RTS Games as Test–Bed for Real–Time AI Research

Michael Buro and Timothy Furtak

Department of Computing Science, University of Alberta,
Edmonton, Alberta, T6J 2E8, Canada

Abstract

This article [1] motivates AI research in the area of real–time strategy (RTS) games and describes the road–map and the current status of the ORTS project whose goals are to implement an RTS game programming environment and to build AI systems that eventually can outperform human experts in this popular and challenging domain.

Key words: Real–time AI, Simulation, Multi–Player Games

1 Introduction

Commercial computer games are a growing part of the entertainment industry and simulations are a critical aspect of modern military training. These two fields have much in common, cross–fertilize, and are driving real–time AI research [11]. With the advent of fast personal computers, simulation–based games have become very popular. Today, these games constitute a multi–billion dollar enterprise. Examples are sports games — in which players control entire hockey, soccer, basketball teams, etc. — and real–time strategy (RTS) games where players command armies which clash in real–time. The common elements of simulation games are severe time constraints on player actions and a strong demand of real–time AI which must be capable of solving real–world decision tasks quickly and satisfactorily. Popular simulation games are therefore ideal test applications for real–time AI research. RTS games — such as the million-sellers Starcraft and Warcraft by Blizzard Entertainment and Age of Empires by Ensemble Studios — can be viewed as simplified

[1] A preliminary and condensed version of this paper has been presented in the IJCAI–2003 poster session.

Email addresses: mburo@cs.ualberta.ca (Michael Buro), furtak@cs.ualberta.ca (Timothy Furtak).

military simulations. In these games several players struggle over resources scattered over a 2D terrain by setting up economies, building armies, and guiding them into battle in real–time. The current AI performance in commercial RTS games is poor by human standards. This may come as a surprise because RTS games have been around for more than ten years already and low–end computers nowadays can execute more than a billion operations per second. The main reasons why the AI performance in RTS games is lagging behind developments in related areas such as classic board games are the following:

- **RTS game worlds feature many objects, incomplete information, micro actions, and fast-paced action.** By contrast, World–class AI players mostly exist for slow–paced, turn–based, perfect information games in which the majority of moves have global consequences and human planning abilities therefore can be outsmarted by mere enumeration.
- **Market dictated AI resource limitations.** Up to now popular RTS games have been released solely by games companies who naturally are interested in maximizing their profit. Because graphics is driving games sales and companies strive for large market penetration only about 15% of the CPU time and memory is currently allocated for AI tasks. On the positive side, as graphics hardware is getting faster and memory getting cheaper, this percentage is likely to increase – provided game designers stop making RTS game worlds more realistic.
- **Lack of AI competition.** In classic two–player games tough competition among programmers has driven AI research to unmatched heights. Currently, however, there is no such competition among real–time AI researchers in games other than computer soccer. The considerable man–power needed for designing and implementing RTS games and the reluctance of games companies to incorporate AI APIs in their products are big obstacles on the way towards AI competition in RTS games.

In what follows, we first take a closer look at AI challenges in RTS games to motivate this domain as being well-suited for real–time AI research. Then, we will describe the ORTS project in some detail whose goals are to implementation of an RTS game programming environment and to build AI systems that can defeat human RTS game players. Finally, we discuss related work and close with concluding remarks.

2 RTS Games and AI Research

RTS games offer a large variety of fundamental AI research problems, unlike other game genres studied by the AI community so far:

- **Adversarial real–time planning.** In fine–grained realistic simulations, agents cannot afford to think in terms of micro actions such as “move one step North”.

Instead, abstractions of the world state have to be found that allow to conduct forward searches in a manageable abstract space and to translate found solutions back into action sequences in the original state space. Because the environment is also dynamic, hostile, and smart — adversarial real–time planning approaches need to be investigated.

A typical strategic problem that illustrates the necessity for adversarial planning is shown in Fig. 1. All corner regions are sealed off by strips of trees. In this Warcraft–2 map lumber and gold are the resources and no air transports are available. It does not take long for human players to realize that it is necessary to cut through the trees right away to claim and defend the gold mine in the center [A]. The computer player (2), however, is clueless and only starts chopping trees after running out of gold. It lost after being sieged by the human player (1) [B,C].

- **Decision making under uncertainty.** Initially, players are not aware of the enemies' base locations and intentions. It is necessary to gather intelligence by sending out scouts and to draw conclusions to adapt. If no data about opponent locations and actions is available yet, plausible hypotheses have to be formed and acted upon.
- **Opponent modeling, learning.** One of the biggest shortcomings of current (RTS) game AI systems is their inability to learn quickly. Human players only need a couple of games to spot opponents' weaknesses and to exploit them in future games. Current machine learning approaches which mostly base on statistics are inadequate in this area.
- **Spatial and temporal reasoning.** Static and dynamic terrain analysis as well

