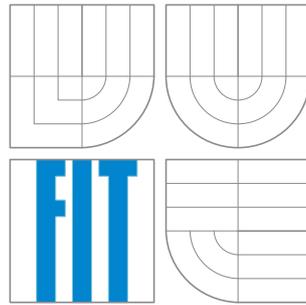


BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY



Agents simulation system

Principles of programming languages
and promotive C++ seminary project

Libor Polčák, xpolca03@stud.fit.vutbr.cz
Boris Procházka, xproch63@stud.fit.vutbr.cz
Martin Rapavý, xrapav00@stud.fit.vutbr.cz
Petr Zemek, xzemek02@stud.fit.vutbr.cz

2007

1 Introduction

Agents2007 is a *simple agents simulation system* that allows users to create simulation scene, put agents and other objects on it and define their behavior to achieve variability in actions these objects perform. Actions, like picking up objects from the ground and transferring them to another place, can be done by defining different behavior scripts. The application has a basic menu, control buttons, an objects palette and allows users to save objects into XML files and use them in another simulation just by adding them to the palette. Program help is also available (default browser has to be set correctly in your system in order to view it from application menu).

2 Teamwork and communication

Our team consists of four members. *Petr* (the team leader) was the main developer who put application parts together and his main area of interest was the simulation logic. *Boris* took care of graphical user interface, including requested graphical toolkit usage. *Libor* designed simulation objects and their configurations that could be saved into a configuration file, including proper XML toolkit usage. *Martin's* main area of interest was to find us a proper interpreter to be used in the program, its implementation and agents behavior description.

Every Monday we had a regular meeting where we were discussing the program design together. This was necessary because the project complexity required good design preparation. In this case, UML was a great design tool that we used.

Internet communication was done by using IRC/Jabber and it was also very important, because we live in different locations, so verbal communication was out off impossible. Source code management was provided by SVN¹, what allowed us to work simultaneously without concerns for rewriting teammate's version. Because of this and the quality reason we have chosen to write the documentation using L^AT_EX typographic system, in order to write clear and professional text. Plain text management was also really easy by using SVN.

We followed specific *coding stadard* (like naming conventions) based on some widely used standards so our sources were easy to read and understand. For the program documentation we used Doxygen² system.

Project design and used implementation methods follow.

3 Project design

We used two approaches when we were designing the application because it was “divided” into two imaginary parts: First part (*top-down approach*) is the graphical user interface (GUI), containing main simulation scene, objects palette and editation windows. Second part (*bottom-up approach*) contains simulation scene on the logical level, objects on the scene and their behavior, so this part is the intermediary between the GUI and the application logic. Only this way the *independance* of functionality, internal structures and data storages could be guaranteed.

Error handling is done by using *exceptions* because they provide a flexible way to react to exceptional circumstances (like runtime errors) in our program. We created a class hierarchy for each sphere of exploitation, for example configuration part has its own exceptions etc.

