Agents for Massive On-line Strategy Turn Based Games

Luís Miguel Landeiro Ribeiro

November 25, 2007

Abstract

Strategy games have two predominant factors that are present in every games. At the highest level, the player impersonates a strategist, managing resources; military and tecnological investiments; global economy. At the lowest level, the player assumes control over operational decisions, e.g. setting up the army disposition; coordinating armies; buys and sells resources. This work focus on the lower level, creating an agent architecture that eases the burden of development. This work shows how massive multiplayer strategy on-line games can be improved by the introduction of an agent oriented engine. Development of complex interactions becomes more manageable; creation of realistic and amuzing environments, while keeping under control the performance requirements, becomes possible.

1 Introduction

Believability is the buzzword in the computer games artificial intelligence [1], games focus on creating the illusion of reality. The longer it takes the players to realize the limitations of the virtual world the longer they will play. Massive multiplayer on-line games [2] face specific problems which affect them harder than other genres. Scalability is often cited as the main issue, with thousand even millions of virtual entities (player controlled or non-player controller) roaming around a virtual world it is hard for performance to be good.

The multi-agent paradigm is increasingly more important in MMOGs because it addresses the main problems, the creation of a virtual world and the distribution of load. Agents act as individuals and constitute the most dynamic parts of the virtual world. Agents who are independent and autonomous can be decoupled from the world core, solving the scaling needs just by offloading their processing to other machines.

1.1 Problem

Turn based strategy MMOGs have an even more peculiar set of problems to solve. Turns are processed at specific dates and therefore have a fixed time duration, during which time only player actions are collected. The game processing only happens at the end of turn. A third issue arises by applying multi-agent systems to turn-based strategy massive multiplayer online games (TBSMMOGs) [3].

1.2 Goals

This work lays down an agent oriented architecture to introduce AI into Almansur. The architecture has the following objectives:

- **Flexibility** - The framework should allow for the development of new content and agents without significant addicional effort
- **Performance** - The framework performance must be good enough to support thousands of agents running at the same time
- **Ease of use** - No special configuration should be needed for things to work out of the box
- **Simulation Realism** - The framework should provide the means to create special unique agents only by customizing their behaviors, which can be influenced by a human player controlling them.

[1] AI - Artificial Intelligence is the science and engineering of making machines, with special emphasis on intelligent computer programs[1]

[2] MMOG - Massive multiplayer on-line games are played over the internet with thousands of human players at the same time

[3] The combination of turn-based strategy games with massive multiplayer online games

1.3 Contribution

The main contribution of this work is the introduction of AI capabilities into turn-based strategy massive multiplayer on-line games. This is new ground, since no such game is known to date. The ability to influence the agents behavior by direct orders of the players is also a contribution to this active research field of human-computer interaction. A new domain specific language, simple yet powerfull, where the agents behavior can be configured without changing the core code or, even more powerfully, it can be changed by the agent itself, to simulate the learning of new abilities or even create offsprings.

2 Almansur 1.0

Figure 1: Almansur Screenshot

Almansur in seven sentences [4]: In the game, each player controls a Land, the objective being to increase the size, richness and power of that Land. Each Land is made of territories (represented by hexes on the map). Territories are the economic and recruitment cells of the Land, and possess natural resources. Each territory can have a Facility of each type and different Populations living there. Facilities can be built in a territory and give economic or military benefits. Resource gathering facilities are only useful in territories which are rich in the specific resource. Armies are formed with contingents of troops. Contingents are recruited from territory populations.

Issues that prevent Almansur from being considered a revolutionary game.

- No single player mode, where the new players can learn the game at their own pace
- Almansur has no AI, which is a distinct trace of next generation games
- Players drop-out and without AI there is no quick replacement for quitters. This degrades other player's experience.
- Dull repetitive tasks
- An overly static world

This work is an attempt to address the previous issues.

