MODELING AGENTS FOR REAL ENVIRONMENT

Gustavo Henrique Soares de Oliveira Lyrio
Roberto de Beauclair Seixas
Institute of Pure and Applied Mathematics – IMPA
Estrada Dona Castorina 110, Rio de Janeiro, RJ, Brazil 22460-320
email: {glyrio,rbs}@impa.br

Computer Graphics Technology Group – TECGRAF
Catholic University of Rio de Janeiro – PUC-Rio
Rua Marqus de So Vicente 255, Rio de Janeiro, RJ, Brazil 22453-900
e-mail: {glyrio,rbs}@tecgraf.puc-rio.br

KEYWORDS
Agents, Game Development, Simulation, Agents Framework and Lua.

ABSTRACT

This work presents Modeling Agents for Real Environments, a framework to build predefined behavior agents and insert then into real environments to run in real time. Coded in Lua, MARE departs from a formal definition of agents and join games techniques such as A* algorithm, finite state machines, terrain tiles and others to build a fully customizable tool for games or simulation creation.

INTRODUCTION

When developing new games programers, in some fase of the project, will face the necessity to build artificial inteligence models. Most times, those models could be reused (at least as the starting base for new ones) from previous game projects. How nice would be if those models could be just imported? And the modifications made in a script level? Modeling Agents for Real Enviroment - `MARE` intent to provide that. Born to simulate any real environment in real time `MARE` offers generic map and agent classes with methods to realize a sort of actions like walk, talk and interact. The main idea behind `MARE` is join popular techniques used in games, generating a tool that provides a lot of methods to create new terrain types, agents, actions and interactions allowing game creation. `MARE` was coded in `Lua`, a new powerful programming language that has become popular in the game industry as a script language. `Lua` is very easy to comprehend and doesn't demand that the programmer spend a lot of time writing extensible code. Also it's extensible and a multi-platform language and `MARE` has inherited these properties. `Lua` was coded in `ANSI C` and is available to any platform that offers such compiler (Windows, Linux, Mac OS, Solaris, etc.). So, `MARE` is available to all that platforms too.
In the next sessions we will describe how `MARE` was modeled and implemented. It is composed by three distinct parts: Environment, Agents and Rules of Interaction

ENVIRONMENTS

All games have at least one environment. It could be a forest, a mountain, a river, a medieval city, etc. In most cases, games have more then one environment. `MARE` environment differs from game environment because it represents the portion of the game environment that is relevant to the agents.
For example imagine a snow covered plain. Is it really relevant to the agents the agents that walk over that plain to know that it's covered with snow? If the answer is yes, then the snow will be part of `MARE` environment, else it will not.

`MARE` environment is divided in two parts: terrain and objects. The terrain is modeled as a two dimensional array (or a matrix) of tiles. A tile is nothing more than a building block of terrain. Put together several tiles and you can make a terrain. Each tile has a lot of attributes and methods to customize the terrain. Objects are anything that isn't a tile or an agent and can be placed over the terrain (constructions, vegetation, rocks, chairs, swords, etc). Agents also can interact with objects.

Terrain

The reason number one to use tiles to model terrain is that tiles saves memory. Consider a terrain with 100 * 100 tiles in it. That results in a total of 10,000 tiles. Let's use one large bitmap for the terrain instead of tiles. To calculate how much memory the terrain requires, you must multiply the total number of tiles by the size of each tile. Let's consider we are using 64x64 pixels tiles. The following demonstrates this concept:

```
100 tiles wide * 100 tiles high = 10,000 tiles
64 pixels wide * 64 pixels high = 4,096 pixels
per tile
10,000 tiles * 4,096 pixels * 1 byte (8-bit) =
40,960,000 bytes (w/ 256 colors)
10,000 tiles * 4,096 pixels * 4 bytes
(32-bit)=163,840,000 bytes (true color)
```

The simple 100x100 terrain will use 163 megabytes of storage if you use true color. Even if you use 256 colors it will use 41 almost megabytes of memory. Let's calculate the memory storage requirements for the same 100 x 100 terrain, using 100 different tiles.

