ON THE DEVELOPMENT OF A FREE RTS GAME ENGINE

Michael Buro and Timothy Furtak
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
email: {mburo,furtak}@cs.ualberta.ca

KEYWORDS
Real-time strategy game, server-client architecture, scripting

ABSTRACT

The genre of real-time strategy (RTS) video games is very popular and poses numerous challenges to AI researchers who want to create systems that play autonomously or aid human players. One obstacle for AI progress in this area is closed commercial software which restricts game access to inflexible graphical user interfaces. In this article we describe the current state of the free RTS game engine ORTS which allows users to define RTS games in form of scripts and to connect arbitrary game client software — ranging from 3d GUIs to distributed AI systems. This flexibility opens up new avenues for RTS game competitions and AI research, of which some are discussed here as well.

Feature	Commercial RTS Games	ORTS
Cost	≈US$ 55	US$ 0
License	closed software	free software (GPL)
Game Specification	fixed	user-definable
Network Mode	peer-to-peer	server-client
Prone to Map-Revealing Hacks	yes	no
Communication Protocol	veiled	open
Network Data Rate	low	low to medium
Unit Control	high-level, sequential	low-level, parallel
Game Interface	fixed GUI	user-definable

Table 1: How ORTS relates to commercial RTS games

BACKGROUND

Real-time strategy (RTS) games such as Starcraft^{tm} and Age of Empires^{tm} are fast-paced war simulations which have become quite popular in recent years. Constructing AI systems that play these games well is challenging because of incomplete information, real-time aspects, and the requirement of long-range planning. Many commercial RTS games feature AI scripts that can win against novice players by being favored in various ways. Examples range from giving AI components access to normally hidden information (such as opponents' unit locations), over executing actions faster, to increasing the influx of resources. While this approach may result in challenging single-player missions for beginners, it is not applicable in fair competitions. Furthermore, it does not tackle the real AI issues such as reasoning, abstract planning, learning, and opponent-modeling. Machines are still inferior to humans in these areas, which is obvious when watching machines play each other repeatedly.

To improve the performance of RTS game AI we made the case for studying real-time AI problems in the context of RTS games in (Buro 2002; Buro & Furtak 2003; Buro 2004; Buro & Furtak 2004). There we also described the design rationales and components of the free RTS game engine ORTS (Open Real-Time Strategy). Table 1 summarizes the major differences between ORTS and current commercial RTS games.

Commercial RTS games software is closed and not expandable. This prevents researchers and hobbyists from tailoring RTS games to their needs and from connecting remote AI modules in order to gauge their playing strength. ORTS, by contrast, is a free software RTS game *engine* which means that its source code and artwork are available free of charge and users can specify their own RTS games.

Furthermore, commercial RTS games as well as the free RTS game engine (Stratagus 2005) utilize peer-to-peer as opposed to server-client technology to reduce network traffic. In peer-to-peer mode the complete game state is maintained on each player's computer – by means of broadcasting all player actions – and the software just hides the invisible part of the game state from the players. By tampering with the client software it is possible to reveal the entire state and thereby gain an unfair advantage. So-called map-revealing hacks are wide-spread and pose a serious problem for online tournaments. We feel that this is unacceptable for playing fair games on the internet. Therefore, we implemented a server-client architecture in ORTS. The entire game state is maintained in the server which repeatedly sends out individual player views, receives player actions, and executes them. (Buro 2002) claims that the resulting system is "client-hack-free" in the sense that client software changes will not benefit attackers. Of course, a truly fair setup also requires trusted servers and trusted communication.

Another advantage of open server-client game architectures is that users can connect whatever client software they like. This openness leads to new and interesting possibilities ranging from fair on-line tournaments of autonomous

AI players to gauge their playing strength to hybrid systems in which human players use sophisticated GUIs which let them delegate laborious or repetitive tasks to AI helper modules. Examples include smart group pathfinding, computing efficient build orders, and small-group combat tactics.

One downside of the server-client operation compared to peer-to-peer implementations is increased network data rates, especially for the server which uploads views to the clients. In ORTS the data requirements are lowered by sending out compressed incremental view updates (Buro 2002), which is sufficient to play games with 1000 visible moving objects at a data rate of 2.5 KB per game tick.

