A Game-based Animation Tool to Support the Teaching of Formal Reasoning

Flávio Soares Corrêa da Silva Filipe Corrêa Lima da Silva

LIDET – Laboratory of Interactivity and Digital Entertainment Technology
Department of Computer Science, University of São Paulo
Rua do Matão, 1010 – São Paulo (SP) BRAZIL 05504-090
{fcs,filipe}@ime.usp.br

Abstract

When teaching formal reasoning in Computer Science courses – e.g. in an Artificial Intelligence or in a Formal Logics course – it is challenging to find compelling practical examples to motivate the students. We introduce in the present article a system to provide students with animations in virtual environments of the interactions of multiple agents, in which the interaction protocols and the behavior of the agents are specified as logical theories.

The system introduced here has been primarily designed as a teaching aid for undergraduate students enrolled in disciplines such as Artificial Intelligence and Formal Logics. We nevertheless envisage different applications for this system, such as a visualization tool for research development in Artificial Intelligence, and a high level specification tool for prototyping of complex computer games.

Keywords: artificial intelligence, games in education

Authors' contact:
Universidade de São Paulo, Depto. De Ciência da Computação. Rua do Matão, 1010 – São Paulo (SP) Brazil 05504-090 – email: {fcs, filipe}@ime.usp.br

1. Introduction

When teaching formal reasoning to Computer Science students – e.g. In an Artificial Intelligence or in a Formal Logics course – it is challenging to find compelling practical examples to motivate the students. It is common that lecturers resort to examples related to safety critical systems – for which formal specification and formal verification are a basic requirement – or to very large systems, such as some modern operating systems – whose design and implementation becomes unmanageable without the disciplined employment of formal methods – or else to widely used systems, such as search engines and systems to support web-based transactions (e-commerce, e-banking, etc.) – which must be permanently available, highly stable and highly reliable, and for which, as a consequence, a provably predictable behavior is most desired.

Even though these examples can be very convincing, it is unlikely that students have the opportunity to experiment with any of them during their courses. It can be frustrating to some students to end up working with highly abstract and/or highly simplified problems that stay quite distant from the motivating examples shown to them.

In the present article we propose artificial agents populating virtual environments as an alternative field of application of formal reasoning techniques. The growing sophistication of software systems for digital entertainment – most remarkably pushed by the computer games industry and interactive digital television – has made these systems akin to two of the classes of examples mentioned above – they are large and complex, as well as very widely used. Hence, we shall see a growing utilization of formal methods to support their design and development. Moreover, these systems can be scaled down to fit into classroom examples and exercises, and hence students can have the feel of working on such systems as well as the satisfaction of seeing their own work completed and finalized as a full working system. Furthermore, given an appropriate interface, these systems provide for truly entertaining experiences, thus offering to the students immediate reward for their efforts.

We introduce a software system specifically designed to support the teaching of formal reasoning techniques to Computer Science undergraduate students. The system has been primarily designed as a teaching aid for undergraduate students enrolled in disciplines such as Artificial Intelligence and Formal Logics. We nevertheless envisage different applications for this system, such as a visualization tool for research development in Artificial Intelligence, and a high level specification tool for prototyping of complex computer games.

The use of computer games development to teach computer science has been advocated by many researchers. Among the most remarkable initiatives, we can mention the Alice project [DANN ET ALLI, 2006], the MUPPETS project [BIERRE ET ALLI, 2006] and the Age of Computing project [NATVIG AND LINE, 2004] for introductory courses. Game development is an engaging activity for Computer Science students, because the end result of their work is appealing and

they can have quick feedback of their efforts to design and implement a software system.

There are indications that artificial intelligence and multiagent systems can improve significantly the quality of the interaction with computer games, contributing to the design and implementation of more believable computer-controlled characters [FORBUS ET ALLI, 2001]. Moreover, computer games have been proposed as the ideal testbed for artificial intelligence techniques [LAIRD AND VAN LENT, 2000].

Based on these considerations, we believe that a system for prototyping of interactions among several computer-controlled agents, in which the interaction protocols and the behavior of agents could be specified using a high level language – e.g. as close as possible to first order logic – can be a useful tool.

We have developed a virtual world in which computer-controlled characters interact according to what is specified in logical theories written as ***PROLOG*** programs.

In section 2 we describe the general architecture of our system. In section 3 we present a sample application, in which a single agent interacts with the environment looking for the shortest feasible path between its present location and a goal location. In section 4 we briefly describe some potential applications of our system in the classroom. Finally, in section 5 we present our conclusions and proposed future work.

The present article is a follow up to the project presented in [SILVA AND CORRÊA DA SILVA, 2005].