100 tiles wide * 100 tiles high = 10,000 tiles
64 pixels wide * 64 pixels high = 4,096 pixels per tile
100 different tiles * 4,096 pixels per tile * 4 bytes per pixel = 1,638,400 bytes (to store all tiles)
10,000 tiles * 1 byte per tile = 10,000 bytes (one byte to index each tile)
10,000 bytes + 1,638,400 bytes = 1,648,400 bytes total (indexes + tiles)

The terrain now uses less then 2 megabytes total. You can use a 1000 tile set and still use less than 20 megabytes of storage.

The second reason to use tiles is that they reduce the workload of terrain creation because the same tile pattern is used multiple times within the same image. Also the tile can be rotated to make new tiles that do not need to be reloaded in memory.

MARE tiles have the following attributes available: index, texture, terrain type elevation and offset:

index identify the tile in the terrain;

texture defines the tile image;

terrain type it's an index for the know types of terrain. That allows MARE to identify the tile's characteristics and associate a tile type with agent's interactions;

elevation describes the height of the tile;

offset is the number of pixels that the image corner differs from the tile corner.

Figure 1 shows an example of terrain with four different properties. The terrain scale again can be defined by the user. MARE for default uses 48x48 pixel tiles representing cells of 1 square meter.

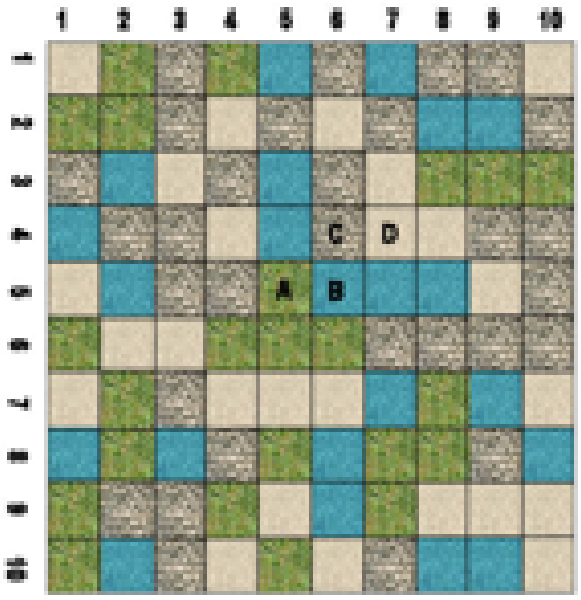

Figure 1: Sample array 10x10 with 4 different tiles modeling a terrain with grass(A), water(B), rock(C) and sand(D) properties. The terrain scale again can be defined by the user. MARE for default uses 48x48 pixel tiles representing cells of 1 square meter.

Objects

Objects are everything that isn't an agent or a tile. It can be rocks, bullets, trees, chairs, clouds, weapons, etc. Objects are placed over the tiles and can interact with the agents. Let's consider a mine for example. It will be placed over a position on the terrain (tile) and if any agent steps on that tile, the mine will run it's interaction function making it explode.

MARE objects have the following attributes:

Time ratio

MARE runs in real time, doing a loop over all the agents that are current inserted on the environment and allowing then to run short time actions. MARE sets the fastest agent movement speed to 1 pixel per time. All the other agents have fractions of that agent's speed. That avoid appear-disappear effect.

For example, let's consider that our tile size is 1 generic unit. Consider also an environment that models tanks and spaceships. The spaceships are the fastest agents, so their speed will be 0.8 tile size. Tanks are lot slower then spaceships. So, the tank speed will be 0.25 tile size.

