microRTSx Documentation
13 December 2012 Revision

1. Introduction

This document provides a general overview for the extended version of microRTS (microRTSx). See the accompanying doxygen generated documentation for in-depth documentation of each class. In the class doxygen documentation, AI authors would be most interested in the documentation for GameState, Unit, and UnitAction, as those are the classes they have access to.

2. Architecture

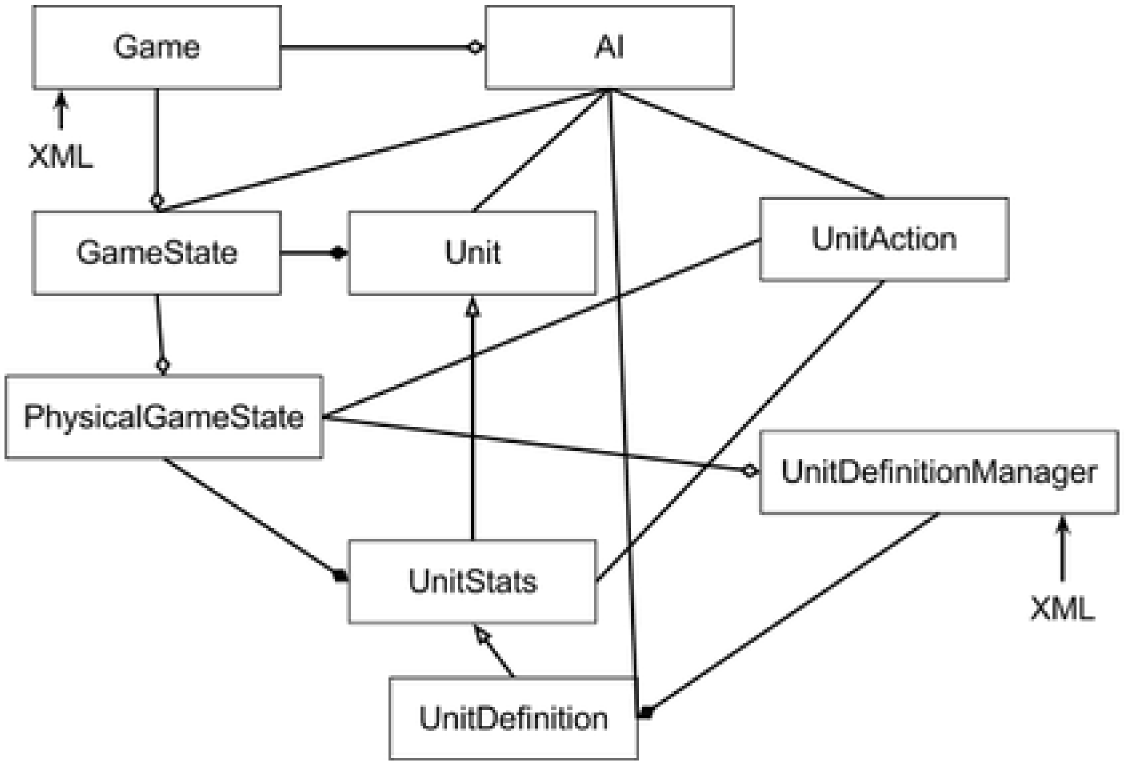

Fig. 1.1. General Architecture of microRTS Changes

To prevent agent cheating, data is stored in a layered architecture. The actual game is run in PhysicalGameState, where GameState provides methods to read certain attributes from the PhysicalGameState. Similarly, the data for a unit is stored in UnitStats (unique per unit) and UnitDefinition (common across all units of the same type) and read access is provided through Unit.

UnitDefinitions are loaded by the UnitDefinitionManager from an XML file.
Game loads turn time limit, max game length, the AIs, and the level map from XML.

3. Turn Time Limit

The agents are given a time limit for each turn. This time limit is passed in every turn so it is feasible to run games where the turn time limit is variable throughout the game. Since the agents must return within a certain time limit, to try to help them from having completely wasted turns, actions are issued to units individually, rather than all being returned together. When an action is issued, it is given a timestamp. All actions issued before the end of the agent's turn are executed in the next `cycle()` of the GameState, and all of the actions issued after the time limit are discarded.