3 Almansur Agent Architecture

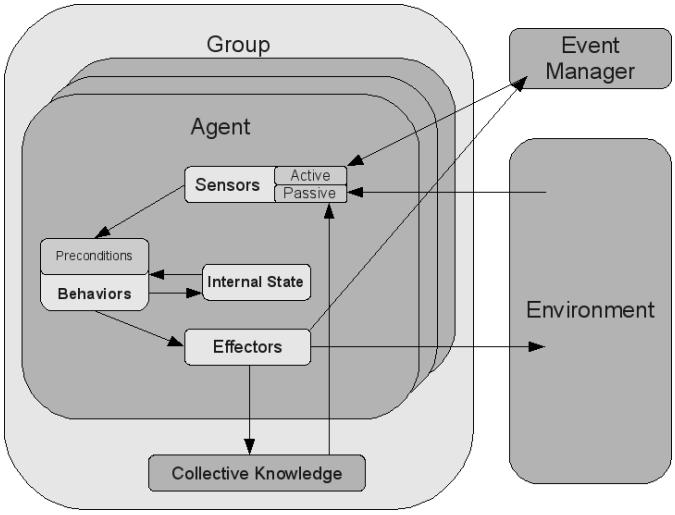

Figure 2: Agent Architecture

Figure 2 shows the outline of the architecture. The environment is accessible to the agents through standard sensors. Also accessible through the sensors is the common knowledge pool, denoted *whiteboard*. The agents can try to modify the environment through their effectors. Effectors can either succeed or fail and when they fail none of their effects are propagated to the world. Through its effectors an agent can modify the content of his group whiteboard.

Each agent can have an internal state, this state can be shared with other agents if the agent allows it. Two types of sensors exist, the passive sensors which are received by the agent by event notification; and the active sensors which are used explicitly by the agent.

Agents can communicate with other agents in two ways. Passively, they simply write information in or pose questions to the whiteboard and check for changes later in pooling mode. Actively, the agent notifies another agent explicitly and waits for his response. The agents are grouped according to the master they work for and only have access to their group's whiteboard.

Intelligence is provided by the behaviors available to each agent, the combination of different behaviors specified and emergent higher level behaviors can be manifested by the interaction of the simpler ones. Each behavior has a set of pre-conditions that need to be valid for the behavior to be enabled. The horizontal nature of the architecture allows multiple behaviors to be started at the same time if the proper conditions are met.

3.1 Environment

The virtual world of Almansur is a partially observable, non-deterministic, sequential, dynamic, discrete and multi-agent environment. It is not fully observable because other players actions or agents from outside groups are not known. The non-deterministic nature also derives from the actions of other players or competitive agents. The actions can have repercutions for many turns, making the environment sequential. It is a dynamic environment because multiple agents compete for the same actions at the same time, causing some agents effectors to fail if they are done too late. While the number of percepts can be extremely large it is still a finite set, so it is a discrete environment. Several agents roam the Almansur world, creating a multi-agent environment. Agents are typically cooperative inside the same group and competitive against different agent groups.

3.2 Agents

Several types of agents exist in this architecture, which will be detailed in the next sections. All of them follow the core design, having a hybrid architecture. The internal state is optional, agents can be fully reactive whenever appropriate. Because the performance is of the utmost importance and because several thousands of games can be active at the same time it is not feasible to have millions of agents active at the same time. The agents thus need to be loaded only when needed and made dormant when there is nothing for them to do.

3.2.1 Behaviors

Behaviors are the core of the agent's mind, an agent being only as intelligent as its behaviors and the emergent interactions they provide. For instance, for an agent to be social it must possess some sort of behavior that enables communication. The behaviors are loaded at each agent's initialization. Connected to the behaviors are the agent's sensors and effectors.

3.2.2 Sensors

Each Agent can have two types of sensors: passive sensors, with which the agent cannot control the action, it is notified when some event is detected and can chose to react to that event or simply ignore it; and active sensors that can be used whenever it deems it necessary to acquire information from the world or other agents or from the common knowledge pool. The agent's internal state is accessed directly, it is not necessary to go through its sensors. The sensors are only required to acquire information from outside the agent.

