ORTS: A Hack-Free RTS Game Environment

Michael Buro

NEC Research Institute, Princeton NJ 08540, USA
mic@research.nj.nec.com

Abstract. This paper presents a novel approach to Real-Time-Strategy (RTS) gaming which allows human players as well as machines to compete in a hack-free environment. The main idea is to replace popular but inherently insecure client-side game simulations by a secure server-side game simulation. Only visible parts of the game state are sent to the respective clients. Client-side hacking is therefore impossible and players are free to choose any client software they please. We discuss performance issues arising from server-side simulation and present ORTS – an open RTS game toolkit. This software package provides efficient C++ implementations for 2D object motion and collision detection, visibility computation, and incremental server-client data synchronization, as well as connectivity to the Generic Game Server (GGS). It is therefore well suited as a platform for RTS related A.I. research.

Keywords: Real-time-strategy game, peer-to-peer, server, client, latency, multi-player

1 Introduction

Real-time strategy (RTS) games have become very popular over the past couple of years. Unlike classic board games which are turn-based, RTS games are fast paced and require managing units and resources in real-time. An important element of RTS games is incomplete information: players do not know where enemy units are located and what opponents plan, unless they send out scouts to find out. Incomplete information increases the entertainment value and complexity of games. The most popular RTS titles so far have been the million-sellers WarCraft-II and StarCraft by Blizzard Entertainment and Age of Empires series by Ensemble Studios. These games offer a wide range of unit and building types, technology trees, multi-player modes, diverse maps and challenging single-player missions. From the A.I. research perspective the situation looks ideal: playing RTS games well requires mastering many challenging problems such as resource allocation, spatial reasoning, and real-time adversarial planning. Having access to a large population of human expert players helps to gauge the strength of A.I. systems in this area – which currently leaves a lot to be desired – and inspires competition. The commercial success of RTS games, however, comes at a price. In order to protect their intellectual property, games companies are disinclined to publish their communication protocols or to incorporate A.I. interfaces into their products which would allow researchers to connect their programs to

compete with peers and human experts. Another reason for keeping game software closed is the fear of hackers caused by software design concessions. Due to minimal hardware requirements the game companies want to meet, the aforementioned commercial RTS games rely on client-side simulations and peer-to-peer networking for communicating player actions. This approach reduces data rate requirements but is prone to client hacking – such as revealing the entire playing field – and threatens the commercial success of on-line gaming as a whole. The ORTS project [2] deals with those two problems: it utilizes an open communication protocol – allowing players and researchers to connect any software client they wish – and implements a secure RTS game environment in which client hacking is impossible. ORTS is an open software project which is licensed under the GNU Public License to give users the opportunity to analyze the code and to share improvements and extensions.

The remainder of this paper is organized as follows: Section 2 presents the ideas behind server-side RTS simulations and takes a detailed look at information hiding and data rate requirements. Section 3 deals with implementation issues and discusses the ORTS kinematics model, visibility computation, and server-client data update. A summary and outlook section wraps-up the article.

2 A Server-Client RTS Game Model

Popular RTS games utilize the client-side game simulation described in Fig. 1. A detailed description of this approach – including various optimizations that mask client latency – can be found in [1]. Clients run game simulations and only transmit user actions to a central server or directly to their peers. At all times game states are synchronized and known to all clients, regardless of what is visible to the respective player. The client software hides information not meant for the respective player. An alternative approach is presented in Fig. 2. Here,