4. Teams

The game engine supports players to be on the same team. It then only becomes necessary to rout the other team rather than all opponents. Currently, teammate's units are placed into the neutralUnits() list. This works for the GeneralAI's logic, but should possibly be moved to a separate teamUnits() list, for speed of access reasons.

5. Running Simulations

5.1. Visual Simulation

The visual simulation is run from tests.GameVisualSimulationTest and is intended primarily for debugging. Within the main function, one should create a Game object, add AI agents to the game, and run the visual simulation:

```
Game game = new Game("MAP_XML",
                     "GAME_DEF_XML",
                     TURN_LENGTH,
                     MAX_GAME_CYCLES);

// for however many AI agents need to be added
game.addAgent(new AI());

game.playVisual(FRAME_HEIGHT,
                FOG_OF_WAR?,
                RENDER_FOG?,
                TEAM_TO_FOLLOW);
```

`"MAP_XML"` is the XML file that holds the map definition. See *Section 7. Map Definition XML* for more information on how this file should be formatted.

`"GAME_DEF_XML"` is the XML file that holds the RTS game definition. See *Section 6. Game Definition XML* for more information on how this file should be formatted.

`TURN_LENGTH` is the length in milliseconds each agent has to issue orders each turn.

`MAX_GAME_CYLES` is the maximum number of turns to simulate before ending the game, using the score to determine the winner.

`FRAME_HEIGHT` is the height of the window where the visual simulation is displayed. The width is automatically determined from the height.

`FOG_OF_WAR?` is a true or false value for whether or not the game should be played with fog of war.

`RENDER_FOG?` is a true or false value for whether or not the fog of war should be rendered in the visualization.

`TEAM_TO_FOLLOW` is the ID of the team you want to follow for when fog of war is enabled (ie- it will only display what Team 0 can see, for instance). The teams are defined in the map definition XML. Use `Game.FOLLOW_ALL_TEAMS` to view the vision of every team.

5.2. Tournament Simulation

To run many games at once, the main entry point from tests.Tournament should be used. This will allow you to set up a round-robin tournament between many agents and collect results on them, rather than having to run each game separately. Although the game engine supports an arbitrary amount of players, the tournament currently works for 2 player matches.

The tournament simulation requires one command line argument, with an optional second argument.

`"TOURNAMENT_XML"` is the XML file that contains the definition for the tournament to run. See *Section 8. Tournament XML* for more information on this file.

`"RESULT_OUT"` is an optional file of where to save the results once the tournament is finished. If this file is not specified, the only output will be to the console. If this file is specified, the console output will be mirrored in the specified file.

6. Game Definition XML

Arbitrary RTS games can be defined by crafting game definition XML files. Some of these tags are taken from how Santi originally made map XML files. Ideally this version of microRTS and his should have compatible XML files, so it may be worth looking into modifying some of the names of tags. The general layout of the game definition file is shown below, followed by some more in-depth explanations of some of the tags.

```
<rts.Game>
     <resources>
          <!-- many <resource> tags -->
     </resources>

     <units>
          <!-- many <unit-def> tags -->
     </units>

     <buildings>
          <!-- many <building-def> tags -->
     </buildings>
</rts.Game>
```

First, to define resources. The resource definition tag can be self-closed.

```
<resource type="TYPE_ID"
          label="LABEL"
          score="SCORE"
          harvest_time="HARVEST_TIME"
          harvest_amt="HARVEST_AMOUNT"
/>
```

`type` is an optional attribute, but may be useful to have in this file for debugging purposes. Each resource is given a type ID, according to their appearance in this XML file. That is, the first resource has type 0, the second has 1, and so on. This type ID is also used later in this XML file when defining costs for units and buildings.

`label` is an optional attribute which may be useful for debugging purposes. AIs can access the label of a unit and print them to the console.

`score` is how many points an agent gets for mining 1 of this resource. This score also influences how many points are earned for building and killing units.