2. General System Architecture

Our system acts as a server that orchestrates the interactions between the 3D Engine, the ***PROLOG*** Engine and the student's ***PROLOG*** client code (Figure 1). We've decided to use the ***Ogre*** open source 3D engine [STREETING 2006], and the open source ***SWI PROLOG*** engine [WIELEMAKER 2006] to develop the system.

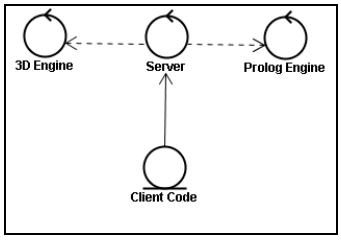

Figure 1: Application Architecture

Ogre is a scene-oriented 3D engine. It is written in ***C++*** and it is designed to facilitate the development of applications that require hardware 3D acceleration. ***Ogre*** abstracts away the details to use the graphical libraries *DirectX* and *OpenGL*, and can be compiled in different platforms, including Windows, Linux and MacOS. ***Ogre*** is not a game engine; rather it is a generic solution for real time rendering. It is an extensively and well documented open source project, and counts on a large and enthusiastic community of users and developers.

SWI PROLOG is one of the most widely used open source ***PROLOG*** implementations. It provides very good quality interpreter and compiler for the ***PROLOG*** language, wrapped in a high quality programming environment.

Both ***Ogre*** and ***SWI PROLOG*** are available under LGPL license.

The ***SWI PROLOG*** has a critical feature that allows us to embed the ***PROLOG*** engine as a DLL into a ***C*** application, thus allowing us to delegate calls to it inside the ***C*** code. As an example, we can load a ***PROLOG*** code in ***C*** with the code snippet presented in Figure 2:

```
1 PlTermv pt(1);
2 pt[0] = "someprologcode.pl";
3 PlQuery q = PlQuery("consult", pt);
4 q.next_solution();
```

Figure 2: A code snippet for loading a ***PROLOG*** code file inside a ***C++*** application using an embedded ***SWI PROLOG*** engine

This snippet loads the `someprologcode.pl` file into memory and into the ***PROLOG*** engine instance. Now we can use the services that are exposed by the engine to query the loaded code using the ***PROLOG*** unification mechanism and dynamically add and retract facts or rules to the engine instance. This way we can effectively delegate computing from the server to the engine.

The server and the ***PROLOG*** client code must cooperate through the enforcement of a 'contract'. That is, the server will expose some functionality for the client code to use when developing the agent's logic. The functionality of the server can be used in the client code through the writing of ***PROLOG*** predicates with specific signatures. It's the client's responsibility to write the predicates exactly as the server expects.

In our case, we want to give the client the possibility of controlling an agent that lives in a 3D world. As an example of interaction, a possible functionality exposed by the server can be driving the agent through the 3D world. The server will load the ***PROLOG*** client code and will try to unify a specific predicate that instantiates a list of ordered nodes. This list corresponds to the path that the agent must travel through to reach his goal. We'll see this process in detail in the next section.

On the server we have built an infrastructure to deal with the problem of orchestrating the system's components. The class diagram is shown in Figure 3.

3. Path Finding as a Logical Problem

Consider a robot in an environment as depicted in Figure 3. The robot has information about its present location (as a pair of coordinates), and receives a goal location (i.e. a second pair of coordinates). To simplify the computation of locations and the formulation of location-related problems, we assume that the environment is discrete, and locations are given based on a square grid. Our proposed problem – to the robot – is to find the shortest path between the present location and the goal location.

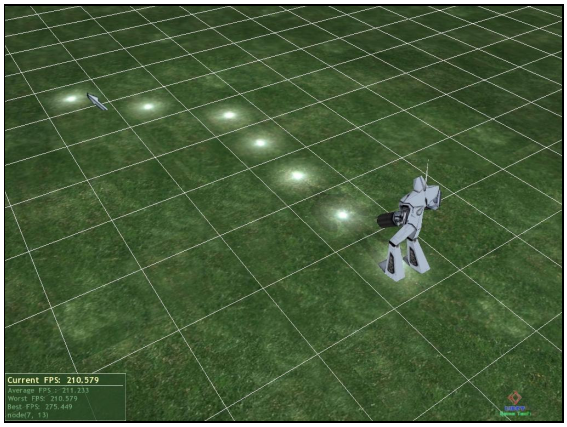

Figure 3: A robot finding the shortest path between its present location and a goal location

This problem can be formulated in many different ways, depending on the language and the formalism to be emphasized during a lecture presentation. For example, we can formulate it as a problem in first order logic:

> *Given a set of axioms that specify the legal movements of the robot, two axioms stating the present location and the goal location of the robot, a set of axioms to capture the intended meaning of a* path *between present and goal locations (namely, a list of connected locations with the present location as first location and the goal location as last location), and the formalization of the notion of* length *of a path (namely, the length of the list representing the path), find a proof for the following first order sentence: there exists a path* $\boldsymbol{p_0}$ *between present and goal locations, such that for all paths* $\boldsymbol{p_i}$ *between the same present and goal locations, the length of* $\boldsymbol{p_i}$ *is greater than or equal to the length of* $\boldsymbol{p_0}$.

