024
Average	849	149474	38562

Table 1: Summary of the collected movement data. Listed are the run number, elapsed time, number of movement updates (MD), and the number of set destination actions (SD).

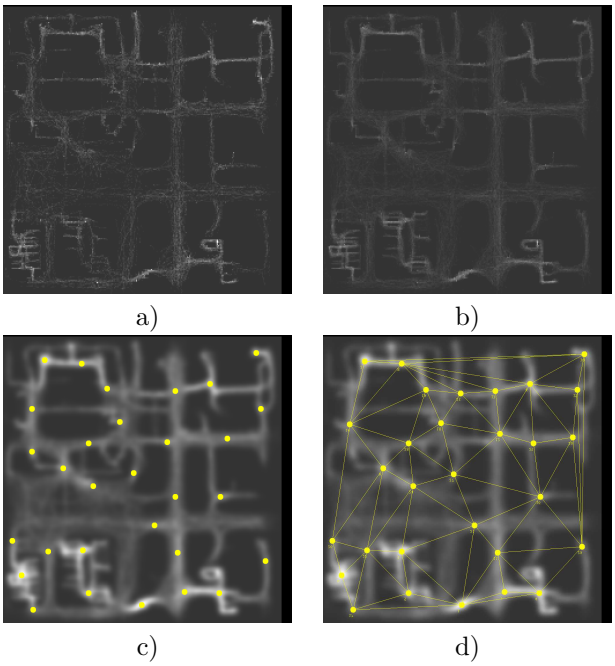


Figure 8: The interest map a) generated from the fourth run b) obtained by the average composition of all usable runs c) processed average composition including interest points, d) processed average composition with Delaunay triangulation of interest points

points using different pathfinding algorithms and under different cache parameters.

A summary of the data gathered from *Operation: Orbius* is given in Table 1. During the first and second games, most players were still getting familiar with the game. Therefore to avoid biased results only runs 3-5 are used for analyses. Figure 8 shows several interest maps and the derived interest points from some of the movement data.

Path-finding Results

Three forms of path workload were considered. RANDOM: a basic workload derived from randomly chosen start and end destinations, MODELED: a workload based on paths chosen from random Orbius points of interest (shown in Figure 9), and EXTRACTED: a workload consisting of paths extracted from the individual player movements in Orbius. These inputs provide a set of workloads intended to be progressive in accuracy, and difficulty of acquisition. Cache sizes were set to 1Meg, and did not fill up in our tests—this data represents the results of an ideal cache environment, with no collisions. All tests were performed on an 8-way Xeon MP 2.7GHz, 8Gig of RAM, using Sun JDK-1.5.0_03 under Gentoo. Tables 2, 3, and 4 show a breakdown of the test results for our three path-finding implementations based on static zoning (SZ), no hierarchy (GR), use of roadmap (RD), and Voronoi zoning (VZ), either alone or in com-

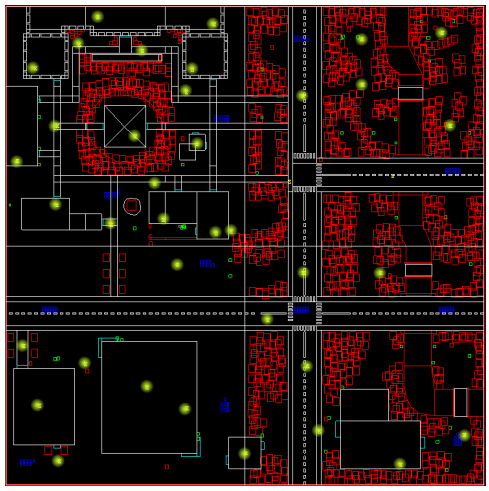


Figure 9: Sample of the destination points used for performance tests.

Test	Nodes/ Search	Dist	Total Time	Speed units/ms	Delay ms
SZ	240	2088	517.3s	0.4	1469
SZ+CC	241	2088	221.3s	0.94	628
SZ+CC+PC	230	2102	212.2s	0.99	602
SZ+CC +sPC	195	2143	150.2s	1.42	424
GR	2031	2076	1585.7s	0.13	15857
GR+CC	2146	2053	660.3s	0.31	6603
GR+CC+PC	1979	2062	618.3s	0.33	6183
GR+CC +sPC	1753	2069	482.4s	0.42	4824
RD	197	2114	936.4s	0.22	931
RD+CC	185	2130	489.2s	0.43	486
VZ	332	2317	2195s	0.1	2469
VZ+CC	332	2317	749.5s	0.3	843
VZ+CC+PC	230	2399	559.3s	0.42	591

Table 2: RANDOM: Benchmark data with random start and destination points. The leftmost column gives the experimental environment. Other columns include average number of nodes per search, total distance of computed paths in game units, total time, distance searched per time unit (speed), and average delay for a path calculation.