`harvest_time` is how many game cycles it takes to mine these resources after executing the `HARVEST` action.

`harvest_amt` is how many of this resource is transferred from the resource field to the worker after the `HARVEST` action succeeds.

Next, to define units. There are some tags which need to be nested within the unit definition tag.

```
<unit-def type="TYPE_ID"
          label="LABEL"
>
     <cost>
          <resource type="R_TYPE_ID"
                    cost="R_COST"
          />
          <!-- more <resource> tags if needed -->
     </cost>
     <stats hp="HP"
            vision="VISION"
             attack_range="RANGE"
             attack_min="ATK_MIN"
             attack_max="ATK_MAX"
             produce_time="PRODUCTION_TIME"
             attack_time="ATK_TIME"
             move_time="MOVE_TIME"
             is_worker="IS_WORKER?"
             is_flying="IS_FLYING?"
     />
</unit-def>
```

`<unit-def>`

`type` Like the resource type ID above, this property is optional, but may be useful for debugging the crafting of this file. This type ID will be referred to by building definitions to determine which units can be produced from which buildings.

`label` Another optional property that may be useful for debugging. This label is also used by the visual simulation to display counts of how many units each agent has.

`<cost>`

No attributes, but contains tags which indicate how many of each resource are needed to build this unit

`<resource>`

`type` The type of resource that is needed to build this unit (could be one of many different types).

`cost` How many of this resource are needed to build this unit.

`<stats>`

`hp` is the maximum HP units of this type should have

`vision` is how far away this unit can see for when fog of war is enabled. This field of vision is straight line distance.

`attack_range` is how many squares away this unit can attack from. This should be 1 for melee units. This range is straight line distance.

`attack_min` is the minimum amount of damage this unit will do with an `ATTACK` action. The actual damage dealt will be between `attack_min` and `attack_max` (inclusive). `attack_min` and `attack_max` should be set to the same thing for the unit to always deal the same amount of damage with every `ATTACK` action.

`attack_max` is the maximum amount of damage this unit will do with an `ATTACK` action. The actual damage dealt will be between `attack_min` and `attack_max` (inclusive). `attack_min` and `attack_max` should be set to the same thing for the unit to always deal the same amount of damage with every `ATTACK` action.

`produce_time` is how many game cycles it takes to build this unit.

`attck_time` is how many game cycles it takes this unit to attack.

`move_time` is how many game cycles it takes for this unit to move to another square.

`is_worker` true/false. Workers are the only units with access to the `HARVEST`, `RETURN` (harvest to stockpile), and `BUILD` commands. Workers can build all buildings.

`is_flying` true/false. Flying units can move through obstructed terrain. Non-flying units cannot.

Like unit definitions, building definitions contain a few nested tags.

```
<building-def type="TYPE"
               label="LABEL"
>
     <cost>
          <resource type="R_TYPE_ID"
                    cost="R_COST"
          />
          <!-- more <resource> tags if needed -->
     </cost>

     <stats hp="HP"
            vision="VISION"
            produce_time="PRODUCTION_TIME"
     />

     <produce>
          <unit type="TYPE" />
          <!-- more <unit> tags if needed -->
     </produce>
</building-def>
```

`<building-def>`

`type` Like the resource type ID above, this property is optional, but may be useful for debugging the crafting of this file.

`label` Another optional property that may be useful for debugging. This label is also used by the visual simulation to display counts of how many units each agent has.

`<cost>`

No attributes, but contains tags which indicate how many of each resource are needed to build this unit

`<resource>`

`type` The type of resource that is needed to build this unit (could be one of many different types).

`cost` How many of this resource are needed to build this unit.

`<stats>`

`hp` is the maximum HP units of this type should have

`vision` is how far away this unit can see for when fog of war is enabled. This field of vision is straight line distance.

`produce_time` is how many game cycles it takes to build this unit.

`<produce>`

Like `<cost>` in that this tag holds a list of other tags.

`<unit>`

`type` the type ID of a unit that can produced from this building.