The ORTS source code is mainly written in C++ with the exception of game specifications and GUI customization for which we developed a simple scripting language. Scripting allows us 1) to change settings without triggering compilation and 2) to use the same executables for different game types. The C++ code uses the following libraries which are available for many systems: SDL, SDL_net, Qt, OpenGL, GLUT, and GLEW. ORTS is being developed under Linux and Cygwin using gcc, but it now also natively builds under Windows and Mac OS X. In addition to the C++ source code, a sample game is provided in the distribution including game specification scripts, a set of 3d models, and user interface customization scripts for the GUI. ORTS software, artwork, and documentation can be downloaded from (ORTS 2005)

In the following sections we give a high-level overview of the major ORTS components with emphasis on the latest developments and scripting. We conclude the paper with a brief discussion of the project's future.

SERVER

The ORTS server is responsible for simulating unit actions and determining what each player is allowed to know about the current state of the world.

Every cycle players can send an action for each unit they control. The server applies these actions in a random order, removing any units that have died. Then the positions of moving objects are updated and colliding objects are stopped. Finally, the region visible to each player is computed and any changed or newly visible tiles are sent along with visible units.

To simplify the description of the world, the terrain is tile-based. Each corner of a tile may be set to an integer height, allowing tiles to be sloped in various ways. Boundaries are automatically generated where two adjacent tiles do not line up or are different types e.g. a land tile next to a water tile. To help make the terrain less blocky we support half-tiles, where a tile is split along the diagonal into two different types and/or the heights on one side of the diagonal do not line up with the heights on the other. The two sides of half-tiles are independent of each other with regard to computing vision; a unit on the lower half may not be able to see a unit on the higher half of the same tile. The default terrain generation produces cliff tiles to ease the transition between different height levels, but this is not required.

Motion

Objects are simple geometric shapes – mobile units are usually circles, buildings are rectangles, and boundaries are line segments. Although object positions are restricted to a fixed grid, collisions for moving objects are computed exactly at a higher resolution, so fast-moving objects won't pass through each other.

Which units can collide with each other is determined by a collision bitmask for each object, set by default to the object's z-category (on land, flying, underwater, etc.). Exceptions to this may be specified in another bitmask, so that special objects can pass through each other without needlessly complicating the default collision rules.

Vision

Visibility is computed in terms of which tiles can be seen from the center of the tile an object is on. If the center of the tile can be seen then that tile is entirely visible and any objects that intersect the tile can be seen. If only a portion of the tile is visible, say a corner or a side, then the type of tile is known but not any units on that tile.

Local visibility for each tile and for the entire map is stored as a bitmap. At the expense of caching the bitmap for each tile after the initial computation, determining visible tiles is quickly done via boolean operations on the bitmaps. A separate visibility computation is performed for cloaked units and the detector units that can see them.

SCRIPTING

The scripting engine performs the interesting game-specific logic and allows for flexible game definitions and client interfaces. High performance tasks common across a large number of possible RTS games such as accurate unit motion and unit vision in the presence of terrain are handled separately by the server. Everything else, such as weapons and special abilities, is scripted as part of the game definition.

The scripting language was designed to provide a convenient way to define unit types and actions. Unit definitions are given in the form of blueprints which list named (usually) integer attributes and actions. The blueprints use a loose multiple inheritance system, allowing them to be combined and nested. New unit types can easily be constructed from functional components. In the client the object creation system is used to create GUI widgets such as buttons and status windows.

When the client receives the game description, which includes unit blueprints, it can locally extend those blueprints by adding extra attributes, sub-objects, or actions. The client can use this functionality to write wrappers for complex actions, add simple background AI, or add event handlers for when an attribute changes. By adding a 3d model sub-object the client specifies how an object will be represented in the world and allows for context sensitive animations.