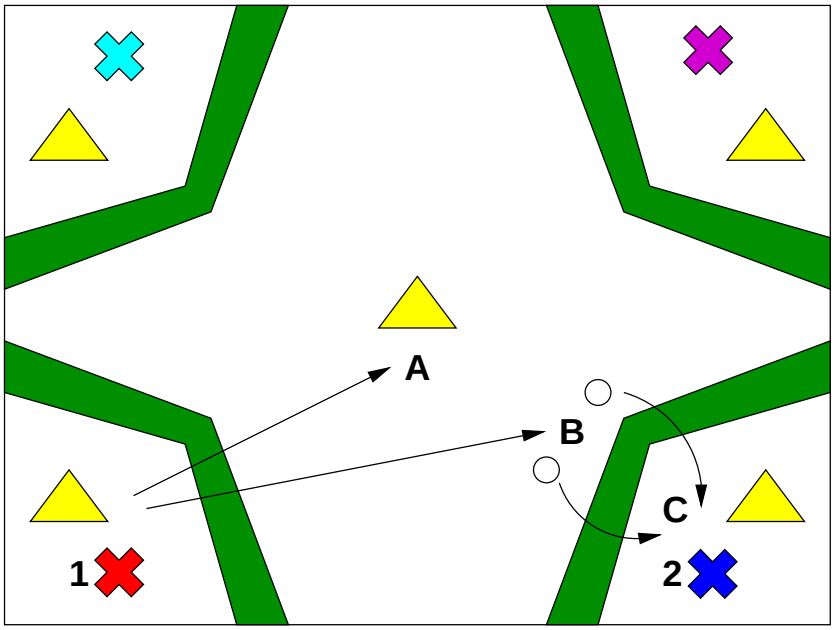

Fig. 1. RTS game scenario. The start locations of up to four players are marked with crosses and are sealed off by strips of trees. Triangles represent gold mines. An acceptable AI system needs to figure out that chopping trees right away to reach the mine in the center quickly is necessary to win the game.

as understanding temporal relations of actions is of utmost importance in RTS games — and yet, current game AIs largely ignore these issues and fall victim to simple common–sense reasoning [8].

- **Resource management.** Players start the game by gathering local resources to build up defenses and attack forces, to upgrade weaponry, and to climb up the technology tree. At any given time the players have to balance the resources they spend in each category. For instance, a player who chooses to invest too many resources into upgrades, will become prone to attacks because of an insufficient number of units. Proper resource management is therefore a vital part of any successful strategy.
- **Collaboration.** In RTS games groups of players can join forces and intelligence. How to coordinate actions effectively by communication among the parties is a challenging research problem. For instance, in case of mixed human/AI teams, the AI player often behaves awkwardly because it does not monitor the human's actions, cannot infer the human's intentions, and fails to synchronize attacks.
- **Pathfinding.** Finding high–quality paths quickly in 2D terrains is of great importance in RTS games. In the past, only a small fraction of the CPU time could be devoted to AI tasks, of which finding shortest paths was the most time consuming. Hardware graphics accelerators are now allowing programs to spend more time on AI tasks. Still, the presence of hundreds of moving objects and the urge for more realistic simulations in RTS games make it necessary to improve and generalize pathfinding algorithms. Keeping unit formations and taking terrain properties, minimal turn radii, inertia, enemy influence, and fuel consumption into account greatly complicates the once simple problem of finding shortest paths.

Playing RTS games is challenging. Even more challenging is the creation of autonomous real–time systems capable of outperforming human experts in this domain. The range of applications of RTS real–time AI modules is by no means limited to creating smart opponents to entertain human players. High–performance simulators are in high demand for training military personnel today and will become the core of automated combat and battlefield decision–support systems of tomorrow. In [22] it is predicted that 20% of the U.S. armed forces will be robotic by 2015. The current state of real–time command and control (C^2) AI, in particular in the RTS games domain, is less than satisfactory: computer opponents do not: smartly adapt to adversaries, learn from their own mistakes, look–ahead in abstracted search spaces, reason about spatial and temporal object relations, nor do they collaborate and communicate well. Human experts, on the other hand, excel in all those areas. This is true not only for the chosen domain. However, by concentrating on a concrete and bounded real–time decision task that features a number of challenging but manageable sub–problems of general interest, we think the chance of accomplishing the goal of creating a strong autonomous C^2 AI system within a couple of years is good. The results of RTS game research will increase our understanding of fundamental AI problems — such as opponent modeling and adversarial real–time planning — and will have considerable impact on the real–time

control domain in general and the computer games industry in particular which is in need of creating credible computer controlled agents. [2]

3 The ORTS Project

RTS games have become quite popular in recent years. For instance, in South Korea alone millions of people enjoy playing StarCraft and Age of Empires — two of the best–selling computer games ever. Tournaments with considerable prize money are being held regularly all over the world. The development of high–performance AI systems can benefit a lot from human expertise. Thus, it is natural to tap into available RTS game resources to learn from human experts and to measure AI performance in tournaments. Unfortunately, games companies are not inclined to release communication protocols or to add AI interfaces to their products which is required to let programs aid human players or play entire games autonomously. Moreover, current RTS game simulations rely on client–side simulations in which all client machines run the entire game simulation and just hide information from the respective players. While this approach saves bandwidth in case the command frequency is small, it is prone to map–revealing hacks which spoil the game experience just as badly as revealing opponents' cards in poker.

3.1 Overview

To overcome these problems, the Open–Real–Time–Strategy (ORTS) project was conceived in 2001 [4]. The short–term project goal is to set up a programming environment for conducting real–time AI experiments. Central to this is the implementation of an RTS game server which allows researchers to connect their AI systems to measure AI performance in real–time domains with the following properties:

- Objects navigate and interact in initially uncharted worlds in real–time. Terrain features include deep seas, rivers, plateaus, and ramps. Objects have radar–like vision which is only obstructed by elevation. They can be airborne, land–based, or naval.
- Players compute actions for a set of objects at their command. The computational model can range from local to global with respect to the objects in order to reflect

[2] At this point it should be stressed that our and the games industry's objectives regarding AI strength differ: while we are interested in creating strong AI systems capable of defeating the best humans, game developers want to maximize the replay value of their titles which excludes average humans losing all the time against computer players. We acknowledge this fact and point out that strong AI systems can be toned down to adjust to weaker players, whereas the other way around is much harder.

given command hierarchies and different levels of physical restrictions imposed by the world.
- Object actions include straight–line motion, attacking objects, gathering resources, trading goods, building, upgrading, and repairing objects.
- Team players can communicate and can share views of the world and object control.
- Top–level goals of the players include: destroying all opponent objects, reaching a designated location first, or gathering as much resources as possible in a given time period.