These axioms can be encoded in a ***PROLOG*** program. Clearly, we are interested in a constructive proof, since we would like not only to know that there exists a path with the desired properties, but also what this path is.

This problem becomes more interesting if we can add some walls to the environment, thus building a maze for the robot to traverse on its way to find the shortest path between present and goal locations. One such maze can be presented as in Figure 4. These walls are constraints to the legal movements of the robot, and hence can be logically represented as some additional axioms.

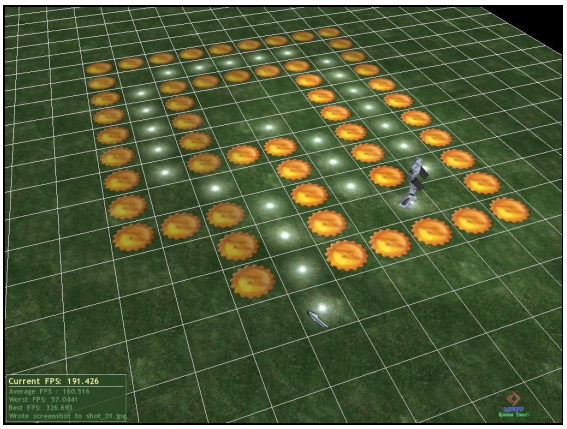

Figure 4: A robot traversing a maze to find the shortest path between its present location and a goal location

Finding the shortest path between two points in a plane containing a maze can be explored in many different ways in different disciplines within a Computer Science course: it can be used to illustrate important issues e.g. in the design and analysis of algorithms, first order logic, formal software design and implementation, and artificial intelligence.

This problem can become much more compelling if the students have the opportunity to appreciate the "practical" results of their work. Our system allows the logical theory that characterizes the problem to be partially specified through interactions with the virtual world inhabited by the robot – namely, the present and goal locations and the geometry of the maze can be drawn directly in the graphical representation of the world. If the constructive proof is successful – i.e. if there exists a shortest path between the intended locations – the shortest path triggers an animation in which the robot walks from its present location to its goal.

It is left to the lecturer to specify what events in the virtual world update the logical theory ruling the behavior of the agents in the environment, as well as what theorems in the logical theory trigger animations in the virtual world. The lecturer can then propose to his/her students the formulation and implementation (as ***PROLOG*** programs) of algorithms / executable specifications / logical theories to connect axiomatic theories with constructive proofs of specified theorems that generate corresponding relevant animations.

