A Platform for Teaching Logic Programming using Virtual Worlds

Spyros Vosinakis  
University of the Aegean,  
Product & Systems Design Eng.  
Syros, Greece  
e-mail: spyrosv@syros.aegean.gr

George Anastassakis  
University of Piraeus,  
Department of Informatics  
Piraeus, Greece  
e-mail: anastas@unipi.gr

Panayiotis Koutsabasis  
University of the Aegean,  
Product and Systems Design Eng.  
Syros, Greece  
e-mail: kgp@syros.aegean.gr

Abstract— Logic Programming (LP) is an essential part of many academic curricula and it is extensively employed in the field of Artificial Intelligence. However, being based on a fundamentally different paradigm and lacking any visual tools for inexperienced users, its teaching may lead to confusion and low student motivation. Virtual Worlds (VWs) may help overcome these obstacles, as they have been successfully used in computer programming education. In this paper we present the MeLoISE platform (Meaningful Logical Interpretations of Simulated Environments) for teaching LP in VWs, through which students can experience a collaborative visual interface to the Prolog programming language. We also present an evaluation of MeLoISE, which was conducted during a learning activity that engaged students in collaborative programming for 2 problem scenarios. Students performed very well and were enthusiastic with the new environment.

Keywords – virtual worlds; collaborative learning; logic programming; Prolog; collaborative programming

I. INTRODUCTION

Logic Programming (LP) is a programming paradigm based on mathematical logic. It is fundamentally different to traditional, imperative programming in that logic programs do not encode algorithmic solutions to given problems but, rather, high-level declarative knowledge about the very nature of the problems in the form of facts and inference rules [1]. Accordingly, logic programs are not executed per se; in LP, solutions are obtained by proving propositions issued as queries (or goals) in the context of the program. One of the most widespread representatives of the LP paradigm is the Prolog language, which relies upon first-order predicate calculus. LP is a powerful computational tool of great use to Computer Science (CS) professionals. It is ideally-suited to situations where arbitrarily-abstract representations, symbolic processing and logical inference are of the essence. For such reasons, LP has been extensively used in the Artificial Intelligence (AI) domain. Knowledge of LP principles can substantially broaden a CS professional’s perspective by leading to a deeper understanding of computer programming and, consequently, LP is an essential element of many academic curricula.

However, teaching LP and Prolog are not without problems. Students often find themselves overwhelmed by having to accommodate new knowledge about the declarative approach next to previously-accumulated programming experience gathered from the imperative standpoint. Furthermore, students may also find themselves confused by the lack of specific, directly executable commands or by the lack of an execution model whose flow of operations and results can be directly anticipated, observed and debugged. As a result, many efforts have been put towards making the process of teaching LP to academic-level students more effective. Some approaches focus on visualizing the proof process [2] while others rely on visualizations of knowledge, syntactical elements as well as inference procedures (for example, [3][4][5]). However, in most cases, visual elements of a rather technical nature (graphs, box diagrams, etc.) are used even when the representations involved are of concrete objects which can be depicted directly and naturally.

Logic programs essentially declare relationships among conceptual or concrete objects of various types. The case with Virtual Worlds (VWs) is strikingly analogous, since they consist of synthetic objects in arbitrary spatial, structural, etc., arrangements. Perceiving objects and verifying their relationships in a VW is visual and as such, it is straightforward and intuitive. In addition, VWs are interactive by definition and thus enable equally straightforward manipulation of objects, including their repositioning and restructuring. Moreover, the evolution of a VW’s state is directly observable and often controllable by means of specialized facilities. For such reasons, VWs seem to be a highly suitable alternative for visualizing logic programs and their operation, as well as for maintaining partial, two-way conceptual mappings between logic programs and synthetic environments. As such, they present an increased potential to support effective LP learning approaches.