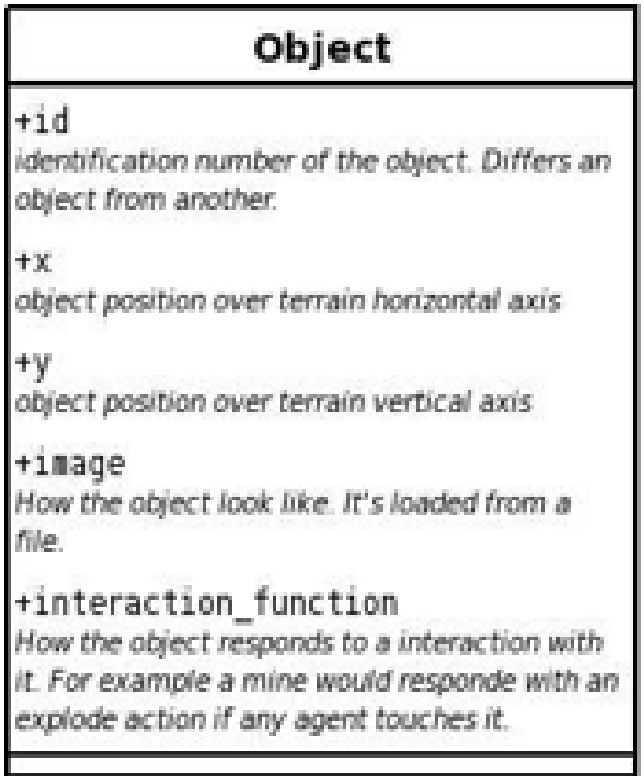

Figure 2: MARE object attributes.

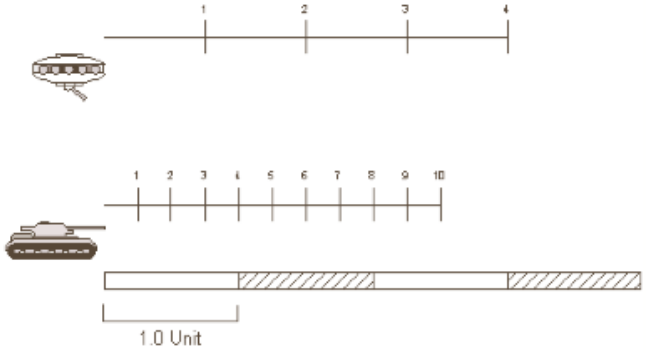

Figure 3: Graphic representation of how agent's speed works in real time. Extracted from (BARRON).

Agents

"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors. (RUSSEL)"

MARE starts from that formal definition to create its agents. Each agent has sensors and effectors. The sensors were modeled with answers that the agent will have to know how to answer with true or false and each effector will be an action that an agent can take over the environment, over other agent or even over himself.

After modeling the agent's sensors the user will group it in states. That states will describe a condition that an agent can be.

Now that we have seen how MARE models sensors, we need to understand how it models effectors. MARE does that, defining actions that the agent will take when in a state.

We also, identify which events may occur on the environment changing our agent's state. We call these events transitions. We saw till now that MARE agents have states, action and transitions. That will lead us to another common technique used in game programming: Finite State Machines.

Finite State Machines

"A finite state machine – FSM consists in a set of states (including an initial state), a set of inputs, a set of outputs, and a state transition function. The state transition function takes the input and the current state and returns a single new state and a set of outputs. Since there is only one possible new state, FSMs are used to encode deterministic behavior. (FUNGE)"

Our FSM initial state will be the initial agent state. The inputs will be the sensors and the outputs will be these sensors updated. Finally, transitions are conditions that have to be fulfilled to allow state transition function to modify the current state.

Agent Class

The agent class has the following attributes:

Predefined Agent actions

MARE has some predefined actions that should be enough to model any kind of agent that users may need for their simulations. If MARE doesn't provide the action that user judges necessary then it's possible to easily write the new action or even replace a existing action for a new

Figure 4: Description of an agent class.

one. The following list shows and explains the actions already available in MARE:

Move the move action is based on a array of way-points returned by an A* algorithm. The agent turns its direction to the first way-point in the array and move each frame by its speed until the next way-point is reached. When the last way-point is reached the action is ended;

Stop identify the next way-point in the agent way-points array. Removes all the way-points that come after it, making the action of move to stop as son as the agent reach the way-point identified;

Flee when the flee position (object or another agent) enters the line of sight the agent will start a move action to the opposite position;

Pursuit receives a position of a target (object or agent) and moves to that position using the move action;

Affect do some action to another agent, object or itself;

Say sends a message to the target. Maybe another agent or even to the screen;

Think Holds the agent's attention to an event or another agent.

Rules of Interaction

This are a set of rules that will define how MARE components (agents, environment and objects) threat events of interacting with other MARE components. These rules allow that the agents response to then can be set directly by the user.