3.2.3 Effectors

Effectors are the standard way for an agent to influence the environment, other agents or to write to the group knowledge pool. They can have two outcomes: either they succeed and the environment reflects all the actions they represent; or they fail and none of the actions they represent are actually performed. This can happen when some other competitive agent performs a conflicting action at the same instant.

3.2.4 Domain Specific Language

The configuration of the behaviors requires some degrees of flexibility. To create a simple yet powerfull way of allowing configuration a custom domain specific language (DSL) was developed. Throughout this work will be refered to as Agent Modeling Language (AML).

This language is used to specify which behavior each agent can have, and which preconditions they have.

3.3 Initialization

When the turn is processed, all the agents need to be loaded. Behaviors are grouped into modules by functionality. Loading a module into an agent causes the agent to learn how to execute all the behaviors in that module.

But only behaviors that are configured are electable to be executed.

The loading process follows the algorithm for each agent:

1. Load custom modules or fallback to default modules
2. Load sensors for configured modules
3. Load effectors for configured modules
4. to_load_behaviors = Custom Behaviors
5. to_load_behaviors = Default Behaviors unless Custom Behaviors
6. for each behavior in to_load_behaviors
   (a) agent_behaviors[behavior] = new Array
   (b) for each precondition in behavior.preconditions
      i. agent_preconditions[precondition] ||= new Array
      ii. agent_preconditions[precondition].push behavior
   (c) end
7. end

If no custom behaviors are specified the agent will fall back to the default behaviors. This feature is useful to avoid repetition of customization of equal agents. A template can be created and all instances will share the same default configuration.

Each agent maintains a list of behaviors. The modules specify which sensors and effectors an agent must have to be able to execute the behaviors present in the module. This ensures an agent is always technically able to execute the configured behavior. When the module is loaded into the agent, the sensors and effectors are also injected into the agent.

3.4 Events

Events are the multi purpose system for communication. All events are under the control of the event manager module.

The event manager is responsible for keeping a record of which agents are registered for a given event type. Any agent can create new events and ask the event manager to process them. The event creator can specify filters

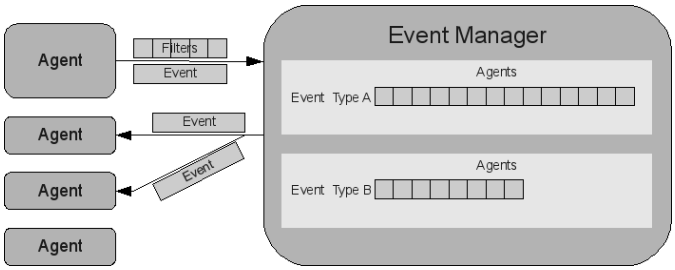

Figure 3: Event System

that must determine a subset of agents entitled to receive such an event. The event can be filled with whatever information the event creator wants, so they can be used to transport all kinds of information back and forth in an object-oriented way.

3.4.1 Synchronous Events

When an agent registers a new event with the event manager he has the chance of specifying that the event be processed in a synchronous form. The workflow below exemplifies what happens:

1. Agent creates a new event
2. Agent sends the event created in 1. to the event manager
   (a) Agent declares the event should be processed in a synchronous form
   (b) Agent sends out the filters each other agent much pass to receive the event
   (c) Agent is blocked until the event manager's execution is finished
3. The event manager receives the event and filters
4. Event manager retrieves all the agents registered to receive notification of event types with the same type as the received event
5. Event manager selects all agents from 4. that pass the received filters
6. Event manager notifies all agents from 5. with the received event
7. Each notified agent processes the received event

8. Event manager sends feedback to the caller agent
9. Agent unblocks

By the time the workflow ends the agent is sure everyone else was notified about the raised event.