Compared with commercial RTS titles the ORTS system has the following advantages:

• **Free Software.** ORTS is released under the GNU Public License (GPL) which means that anyone can download the source code at no cost in order to learn how the system works and to contribute to the project by submitting bug fixes and adding new features. It also means that projects that incorporate ORTS code need to release their source code as well. It is important to point out that this does not prevent users of GPL'ed code from selling their software. The benefit of GPL'ed software releases to the community is huge as witnessed by the success of the Linux, KDE, and Mono projects. We invite games companies to release source code of their popular but no longer sold or maintained games to the community so that we all can learn.

• **Flexible Game Specification.** ORTS is a generic RTS game programming environment. It provides the infrastructure for RTS games including a server and a graphics client. However, the actual game played when using the ORTS system is not fixed — as in commercial RTS games — but scripted. A script is used to define unit properties — such as size, sight range and maximum speed — and unit actions. The freedom to adjust RTS games to the needs of AI researchers is important. However, we also acknowledge the importance of providing standard setups in order to attract human players and to spark competition in form of tournaments. The community can help here because ORTS is free software.

• **A Hack–free server–side simulation.** The ORTS game server maintains the entire world state and sends only visible information to players which connect from remote machines. Map–revealing client hacks that are common in commercial client–side simulations are therefore impossible. The additional bandwidth requirement is mitigated by compressed incremental updates.

• **Players are in total control.** Today's commercial RTS games confine users to single view graphical user interfaces, fix low–level unit behavior, and sometimes use veiled communication. The ORTS system, on the other hand, employs an open message protocol that allows AI researchers and players to connect *whatever* client software they like — ranging from split–screen GUIs, over hybrid systems in which AI components aid the human player, to fully autonomous AI systems. ORTS

clients have complete knowledge of all their units and visible terrain at all times. Therefore, there is no need for switching focus back and forth in case of multiple simultaneous battles. Another big advantage is that in ORTS there is no prescribed low–level unit behavior, which in commercial RTS games often is too simplistic and awkward. Instead, clients send each and every micro action — including breaking down paths into straight line segments — to the server. Furthermore, in each simulation cycle actions can be generated for *all* units unlike in client–side simulations where the command frequency is very limited. ORTS clients are therefore in total control of their units.

• **Remote AI.** In commercial client–side simulations the AI code for all players runs on all peer nodes to save bandwidth and to keep the simulations synchronized. This creates unwanted CPU load. Moreover, user configurable AI behavior is limited to simple scripts because there are no APIs to directly connect AI systems that run on remote machines. By contrast, conducting AI experiments in the ORTS environment is easy. Its open message protocol allows users to even connect super computers to either play RTS games autonomously or to aid human players.

Popular games in which human players still have the upper hand are ideal test–domains for AI research. By providing an open source RTS game programming environment we hope to spark interest in the C^2 domain among real–time AI researchers. The open design allows the construction of hybrid AI systems in which human players are aided by AI modules of growing capabilities. Competitive game playing on an ORTS Internet game server is therefore likely to improve AI performance and ergonomic GUI design.

3.2 A Command and Control AI Research Agenda

The long term goal of the ORTS project is the creation of AI systems whose performance is surpassing human experts in real–time command and control domains. Even before the simulator is completed we have initiated research on C^2 AI in two directions:

• **Bottom–up.** A good local unit performance is crucial to the overall success of a C^2 system because generals are overburdened if they have to issue low–level instructions to all objects under their command. Instead, objects are required to handle the most basic problems they face autonomously and quickly. Examples include finding safe routes to a given destination, concentrating attacks, and fleeing in the face of overwhelming opposition. Simple reactive systems such as finite automata, rule–based systems, and decision trees are adequate for many local decision problems. Some of these systems are suited for machine learning and combinatorial optimization. In [13], for instance, genetic algorithms were used to optimize cellular automata that control objects to study emergent strategical behavior in the

C^2 domain. A layered learning approach is presented in [18] which statically maps plan structures into a hierarchy of decision components that can be trained independently. Preliminary TD–learning experiments conducted in the ORTS system have revealed that even for managing mundane local tasks, like fleeing, look–ahead is necessary. In order to improve the local decision quality we intend to bootstrap evaluation functions that are used in conjunction with shallow heuristic searches to compute object actions. Local simulations generate the necessary training data for various function approximation techniques — such as neural networks, support vector machines [7], and the Generalized Linear Evaluation Model introduced in [3] — that we want to utilize.

• **Top–down.** On the other end of the spectrum, planning is important to deal with vast state spaces. In the C^2 domain heuristic search conducted in the original state space on a global scale is infeasible due to micro actions. Instead, the search space needs to be abstracted while retaining sufficient information to map found solutions in the abstracted space to actions in the original space. The biggest challenges in this project are the presence of adversaries, very little time for decisions, and incomplete information which causes frequent re–planning [6,9] when new information about the current world state is revealed. A way to tackle these problems at once is to find suitable abstractions that lead to manageable search spaces which allow to reconstruct solutions and can be traversed quickly by traditional heuristic search techniques. Unlike general planning tasks, navigating in 2D environments and building simplified economies provides many opportunities for exploiting domain knowledge. For instance, terrain analysis modules [16] can generate sets of important way points [21] — which are useful for planning attack routes [17] and setting up formations [5] — and analysing asset type dependency graphs [20] can provide valuable information for economic planning.

Human expertise in the C^2 domain is also very helpful. Because the goal is to perform better than human experts there is no reason to demand tabula rasa approaches that create systems from first principles by coming up with evaluation features or search space abstractions on their own. All is fair — we can make full use of human knowledge and reasoning abilities and build a strong C^2 system based on machine learning and planning on top of it.

