Computer-Aided Catalan Learning Application

Toni Sagristà

Final Year Project Report
2006/07

PART 1 of 2

Supervisor: Dr Ehud Reiter

Department of Computing Science
University of Aberdeen
King's College
Aberdeen AB24 3UE

Declaration

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Toni Sagristà

Date:

Acknowledgements

I would like to thank first and foremost my supervisor, Dr. E. Reiter for his advice and guidance, that surely helped me during the development of this project.

I'd also like to thank Francesc Alias from the university of La Salle for kindly giving me a hand with the Catalan text-to-speech. I would also like to give my thanks to Paula, Joseto and all the others that agreed to become testers of the project. Their feedback has been both useful and helpful.

Finally, I'd like to thank Paula again for always being there.

Abstract

Sometimes learning a new language from scratch using the traditional method (academy, school...) may become a boring or even a tough activity, specially when the student is on his first steps and must deal with loads of new vocabulary and grammatical rules.

The Computer-Aided Catalan Learning Application puts the student in a virtual world where he must manage fully in the target language, performing actions, looking at the world around, speaking to people, etc. This way the student is forced to wise up in a motivating user-friendly graphical environment whose purpose is to take the most advantage of the time. The more motivated the student, the quicker his learning.

Such learning tools are not aimed to replace the traditional methods but to complement them in the sense that they may help prevent the student's language skills from decaying.