3.4.2 Asynchronous Events

Asynchronous events do not block the caller agent while they are processed. Furthermore, the caller agent has no callback to notify that the event is indeed processed by everyone else. They follow the workflow below:

1. Agent creates a new event
2. Agent sends the event created in 1. to the event manager
   (a) Agent declares the event should be processed in a asynchronous form
   (b) Agent sends the filters each other agent much satisfy to receive the event
3. The event manager receives the event and filters
4. Event manager retrieves all the agents registered to receive notification of event types with the same type as the received event
5. Event manager selects all agents from 4. that pass the received filters
6. Event manager notifies all agents from 5. with the received event

3.5 Whiteboards

The second form of communication are the whiteboards. Whiteboards are simply a shared pool of knowledge that can be accessed by a group of agents. Agents are denied access to whiteboards outside their group. They do not even know they exist so all whiteboard information can be considered if all members of the group are honorable and trustworthy.

3.6 Event Processing

Event processing in multi-agent systems requires some form of arbitration to resolve conflicting actions.

3.6.1 One-Phased cycle

For performance reasons, when it is clear that an event will not cause multiple action to conflict, the event manager can notify agents to perform a combined feel-effect cycle. Agents can act on their percepts without worrying about them getting outdated. This mode follows this workflow:

1. Event manager receives the event and filters
2. Event manager retrieves all the agents registered to receive notification of event types with the same type as the received event
3. Event manager selects all agents from 2. that pass the received filters
4. Event manager notifies all agents from 3. with the received event and specifying its a feel-effect cycle.
5. Agents receive an event with a feel-effect cycle.
6. Agents uses their active sensors to extract information from the environment
7. Agents uses their effectors to perform changes on the environment

3.6.2 Two-Phased cycle

When actions from multiple agents may be conflitcting, all agents need to be coordinated to acquire information from the environment at the same time. Otherwise, some agents would have an unfair advantage over others since the accuracy of the information gathered would change. Essentially, the processing is done in two steps: a feel phase where all agents update their information about the current state of the world; and an effect phase, where the agents use the knowledge acquired from the previous phase to select what actions to take, and to effectively send them to the environment.

This mode follows this workflow:

1. Event manager receives the event and filters
2. Event manager retrieves all the agents registered to receive notification of event types with the same type as the received event

3. Event manager selects all agents from 2. that pass the received filters
4. Event manager notifies all agents from 3. with the received event and specifying its a feel phase.
5. Agents receive an event with a feel cycle.
6. Agents uses their active sensors to extract information from the environment
7. Event manager notifies all agents from 3. with the received event and specifying its a effect phase.
8. Agents receive and event with a effect cycle.
9. Agents uses their effectors to perform changes on the environment

4 Almansur 2.0

Version 2.0 of Almansur departs from the old central paradigm, where everything at turn processessing was pooled by the game. Now, at the core we have a virtual world that stimulates its agents to act. The game flow is controlled by what type of events the world generates, and the game impartial features are under the belt of special agents called managers. The managers can be seen as demi-gods in the virtual world, whose task it is to ensure the agents abide by their rules by any means necessary. For this reason the managers live outside the effectors/sensors jail and can manipulate any other agent's internal state directly.

4.1 Turn flow 2.0

Turn flow uses the introduced events to get the core going. During the processing of the core events the agents can raise their own events and create a recursive feedback loop. This effect needs to be taken into consideration when electing an agent to raise events, or else we can raise an infinite feedback loop and bring the game down.

The Almansur 2.0 core is run from the simple pseudo code below.