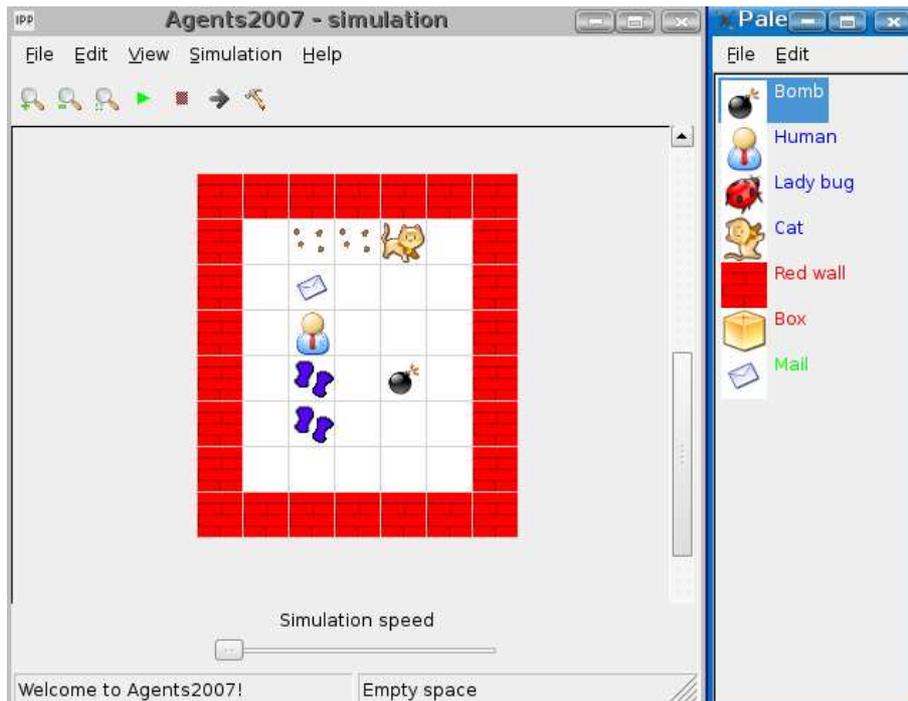


Figure 1: Agents simulation system screen shot

3.1 Graphical user interface

The graphical user interface is based on wxWidgets³, which is free, open-source cross-platform toolkit. GUI consists of four parts: application window, simulation scene, palette and editation windows. The core, application window, acts like a wrapper for whole application and provides option lists used to control the simulation. It has two modes: editing mode (allows manipulation with objects) and simulation mode (shows running simulation). Application logic is exposed to the GUI through the simulation scene, where it pumps all needed information, to draw the scene. Palette is represented as a container of objects and allows a user to put them into the scene. It also allows creation of a new one. Objects in the palette can be manipulated by loading/saving them from/into files, erasing or editing. Finally, editation windows provide a way how objects in the matrix or palette can be modified.

3.2 Simulation logic

On the logical level the simulation scene is represented by “two” dimensional array (*matrix*). The word two is in quotation marks since we had to implement *pseudo three dimensional array* because of some possible situations, like agent standing on another agent’s trail etc. The matrix is responsible for moving objects, performing their actions (like picking up a transferable object or putting trail underneath itself) and sending collisions to objects. It provides simple interface like putting object or destroying object on the selected place, making next simulation step and some member functions ascertaining simulation state.

We used *iterator design pattern* there, because of the independence between the GUI and the internal representation of the matrix. The third dimension is represented by a *container* on each

¹<http://subversion.tigris.org/>

²<http://www.stack.nl/~dimitri/doxygen/>

³<http://www.wxwidgets.org/>

non-empty place (*adapter design pattern* used there), also with basic interface. Because we need to keep some additional information when performing specific actions over the matrix (like saving objects configurations), *functor design pattern* is used. We are using *priority system* while deciding whether it is possible to put an object on the selected place or not.

3.3 Configurations

When we were discussing how to handle objects creation in the scene from the palette and XML files, we decided to use special class hierarchy, so called *configs*. It is a little bit similar to the *prototype design pattern*. This hierarchy has given us easier way to create the objects.

For working with XML files we have used *TinyXML*⁴ parser. Communication is done through our *ConfigProcessor* class, which has ability to open and save XML files.

We have decided to create every object in the matrix from its config. But not only objects are created this way. Whole matrix, simulation scene and application are created from configs. These objects can also store their settings into configs.

Configs also provide the way of storing settings into *XML* files. They have special virtual member function to save everything for later usage. These virtual member functions can be called directly from *ConfigProcessor* or “recursively”. For example, matrix contains many cells and every cell can contain more than one object, which can be an agent carrying another object.

Loading is done likewise. Every config has constructor with one parameter, which is a pointer to the *TinyXML* element class. It can check the validity of the XML file and create itself. When the element contains nested elements of another config, all configs are created recursively.

3.4 Objects on the scene

Each object is characterized by its name, description, direction (which way is the object heading to), action (what is the object doing) and image (how does it look like on the palette or on the scene). Specialized objects have of course more details. There are also various types of objects, which we will try to describe more precisely. All of them belong to our object class hierarchy.

Fixed objects cannot move, so their scope of actions is limited just to a “wait” action. Nothing special in here.