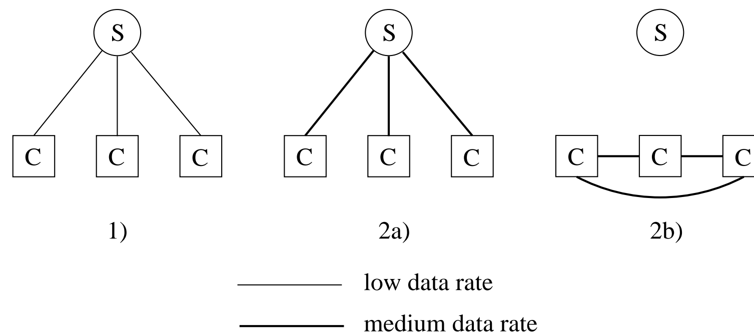

Fig. 1. Common client-side game simulation: clients first establish contact to a central server (1). When a game is created, all clients start simulating world changes simultaneously and send issued user commands either back to the server (2a), which broadcasts them to all other clients, or directly to their peers (2b) using a ring or clique topology. The data rate requirements are modest if the number of players and the number of commands they issue are small.

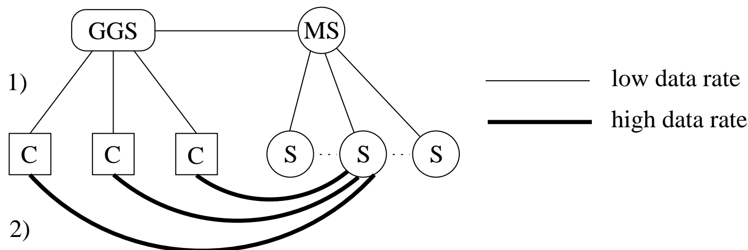

Fig. 2. ORTS server-side game simulation: clients and a master server (MS) are connected to a central server – GGS in this case – and several worker servers (S) are connected to the master server (1). When a game is created on MS, it schedules an idle worker server on which the game is to be simulated. This server then connects to the clients to send the individual game views and to receive unit commands (2).

a central server runs the simulation and transmits only the part of the game state the respective client is entitled to know. In the following two subsections we compare both models with regard to information hiding and data rate issues.

2.1 Information Hiding

Playing games with incomplete information at a competitive level on the internet demands that information can be physically hidden from clients. To see this one just needs to imagine a commercial poker server that sends out all hands to each client. Hacking and losing business is inevitable. Translated into the RTS world, information hiding rules out client-side simulations in theory. Regardless of how clever data is hidden in the client, the user is in total control of the software running at her side and has many tools available to reverse engineer the message protocol and information hiding measures, finally allowing her to reveal "secret" game state information during a game. [5] discusses means of thwarting such hacking attempts. Despite all the efforts to secure clients, hacks – which spoil the on-line game experience – are known for all popular RTS games based on client-side simulation. Perhaps encryption and information hiding schemes that change with every game and require analysis that takes longer than the game itself lead to a practical solution to the information hiding problem. However, uncertainty remains and we argue that game designers do not even have to fight this battle anymore when up-to-date hardware is used.

2.2 Data Rate Analysis & Measurements

Synchronized client-side simulations require only player commands to be transmitted to peers, which keeps the data rate low if only a few commands are generated during each frame and the number of players is small. Independent of the chosen communication topology and player views during the game, the

client input data rate $d^{(\mathrm{in})}$ – measured in bytes/frame – depends on the number of participating players and the rate they issue commands:

$$d_k^{(\mathrm{in})} = \sum_{i=1, i \neq k}^{n} d_i^{(\mathrm{out})}. \tag{1}$$

In the server-side simulation model the input data rate grows linearly in the number of objects the players see in the current frame:

$$d_k^{(\mathrm{in})} = D \cdot \#\text{objects visible to player } k, \tag{2}$$

where D is the average data rate generated by one visible object. To compare the i/o requirements of both models we look at the extreme cases: 1) small vs. large number of players and unit commands per frame and 2) overlapping vs. disjoint player views. Client-side simulation has a lower input data rate requirement if a small number of players only generate few commands during each frame. Server-side simulation excels if the number of players and unit commands is high and the player views are mostly disjoint.

Data rates in the server-side simulation model can be decreased by incremental updates, compression, and partial client-side simulation of visible objects. The empirical results presented in Fig. 3 indicate that even without partial simulation it is possible to play RTS games at 5 frames/sec featuring hundreds of visible objects over conventional DSL or cable modem lines.