The client extends the scripting language functionality by registering special functions that allow access to OpenGL commands for drawing bitmaps and then simply calling

```
blueprint marine
  # include a set of common attributes and default values
  is generic_unit

  # create a sub-object of type "kevlar" named "armor"
  class kevlar armor

  # the rifle sub-object has already been defined and
  # has a "shoot" action defined
  class rifle weapon

  # make zcat constant and assign it the enum ON_LAND
  setf zcat ON_LAND
  setf max_hp     100
  set  hp         100
  setf sight      6
  setf radius     5
  set  max_speed  3
end
```

Figure 1: Marine blueprint

those functions within the script. Mouse and keyboard events received by the client are transferred to the script by calling the actions of a special root GUI object, passing the event information as parameters. This object recursively calls the interface actions of its children until it is handled.

Since the scripting language was designed to be able to perform reasonably complicated game logic, eventually errors will occur that cannot be simply debugged by inspection. At this point it becomes invaluable to have some way for the script to write information to the console or to inspect the current state. As a compiler option the interpreter can maintain a stack trace of the current execution with a printout of line numbers and the statement being evaluated at each step. This trace is automatically printed when a trappable error occurs in the script, and can be printed manually from inside a debugger such as gdb.

Because it is relatively trivial to extend the scripting language by adding external C functions it is tempting to do so whenever additional functionality is needed. This can quickly lead to numerous special purpose functions and bloated syntax. Consider the problem of implementing an STL-like vector container. One option is to try to force the language to do something it was never intended to, perhaps by implementing a complicated linked list. Another is to add a C function that returns a pointer to an actual STL vector, with additional functions for adding to it, sorting, etc.

To help make the scripting language extensible, objects in the script are all derived from a common base class, with game objects being only one possible option. To address the previous concern, wrappers have been written for STL vectors and sets, allowing them to be created in the same manner as classes described by blueprints. By modifying the new objects' incremental update functions the container can be used as a sub-object within game units. The graphical client uses derived classes for 3d models and particle systems to attach these things to objects in the game.

Script actions take generic script variables as parameters, which may be object pointers, integers, or something else.

```
blueprint missile
  has  core_attr
  has  movement
  setf shape       CIRCLE
  setf radius      3
  setf max_speed   20
  set  speed       0
  setf zcat        IN_AIR
  setf targetable  0
  setf invincible  1

  var hidden det_range    3
  var hidden blast_range  20
  var hidden min_dmg      200
  var hidden max_dmg      350

  # set the collision mask to ignore all other objects
  var collides 0

  # this action takes one object as a parameter, no
  # integer variables, and no hidden variables.

  action track_obj(targ;;) {
    gob e;
    int dmg, damage_type;

    damage_type = this.damage_type;

    if (targ.targetable < 1) break;
    if (distance(this,targ) <= this.det_range) {
      # "-1" -> not owned by any player
      e = create("explosion", -1);
      e.x = this.x;
      e.y = this.y;
      e.zcat = targ.zcat;
      e.radius = this.blast_range;
      e.damage_type = EXPLOSIVE;
      # add the "boom" action to the action queue and
      # execute it sometime in the current tick
      e.boom(;this.min_dmg, this.max_dmg, 0;) in 0;

      # mark the missile as dead - it can still act,
      # but cannot queue any more actions, and will
      # be deleted at the end of the current tick
      kill(this);
    } else {
      # move events are handled after script actions.
      # the object isn't teleported, it walks/flies to
      # the target location at its speed
      move(this; targ.x, targ.y);

      # accelerate the missile - applies to above command
      this.speed += 4;
      if (this.speed > this.max_speed)
        this.speed = this.max_speed;

      # execute this action again in 1 tick
      # without "in 1" action would be called immediately
      this.track_obj(targ;;) in 1;
    }
  }
end
```

Figure 2: Missile blueprint

When evaluating scripts in the client, actions that are part of the original game description are not evaluated locally, but are automatically placed in the outgoing action list to be sent to the server.

The game simulation is tick-based, and a large number of object actions naturally depend on time constraints e.g. weapon cool-down, construction times. To better support time in the scripts the language is able to specify that actions are to occur some number of ticks in the future. These actions are stored in a priority queue until they need to be evaluated. A small amount of bookkeeping is required to ensure that dead objects still referenced by a pending action are not deleted until they are no longer pointed to. A dead object can no longer perform actions, but functions can check if it is still alive.