VWs have been used as an educational tool in general and for the purposes of teaching computer programming in particular on numerous occasions until today in both academic and lower-grade courses (for instance, see [6][7]). Alice, for example, is an interactive 3D environment in which students create programs in a simple scripting language to manipulate objects (e.g., animals and vehicles) and thus observe the results of their code directly [8]. The goal is to increase understanding of the function of various programming language constructs. Another example is the InterReality Portal, a mixed reality collaborative learning environment used in problem-based learning scenarios in which geographically-dispersed students pursue learning objectives that require the combined use of both hardware and software modules [9]. These platforms are examples of dedicated implementations designed and built so as to support specific computer programming learning scenarios. Other efforts (for example, see [10][11]) rely on general-purpose solutions, most notably the online worlds of Second
Life (http://secondlife.com) and its open source alternative, OpenSimulator (OS) (http://opensimulator.org). However, despite the fact that VWs have been and are still being used as tools for teaching computer programming, their potential to support LP learning does not seem to have been extensively investigated.

In this paper we present the MeLoISE platform (Meaningful Logical Interpretations of Simulated Environments) for supporting LP education using the Prolog language in OS. Our approach combines declarative thinking, problem-based learning practices and program-based examples, thus including elements of a variety of established Prolog learning trends [12]. MeLoISE aims to:

- enable the visual interpretation and verification of Prolog program results in a straightforward and intuitive fashion;
- facilitate students in getting started with LP by automatically generating a set of facts about the contents of the 3D environment;
- encourage students to adopt a collaborative approach towards attaining learning milestones; introduce an intuitive, natural modality for user-interaction with a Prolog knowledge-base; last but not least, make students enjoy their learning experience.

We have conducted an evaluation of MeLoISE during a learning activity that engaged students in collaborative programming for 2 problem scenarios. The results are very encouraging since students performed well and were enthusiastic with the new environment.

II. DESCRIPTION OF THE LEARNING PLATFORM

MeLoISE comprises of a set of tools that enable the interconnection between a Prolog engine and multiple educational scenarios running in the OS environment. Each scenario takes place in a part of the VW, the scenario space, and involves only the avatars and objects inside it. The platform can automatically generate a symbolic description of the space based on the type, appearance and spatial arrangement of its contents. This description is expressed using appropriate Prolog facts and rules. Users can then enhance it by adding their own code and produce more complicated representations that involve further properties and relations describing the scenario. E.g. in an museum scene one could add facts declaring that some of the objects are works of art and rules that decide whether a person is looking at one based on her relative position and orientation to it. While immersed in the VW users can ask queries about the scenario and read the reply from the Prolog engine. The contents of a scenario space may be rearranged by moving, adding or removing objects and users, and the code may be tested again in the new circumstances.

A. Architecture

The architecture of MeLoISE follows a client-server model, in which a number of interface units placed in the centre of each scenario exchange data with a knowledge base server hosting the Prolog engine (Fig. 1). The interface unit is a visible 3D object in the VW, whose behavior is scripted in LSL/OSSL, the scripting language of OS. It handles both the communication with the users and the connection with the remote server. The server is implemented in Java and is responsible for communicating with the Prolog engine to submit user queries, and sending back the results to the interface unit. For each scenario, the server utilizes a respective Scenario Manager component, which manages and updates the required data structures for submitting the scenario description to the Prolog engine. These are: a) an intermediate geometric representation of the environment, b) the facts that have been automatically generated for all scenario entities and c) the user-submitted code. The Prolog engine is running in SWI-Prolog (http://swi-prolog.org) and the jpl API has been used for its interface with Java. All technologies used are open source.

The automated representation of a scenario in Prolog involves three steps: the transmission of the current status of the scenario space to the knowledge base server, the update of its geometrical representation, and the production of respective facts. Initially, the interface unit scans a spherical area around it for objects and avatars, collects their geometric properties and relations and submits them to the server. The latter calls the appropriate Scenario Manager to update the geometric description of the space based on the received data. In the geometric description all entities of the scene are represented as oriented bounding boxes, which are defined by their current position, orientation and size. This structure is useful for determining spatial relations and properties of entities that will be used by Prolog rules, as we shall describe later. Finally, all newly identified entities are assigned a unique name that complies with Prolog restrictions for naming objects (atoms), and the following facts are generated about them:

- object(<name>) or avatar(<name>), depending on their type, where <name> is their assigned name
- shape(<name>,<shape>), for objects only, where <shape> describes their shape, e.g. box, cylinder, prism, etc.
- color(<name>,<color>), for objects only, where <color> describes their color using the nearest name that matches its RGB color values, e.g. red, orange, brown, etc.
- sitting_on(<name>,<obj>), if <name> is an avatar, <obj> is an object and the avatar is sitting on it.