Agent x Agent interaction

This rules will describe how and agent will respond to an event generated by other agent. With this simple set of rules MARE intend to cover all possible interaction between two distinct agents. Again, if the user finds that any other rule is still necessary, MARE will provide methods for him to code and insert as many rules he thinks he will need.

onOverTile this agent steps over a tile

onObjectAppear an object enters the line of sight.

onObjectDesappear an object leaves the line of sight

onObjectAffect an object does something that directly affects the agent. This can look strange at first time, but let's consider a mine for example: it's a MARE object, as we saw above, and it could explode affecting one or more agents next to it.

Results

This session will present an examples on how to use MARE to build agents for games, presenting creatures that live

on a hole.

Creatures on a Hole Model

Let's consider hungry creatures that live on a hole. They remain randomly moving until another specie of creature falls on the hole. Then the hole creatures will try to eat that creature. Let's see how that kind of agent can be modeled by `MARE`.

For this example we will build a hole_creature class that will have an agent class as one of its attributes:

Hole Creature attributes:

- MARE agent

Hole Creature methods:

- random move()
- pursuit target()
- attack()
- look for target()
- can_reach_target()

After that, we need to consider which sensors are needed for our agents. Fortunately two sensors will do the job. The questions that our agents will have to be able to answer is: Can i see some other creature that's not a hole creature? Can i reach the creature? Naturally if the answer is yes (`true`) for the first one, the agent will pursuit the creature. Else if the answer is no (`false`) the agent will remain in random move through the hole. If the answer is yes to the second one the hole creature will attack, else it will remain pursuit while it remains in the line of sight.

Now that we have sensor and effectors we can define three different agent states: random move state, pursuit state and attack state. As we saw above states demand state actions and transitions.

While in our first state (random move) the agent will have its sensors with both negative answers. It cannot see a different kind of creature, and, of course, it cannot reach it. The agent will execute random move() function, that consists in a call to `MARE` predefined action move() with a random environment position as destiny. The state condition function will be look for target() that calls a `MARE` interaction rule onAgentAppear verifying if the agent type isn't another hole_creature. Finally we will set the state transitions. If conditional function returns yes then a transition to pursuit state will be made. If the answer is no then the agent will remain in random move state.

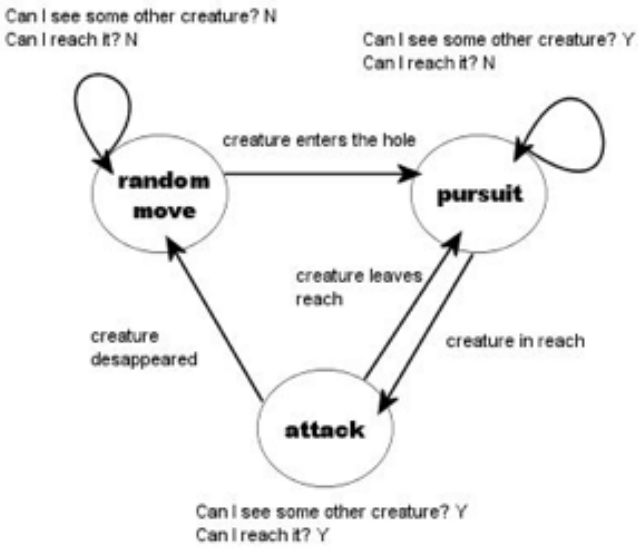

Figure 5: Graphic representation of hole creature agents `FSM`.

For the second state (pursuit state) the agent will call can reach target() method , which will call another of `MARE`'s predefined rules of interaction: onOverTile; to verify if its positioned next to the target. If yes it will change to attack state, if not it will remain in pursuit.

For the third and last state (attack state) the agent will call method attack() that consists in another call to one more predefined `MARE` action: affect() with the target creature as parameter. The agent will remain in that state until the creature dye or until it leaves the agent's reach.

Conclusions

The fact that `MARE` was builded above a formal definition of agents guarantees that it's able to construct any kind agent the user may need. Actually `MARE` provides tools to generate only pre-defined behavior agents. Although artificial intelligence has been growing a lot on games, these kind of agent still are the most common type of agents, because they are simple to code, understand, maintain, don't demand high computation power and can generate very complex behaviors.