For the tests the motion of up to 1500 circular objects were simulated. In the initial configuration objects of diameter 16 were distributed evenly on an empty 800x800 playing field. Then object ownership was assigned randomly to two players. During the game both players picked new random headings for their objects whenever they collided. All objects had constant speed of 4/frame and a

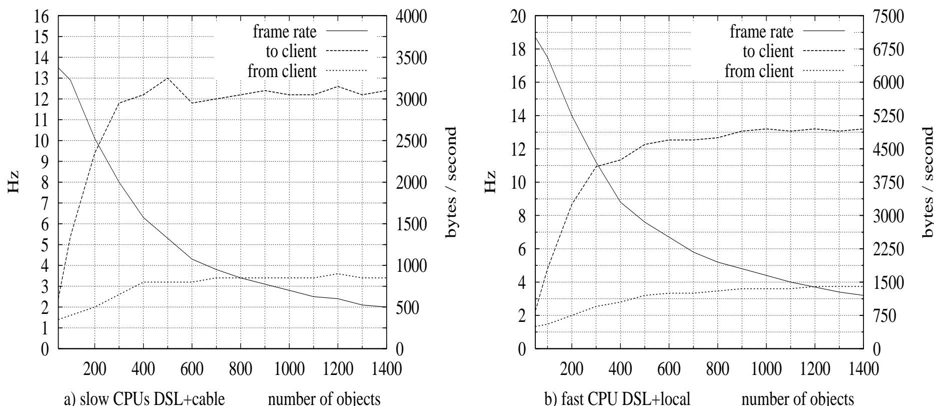

Fig. 3. Data and frame rates dependent on the number of objects. Slow (500 MHz) client CPUs, DSL and cable modem connection with 80 msec ping time (a). Fast (850+ MHz) client CPUs, DSL and local connection (b).

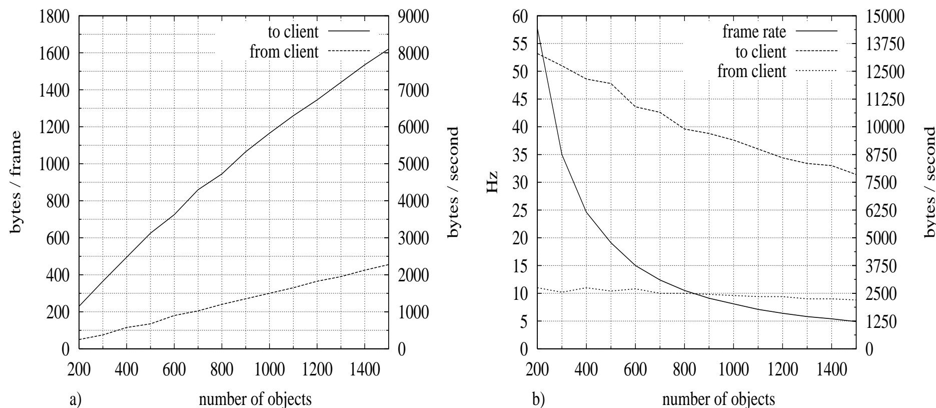

Fig. 4. Data and frame rates dependent on the number of objects: Bytes per frame and bytes per second at a fixed simulation rate of 5 Hz (a). Total frame and data rates measured on a dual Pentium-3/933 system for the entire simulation including object motion, collision test, (de)compression, and data transmission (b).