1. Create a new Event Manager
2. Create a new Agent Manager
3. Request the Agent Manager to load all Agents
4. Loads whiteboards for the current game from the persistent support
5. Raise a synchronous *start turn* event
6. Retrieve current day from Game Agent
7. For each day in number of days per turn
   (a) Raise a synchronous *start day* event with current day
   (b) Request event manager to raise all in-game events scheduled to happen at current day
   (c) Increments current day
8. Raise a synchronous *production* event
9. Raise a synchronous *market* event
10. Raise a synchronous *reproduction* event
11. Raise a synchronous *economy* event
12. Raise a synchronous *end turn* event
13. Request event manager to raise all queued events
14. Saves modifications to whiteboards to a persistent support

5 Tests and Results

In this section version 2.0 is tested under the same environment of version 1.0 and both are compared in performance issues and simulation issues.

5.1 Test Machine

All the tests where done under the same environment. Hardware - Intel(R) Core(TM)2 CPU 6400, 2GB of RAM Software - Ubuntu 7.04, Ruby 1.8.5 (2006-08-25) and lighttpd-1.4.13

5.2 Scenarios

The smallest scenario, 2HNoSea, has a reference point for a bare minimum scenario to be played in a multiplayer set. The timings achieved in this scenario are the best and represent the weight the architecture carries.

Game	Populations	Contingents	Units	Territories	Lands	Facilities	Resources	Personalities	Agents	Behaviors	Processed	Unprocessed
FTS4	45	28	4	99	4	12	159	44	223	1854	1359	495
CM40	2362	190	40	1989	40	80	2036	440	5024	26404	14723	11681
AB101	876	515	101	1881	101	2130	10790	1111	4487	34779	28038	6741
WOT20	589	130	25	1188	25	849	4025	275	2210	13138	8681	4457
Crejak10	341	40	10	528	10	227	1321	110	1032	5546	3040	2506
QI	980	644	107	1815	107	1285	8798	1177	4726	39584	29395	10189
Final	2144	351	60	1419	60	1025	9479	660	4637	29885	19789	10096
Juri	84	84	12	340	12	312	984	132	655	5350	3838	1512
OF6	54	33	6	280	6	72	492	66	442	2911	1810	1101
OF20	637	110	20	990	20	320	2916	220	1980	11548	6794	4754
2PL	17	14	2	140	2	26	190	22	198	1312	840	472
2PLT	2	14	2	56	2	14	46	22	99	916	690	226
2HNoSea	2	14	2	2	2	14	46	22	45	700	636	64

Table 1: Test scenarios

5.3 Simulation

To measure the simulation improvements of version 2.0 over version 1.0 the scenario 2PL is putted to the test.

Figure 4: Scenario 2PL

The test begin with a vanilla scenario, it has two lands slice and dice. The player ruling slice increases all territory taxes to 99% while dice maintains the taxes at the default value 5%. Three turns are processed on each version of the game, after each turn the different behaviors are interpreted. Because populations are the most affected from the tax change, a visual representation of the number of citizens living in each territory is presented. This visual representation varies from dark orange (low population density) to bright yellow (high population density). Figure 5 represents the initial distribution of the populations.

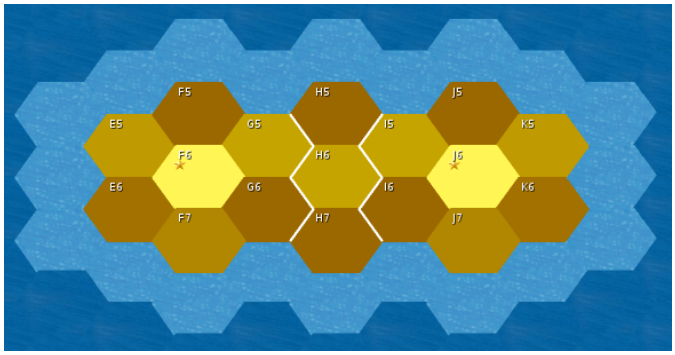

Figure 5: Initial population distribution

To better compare the inner works of the game, the following properties are measured between each turn processing:

- Lawfull - The number of law abiding citizens in the territory [0 - everybody is a criminal, 1 - everybody follows the rules]