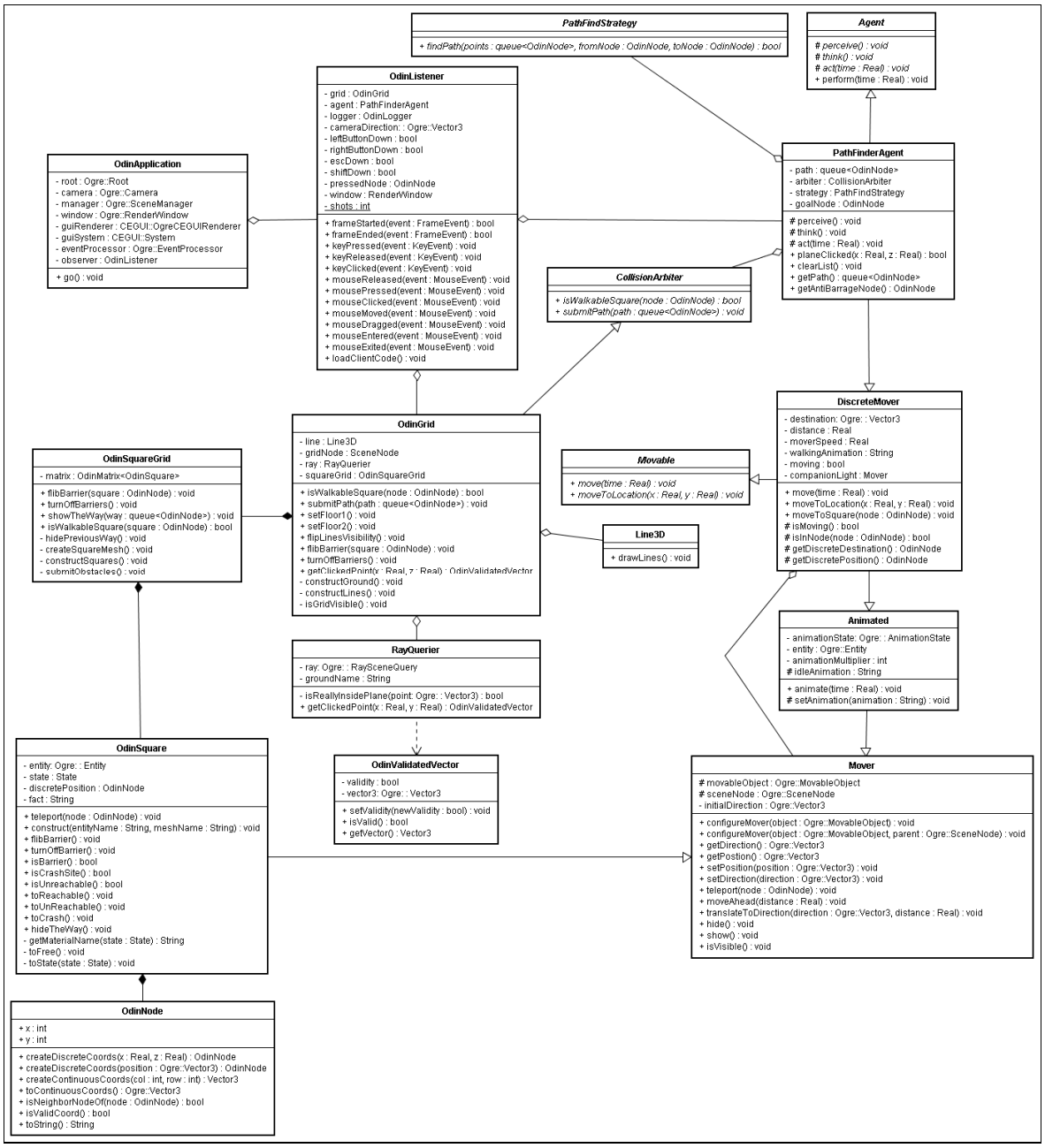

Figure 5: Class Diagram of the System

In our example, whenever a block is added to a cell in the grid to constitute a maze, a ***PROLOG*** fact is added to the theory, of the form:

```
obstacle(node(X, Y)),
```

which indicates that the node on position X, Y of the grid is an obstacle.

In order to select a goal location in the grid, the user has to click on the desired spot on the grid. This adds a ***PROLOG*** fact to the theory, of the form:

```
goal(node(X, Y)),
```

which indicates that the node(X, Y) is the goal node of the agent. The present location of the robot is passed through the ***PROLOG*** goal clause of the form

```
findpath(From, Path),
```

which is triggered whenever a new goal location is selected. Here `From` is the robot's current position in the form `node(X, Y)`. `Path` will represent the instantiation of a list of nodes.

Depending on the logical theory that is implemented in the rest of the ***PROLOG*** program, a path is found connecting the present location with the goal location. This path is then passed to the animation engine, which effectively takes the robot from its present location to the goal location. Then the cycle can be restarted: blocks can be added or deleted from the maze, the last goal location becomes the new present location, a new goal location can be selected, the ***PROLOG*** goal clause is triggered, a new path is generated and the robot walks from present location to goal location.

We provide the class diagram of the system in figure 5. The invocation of the client code happens on a concrete subclass of *`PathFindStrategy`*.

4. Virtual Worlds in the Classroom

The simulation described in section 3 certainly opens up for interesting opportunities for lectures in AI courses. But while it is interesting to have a ***PROLOG*** interface to program a virtual agent, the system is not limited to simulate an environment with only one. We can have multiple agents dwelling in this virtual scenario.

A simulation with multiple virtual agents brings important problems into the system. A first consequence is the problem of how do you say to do server which set of predicate maps to which agent in the simulation in the client code.

For example, if we have two types of agents in the environment and we would like each one of them to have different algorithms for path finding, we'd have to design two different signatures for the `findpath(From,Path)` predicate. This happens because the ***PROLOG*** engine loads different code files into the same program space while each agent instance must have its own scope, or view of the world.