sight range of 60 which ensured that a large fraction of enemy units was visible at all times (Fig. 7). All experiments were conducted on a dual Pentium-3/933 system under Linux. Fig. 4a) shows the generated data rates dependent on the number of objects when using message compression. In average approximately 1.2 bytes per visible object per frame is sent to each client and ca. 0.6 bytes per own object per frame is returned to the server. The details of the utilized compression scheme are described in the next section. Fixed at 5 frames/sec the resulting data rates can be handled by current DSL and cable modem technology. Neglecting latency it is even possible to run a server for a two-player game with up to 400 visible objects on systems with 6 KB/sec upload data rate. The graphs in Fig. 4b) take server and client CPU load latencies into account. They show the total performance and data rates when two clients and a server are running on the same dual processor machine. The frame rate drops from 57 Hz when 200 objects are simulated to 5 Hz in case of 1500 objects. During these experiments the CPU loads for the server and both clients stayed around 60%/40%/40%. To increase the frame rate, latency caused by simulation, message (de)compression, and data transmission has to be minimized. The current implementation is not well optimized. In particular, data (de)compression and transmission use stream and string classes which slow down computation by allocating heap memory dynamically. Another approach for increasing the frame rate at the cost of command latency is to delay actions by a constant number of frames [1]. This allows the server to continue its simulation after sending out the current game views without waiting for action responses. Whether built-in command latency is tolerable is game dependent. Currently, this technique is not employed in ORTS.

To check how server-side simulation performs in conjunction with smaller channel capacities and higher latencies we ran two external clients on slower

(500 MHz) machines which transmitted data over a 128/768-kBaud DSL and a 240/3200-kBaud cable modem line. The frame rates we measured are shown in Fig. 3. Apparently, latency caused by transmission and slower client side computation rather than the available data rate is the bottleneck in this setting. Nevertheless, we can conclude that playing RTS games in a hack-free environment featuring hundreds of moving objects is possible using up-to-date communication and PC equipment. It is important to note that the reported frame and data rates are lower bounds because in actual RTS games player views usually do not coincide (Fig. 7). On the other hand, data rates will increase if features are added and more object actions become available. However, the data rate increase is expected to be moderate because compression routines have access to all feature vectors and the entire action vector and can therefore exploit repetitions.

3 Implementation Issues

3.1 ORTS Kinematics

ORTS objects are circles placed in a rectangular playing field. Each object has a fixed maximal speed and can move in any direction or stop anytime. ORTS kinematics is simple: there is no mass, no acceleration, and no impulse conservation. When two objects collide they just stop. This basic model simplifies motion and collision computation but already covers the important motion aspects of RTS games. Object motion is clocked. In each time interval objects move from their current location to the destination location which depends on the object's heading and speed. The algorithm presented in Fig. 6 detects all collisions that occur in one time interval and moves the objects accordingly. The algorithm first computes the motion bounding spheres Fig. 5 for all objects. Then it constructs the sphere intersection graph G in which nodes represent bounding spheres and

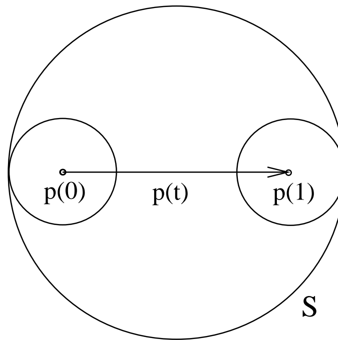

Fig. 5. Motion bounding sphere. A circular object moves from position $p(0)$ to $p(1)$. The motion bounding sphere S covers all points occupied by the object in time interval $[0, 1]$. Spheres are good approximations if the motion vector is short compared to the object radius – which is usually the case in RTS games.

```
compute motion bounding spheres
compute sphere intersection graph G
forall connected components c in G {
  empty H  // pair-heap of edges and collision times
  empty S  // stopped nodes collected here
  forall active edges (u,v) in c {      // at least one moving node
    if (u and v intersect) { stop u and v at time 0 and add them to S }
    add ((u,v), NextCollisionTime(u,v,0)) to H
  }
  T := 0                                // current collision time
  forever {
    while (S not empty) {
      empty N                           // newly stopped nodes collected here
      forall u in S {
        forall neighbors v of u in G {
          if (v is moving) {
            d := NextCollisionTime(u, v, T)
            if (d = 0) {
              stop v at time T          // instant collision
              add v to N
              remove ((u,v),?) from H
            } else if (T+d >= 0 && T+d <= 1)
              add ((u,v),T+d) to H      // update collision time
            else
              add ((u,v),2) into H      // no collision anymore
          } else
            remove ((u,v),?) from H     // remove inactive edge
        }
      }
      S := N;
    }
    finished := false;
    forever {                                   // find next collision time
      if (H empty) { finished := true; break; } // all done?
      retrieve ((u,v),t) with minimum t from H
      T := t;                                   // next collision time
      if (T > 1) { finished = true; break; }    // no more collisions?
      if (u or v is moving) break;              // edge active? -> handle collision
      remove ((u,v),?) from H                   // remove inactive edge
    }
    if (finished) break;                        // component done
    if (u is moving) { stop u at time T; add u to S }
    if (v is moving) { stop v at time T; add v to S }
  }
}
```