7. Map Definition XML

Some of these tags are taken from how Santi originally made map XML files. Ideally this version of microRTS and his should have compatible XML files, so it may be worth looking into modifying some of the names of tags. The general layout of the map definition file is shown below, followed by some more in-depth explanations of some of the tags.

```
<rts.PhysicalGameState width=”WIDTH”
                        height=”HEIGHT”
/>
     <terrain>MAP_TERRAIN</terrain>
     <players>
          <rts.Player ID=”ID”
                      team=”TEAM_ID”
          >
               <resource type=”R_TYPE”
                         amount=”R_AMOUNT”
               />
               <!-- more innitial resources if needed -->
          </rts.Player>
          <!-- more players if needed -->
     </players>
     <units>
          <resource type=”R_TYPE”
                    x=”X”
                    y=”Y”
                    amount=”R_AMOUNT”
          />
          <!-- more resources to be placed on the map -->

          <building type=”B_TYPE”
                    player=”PLAYER”
                    x=”X”
                    y=”Y”
          />
          <!-- more buildings to be placed on the map →

          <unit type=”U_TYPE”
                player=”PLAYER”
                x=”X”
                y=”Y”
          />
          <!-- more units to be placed on the map →
     </units>
</rts.PhysicalGameState>
```

`<rts.PhysicalGameState>`

`width` The width of the map

`height` The height of the map

`<terrain>`

Terrain tag contains text of width x height 0s and 1s. 0s indicate open terrain. All units can move through open terrain and buildings can be placed on open terrain. 1s indicate obstructed terrain. Only units with the attribute flying set can move through obstructed terrain.

`<players>`

This tag just contains a list of players in the map

`<rts.Player>`

`ID` Optional attribute to indicate which ID this player has. The ID is actually set by the game engine according to the order the players are listed in this file. So the first one has ID 0, the next one has ID 1 and so on. This ID is used to assign units and buildings on the map to this player.

`team` Which team this player is on. Multiple players can be on the same team. Team IDs start at 0 and count up by one.

`<resource>` A player's starting resources

`type` The resource type ID, as defined in the game definition XML. To indicate how much of this resource type the player starts with

`amount` How much of this resource to start with

`<units>`

Contains a list of units, buildings, and resources to place on the map

`<resource>`

`type` The resource type ID

`x` The x coordinate to place this resource field

`y` The y coordinate to place this resource field

`amount` How much of this resource is in this resource field

`<building>`

`type` The building type ID, according to the game definition XML

`player` Which player ID owns this building

`x` The x coordinate to place this building

`y` The y coordinate to place this building

`<unit>`

`type` The unit type ID, according to game definition XML

`player` Which player ID owns this unit

`x` The x coordinate to place this unit

`y` The y coordinate to place this unit

8. Tournament XML

The general layout of the tournament XML file is shown below, followed by some more in-depth explanations of the tags.

```
<tournament fog="FOG_ON_OFF"
            games="NUM_GAMES"
            turn_length="TURN_LENGTH"
            max_game_cycles="MAX_GAME_CYCLES"
/>
     <game xml="GAME_DEF_XML" />
     <map xml="MAP_DEF_XML" />
     <players>
          <player agent="AGENT_CLASS" lesion="LESION" />
          <!-- as many more AIs as you want -->
     </players>
</tournament>
```

`<tournament>`

`fog` on/off. Whether or not these games should be played with fog of war.

`games` how many games to play between each player. This number is actually doubled, since the players will be simulated from both starting positions in the map.

`turn_length` how many milliseconds each agent has to make a turn.

`max_game_cycles` the maximum amount of turns to simulate before just using the score to determine the winner.

`<game>`

`xml` the game definition XML. See *Section 6. Game Definition XML* for more information on this file.

`<map>`

`xml` the map defintion XML. See *Section 7. Map Definition XML* for more information on this file.

`<players>`

Has no attributes, but contains a list of `<player>` tags.

`<player>`

`agent` The classname of the agent for this player. Some examples would be "ai.RandomAI", "ai.general.GeneralAI", or "ai.montecarlo.MonteCarlo"

`lesion` The lesion to apply to this agent. This only does anything if the AI agent itself defines its lesions.