Thus, even if we have two exactly equal robot agents, according to the proposed scenario in section 3, we would have to change the predicates because each of the robots must have its own goal predicates. For example, we could use a goal predicate of the form:

```
goal(node(X, Y), AgentIdentifier),
```

and we would have to change the `findpath` predicate implementation to deal with the new parameter.

The second consequence is that the dynamism brought by new agents into the environment has to be taken into account by the student when he is designing his logic. A shortest path found by a computation few seconds ago has a chance of not being the shortest path anymore as of now because of the new position assumed by other agent in the nearby surroundings.

The third consequence is that dynamic possibilities of collision come into play allied with the problem of correctly managing the continuity and discrete aspects of the environment. The animation aspect of the simulation is continuous, the agent will incrementally move from square to square, frame by frame. But the agent, and hence the student coding his behavior, sees the world as a grid formed by squares, a discrete perception. This brings synchronization problems when dealing with collision between many agents.

The collision problem is illustrated in figure 6: the agent a1 calculated his path p1 from square s1 to s8, while the agent a2 calculated his path p2 from square s6 to s4. We can see clearly a possibility of collision between the two agents on the square s5, depending on their individual speeds. Neither of the agents knows about this possible collision because they do not precisely know each other trajectories.

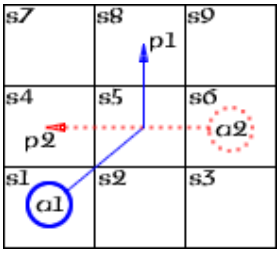

Figure 6: Multiple Agents Collision Problem

We can also have one square discarded by an agent calculating his path, because the square is currently occupied by other agent. To illustrate this, we can look at figure 7, and assume that agent a1 and a2 have the same speed. We can see that even though the square s5 will be available by the time the agent a1 arrives, because of the path p2 chosen by a2, agent a1 will not be able to calculate his path p1 due to the temporarily square s5 occupied by a2.

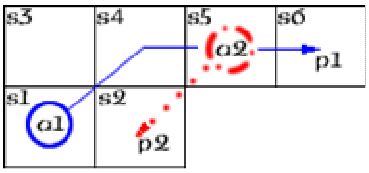

Figure 7: Multiple Agents livelock problem

This situation has the ingredients for a livelock, where agent a1 and a2 keep changing their positions and effectively blocking each other way.

5. Conclusions and Future Work

In the previous section, we exposed the problems that arise when turning our system into a multiple agent simulation environment. The next step in our work is to work on these problems with a concrete example, that we think it is interestingly enough to be taken to the classroom. That would be the problem of the sheep, the shepherd and the starving wolf.

In this scenario we have three types of agents and each one has one specific long term goal. The objective of the shepherd is to take the sheep from one corral on the side of the scenario to the other, while trying to protect the sheep from the wolf. The objective of the sheep is trying to survive the wolf. The objective of the starving wolf is to survive, according to the food chain.

This allows for some interesting interactions among the students. We could have one group of students to code the logic for the shepherd, and other group coding the logic for the wolf. We could explore communication problems between the sheep, and also the problem of walking in formation.

We have shown that our system poses as an interesting approach for teaching AI because it builds a bridge between the algorithms and its application with a real example. With the multi-agent version we hope that the system turns out to be a great tool for educators to use in AI courses in order to build a motivational learning environment.

Acknowledgments

This work has been partially funded by Microsoft Research.

References

BIERRE, K., VENTURA, P., PHELPS, A., EGERT, C. Motivating OOP by Blowing Things Up: an Exercise in Cooperation and Competition in an Introductory JAVA Programming Course. ACM SIGCSE Workshop. USA, 2006.

DANN, W., COOPER, S., PAUSCH, R. Learning to Program with Alice. Prentice-Hall, 2006.

FORBUS, K., MAHONEY, J., DILL, K. How Qualitative Spatial Reasoning can Improve Strategy Game AIs. AAAI Spring Symposium on AI and Interactive Entertainment. 2001.

LAIRD, J. E., VAN LENT, M. Human-level AI's Killer Application: Interactive Computer Games. Invited talk, AAAI Conference. 2000.

NATVIG, L., LINE, S. Age of Computers: Game-based Teaching of Computer Fundamentals. ACM SIGCSE Bulletin, v. 36(3), pages 107-111. 2004.

SILVA, F. C. L., CORRÊA DA SILVA, F. S. Um Ambiente Virtual Baseado em Jogos para o Aprendizado de Inteligência Artificial. IV Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital. Brasil, 2005.

STREETING, S. OGRE: Object Oriented Graphics Rendering Engine. http://ogre3d.org. 2006.

WIELEMAKER, J. SWI Prolog. http://swi-prolog.org. 2006.