Fig. 6. Pseudo-code for object motion and collision test. NextCollisionTime(u,v,t) returns the elapsed time until the next collision of objects u and v occurs after time t. If the objects do not collide during the current time interval the function returns a value greater than 1.

edges indicate intersections. In order to minimize the sphere distance computation – which consumes quadratic time if implemented naively – spheres are first assigned to grid sectors and then all distances between spheres in each sector are determined to form G. Motion and collisions can now be handled local to the connected components of G because objects do not collide if their motion bounding spheres are disjoint. The algorithm then generates the sequence of collision times for each component of G separately and moves the objects to their respective final positions. The central data structure is an augmented heap which gives access to the edge with the earliest collision time and allows adding, updating, and removing of edge-time pairs in logarithmic time. An additional mapping from edges to time allows to perform delete operations of pairs with unspecified collision time which is used in several placed throughout the subroutine. The algorithm starts by computing local collision times ignoring global effects at first. Then, starting with the earliest collision, it moves colliding objects to the collision location, stops the objects there, and updates the collision times between newly stopped objects and objects in their neighborhood. When all collisions are handled the remaining non-colliding objects are moved to their final location. Compared with the visibility computation we describe in the next subsection the running time for object motion and collision test is negligible.

3.2 Visibility Computation

In ORTS objects have circular vision. Enemy objects are reported to a player if it is in sight of at least one friendly object ("Fog of War", Fig. 7). Similar to object motion, a naive implementation leads to quadratic run time. Moreover, the sector approach we adopted for object motion is slow in case of large sight ranges because many objects fall into single sectors. The ICollide software package [3] implements a fast on-line collision test which can also be applied for visibility computation. It is based on the fact that axis-aligned rectangles intersect if and only if their projections onto the x and y axes overlap, and exploits that the order of projection intervals only slightly change over time. Although this algorithm is fast in sparse settings – where it takes only linear time in average – its worst case run time is quadratic independent of the actual number of intersections. To increase the worst-case performance when objects are crowded we make use of a well known line-sweep technique for computing rectangle intersections. Objects and vision areas are approximated by bounding squares. Square intersections are detected in a left-to-right sweep by maintaining two priority search trees [4] which contain the current set of vertical intervals of sight and object squares. Whenever a new square appears in the sweep its vertical projection is checked for intersections with the other type of squares and then added to the respective set of intervals. At exit of a square its vertical projection is removed from the respective priority search tree. Adding and removing intervals from a priority search tree takes $O(\log n)$ time, while reporting all K_i intersections at time step