The next milestone is the design of an RTS C^2 hierarchy and associated computational decision models suitable for machine learning and planning. Starting with the training of low–level object behavior, support systems will then be developed to aid human experts — in the role of coordinators — and freeing them from low–level decisions. By also training the upper parts of the C^2 hierarchy the final goal is to be able to replace the human coordinator in RTS games by a autonomous real–time AI system of greater performance.

One property that makes games ideal test domains for AI research is the possibility of direct confrontation in well–defined arenas which allows to gather significant

statistical performance data and to evaluate progress objectively. The initial project stage is focused on low–level unit behavior whose effectiveness can be measured by either local simulation or the success rate of human/machine hybrid teams in which the human player assumes the role of a general who delegates micro actions to the subordinate AI. Later, when the focus shifts to global tasks, software performance will be measured by either letting autonomous systems play against each other or against human/machine hybrid teams. This evaluation methodology relies on the presence of open RTS simulation software and ergonomic graphical user interfaces as well as their acceptance in the research and player communities — which is likely given the eminent interest in both multi–agent research and RTS gaming. Competition is a driving force for research.

This project can be broken down into a number of well–defined sub–tasks such as implementing an RTS game system, training of low–level object behavior, finding features and state space abstractions for heuristic search and planning purposes, terrain analysis, and efficient pathfinding. These tasks are focused, mostly independent, and their complexity range from undergraduate– to doctorate–level which makes them suitable for programming project and thesis topics.

4 ORTS Software Components and Performance

In this section we discuss various implmentation issues and provide experimental results that underline ORTS's efficient simulation and data communication.

4.1 Vision

Each cycle, the determination of what information is visible to each client must be made. This determination is based on which regions of the world are currently visible to a client. This is the union of what can be seen by all objects currently under the client's or an ally's control. The objects can see only what is within their defined range of vision, taking into account obstructions caused by terrain features such as plateaus.

The tile–based nature of the ORTS world naturally lends itself to describing an object's visual field in terms of the regions visible from the tiles it is occupying. Regardless of the criteria used to determine visibility, the highly static nature of terrain obstructions allow the results of those visibility computations to be reused for any object looking out from that tile afterward. This ability to avoid re-computation for minor changes in position greatly mitigates any coarseness in the visual description. Moreover, the level of coarseness may be adjusted by changing the number of tiles used to describe the world.

An incremental update solution — where each tile maintains a count of the number of objects that can see it — is able to optimize calculations for situations where there is little movement. However if there are a large number of moving objects then computing a client's view becomes much more expensive. For every object that moves to a different tile, a potentially large number of counters must be updated. Storing the changes to the view which occur when an object moves from one tile to another becomes quite complicated, especially when considering objects that can move more than one tile per cycle. In these worst case scenarios, computing the entire view in an optimized manner becomes faster than the bookkeeping needed for incremental updates.

Unlike an incremental solution, where the change in position of each unit must be taken into account when determining the visible tiles, ORTS generates the entire view independent of the last cycle. As a result, it is unaffected by changes in unit positions or by large portions of the view being quickly hidden and revealed as would be the case when moving troops through highly obstructed terrain. Because the number of interactions (collisions, enemy encounters, etc.) increases with the number of moving objects, computations have been optimized to reduce the resources needed in this worst case. The execution time of the vision computation is linear in the number of objects (N), the number of tiles (T), and the number of players (P), for a time complexity of $O\left(N + P{\cdot}T\right)$ (Fig. 2) per simulation cycle.

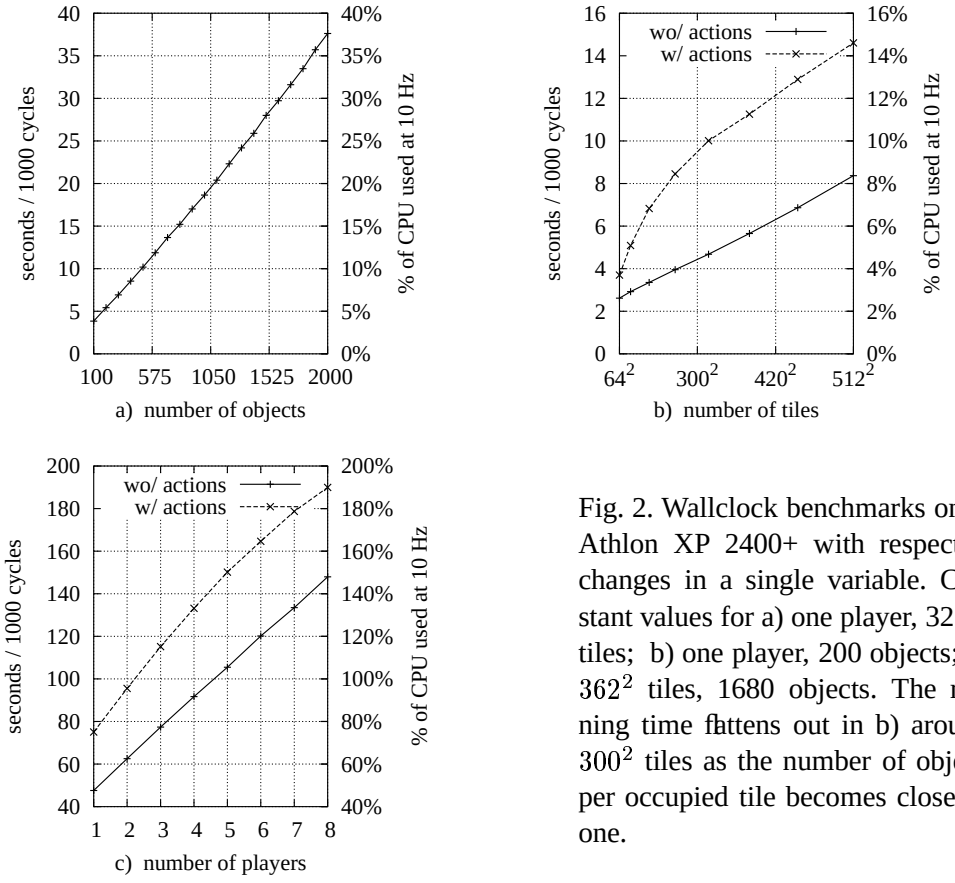

Fig. 2. Wallclock benchmarks on an Athlon XP 2400+ with respect to changes in a single variable. Constant values for a) one player, 32768 tiles; b) one player, 200 objects; c) 362^2 tiles, 1680 objects. The running time flattens out in b) around 300^2 tiles as the number of objects per occupied tile becomes closer to one.

