ORTS Competition: Getting Started

Tapani Utriainen & Michael Buro

March 4, 2007

1 Introduction

The ORTS AI competition is going to be held prior to the AIIDE 2006 conference in June 2006. The deadline for registering a submission is first of June, and the entries are to be submitted a few weeks later. Anyone can attend. There are no fees involved and the authors do not have to be present at the conference.

This document describes how the ORTS system works at a high level and gives an overview of the three game categories played in the competition. For more detailed information please see the competition documentation files, the game definition scripts, ORTS documentation, and the ORTS source code.

2 Game Rules

In the games players control a number of objects, dividable into units (workers, marines and tanks) and buildings (control centers, barracks and factories). Units are small, circular, and can move and fight. Buildings are large, rectangular, and can produce units.

2.1 Simulation Cycle

Time in the games is measured in discrete frames of equal duration. In the competition a pace of 8 frames per second is used. Once per frame the game server sends individual game views to all clients (the player software) which then can specify at most one action per game object under their control and send this vector of actions back to the server. All received actions are then randomly shuffled and executed in the server.

2.2 World Representation

Internally, the terrain is represented by a rectangular array of tiles. Each tile is defined by a corner height value and a type type ("ground" or "cliff" in the competition, where ground tiles are traversable but cliff tiles are not). Tiles can also be split into two triangles (`|\|` or `|/|`) in which case both triangles can have different types. The height field defined by the tile corner heights is continuous, i.e. corners shared by neighboring tiles have identical heights.

Objects in the world have a shape and a position on the terrain. The position and size of objects are represented by integers using a scale of 16 points per tile ("tile points"). Shapes are circles or axis-aligned rectangles. Circles are specified by their center and radius. Rectangles are defined by their upper left and lower right coordinates.

A special form of objects are *boundaries*. These are line segments that objects cannot pass through which are computed from the tile-based terrain representation by considering ground-cliff tile transitions. The server uses these boundary line segments for efficient collision detection. Both tile and boundary information is maintained in the client.

2.3 Visibility

All objects controlled by a player have a visibility range, that is the distance how far they can see. Visibility is defined in terms of tiles to speed up its computation. Tiles that have some part within the range from the tile that contains the object center are visible to the player. The same goes for objects. If any part of the object is in the visibility range, the object is visible.

All map objects are represented by a unique object id (an integer). Once an object goes out of view, and comes back into view, it gets assigned a new object id (this applies to minerals as well!).

2.4 Objects and Movement

Objects can be divided into "buildings", which are stationary axis-aligned rectangular objects, and "units" — smaller, circular mobile objects (with the exception of minerals). All units have a maximum speed with which they can move. This means that within each simulation frame, the valid moves for an object are constrained to the integer coordinates that are within a distance of less or equal to the speed of the object.

Every move is assumed to go in a straight line from the objects current position to its destination. Objects move simultaneously and their current locations are rounded to tile points before being sent to the clients which can lead to temporary and small object overlap on the client side. In case of collisions with boundaries or other objects, the moving objects are stopped at the collision location and no damage is inflicted. It is not possible to do several moves in one tick in order to avoid obstacles, even if the total distance of the moves is less than the maximum speed of the unit.

It is not possible to move inside another object or building, however if somehow trapped inside one, it is possible to move out. An example are neutral sheep that in the competition games can start inside control centres and move out.

2.5 Combat

Units can engage in combat with other units and with buildings. Marines and tanks can attack from a distance, while workers need to stand close to the attacked object. It suffices for any part of the attacked object to be in range [CONFIRM?]. After attacking the weapons are subjected to a cooldown period before they can attack again. The cooldown time is specified in simulation frames. Objects have hitpoints (HPs) indicating how much damage they can take before being destroyed. Lost HPs cannot be regained. Units and buildings can also have armor which decreases the individual damage by subtracting a constant. Damage values are uniformly distributed over certain intervals (see object stats tables below). Units only die after a simulation frame has been completed when their HPs have dropped below 1. This ensures that the order of executing attack actions is irrelevant.

2.6 Resource Gathering

For simplicity, there is only one resource called minerals. Minerals are spread out on the map in clusters. Every mineral patch is a circular object. A worker unit can gather minerals when being close to a mineral patch. A maximum of four workers can simultaneously harvest one mineral patch. Resources are to be delivered to any control center, and can be dropped off at once when being close to one. A worker can carry a maximum of 10 minerals at any time. Each mineral patch contains a finite amount of minerals and the amount is decreased by 1 every 4 simulation frames when mining.

2.7 Game Overviews

In the object value tables below, the vision range unit is tiles, build times and cooldown periods are given in simulation frames, costs are in minerals, the speed unit is tile points per simulation frame, and object sizes are given in tile points.

2.7.1 Game 1: "Cooperative Pathfinding"

A single player starts with 20 workers and one control center randomly positioned in a random terrain that features small plateaus and several mineral patch clusters. The world is populated with neutral and invincible sheep that move randomly. No objects can be built and all objects and terrain is visible. The objective is to gather as much minerals as possible in single-player mode.