CLIENT SOFTWARE

Unlike commercial RTS games, ORTS players can connect *whatever* client software they like and can issue commands to *all* of their units in each game tick (usually more than 8 times a second). Consequently, ORTS clients have much more control over game objects which greatly impacts game design. Consider default unit-behavior. In Starcraft$^{\text{tm}}$ for example, tanks automatically fire on enemy units within range. But very powerful spells like “lock-down” and “psionic storm” have to be cast manually by the player, thus limiting their effectiveness. In ORTS, all units can become so-called auto-casters by letting client AI modules decide when and where an object acts without having to wait for slow-paced player instructions. Thus, the cost of ORTS game objects has to be balanced in light of ubiquitous auto-casting.

The ORTS software currently provides basic client functionality such as communication with the server, maintaining the game state, a GUI, and some low-level AI modules, which are discussed below. The main focus of future client software additions will be on making AI components smarter to allow players to concentrate more on high-level strategic decisions, and eventually let the AI play games autonomously.

Maintaining the Game State

Because the ORTS server sends out incremental and compressed view updates and receives compressed action sequences, it is helpful to encapsulate the game state and communication in classes for everybody to use. Another advantage is that the communication protocol and compression can be changed without breaking client code. Server and client share the same Game class. In clients, this class represents the current game state in view of the player, and provides access to tiles and visible game objects. The Game class is part of GameStateModule, which communicates with the server, updates the state, and informs registered users about server messages by invoking event handlers. Each game object has an action member which can be set either by AI modules or the GUI as a result of user actions. In each game tick, actions for all objects under player control are sent to the server by invoking a function in the GameStateModule class.

Graphical User Interface

For interacting with human players and AI demonstration purposes a graphical user interface is essential. We have implemented a client component (class GfxModule) that uses OpenGL to render arbitrary 3d views of the current game state in a window together with a minimap, an information panel, and action button panel (Fig. 3). Moreover, rectangular overlays can be created to display additional information such as pathfinding results and influence maps. The widget layout, keyboard command shortcuts, and actions attached to buttons are scriptable. The graphics module communicates with the server through GameStateModule.

Low-Level AI Components

The server does not provide any default high-level functionality, so any tasks involving multiple low-level actions must be coordinated by the client. Basic gameplay tasks such as pathfinding, gathering resources, and automated defenses are implemented client-side via pluggable C++ modules. These components communicate with the GUI and with each other via a simple message passing system. When a user sends a unit to a location a pathfinding event is generated. The pathfinding module then plans a route to the target and babysits the unit, sending move commands for each leg of the path. As the world is explored the pathfinding module receives messages notifying it of new obstacles and units, letting it update its map of the world. The resource gathering module, once initiated by the client, works with the pathfinding module. It broadcasts a pathfinding message to send a unit to a given resource, receives confirmation of arrival, and then orders the unit to start mining. Similarly for returning resources to the base once collected.

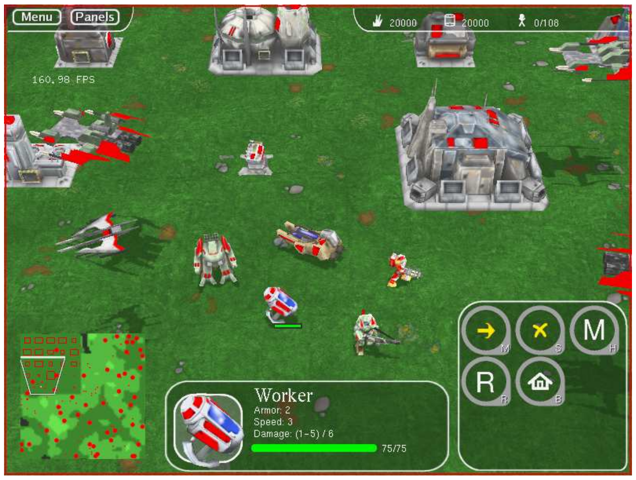

Figure 3: GUI screenshot

ORTS.NET

A recent addition to ORTS is the ORTS.net internet game service where players can meet and initiate ORTS games by communicating through a generic game server (GGS). ORTS.net is comprised of three programs:

netservice: ORTS.net game manager. Stores player data such as buddy lists and ratings. Also maintains a list of networkers, sets up games, and assigns networkers to host them.

netclient: Graphical (Qt) front-end of the ORTS.net service featuring log-on and chat dialogs and more. Communicates with netservice and players via GGS.

networker: ORTS server controlled by netservice. It hosts ORTS.net games and clients, such as ortsg, connect to it directly.