Users may also declare the type of an object by adding it to the object’s description in the VW. In that case, a
respective fact is also generated, e.g. if the type of the object my_chair is declared as ‘chair’, the fact ‘chair(my_chair)’ is added to the set of facts that describe the scene.

Scenario descriptions are enhanced with Prolog rules for determining spatial relations and geometric properties of entities. Upon call, these rules use the jpl API to trigger respective Java methods, which evaluate them based on the current geometric representation of the scenario. The supported relations and properties are summarized in Table 1. The first four relations (front_of, behind, left_of and right_of) are directional: they are calculated based on the position of the first entity with respect to the position and orientation of the second. The next three (above, below and between) are based on the absolute positions of the entities involved. The relations ‘near’ and ‘far’ are calculated based on the distance of the two entities with respect to the second entity’s size. The ‘on’ relation holds if the lower part of the first entity is touching the upper part of the second, and ‘intersect’ holds if the two bounding boxes collide with each other. The remaining rules return geometric property values: the size of an entity, as a volume (size_of) or in every individual dimension (size_x, size_y and size_z), the position of an entity in the three dimensions (pos_x, pos_y and pos_z) and the distance between two entities (distance).

TABLE I. SUPPORTED RELATIONS AND PROPERTIES.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>front_of</td>
<td>Entity &lt;e1&gt; is in front of / behind / left_of / right_of of entity &lt;e2&gt;</td>
</tr>
<tr>
<td>behind</td>
<td></td>
</tr>
<tr>
<td>left_of</td>
<td></td>
</tr>
<tr>
<td>right_of</td>
<td></td>
</tr>
<tr>
<td>above</td>
<td>Entity &lt;e1&gt; is above / below entity &lt;e2&gt; / between &lt;e2&gt; and &lt;e3&gt;</td>
</tr>
<tr>
<td>below</td>
<td></td>
</tr>
<tr>
<td>between</td>
<td></td>
</tr>
<tr>
<td>near</td>
<td>Entity &lt;e1&gt; is near/far entity &lt;e2&gt;</td>
</tr>
<tr>
<td>far</td>
<td></td>
</tr>
<tr>
<td>on</td>
<td>Entity &lt;e1&gt; is on entity &lt;e2&gt;</td>
</tr>
<tr>
<td>intersect</td>
<td>Entities &lt;e1&gt; and &lt;e2&gt; intersect</td>
</tr>
<tr>
<td>size_of</td>
<td>The b. box volume of &lt;e&gt; is &lt;s&gt;</td>
</tr>
<tr>
<td>size_x</td>
<td>The b. box size of &lt;e&gt; in the x/y/z dimension is &lt;sx&gt; &lt;sy&gt; &lt;sz&gt;</td>
</tr>
<tr>
<td>size_y</td>
<td></td>
</tr>
<tr>
<td>size_z</td>
<td></td>
</tr>
<tr>
<td>pos_x</td>
<td>The x/y/z value of the center position of entity &lt;e&gt; is &lt;px&gt; &lt;py&gt; &lt;pz&gt;</td>
</tr>
<tr>
<td>pos_y</td>
<td></td>
</tr>
<tr>
<td>pos_z</td>
<td></td>
</tr>
<tr>
<td>distance</td>
<td>The distance between the centers of &lt;e1&gt; and &lt;e2&gt; is &lt;d&gt;</td>
</tr>
</tbody>
</table>

Users can refer to the generated facts and to the above-mentioned relations and properties in the code they submit and the queries they ask. E.g. a user may add a rule to the scenario code which implies that all red spheres within some size limits are balloons:

\[ \text{balloon}(X) \leftarrow \text{size}(X, 3), \text{color}(X, \text{red}), \text{size_of}(X, Z), Z \geq 0.5, Z \leq 0.1. \]