The performance is initially greater for small map sizes due to a greater number of objects being on the same tile. This reduces the number of tileviews used relative to the number of objects. The space complexity is also $O\left(N + P{\cdot}T\right)$ if one assumes a fixed maximum sight range.

For the purposes of ORTS, the visibility status of a tile may be one of three types:

- **unknown** – no information is offered about the terrain or unit activity on the tile.
- **known** – partially visible, the complete terrain description may be seen.
- **visible** – the terrain may be seen, along with any objects or events which are on or intersect the tile.

These descriptions are used to describe the visibility of tiles within the world as they relate to a client's view of the world — a "mapview" — as well as the visibility of the world around a particular tile — a "tileview".

To determine a client's view the tileview for each object, which is actually composed of two bitmaps for visible and known tiles, is computed. The tileviews are then merged into two mapviews. Finally the difference between the current frame's view and that of the previous frame is determined, and the terrain of any newly explored tiles are included in the information sent to the client (Fig. 3 & 4).

Determining whether an object is visible to a client or not is done by computing the tiles that the object in question intersects and then accessing the mapview to check whether those tiles are visible to the client.

The ability of some objects to detect hidden or otherwise cloaked objects requires another visibility layer for determining which tiles are currently detected, but does

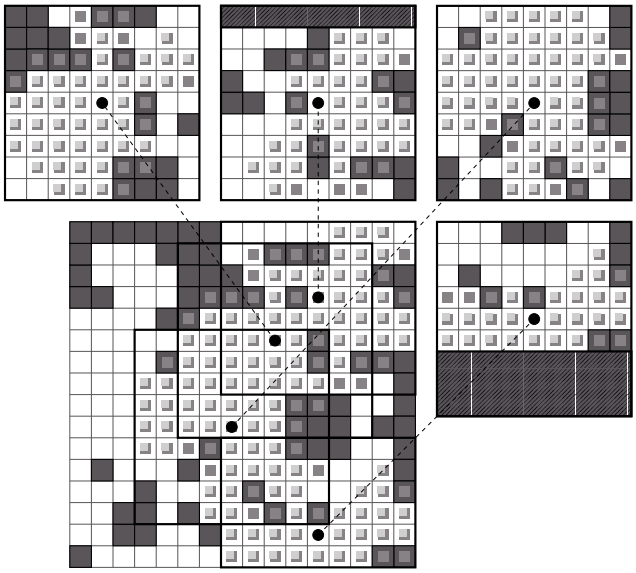

Fig. 3. Computing a client's view. The area visible from the tile under each object's center is generated, and the results are merged into a visibility map for the entire region.

```
FORALL t in Tiles {
  t.v1 = t.v2 = 0;    // reset the maximum ground and air sight ranges
}

// set the visibility of each tile based on the objects currently on it
FORALL x in Objects {
  if (x is flying)  t[center of x].v2 = max{t[center of x].v2, x.sight}
  else              t[center of x].v1 = max{t[center of x].v1, x.sight}
}

FORALL t in Tiles {
  if (t.v1 || t.v2) {
    compute t.bitmap  // the ground view up to the maximum sight range
    bitmap b = full_view[t.v2] | (t.bitmap & full_view[t.v1])
    mapview.merge(b)
  }
}
```

Fig. 4. Outline of the view computation for a client.

not otherwise affect the vision computations.

In descriptions relating to tile vision, the origin will be used to refer to the center of the tile from which visibility is being computed. In ORTS, a tile is defined to be visible relative to another tile if a line may be drawn through the centers of the two, without intersecting an occluding wall. A tile is considered known if a line may be drawn from the origin to any point in the interior of the tile without intersecting an occluding wall. Occluding walls are considered to exist at tile edges where the height of the tile is greater than can be seen over from the viewing tile. Specifically, objects may look down onto lower regions, but cannot see areas which are more than a set amount higher than their current location.

Because intersections that occur at the corners of occluding tiles can still leave half of a tile seen, the definition of visibility is modified to allow for drawing a sight line from the origin to any point within an arbitrarily small circle at the center of the observed tile.

To compute the states of the surrounding tiles, the visual field is partitioned into ranges of angles between which lines of sight are either entirely obscured or visible. Working outward from the viewing tile, encountering tiles in order of increasing

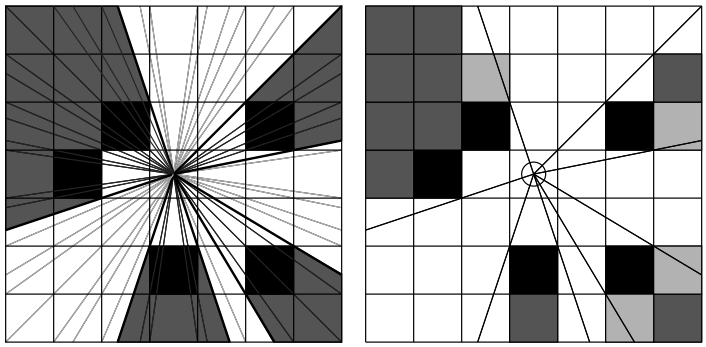

Fig. 5. Determining occluded regions. The angles obscured by each plateau tile (black square) are masked out, and every tile is checked to see how much of it is visible. Tiles with unshadowed centers are visible, tiles with any portion visible are known.

distance from the origin, the visibility of the tile is determined and the range of angles it obscures merged with the current set of ranges (Fig. 5).