- Loyalty - How loyal a population is to its territory [0 - rebelion about to happen, 1 - they love the territory where they live]

Turn	Territory	Lawfull	Loyalty	Population	Stability	Tax Base	Tax
0	"H6"	0.95	1	20000	1	19900	0.05
0	"F6"	0.95	1	30000	1	29900	0.99
0	"G5"	0.95	1	20000	1	19900	0.99
0	"I5"	0.95	1	20000	1	19900	0.05
0	"J6"	0.95	1	30000	1	29900	0.05
1	"H6"	0.96	1	20517	1	20619	0.05
1	"F6"	0.84	0.5	30848	0.59	27356	0.99
1	"G5"	0.84	0.5	20517	0.53	18157	0.99
1	"I5"	0.96	1	20517	1	20619	0.05
1	"J6"	0.96	1	30848	1	31058	0.05
2	"H6"	0.97	1	21047	1	21361	0.05
2	"F6"	0.62	0.25	31479	0.36	21006	0.99
2	"G5"	0.6	0.25	20881	0.3	13474	0.99
2	"I5"	0.97	1	21047	1	21361	0.05
2	"J6"	0.97	1	31719	1	32257	0.05

Table 2: 2PL scenario test on version 1.0

- Population - The number of citizens living in the territory
- Stability - How peaceful and stable the territory is
- Tax Base - The potential value a territory can collect in taxes
- Tax - The actual tax value for the territory

5.4 Version 1.0

After the third turn the population distribution is still very similar between Slice and Dice as it can be seen in figure 6. Stability and Loyalty fall by similar amounts, which the lawfull property decreases slower. Still after three turns the population reproduction difference is small and the only the tax base is much better for the Dice player. Slice has collected much more money in taxes, and the populations keep on paying even though on smaller percentages.

5.5 Version 2.0

Version 2.0 starts with the same population distribution, but because the population effects are much more noticeable, a visual representation after each turn is provided.

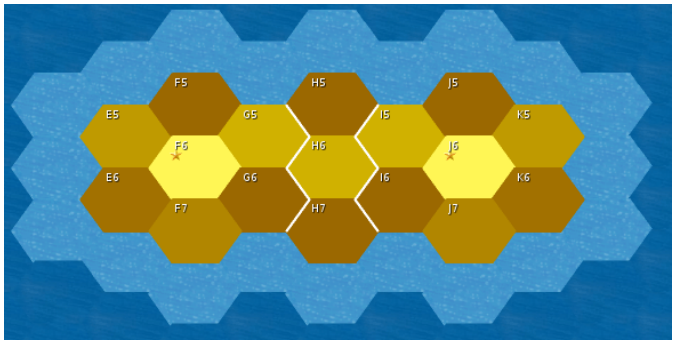

Figure 6: Population distribution for v1.0 after 3rd turn

Because version 2.0 is more complex and harder to analyze, it's necessary to check the log files to explain the current state.

Partial log from turn 1:

```
37 migrating from F5 to E5
61 migrating from F5 to F6
56 migrating from F5 to G5
```

Turn	Territory	Lawfull	Loyalty	Population	Stability	Tax Base	Tax
0	"H6"	0.95	1	20000	1	19900	0.05
0	"F6"	0.95	1	30000	1	29900	0.99
0	"G5"	0.95	1	20000	1	19900	0.99
0	"I5"	0.95	1	20000	1	19900	0.05
0	"J6"	0.95	1	30000	1	29900	0.05
1	"H6"	0.95	0.95	21734	1	21484	0.05
1	"F6"	0.73	0.03	32058	0.59	24803	0.99
1	"G5"	0.73	0.03	18487	0.53	14396	0.99
1	"I5"	0.96	1	20517	1	20596	0.05
1	"J6"	0.96	1	30819	1	30986	0.05
2	"H6"	0.94	0.93	23391	1	22903	0.05
2	"F6"	0.57	0.02	33225	0.36	20328	0.99
2	"G5"	0.55	0.02	16390	0.3	9914	0.99
2	"I5"	0.97	1	21047	1	21315	0.05
2	"J6"	0.97	1	31658	1	32108	0.05
3	"H6"	0.95	0.96	24007	1	23745	0.05
3	"F6"	0.42	0.01	2151	0.23	2298	0.99
3	"G5"	0.39	0.01	131	0.18	951	0.99
3	"I5"	0.97	1	21589	1	21841	0.05
3	"J6"	0.97	1	32518	1	32942	0.05