Table of Contents

List of Figures

List of Code extractions

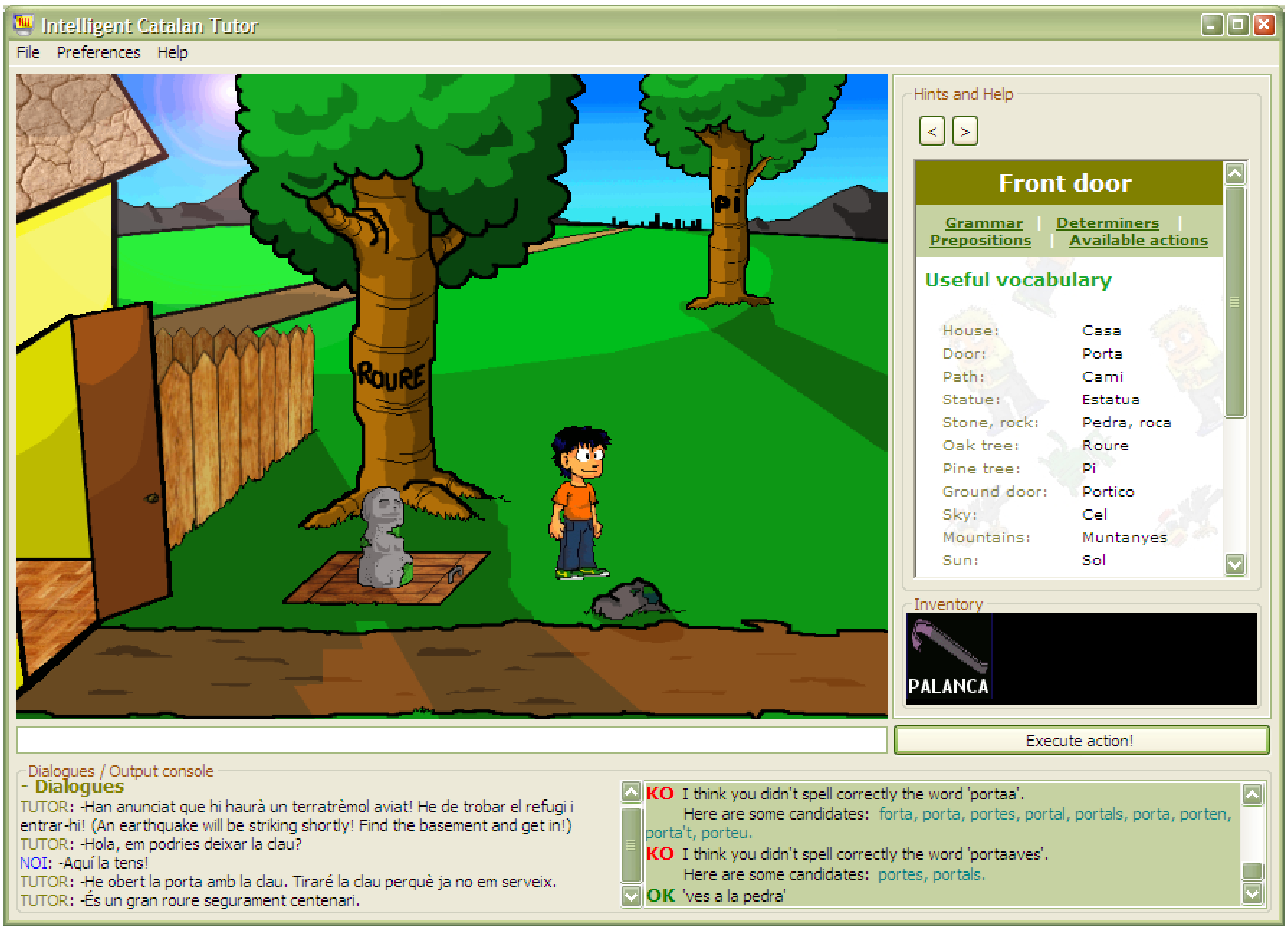
1 Introduction

This introductory chapter tries to give the reader a bird's eye view at the whole project, without entering in too much detail in any issue, discusses its objectives and goals and finally presents my motivations for taking it on.

1.1 Project overview

Learning a new language may become a boring or even a tough activity for the student, who has to deal with loads of new words and new grammatical rules. Traditional computer-based systems for helping people learn new languages (formally called *Computer-Assisted Language Learning* systems, *CALL*) just replaced the exercises one can find in any workbook intended for students learning a language and consisted, mostly, of fill-the-blanks or tick-the-correct-answer like exercises. However, there's still a newer set of *CALL* systems other than the traditional, called *ICALL* (which stands for *Intelligent Computer-Assisted Language Learning*), and that try to put more imagination and find better and more effective ways to teach a language.

The Computer-Aided Catalan Learning Application is located in the latter set of systems, and its aim is to immerse the user into a world where everything is designed for helping him learn the target language (see screenshot of Figure 1). The user will play a role in an adventure game (I like to call it a language game) where everything is in the target language, in our case, Catalan; everyone speaks to him in Catalan and even the character he's controlling produces only Catalan speech. This should be at least challenging for the student and he should feel motivated to advance through the game while he learns, since games are viewed as intrinsically motivating.



Front door

Grammar | Determiners | Prepositions | Available actions

Useful vocabulary

House:	Casa
Door:	Porta
Path:	Cami
Statue:	Estatua
Stone, rock:	Pedra, roca
Oak tree:	Roure
Pine tree:	Pi
Ground door:	Portico
Sky:	Cel
Mountains:	Muntanyes
Sun:	Sol

- **Dialogues**

TUTOR: -Han anunciat que hi haurà un terratrèmol aviat! He de trobar el refugi i entrar-hi! (An earthquake will be striking shortly! Find the basement and get in!)
TUTOR: -Hola, em podries deixar la clau?
NOI: -Aquí la tens!
TUTOR: -He obert la porta amb la clau. Tiraré la clau perquè ja no em serveix.
TUTOR: -És un gran roure segurament centenari.

```
KO  I think you didn't spell correctly the word 'portaa'.
      Here are some candidates:  forta, porta, portes, portal, portals, porta, porten,
porta't, porteu.
KO  I think you didn't spell correctly the word 'portaaves'.
      Here are some candidates:  portes, portals.
OK  'ves a la pedra'
```

Figure 1: Screenshot of the application

To become fluent in a language, one should exercise both the comprehension (whose main activities are listening and reading) and the production (which involves speaking and writing). This application helps the student get a bit used to Catalan with three of them: listening, reading and writing.

Of course, I don't pretend to say Computer Tutors will replace classroom communication or real immersion experience, but at least can help submitting a new learning experience.

1.2 Objectives

The Computer-Aided Catalan Learning Application abstract objective is basically to become a helpful tool for language students, specially those that are leaning Catalan. But there's a very important part that must be taken into account. The application and the process of learning must appear attractive to the student and he must feel motivated to use it. That's why the graphical stuff is as important as the error feedback or the spell checking themselves. Basically, the project aims to build a virtual world where the user can immerse himself and live in it completely in Catalan. Primary and secondary objectives are discussed in depth in the following subsections.