`MARE` is a agent framework that is not restricted to the game development. It can provide tools to researchers build simulation environments. Currently `MARE` has been used to simulate enemy behavior (conventional forces and guerrilla) for military training on Brazilian Marines Simulation System.

`Lua` language is commonly used as a scripting language due to its simplicity and short time learning curve. With

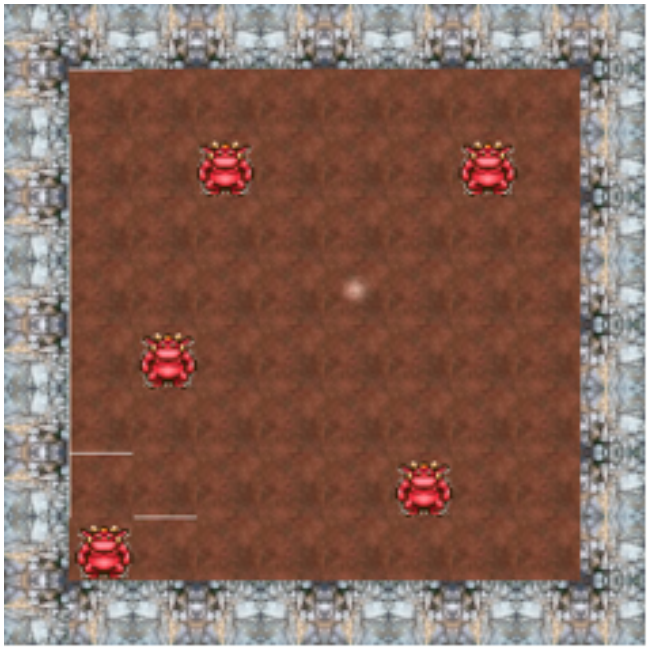

Figure 6: Possible result of implementation of creatures on a hole.

it's core developed in `Lua` language `MARE` not only provides a platform independent tool to create agents behavior but also a simple language to users create their agents. `MARE` users will get quickly familiar with Lua, and that will make then able to improve and even upgrade `MARE`'s core with no effort.

Future Works

Insert a fuzzy logic module will allow the agents decide when they want to take an action improving the framework a lot and insert sockets (by `Lua` sockets) to support network games.

REFERENCES

[RUSSEL] S. Russel P. Norvig, *Artificial Intelligence, A Modern Approach. Second Edition*, Prentice Hall, 1995.

[FUNGE] J. David, *AI for Games and Animation: A Cognitive Modeling Approach*, AK Peters, 1999.

[BARRON] T. Barron, *Strategy Game Programming With DirectX 9.0*, Wordware Publishing Inc., 2003.

[MULHOLLAND] A. Mulholland T. Hakala, *Developers guide to multi-player games*, Wordware Publishing Inc., 2001.

[REYNOLDS] C. Reynolds, *Steering Behaviors for Autonomous Characters*, Proceedings of Game Developers Conference, 1999, 763-782.

[JIANG] B. Jiang, *Agent-based Approach to Modeling Environmental and Urban Systems within GIS*, In Proceedings of 9^{th} International Symposium on Spatial Data Handling, Beijing,2000.

AUTHOR'S BIOGRAPHY

Roberto de Beauclair Seixas works with Research and Development at Institute of Pure and Applied Mathematics - IMPA, as member of the Vision and Computer Graphics Laboratory - Visgraf. He got his Ph.D. degree in Computer Science at Pontifical Catholic University of Rio de Janeiro - PUC-Rio, where he works with the Computer Graphics Technology Group - TeCGraf. His research interests include Scientific Visualization, Computer Graphics, High Performance Computing, GIS, Simulation Systems and Warfare Training Games. Currently he is the advisor of the Warfare Games Center of the Brazilian Navy Marines Corps.

Gustavo Henrique Soares de Oliveria Lyrio works with the Computer Graphics Technology Group - TeCGraf. He got his B.Sc. in Computer Engineering at Pontifical Catholic University of Rio de Janeiro - PUC-Rio. His interests include Computer Graphics and Warfare Training Games. Currently he is developer of the Warfare Games Center of the Brazilian Navy Marines Corps.