Test	Nodes/ Search	Dist	Total Time	Speed units/ms	Delay ms
SZ	213	2057	490.7s	0.41	1268
SZ+CC	213	2058	201.8s	1.01	521
SZ+CC+PC	173	2074	159.9s	1.29	403
SZ+CC +sPC	138	2114	95.2s	2.21	234
GR	1758	2019	1338.9s	0.15	13389
GR+CC	1778	2018	463.7s	0.43	4637
GR+CC+PC	1262	2041	313.3s	0.65	3133
GR+CC +sPC	963	2060	191.7s	1.07	1917
RD	108	2162	451.7s	0.47	442
RC+CC	98	2162	215.2s	1	211
VZ	587	2182	3940.6s	0.05	4663
VZ+CC	587	2181	1244.1s	0.17	1472
VZ+CC+PC	443	2286	978.4s	0.23	1108

Table 3: Modeled: Benchmark data with start and destination points selected from Orbius data (outside points), along with some random interior points.

Test	Nodes/ Search	Dist	Total Time	Speed units/ms	Delay ms
SZ	386	2052	966.8s	0.21	2222
SZ+CC	503	2051	551.3s	0.37	1388
SZ+CC+PC	534	2052	595.9s	0.34	1475
SZ+CC+ sPC	273	2052	203.9s	1	635
GR	1527	2050	1598.5s	0.12	11841
GR+CC	1887	2062	768.4s	0.26	5528
GR+CC+PC	1267	2053	437.7s	0.46	3647
GR+CC+ sPC	932	2063	292.4s	0.7	2249
RD	186	2050	1064.1s	0.19	1019
RD+CC	196	2065	547.3s	0.37	526
VZ	599	2052	3901.4s	0.05	4763
VZ+CC	461	2053	930.2s	0.22	1177
VZ+CC+PC	588	2066	1298.5s	0.15	1603

Table 4: Extracted: Data when paths are extracted from the Orbius movement data.

bination with collision caching (CC), path caching (PC), or saturated path caching (sPC). The latter adds the presumption of a partially-filled cache at the start of testing.

Under all workloads a single level A* approach, as observed by using only the grid level (GR), performs much slower than static zoning or roadmap approaches. Caching improves overall time to closer to that of static zoning, although response time remains objectively quite high—far more nodes are searched for a single-level path-finding approach. The GR approach usually returns the shortest paths. The improvement in path quality (distance) over the other approaches is not large, however, and the difference between GR and the best variation of SZ is quite marginal in all situations.

The Voronoi zoning scheme performed the slowest. This is perhaps largely due to the difference in terrain conformance between the Voronoi and static zoning models: static zones respect building and other boundaries, while Voronoi zones largely ignore the structure of the underlying map. A given Voronoi zone may actually contain a maze of obstacles making navigation through it very expensive or perhaps even impossible. A greater emphasis on connectivity, including precalculation of efficient cross-zone paths, as well as use of a *constrained* Voronoi diagram, better respecting map obstacles would greatly improve performance.

The effects of the collision cache (CC) are visible in all three sets of experiments. We observe significant speedups, with total time reducing by a factor of 1.8 to 4.2, depending on the type of path-finding approach used and workload applied. The specific magnitude of this reduction does of course strongly depend on the cost of collision detection, but mirrors the expected density of collisions the different algorithms would encounter. Roadmaps by nature avoid collisions due to being based on actually travelled routes, and thus benefit the least. Voronoi makes the most use of the CC; collisions are more frequent, again due to the relative lack of map conformance in the zones.

The benefits of the path cache (PC) are also noticeable, if less drastic. Path caching reduces the number of nodes expanded by A*, and this translates into a further 1.0 to 1.5 factor reduction in total time. To get a better appreciation of the path cache under long-term usage, we performed a further experiment where the cache was partially preloaded with values from random pathfinding queries (sPC). When there is existing data to exploit performance is even more improved, a factor of 1.4 to 2.4 over plain collision caching.

Without path caching the roadmap extension to the hierarchical approach yields the best overall results, at a potential small cost to path optimality as observed by the increase in total distance travelled. With path caching static zoning shows significant further improvements under all workloads, more so if the path cache is primed or already partially-filled.