The user-generated code is written in the VW and transmitted through the interface unit to the knowledge base server causing the respective scenario manager to update the set of commands. If a query is asked, the scenario manager loads the generated facts and the user code to the Prolog engine, asks the query, and transmits the results through the server back to the interface unit. The reply is shown to the users inside the VW. E.g. user Alice could submit a query about the balloons that are in front of her:

\[ \text{balloon}(X), \text{front_of}(X, \text{alice}). \]

B. User Interface

The user interface of the learning platform is mostly command-line through the public chat channel of the VW. We decided to use the public chat to enhance awareness of other users’ activities in synchronous, collaborative scenarios. Users can type their commands to the platform in the chat box and read the replies from the server as a public chat message sent by the interface unit. The user code is written in a text file (notecard), which is placed in the contents of the interface unit and can be accessed by clicking on the object. The code file is visible and editable by all users who take part in the scenario. Finally, users may declare the type of scene objects by modifying their description. If the description starts with a dollar sign ($) the knowledge base server treats the rest of the description as the type of the object and adds the respective fact.

All user commands to the interface unit start with the word ‘kb’ (knowledge base) to easily distinguish them from the rest of the public chat messages. The commands are: kb update_scene to update the scene description, kb update_code to update the user code, kb query <query> to ask a specific query and get a single result, kb next to get the next result of the query and kb query_all <query> to get all the query results at once.

C. Educational Use

MeLoiSE can be used for teaching LP through demonstration of typical examples and solutions, collaborative problem solving, and student-driven construction of problem scenarios. In the first case, the instructor may create the scenario and use it to present alternative solutions and what-if cases by rearranging scene elements and/or altering the code. It is common in introductory AI courses to use graphs, tile-based worlds (e.g. Wumpus world) and simple puzzles (blocks world, towers of Hanoi, N-Puzzle, etc) as examples [14]; these can be easily constructed in OS. In the second case, students may be placed in pre-constructed scenario spaces, for which they are asked to simulate the facts and rules that can successfully address specific queries. These scenarios do not have to be limited to abstract primitive-based environments; they could also represent more realistic situations, such as the cafeteria scenario we used in our evaluation. The third case is a more constructive and playful variation of this approach, in which students both construct the scenario space and write the appropriate code to answer some queries. This would allow them to collaborate, explore various alternatives and progressively refine their environment and code. Finally, an extended version of this platform could support automatic assessment of student solutions. A scripted VW object could assess the student code by submitting various queries and checking the answer and by producing variations of the scenario world to test the generalizability of the solution.
III. LEARNING ACTIVITY AND EVALUATION

A. Learning Activity, Setting and Participants

The aim of the learning activity was to identify how students with basic skills of LP can benefit from the use of MeloISe during a typical laboratory course. The learning activity was designed with a focus on having students perform collaborative programming tasks in the VW in two problem scenarios.

![Image of a cafeteria scenario](image)

**Figure 2.** A screenshot of the cafeteria scenario

The activity lasted for a total of 3 hours in terms of the course (omitted for blind review). The participants were 14 MSc students, which were divided in 4 groups (of either 3 or 4) depending on their logic programming and VW skills. All students were quite familiar with Prolog, but a few of them (4 out of 14) had some experience with the use of VWS (each one of these students was appointed to each group to balance group skills). The learning activity involved:

1) **VW tutorial (30’).**
2) **Introduction to the learning platform (45’):** An introduction to the interface and functionality of the platform using simple examples.
3) **1st problem scenario - ‘Cafeteria’ (45’):** The scenario setting was a cafeteria with customers, chairs and tables (Fig. 2). The aim was to insert appropriate code to answer the following queries: (a) if a chair belongs to a table; (b) if a table is taken; (c) if two users are having a drink together.
4) **2nd problem scenario - ‘Composite Objects’ (45’):** The requested code should be able to recognize composite objects made up with primitive shapes: (a) if three spheres are at a pile; (b) if 6 primitive objects compose a snowman.
5) **Wrap-up and questionnaire (15’):** At the end of the learning activity, a semi-structured discussion was conducted about students’ experience with the use of the platform. They provided their views about the value of this approach, problems faced and if they would like to further explore the system in subsequent courses. In addition students filled in a questionnaire.