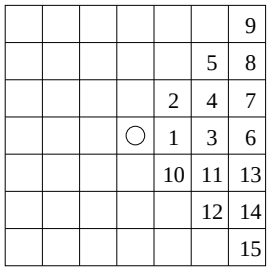

Fig. 6. Order of tile traversal.

Encountering tiles in increasing order of distance from the origin guarantees that for any tile, all tiles that could possibly obscure it have already been checked and their occluded ranges taken into account. This is accomplished by encountering tiles as strips of increasing distance from the origin, each strip moving outward from the axis (Fig. 6).

Because occluding walls only occur at the boundaries between tiles, the number of possible angles where a transition between obscured and visible ranges can occur is restricted to the set of angles from the origin through the corners of each tile within a maximum visibility radius. By pre–computing all such angles, the visible ranges at any point may be expressed in terms of a bit vector, with each bit representing the visibility status between one transition angle and the next. In practice the bit vector is stored as an array of word–sized elements, to maximize the number of concurrent operations.

The visible status of a tile is determined by checking whether the angle of a line through the origin to its center lies within or on the boundary of a visible range. Similarly, a tile is known if the range of angles the tile occupies intersects the currently visible ranges. Having a tile obscure vision is accomplished by merging its occluding range with the current occluded ranges (Fig. 7).

By converting all of the range data used for the visibility determination into bit vectors, range operations (merges and intersections) to determine intersections with the visible region and to mask out obscured regions are reduced to efficient boolean operations (ANDs and ORs). Similarly, once the tileviews have been converted into bitmaps they can be quickly merged into a larger bitmap for the entire world.

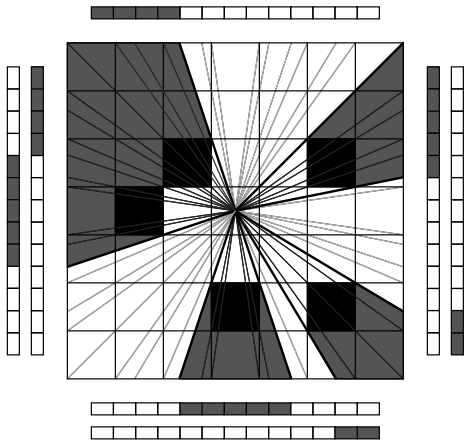

Fig. 7. Generating a tile view. Bit vectors of the range of slopes each tile takes up are used to mask the visible region as they are encountered.

```
blueprint marine
  is generic_unit      # include a set of common attributes and default values
  class kevlar armor   # create a sub-object of type "kevlar" named "armor"
  class rifle  weapon  # rifle already knows how to shoot, the "shoot" action
                       # is assigned an index in the marine's list of actions
  setf zcat  ON_LAND   # "set final" - lock the z-category (used by the server)
  setf max_hp    100
  set  hp        100
  set  sight     6
  setf radius    5
  set  max_speed 3
  set  speed     3
end
```

Fig. 8. Sample blueprint description.

4.2 Game Objects and Scripting

Objects in ORTS are abstract containers for integer variables, actions, and sub–objects. From these abstract containers, a set of blueprints is defined for the current game. These blueprints describe the names and initial values of attributes, the available actions, and the structure of an object (Fig. 8). All objects in the game are instances of one of the blueprints. The blueprints are read from a file at the start of a game, and the descriptions are sent to each of the clients. An object is described in terms of the actions and constant values of the blueprint along with the current values of the variables.

The integer elements describe all of an object's attributes, and can be declared as constant or hidden. Constant attributes are used in determining how the object interacts with the environment, but since their values never change there is no need to include them in the object descriptions sent to the client. Hidden attributes describe values which are visible to the owner of the object and allied players, but which are not observable by enemies, such as when a weapon was last fired.

With the exception of certain variable names, the server does not infer meaning from these elements. Rather, it is the actions attached to each object, all of which may be scripted, that define the interactions within the world. This allows for a wide range of games to be described and simulated in the ORTS environment without the need to add special extensions. The action scripts refer to attributes, components, and actions by name and so do not depend on any knowledge about the object's type. The scripts themselves are able to perform complex actions — as in Fig. 9 — including:

- **creating new objects** – permanent and temporary objects can be created from a blueprint and assigned an owner.
- **queuing an action** – either for immediate evaluation or for a later cycle.
- **cancelling actions** – an object's actions can be individually cancelled if any are pending in the action queue.
- **iterating over objects in a region** – allowing the scripting of area effects.

```
blueprint missile
  has core_attr                  # the server will look for these attributes
  has movement                   # basic coordinates and settings
  setf shape       CIRCLE
  setf radius      3
  set  speed       4
  setf max_speed   20
  setf zcat        IN_AIR

  var hidden det_range    5       # minimum distance to target
  var hidden blast_range  10      # size of the explosion
  var hidden min_damage   200
  var hidden max_damage   350
  var hidden damage_type  NORMAL
  var hidden fuel         60      # number of frames to track before self-destruct

  action track_obj(targ;;) {            # one object as a parameter
    gob e;                              # unassigned object pointer

    if (!targ.targetable) break;        # the object might be intangible

    if (distance(this,targ) <= this.det_range) {

      create (e, explosion, -1);        # construct an "explosion" object
                                        # with owner -1 (unowned) and
                                        # assign e to the new object
      e.x = this.x;
      e.y = this.y;
      e.zcat = targ.zcat;
      e.radius = this.blast_range;
      e.damage_type = EXPLOSIVE;

      # activate the explosion right now, it will damage units in its
      # blast radius and then destroy itself
      e.boom(;this.min_damage, this.max_damage, 0;) in 0;

      kill(this);                       # destroy the missile

    } else {
      this.fuel -=1;
      move(this, targ.x, targ.y);       # set a new motion target
      this.speed += 4;
      if (this.speed > this.max_speed) this.speed = this.max_speed;

      if (this.fuel < 0) {
        create (e, explosion, -1);
        e.x = this.x;
        e.y = this.y;
        e.zcat = targ.zcat;
        e.radius = this.blast_range/2;  # smaller visual effect
        e.boom(;0, 0, 0;) in 0;         # explode with no damage
        kill(this);
      } else {
        this.track_obj(targ;;) in 1;    # requeue this action for next cycle
      }
    }
  }
end
```