Object stats table:

Object	**Spd**	**Size**
Worker	4	r=3
Control	-	62×62
Sheep	2	r=4

Unit actions: workers can move, mine minerals, and drop-off minerals at the control center.

2.7.2 Game 2: "Small-Scale Combat"

Two players start with 5 control centers each and 10 tanks located around each control center. Terrain and base locations of one player are randomized. The bases of the other player are positioned symmetrically. There is no fog of war: the entire playing field and all objects are visible. As in game 1, neutral sheep are roaming the map. The objective of the game is to destroy as many buildings as possible.

Object stats table:

Object	**HP**	**Spd**	**Size**	**Rng**	**Armor**	**Dmg**	**Cool**
Tank	150	3	r=7	112	1	26–34	20
Control	4000	-	62×62	-	2	-	-
Sheep	∞	2	r=4	-	-	-	-

Unit actions: tanks can move and attack simultaneously.

2.7.3 Game 3: "A Real RTS Game"

This "real" RTS game features resource gathering, a small technology-tree, and fog of war. Two players start with 6 workers, a control center, and 600 minerals each, and a nearby mineral patch cluster on randomized terrain. Start locations are not symmetric and again, invincible sheep are roaming the map randomly. The objective is to wipe-out all enemy buildings.

Object stats table:

Object	**Cost**	**HP**	**Spd**	**Size**	**Build**	**Vis**	**Rng**	**Armor**	**Dmg**	**Cool**
Worker	50	60	4	r=3	56	6	4	-	4–6	8
Marine	50	80	3	r=4	64	6	64	-	5–7	8
Tank	200	150	3	r=7	128	7	112	1	26–34	20
Control	600	1700	-	62×62	304	4	-	2	-	-
Barracks	400	1150	-	62×46	200	4	-	2	-	-
Factory	400	1400	-	62×46	200	4	-	2	-	-
Sheep	-	∞	2	r=4	-	-	-	-	-	-

Technology Tree:

- Workers can build barracks only if a control center exists
- Workers can build factories only if a control center and a barracks exist

Unit actions:

- Workers, marines, and tanks can move and attack simultaneously
- Workers can also build control centers, barracks, and factories. In addition, they can also mine minerals and drop them off at control centers

3 Using the ORTS framework

This explanation assumes a GNU/Linux environment.

First, start off by compiling the ORTS project (read the toplevel `README`, and install all the packages it requires). Many compile errors are avoided by upgrading your existing software (for instance, having the wrong version of GNU make, appears like a compile error due to bad programming).

One way of creating a client is to use the existing ORTS framework, and the code that comes with it. It is recommended to start with the sample AI found in `apps/sampleai/src/` directory. The sample AI is compiled by `% make sampleai` in the ORTS directory.

In order to make your own AI, you could create a directory for your AI, for instance `apps/yourai/src`. The easiest is to copy the contents of the sampleai directory. The file containing the main function must be named `yourai_main.C`. This way, your code is compiled by `% make yourai`. Any additional libraries used can be added by editing app.mk

Inside the `compute_actions()` you can read all changes from the last tick (done inside the `if` when `minfo` is `true`). Most of the vectors contain pointers to `GameObj`, and useful information can be extracted as follows:

```
/* X and Y coordinates of objects center */
int GameObj_X(GameObj *gob) { return *gob->sod.x; }
int GameObj_Y(GameObj *gob) { return *gob->sod.y; }

/* Upper left and lower right corner of rectangular objects, start and end of boundaries */
int GameObj_X1(GameObj *gob) { return *gob->sod.x1; }
int GameObj_Y1(GameObj *gob) { return *gob->sod.y1; }
int GameObj_X2(GameObj *gob) { return *gob->sod.x2; }
int GameObj_Y2(GameObj *gob) { return *gob->sod.y2; }

/* Player id of owner (0,1 = players, 2 = neutral) */
int GameObj_Owner(GameObj *gob) { return *gob->sod.owner; }
```

```
/* Radius of circular objects */
int GameObj_R(GameObj *gob) { return *gob->sod.radius; }

/* What object is it (marine? mineral? ...) */
int GameObj_Type(GameObj *gob) { return gob->code(); }

/* Unique integer identifier */
int GameObj_Id(GameObj *gob) { return ScriptObj::get_obj_id(gob); }

/* Current action */
int GameObj_Action(GameObj *gob) { return gob->cur_action.id; }

/* Hit points */
int GameObj_HP(GameObj *gob) {
   const sint4 *hp = gob->get_const_int_ptr("hp");
   if (hp != NULL) return *hp;
   else return -1;
}
```

For player state information the following has turned out to be useful:

```
/* Which player am I? (0 or 1) */
const Game &game = state.gsm->get_game();
my_id = game.get_client_player()

/* Number of players */
nof_players = game.get_player_num();

/* My current minerals */
minerals = game.get_player_info(game.get_client_player()).global_obj("player")->get_int("minerals");
```

[TODO how to send all possible actions]

4 Known issues

gcc-4.0.2 runs into an internal error while compiling ORTS. Both 3.4.5 and 4.1.0 works though.