Table 3: 2PL scenario test on version 2.0

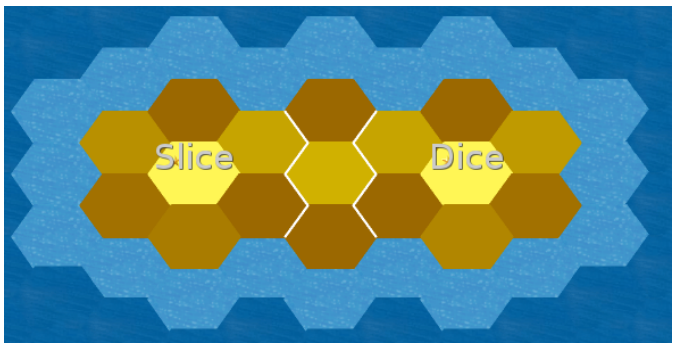

Figure 7: Population distribution of v2.0 after turn 1

After the first turn is processed, the populations of Slice start migrating to territories with better taxes. Since Slice's capital is surrounded by other territories with high taxes, they don't have where to go, and that explains why the populations increases in territory F6. Loyalty now falls much faster than in version 1.0, stability varies slower than loyalty but still faster than in version 1.0. The percentage of lawfull population also changes faster than in version 1.0, but slower than loyalty or stability.

Figure 7 shows a slight darker color on the slices side, a very small difference yet.

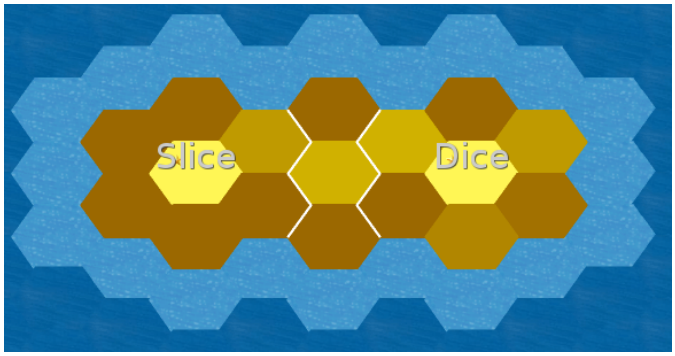

Figure 8: Population distribution of v2.0 after turn 2

Partial log from turn 2:

```
The Humans are unhappy, and 2407
 of them decided to migrate
The Humans are unhappy,
 some of them are tempted to migrate
The Humans in E5 are revolting against you
The Humans in E6 are revolting against you
```

The second turn the populations start to revolt against Slice, figure 5.5 show a massive contrast from the previous turn. Also the number of populations migrating increased to much larger numbers.

Figure 9: Population distribution of v2.0 after turn 3

Figure 9 shows now a darker picture for Slice's side, it possesses almost no population, and will soon disappear.

Partial log from turn 3:

```
The Humans in G6 are revolting against you
The Humans in F7 are revolting against you
The Humans in F6 tried to revolt against you,
 armies loyal to you stopped this uprising
The Humans in E6 are revolting against you
The Humans in F6 are revolting against you
```

After the third turn with a taxes at 99%, almost all the population of Slice has abandoned him. They also formed armed militias that are now enemies of Slice. That's why the population dropped so much on every territory. In F6, Slice has it's army and it's able to stop smaller revolts from taking place, but the large numbers of revolting populations are finally able to defeat Slice's army.