Fig. 9. Example of blueprint definition and action scripting for a homing missile.

The scripting language is defined recursively in terms of integer values (either values or object attributes), basic arithmetic and logical connectives, conditional expressions, and pre–defined functions. These functions are inserted as compiled expressions within the script and are used as helpers to provide access to low–level data structures and commonly used internal functions. It is often the case that when

the scripting language is unable or is too awkward to use for a specific task, a helper function may be quickly created by defining a keyword and associating a function within the script parser.

In addition to scripted actions defined within the blueprints, actions may be compiled functions within the server that are explicitly added to a list of available actions. These actions can then be referenced by name within the object description and a link to the compiled function added to the internal blueprint.

4.3 Communication and Networking

In each cycle the server sends the state of the world to each client as they perceive it and the client responds by sending a list of actions for the objects it has control of. As the world is explored and portions of the map are revealed, the server sends a list of the newly visible tiles along with a description of their topography (height, ground type, whether or not the tile slopes in a particular direction). Objects are entirely described by the index of the blueprint used to create the object, and a vector of their current attribute values. Subsequent viewings of the same object (if the object has not been lost from sight) are given in terms of the attribute changes from the last frame.

With the increasing speed of network communication, such data rates are within the limits of current high–speed Internet connections. By applying a moderate amount of compression to the client's view prior to sending it, the necessary throughput is greatly reduced to acceptable levels for even large multi–player games (Fig. 10).

The experimental results reported in [4] suggest that the main communication bottleneck when using server–side simulation and high–speed connections such as

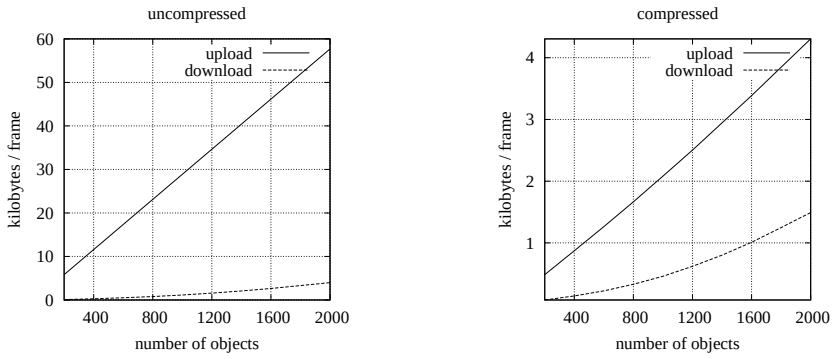

Fig. 10. Server–side data rates —with and without compression —averaged over 1000 frames with client generated actions. Each object has on average 26 attributes, and requires approximately 1 byte per attribute to encode the change from the previous cycle. The sending of tile information is included in these measurements, although their contribution to the average approaches zero since each tile is only sent once.

cable–modems or DSL is lag rather than data throughput. Lag is induced by data transfer over networks and by associated computational overhead such as message compression/inflation and updating data structures on both communication ends. For the sake of simplicity the first ORTS implementation totally ignored network lag and indeed used blocking TCP I/O on both the server and client side. As we are now moving towards a fully functional RTS game environment latency issues now need to be addressed. The most pressing question is what should happen when during a game one or more players experience unusually high network lag. In the current implementation the server gathers statistics about the clients' response times. Whenever a client exceeds a maximum lag time accumulated over a certain period of time it gets disconnected. This strategy works quite well when playing on LANs but is unsuitable for wide area networks on which lag varies a lot.

We are currently working on an alternative scheme which tries to avoid the decision on whether or when to kick out players based on network lag altogether. The idea is that rather than disconnecting clients when they do not answer in time they just forfeit the opportunity to issue actions to their units in that particular simulation frame. Two problems remain: what should happen when the server blocks while sending out the current view to a client or at one time the server receives multiple action messages? Both problems can be solved by using buffered I/O: in the first case messages accumulate in the server's send buffer. In order to quickly recover from heavy network lag, several consecutive differential view messages in the send buffer can be replaced by single messages that encode the current view without referring to earlier messages. For resolving the second problem the server needs to be able to remove multiple messages from the receive buffer in a single simulation frame because otherwise clients that experienced lag can never catch up. A simple strategy is to just consider the youngest action message in the receive buffer and to dispose of the other messages. The outlined asynchronous communication scheme has another big advantage: combined with proactive action messages sent by lagging clients it solves the lag issue in RTS games altogether — provided that a good response to the next world view(s) is highly predictable. In this regard the start/stop nature of actions in ORTS is helpful because even if action messages are delayed, units will — for instance — continue moving or attacking targets once those actions have been initiated.