Transferable objects can be moved by agents from one place to another. Our solution of picking and dropping objects is following: When an agent takes a transferable object he moves to its position and information about the picked up object is saved. Then, when a “drop” action is set, he drops the carrying object underneath itself and moves to the first free place (clockwise testing sense). If there is no free place left, the dropping action is suspended until there is a free place to go.

Agents are the only objects that can perform almost every action. They are also scriptable (see page 5). If they have no behavior defined, they just go straight forward.

Agents can put *trails* underneath themselves, but only if they (agents) have set this ability (trail duration is greater than zero). There can be more trails on one place and when the top trail disappears, the second trail (if any) appears instead and so on. Trails have their “duration counter” decreased every simulation step, so it is easy to implement lifetime of a trail.

For objects creation we use special class based on *factory and singleton design patterns*, because this solution is extensible and utilizes from one of the basic object-oriented programming paradigms - polymorphism. We are creating objects based on their class, but no “switch” is used there. Factory implementation was taken from the *Loki* library (see page 6).

⁴<http://sourceforge.net/projects/tinyxml>

3.5 Scripting - defining agents behavior

Agents behavior can be extended via simple XML scripts. We use *Simkin*⁵ to parse and interpret these behavior scripts. Decision to use Simkin was based mainly on its ease of use and its features, especially ability to parse scripts directly from XML and access to underlying C++ objects. Last named feature is widely used in our application to create interface between a behavior script and application itself. *Singleton design pattern* is used to create one instance of script interpreter in whole program.

Simkin is a multi-paradigm interpreted language. It supports imperative and object-oriented programming. In our program we use only its imperative basis so it's quite easy to write behavior scripts also for non-programmers.

User's elementary task is to implement bodies of few predefined behavior methods, which agents use to set their next action during collisions or in the next action decision. User can define as many other methods as needed and call them as he likes. We provide user with simple interface to control agents in their behavior scripts. The interface consists of several agent's methods (without parameters), which the user can use to achieve desired behavior.

Example of a behavior script:

```
<function name = "ObjectAgentCollision">
    // method called when the agent collides with another agent
    if (HasObject()) { // calling built-in function returning bool
        TurnLeft();
    }
    else {
        TurnBack();
    }
</function>

<function name = "ObjectTransferableCollision">
    // method called when the agent collides with transferable object
    if (not HasObject()) {
        TakeObject();
    }
    else {
        RandomAction(); // agent performs pseudo-random action
    }
</function>

<function name = "ObjectFixedCollision">
    // method called when the agent collides with fixed object
    TurnBack();
</function>

<function name = "NotInCollision">
    // method is called when the agent is deciding its new action
    // (when it is not in collision)
    Go();
</function>
```

More detailed scripting description is available in the program help.

⁵<http://www.simkin.co.uk/>

4 Conclusion

All specifications from the assignment were observed during the project development. Each member of our team was interested in the project and critical parts of the design were discussed together. We were focusing on the application logic (including object design) and usability, because these parts are very important in the result. Main parts were designed considering possible future development and extensibility, like creating more object types that can be put on the scene or adding more built-in functions that can be used in scripts. The application was successfully tested on the faculty server *merlin*⁶ (Linux, x86-64) and on our systems (Linux, x86).

Used libraries

Standard C++98 library and STL

wxWidgets 2.8.0, <http://www.wxwidgets.org/>

TinyXML 2.5.2, <http://www.grinninglizard.com/tinyxml/>

Expat 2.0, <http://expat.sourceforge.net/>

Simkin 2.23, <http://www.simkin.co.uk/>

Loki 0.1.6, (Factory.h) <http://loki-lib.sourceforge.net/>

Loki library

Copyright (c) 2001 by Andrei Alexandrescu

Permission to use, copy, modify, distribute and sell this software for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

Metrics

Source files: 68

Source lines of code: 10364 (without empty lines)

Static data size: 26 KB

Executable file size: 896 KB

(Linux, x86-64, g++ 3.4.6, compilation without debugging information, no explicit optimizations)

⁶<http://merlin.ft.vutbr.cz/>