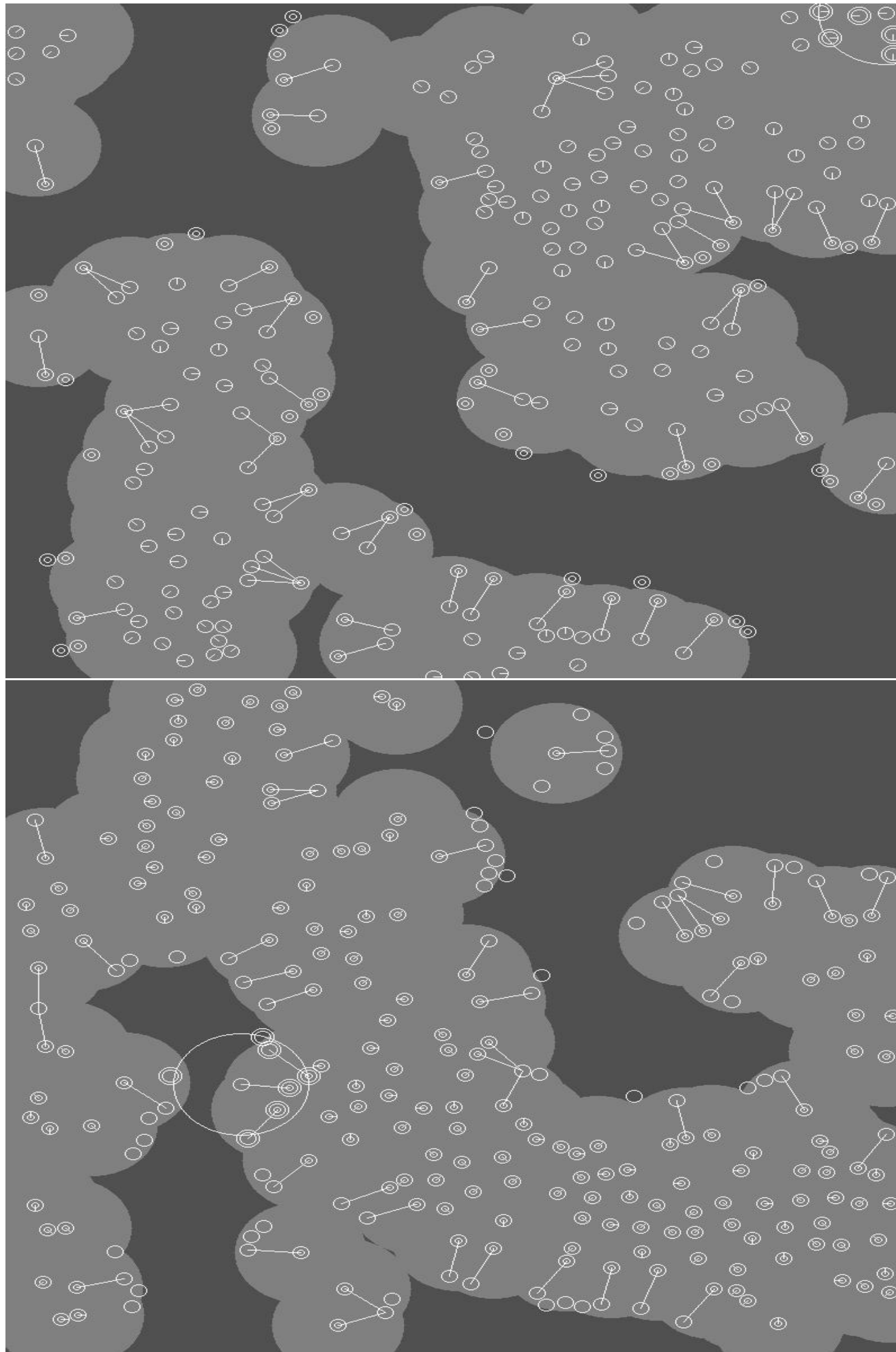

Fig. 7. Two views of a two player ORTS game. Straight lines represent attacks. Object sight ranges are indicated by filled ovals.

i is possible in time $O(\log n + K_i)$. Including the initial sort of the x-coordinates the total worst-case run time amounts to $O(n \cdot \log n + K)$, where n is the total number of objects and K the number of square intersections.