B. Data Collection Methods

Data collection took place during the evolution of the 2 problem scenarios, when we closely examined the students’ collaborative learning experience with the methods of:

1) **Observation:** Student behavior was continuously monitored both in the computer lab and the VW to assess their engagement, progress, problems-faced and outcomes.
2) **Concurrent probing:** Collaborative programming is a vibrant process of exchanging ideas, code writing and testing. Thus, we also took an active stance of posing questions to groups when they seemed to slow down or stuck to help them find their direction to a solution.
3) **Students’ self-reporting:** Firstly, during the problem scenarios each group wrote down a single-page table which listed: (a) knowledge description in natural language, (b) Prolog code, (c) result. Secondly, at the wrap-up, each participant filled in the questionnaire.
4) **Platform data and information:** Various data about student work, results and communication was recorded in:
   (a) VW chat logs (both student communication and commands to the KB), (b) KB code listing, (c) KB server log, (d) VW server logs.

C. Results

We report on the group task performance for each one of the problem scenarios based on group documentation and code listings and then on collaborative learning experience issues as these were perceived by students on the basis of their questionnaire answers and the wrap-up interviews.

1) **Group performance on collaborative programming for the 2 problem scenarios**

<table>
<thead>
<tr>
<th>Group</th>
<th>Scenario #1 (Cafeteria)</th>
<th>Scenario #2 (Comp. Obj.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q1</td>
<td>Q2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>.9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>.8</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group</th>
<th>Scenario #1 (Cafeteria)</th>
<th>Scenario #2 (Comp. Obj.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q1</td>
<td>Q2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>.9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>.8</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Evaluation is based on documentation and code; Q: Question; GP: Group Performance.

Group student performance was satisfactory overall, given the experimental conditions of the evaluation and the lack of familiarity with the environment. Two groups were significantly better and achieved to deliver almost all tasks assigned within a tight timeframe. The other two groups were comparatively deficient especially at the first problem-scenario. Since that group skills were balanced, we investigated at the end why these groups did not perform so well: their response was simply that they needed more time to absorb the basics of this course, to get familiar with the environment and basic syntax; for example, they came up with several na"ive errors at the beginning that hold them off for some time. Also, they required time to communicate and coordinate especially at the beginning of each problem scenario. We were encouraged to see that group performance improved significantly through time, and that all groups improved their performance in the second scenario.

2) **Perceived Collaborative Learning Experience**

It was obvious that all students were enthusiastic with this new environment for logic programming since they all worked intensely during the two problem scenarios. At the
end, they reported that they would certainly prefer the VW and MeLoISE as the programming platform for their course.

Students also filled in a questionnaire to measure important elements of the collaborative learning experience. The questionnaire comprised of a total of 15 questions concerning the three dimensions of: (a) learning effectiveness, (b) collaboration and (c) user experience. For each question students provided their opinion in a 5-point Likert scale. The analysis of responses followed the norm of analyzing self-reported data in user experience studies (e.g., [12], p. 127) including the computation of averages (overall and for each dimension) for all participants and between groups as well as the grouping of responses into the top-2 (positive responses) and bottom-2 boxes.

To provide a quantitative measure of the overall user experience, the averages of all student responses were computed and scaled up to the interval of [0,100] for each dimension. The average of student responses for learning effectiveness was 81.

Figure 3: Average user scores (for all users and per group) for dimensions of learning effectiveness.

![Figure 3](image)

Figure 4: User distribution by type of response (positive, neutral, negative) about dimensions of learning effectiveness.

![Figure 4](image)

IV. CONCLUSIONS AND FUTURE WORK

The MeLoISE platform was evaluated in a laboratory course that included teaching by demonstration and collaborative programming in problem-solving scenarios. There was high student motivation and engagement due to the collaboration style and the interactivity of the environment. Students did not face particular usability issues, although this should be further explored in future uses of the system. They also achieved their problem-solving assignments at a satisfactory level, given the experimental conditions of the learning activity.

We plan to continue exploring the educational potential of MeLoISE by deriving additional scenarios and evaluating them against student groups of various backgrounds in computer programming and at different levels of academic education. We also plan to formalize and extend the platform functionality with a bi-directional Prolog-OS interaction API which will enable specialized Prolog predicates to actually affect the VW’s state and, thus, make the visualization of the operation and results of logic programs possible.

REFERENCES