Figure 4 shows how these programs are connected. Central to ORTS.net is GGS, a message passing server which can be downloaded from `www.cs.ualberta.ca/~mburo`. GGS allows connected parties to exchange messages using a simple text-based protocol. Before an ORTS game can be initiated, netservice and one or more networkers have to be connected to GGS. Networkers register themselves with netservice to indicate that they are available for hosting ORTS games. After players connect to GGS using netclient, they can chat with each other and arrange ORTS games by sending messages to netservice. When netservice creates a game it selects an available networker and sends its IP address to the netclients along with a one-time password. The netclients then launch ortsg which connects to the networker to start the game. Finally, when the game is over, the networker sends the result back to netservice, disconnects the clients, and becomes available for hosting another game.

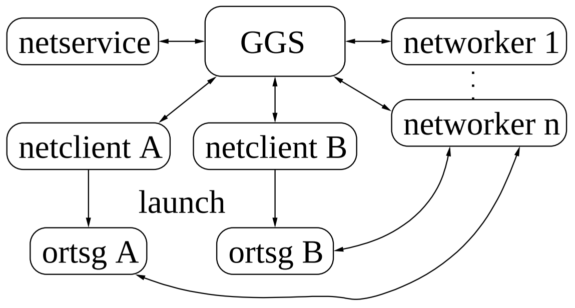

Figure 4: ORTS.net network topology

OUTLOOK

With all major components now functional, the ORTS software has reached the point where it can be used as platform for real-time AI research, the development of new RTS games, and on-line competitions.

ORTS can still be improved in various ways. For instance, the game state currently cannot be saved, GUI customization is incomplete, the graphics performance needs to be improved. Moreover, work on RTS game AI that is executed in ORTS clients has just begun.

Currently our research group is looking at pathfinding, small scale combat, optimizing build orders, and high-level planning based on Monte Carlo simulations. We hope that the availability of a (hack-) free RTS game engine sparks more interest in RTS game AI and competition among researchers, students, and hobbyists. (Molineaux 2005) reports that work already has begun to interface ORTS with (TIELT 2005), a testbed for integrating and evaluating learning techniques in real-time games.

ACKNOWLEDGMENTS

We thank Keith Yerex and Sami Wagia-alla for their numerous contributions to the ORTS project. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

Buro, M., and Furtak, T. 2003. RTS games as test-bed for real-time research. In *Proceedings of the JCIS Workshop on Game AI, extended version at `www.cs.ualberta.ca/~mburo/orts`*, 481–484.

Buro, M., and Furtak, T. 2004. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)*, 63–70.

Buro, M. 2002. ORTS: A hack-free RTS game environment. In *Proceedings of the Third International Conference on Computers and Games*, 156–161.

Buro, M. 2004. Call for AI research in RTS games. In *Proceedings of the AAAI Workshop on AI in Games*, 139–141.

Molineaux, M. 2005. NRL (Washington D.C.) personal communication.

ORTS. 2005. Free Server-Client RTS Game Engine at `http://www.cs.ualberta.ca/~mburo/orts`.

Stratagus. 2005. Free Peer-to-Peer RTS Game Engine at `http://stratagus.sourceforge.net`.

TIELT. 2005. Test-bed for integrating and evaluating machine learning techniques in real-time games at `http://nrlsat.ittid.com`.

MICHAEL BURO is an associate professor for computer science at the University of Alberta. After receiving his Ph.D. in Germany he worked as a scientist at the NEC Research Institute in Princeton for seven years before he moved to Edmonton in 2002. His main research interests are heuristic search and planning in AI and machine learning applied to games. He is the author of LOGISTELLO — a learning Othello program that in 1997 defeated the human World-champion 6–0.

TIMOTHY FURTAK is entering the masters program at the University of Alberta's computing science department. He has spent the last two years developing the ORTS game engine at the University of Alberta and is interested in applying machine learning to games.