5.6 Turn processing times

During the turn processing phase version 2.0 is consistently slower than version 1.0. The added overhead from the agent's framework can be shaken off by the improvements at the schema level. CM40 is the scenario with the highest agents:lands and agents:territories ratio, and it demonstrates most heavily the agent's overhead. The global picture looks promising for, even though many new behaviors have been added, version 2.0 is only 10% slower.

Another important factor to consider is, version 2.0 can process multiple games at the same time. On a machine with multiple cores version 2.0 can theoretically provide an almost linear with each added cpu.

Game	1.0	2.0	Speedup	Var
AB101	289.67	295.65	0.98	-2%
QI	191.85	192.5	1	0%
Final	146.16	161.22	0.91	-9%
CM40	92.05	150.71	0.61	-39%
WOT20	92.55	96.99	0.95	-5%
OF20	42.61	54.06	0.79	-21%
CJK10	20.33	26.85	0.76	-24%
Juri	20.41	19.71	1.04	4%
OF6	10.17	11.88	0.86	-14%
FTS4	7.55	6.03	1.25	25%
2PL	5.11	6.52	0.78	-22%
2PLT	3.56	3.09	1.15	15%
2HNoSea	3.52	2.04	1.72	72%
Total	925.54	1027.25	0.9	-10%

Table 4: Turn Processing Comparison

6 Conclusions and future work

The AAA was applied to Almansur creating version 2.0 of the game 4. Two main areas where tested to compare version 1.0 and 2.0: simulation and performance. Simulation tests try to identify if version 2.0 really improves over the world of version 1.0. The performance tests are focused on the turn processing times, which is the main bottleneck of the game.

6.1 Simulation

The basic blocks for developing AI in Almansur were accomplished. The virtual world is now more dynamic and with the improved behaviors of populations and other agents, it's expected to be more interesting for players.

Version 2.0 resolved some of the main issues of simulation present in version 1.0.

- Populations were easy to exploit, now they have a mind of their own and players must keep them happy if they want populations to stay with them
- The contingents used to magically provide food for themselves, this lead for major balance issues. The richer lands could simply recruit huge stacks, until they overpower the poor ones. Now it's necessary to feed every alive agent, making it harder to sustains huge armies.

The other great advantage of version 2.0 is the clear separation of modules. New agents and behaviors can be implemented without changing the game core. It's now easier to produce new content for the game without having the risk of introducing new bugs on other parts of the game.

6.2 Turn Processing

Even with the overhead introduced by AAA the game performance stayed at the same levels. The increased flexibility of version 2.0 allows for futher developments to be easily introduced, no more changes to the game core are necessary.

This work fixed the major issues with the turn processing of version 1.0, the major issue that remains is the time it takes to process each turn. A few hypotheses are possible to improve that area,

- Change the implementation language. Currently the game core is implemented in ruby [4] which is a beautiful and elegant programming language, but it is dead slow. Reimplementing the core on other language like c++ or java might be the only safe way to really scale the game to millions of players.
- Change the ruby interpreter. Multiple implementations of ruby are surfacing today, within some time they might solve the performance issues on their own.

References

[1] John McCarthy, *What is artificial Intelligence*, Online,, *last accessed on 1 September of 2007*

[2] An Introduction to MultiAgent Systems, Michael Wooldridge, 2002, John Wiley and Sons, ISBN 047149691

[3] http://www.almansur.net/game/, online, last access 9th September 2007

[4] http://www.almansur.net/forum/viewtopic.php?pid=1554, online, last access 30th September 2007

[5] http://www.ruby-lang.org/en, online, last access on 10th Setember 2007

[6] http://softwareengineering.vazexqi.com/files/pattern.html , online, last access on 10th Setember 2007

[4] Ruby is a language of careful balance. Its creator, Yukihiro "matz" Matsumoto, blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balanced functional programming with imperative programming.[5]