Workload Differences

The effect of different workloads can be seen in the three data Tables. In a general sense actual player movements seem to be more complex and less predictable (ie less cache-able) than more artificial data. The node searches in the Extracted data set are much larger than in Modeled or Random, and this results in larger total times as well. This can have a noticeable impact; in the case of Extracted path caching overhead is sufficient to cause a reduction in performance when introduced naively in the SZ and VZ cases. Once the cache is more filled cache hits more than balance out the overhead.

Performance of the caching and of the individual algorithms, except Voronoi, is overall best on the Modeled

data. This reflects the nature of the generated workload. Only 30 source and destination points are used for the paths based on the MODELED data, whereas EXTRACTED paths are based on a much larger set of player coordinates, and RANDOM paths are drawn from any map point. A smaller, more controlled and well-balanced sample space has a better chance of permitting cache hits in our caches, and the underlying machine's as well.

Interestingly, the impact of the hierarchy is greater on RANDOM data, with progressively less relative impact on the MODELED and EXTRACTED sets. This also mirrors the structure of the input data. Random data points are well-distributed, and thus make good use of the hierarchy; SZ is over 3 times faster than GR. Orbius interest points are also reasonably well-spaced on the map, but many are outside in the single large outdoor zone, and the hierarchical gain is reduced to between 2.0 and 2.7 depending on cache choices. Extracted paths correspond to the actual game-play of Orbius, and so are primarily outside, reducing the gain to between 1.4 and 1.7.

CONCLUSIONS & FUTURE WORK

Workload experiments show the difference in scale and variation an algorithm may experience. Here, surprisingly, while there are significant differences a randomized model is reasonably accurate, at least when considering the relative performance of algorithms. For path-finding the workload choice is not a dominant one, and algorithm design is much more important. Hierarchical implementations are unsurprisingly best, but only if reasonably well tailored to the underlying map, and with some dependence on the choice of measurement workload.

We have attempted to incorporate genre or game-specific behaviour into understanding the behaviour of different path-finding approaches. There are many other game and path-finding aspects worth considering. Larger and different kinds of maps, different games, and so forth would be interesting to pursue. We are interested in the effect of dynamic collisions, re-pathing and collision avoidance on path-finding efficiency. Adapting algorithm usage to to high level changes in game strategy, game "phases," may also help improve and further scale performance.

ACKNOWLEDGMENTS

This research has been supported by the National Science and Engineering Research Council of Canada.

REFERENCES

Bayazit, O. B., Lien, J.-M., and Amato, N. M. (2002). Roadmap-based flocking for complex environments. In *The Pacific Conference on Computer Graphics and App. (PG)*, pages 104–113.

Botea, A., Muller, M., and Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28.

Chen, D. Z., Szczerba, R. J., and Jr, J. J. U. (1995). Planning conditional shortest paths through an unknown environment: a framed-quadtree approach. In *IEEE/RJS International Conference on Intelligent Robots and Systems (IROS 95)*, volume 3, pages 33–38.

Davis, I. L. (2000). Warp speed: Path planning for star trek armada. In *AAAI 2000 Spring Symposium on Interactive Entertainment and AI*, pages 18–21.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4 (2)*, pages 100–107.

Holte, R. C., Perez, M. B., Zimmer, R. M., and MacDonald, A. J. (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *The Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 530–535.

Kavraki, L. E., Svestka, P., Latombe, J. C., and H.Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics & Automation*, pages 566–580.

Khatib, O. (1985). Real-time obstacle avoidance for manipulators and mobile robots. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 500–505.

Korf, R. (1985). Depth-first iterative deepening: An optimal admissible tree search. In *Artificial Intelligence*, pages 97–109.

Maio, D. and Rizzi, S. (1994). A hybrid approach to path planning in autonomous agents. In *Second International Conference on Expert Systems for Development*, pages 222–227.

McGill University (2005). Mammoth: The massively multiplayer prototype. `http://mammoth.cs.mcgill.ca`.

Niederberger, C., Radovic, D., and Gross, M. (2004). Generic path planning for real-time applications. In *Computer Graphics International (CGI'04)*, pages 299–306.

Patel, A. (2003). Amit's thoughts on path-finding and A-star. `http://theory.stanford.edu/~amitp/GameProgramming/`.

Pottinger, D. C. (2000). Terrain analysis in realtime strategy games. In *Computer Game Developers Conference*.

Reynolds, C. (1999). Steering behaviors for autonomous characters. In *Computer Game Developers Conference*, pages 763–782.

Sturtevant, N. and Buro, M. (2005). Partial pathfinding using map abstraction and refinement. In *The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*.

Yap, P. (2002). Grid-based path-finding. In *AI '02: Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pages 44–55. Springer-Verlag.