3.3 Server-Client Data Update

A server-side simulation cycle consists of sending the respective game state views to all clients, waiting for their object action responses, executing all actions, moving objects, and resolving collisions. To ensure high frame rates, latency and required data rates have to be minimized. For slow data connections compression is essential because clients need to be informed about a potentially large number of visible objects. In ORTS each object has an associated numerical feature vector with the following components:

(Object ID, Owner, Radius, Sight Range, Min. Attack Range, Max. Attack Range, Speed, Attack Value, Replicating, Hit Points, Moving, Position)

Most of these values stay constant during simulation or vary only slightly. Before compression algorithms such as LZ77 [6] are applied it is therefore beneficial to reduce entropy by computing differential vector updates first. Compression and decompression increase CPU loads and latencies in the server and the clients. Decreasing the compression rate in favor of lower compression times may therefore result in a better overall performance. The better performance when using a fast CPU at the client side (Fig. 3b) indicates that compression induced latencies currently form the bottleneck in the ORTS implementation. We will deal with (de)compression speed optimizations and implement the command delay approach to increase the simulation rate in future ORTS releases.

4 Summary & Outlook

ORTS is a hack-free RTS game toolkit. Its open message protocol and available client software enable A.I. researchers to gauge the performance of their algorithms by playing RTS games in a secure environment based on server-side simulation. Even though popular RTS titles do not provide dedicated A.I. interfaces, programs can – in principle – be constructed to play those games by accessing the frame buffer and audio streams and generating mouse and keyboard events. However, only having indirect access to the game state and being restricted to a sector view imposed by popular GUIs slows down computation and communication by forcing the A.I. to switch focus often. It also limits the command rate considerably because only objects within the current sector can receive instructions. The server-side simulation discussed here removes these artificial limitations at the cost of higher data rates and latencies which, however, can be handled by modern hardware quite easily. Human players also benefit from the open game concept as they are no longer confined to static user interfaces and predefined low-level object behavior. Instead, players can utilize self-made or third-party client software for low- or mid-level object control –

Fig. 8. StarCraft User Interface. The main screen area shows a detailed view of a playing field sector. Its location is indicated on the “mini-map” in the lower left part.

such as path-finding and multi-unit attack/defense. Improved A.I. frees players from cumbersome hand-to-hand combat and lets them concentrate on strategic decisions. Moreover, GUIs can be chosen freely because server-side simulations are not prone to client-side hacking. For instance, players may want to replace the usual detailed view of a single playing field sector (Fig. 8) by several low resolution views which together cover much more space and allow to control multiple sectors more efficiently.

ORTS provides basic RTS game functionality and can be extended easily. Currently, objects are restricted to moving circles placed on a rectangular playing field, there are no landmarks or resources, motion is acceleration free, objects can replicate, have limited vision and just stop when they collide, and object interaction is restricted to attacking objects in a given attack range. This simple RTS game is already challenging and playing it well requires understanding of multi-object attack/defense formations, scouting, and motion planning. We are currently working on a machine learning approach for basic unit behavior and think about similar ideas to train the other components of a hierarchical command and control structure.

Acknowledgment

The ORTS project has benefited from many fruitful discussions with Susumu Katayama and Igor Durdanovic.

References

1. P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. *Gamasutra* `http://www.gamasutra.com/features/20010322/terrano_01.htm`, March 2001.
2. M. Buro. ORTS project. `http://external.nj.nec.com/homepages/mic/`, March 2002.
3. J. Cohen, M. Lin, D. Manocha, and K. Ponamgi. I-Collide: An interactive and exact collision detection system for large-scaled environments. *Proceedings of ACM Int. 3D Graphics Conference*, pages 189–196, 1995.
4. E.M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.
5. M. Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it. *Gamasutra* `http://www.gamasutra.com/features/20000724/pritchard_01.htm`, July 2000.
6. J. Ziv and A. Lempel. A universal algorithm for sequential data compression (implemented for instance in zlib http://www.gzip.org/zlib). *IEEE Transactions on Information Theory*, 23:337–342, 1977.