5 Related Work

Research on computer soccer pursues goals similar to those outlined in this paper and has become quite popular [19]. This domain can be regarded as a simplified RTS game with only a few objects, no economy, unremarkable terrain features, and more or less complete information. Another big difference is that agents in computer soccer are required to compute their actions locally. While this decision makes perfect sense when studying collaboration in autonomous multi–agent set-

tings, there is no way of strictly enforcing it in games played on wide–area networks. In RTS games the player's role is a manager who by definition has a global view and needs to think globally. Given its limited view, no single object in RTS games can even approximate global plans because the playing field is big and possibly many local battles in different regions are fought simultaneously. Thus, the AI focus in RTS games has to be on planning on the global scale and the AI system does not need to be restricted to computation local to the units. It is therefore possible to connect any client software running on remote machines to the server without worrying about cheating.

ORTS is not the first and only free software RTS game project. Most of these projects, however, are not well maintained and still in the design phase. The notable exception is Stratagus (www.nongnu.org/stratagus) — formerly known as FreeCraft. Stratagus uses client–side simulation and is therefore prone to client hacks and not suited for real–time internet AI competitions. Nevertheless, it has recently been used as test–vehicle for MDP related planning research [10].

Many articles on robot motion, planning, temporal and spatial reasoning, and learning are relevant to constructing AI systems for RTS games. The SOAR architecture — for instance — and its application to first–person shooter games [14,15] as well as M. Atkin's work on the GRASP system that is applied to a capture–the–flag war game [1,2] are highly significant. Both projects have created high–performance game programs and represent the state–of–the–art in planning research applied to games.

The other large body of literature relevant to this work is on military analyses and applications. Research in this area spans from mathematical combat models [12] over computer generated forces — which are used in simulation and training — to decision–support systems that aid commanders and troops on the battle–field or even control entire weapon systems autonomously. This project brings both research communities together.

6 Conclusion

In this paper we have motivated AI research in the domain of RTS games. We also described the current state of the ORTS project whose goal it is to implement a programming infrastructure for RTS game AI research and to build AI systems that eventually outperform human players. The rich set of research problems that have to be tackled in order to reach human performance in these games span from pathfinding over temporal reasoning to adversarial real–time planning. Most of these problems have applications outside the game domain. Examples include autonomous robot navigation in hostile environments and simulators for training military personnel. RTS games are also well suited for team research because work on

various modules can mostly proceed independently. We encourage AI researchers to consider RTS games as test–domain and invite programmers to join our efforts to make our free software RTS game system attractive to both human players and researchers. The resulting competition then likely drives real–time AI performance to new heights.

Acknowledgments

We thank Harry Wentland for contributing to the ORTS client software. This research is partly funded by the first author's NSERC discovery grant and the second author's NSERC scholarship.

References

[1] M.S. Atkin. AFS and HAC: Domain-general agent simulation and control. In *AAAI Workshop on Software Tools for Developing Agents*, 1998.

[2] M.S. Atkin and P.R. Cohen. Physical planning and dynamics. In *Working notes of the AAAI Fall Symposium on Distributed Continual Planning*. 1998.

[3] M. Buro. From simple features to sophisticated evaluation functions. In *LNCS volume 1558 – Proceedings of the First International Conference on Computers and Games*, pages 126–145. Springer–Verlag, 1998.

[4] M. Buro. ORTS: A hack-free RTS game environment. In *Proceedings of the Third International Conference on Computers and Games*, pages 156–161, 2002. Software: http://www.cs.ualberta.ca/˜mburo/orts/orts.html.

[5] C. Dawson. Formations. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 260–271, 2002.

[6] M.E. des Jardins, E.H. Durfee, C.L. Ortiz, and M.J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 20(4):13–22, 1999.

[7] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. John Wiley and Sons, 2001. 2nd edition.

[8] K.D. Forbus, J.V. Mahoney, and K. Dill. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17(4):25–30, July 2002.

[9] J. Gratch and R. Hill. Continuous planning and collaboration for command and control in joint synthetic battlespaces. In *Proceedings of the Sixth Conference on Computer Generated Forces and Behavioral Representation*, 1998.

[10] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, 2003. Software available at http://dags.stanford.edu/Freecraft/.

[11] J.C. Herz and M.R. Macedonia. Computer games and the military: Two views. *Defense Horizons, Center for Technology and National Security Policy, National Defense University*, 11, April 2002.

[12] A. Ilachinski. Land warfare and complexity. CRM 96-68, Center for Naval Analysis Research, 1996.

[13] A. Ilachinski. Irreducible semi-autonomous adaptive combat: An artificial-life approach to land warfare. Memorandum 97-61.10, Center for Naval Analysis Research, 1997.

[14] J. Laird. Using a computer game to develop advanced AI. *Computer*, 34(7):70–75, July 2001.

[15] J. Laird, A. Newell, and P.S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence Journal*, 33(3):1–64, 1987.

[16] D. Pottinger. Terrain analysis in real–time strategy games. In *Proceedings of Computer Game Developer Conference*. 2000.

[17] D. Reece, M. Kraus, and P. Dumanoir. Tactical movement planning for individual combatants. In *Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation*. 2000.

[18] P. Stone. *Layered Learning in Multi–Agent Systems*. Computer Science Department, Carnegie Mellon University, 1998. Ph.D. Thesis CMU–CS–98–187.

[19] P. Stone. Multiagent competitions and research: Lessons from RoboCup and TAC. In *RoboCup International Symposium*, Fukuoka, 2002.

[20] P. Tozour. Strategic assessment techniques. In M. DeLoura, editor, *Game Programming Gems 2*, pages 298–306, 2001.

[21] W. van der Sterren. Terrain reasoning for 3D action games. In M. DeLoura, editor, *Game Programming Gems 2*, pages 307–323, 2001.

[22] S. von der Lippe, R.W. Franceschini, and M. Kalphat. A robotic army: The future is CGF. In *Proceedings of the 8th Conference on Computer Generated Forces and Behavioral Representation*, Florida, USA, 1999.