1.2.1 Primary objectives

The main primary objective is, basically, to build an application composed by mainly two parts (the graphical scene and the user interface) with the basic functionality described below. The *user interface* must have at least a field that allows the user to input commands and an output console. The scene must actively react to the sentences the user type in, as long as their correctness has been checked and ensured. If the sentence is not correct, then error feedback must be given to the user informing him of what is wrong.

We can also set as a primary objective the completeness of the graphical engine. Let me explain myself. Completeness means that anything reacting in the scene must appear attractive to the user so that using good-looking animations, sequences of animations or even sequences of actions might be a good practice. Each entity must appear live in the scene and should have, if possible, at least one idle animation. This is mandatory since a good-looking scene is deemed to attract the interest of students. The interest for seeing how the scene reacts to what one types in should keep one's motivation high.

Another important primary objective is to build the application in such a manner that the language it is actually teaching can be easily changed. It would be great to be able to adapt the application for teaching languages other than Catalan such as Italian, French or

Portuguese. Obviously one can do that given any design, even the worst, but the objective is to minimize the work needed to port the application to another target language.

1.2.2 Secondary objectives

Secondary objectives mainly consist of additional features that may be included in the system to improve the user experience and the teaching effectiveness. However, there are another set of secondary objectives that would imply a radical change in the way the application is actually working and that could now be classified as future work. These are, for example, reverting the flow of information, in the sense that it would be now the system who tells the user what to do and not vice versa (the system takes the initiative, in few words). Another example for this would imply a graphical interaction between the user and the scene so that the user would be allowed to perform some actions directly on the scene (like moving things around with the mouse) and the system would print a message in the target language describing the action the user just took.

The additional features set as secondary objectives are:

- Including a help window in the graphical user interface that displays some basics in Catalan grammar and vocabulary useful for the user to get in touch with the language.

- Including speech output so that the user can improve his listening skills too. If this objective is accomplished, some way to control the volume would be useful. Some background music may also be included to get a fancier application and enhance the user experience.

- Split the output console into two consoles, one for the dialogues the characters produce and another for the error feedback.

- Add a goal to the adventure. This means that the adventure game has a goal that must be achieved to clear it. It would give the user some extra motivation.

- Add a spell checker displaying some candidates in the output console when the user mistakes writing a word.
- As long as the project is programmed using *Java* and *Java* is a platform-independent language (code is compiled to an intermediate code which is later run by the *Java Virtual Machine*), the application should run in both *Windows* and *Linux* systems. It would also be useful to be able to run it on *Macintosh* systems (from *Mac OS X* on) but I won't include it in the secondary goals because I don't have access to any *MAC* machine to create the installation for it.

1.3 Motivations

In this section I'll discuss the motivations that brought me to take on this project.

1.3.1 Interests in NLP

In my home university in Barcelona I took a couple of subjects about compilers. They taught me the basics on building a compiler for a programming language. Actually, the practical for this subject took the student through the stages involved in building a fully-featured compiler for a functional programming language similar to Pascal. Soon I realized that the same background philosophy could be applied to build a parser for natural language recognition. Moreover, I think this area is quite interesting and a lot of research must still be done due to the indeterministic nature of the natural language. Researching on this field is the only way to get computers being able to understand human language and, who knows, maybe we could expect programming languages to become much similar to human language than they are now in the future.

1.3.2 Getting knowledge on game programming

Game programming has always interested me as well. The application takes the form of a very simple adventure game so that most of the concepts involving the development of such games appear in some sense in this project. Timing control, animations, sequences of actions, music and sound control and other features are also nice subjects to explore for me.

Furthermore, I consider I'm not bad at drawing and I think I can create good-looking graphics and images for the application.

1.3.3 Explore new ways to learn languages

Personally, I really reckon that there must be other ways to teach a language better than the traditional ones, or, at least, complementary ways to complete the traditional teaching. Such ways should be more motivating for the student and should get him involved with the target language. I emphasize this because keeping the student motivated is essential from my point of view and from my experience as a foreign language student at both school and academy.

1.3.4 The Catalan language background

The Catalan is a small language in terms of number of speakers. It is reported to have about 10 million of possible speakers (people who can understand, write and speak it correctly) spread throughout the territories of Catalonia, the Valencian Country, the Balearic Islands, the south of France, Andorra, the Aragonese border with Catalonia (called *La franja de ponent*) and the Alghero, a town in north-western Sardinia, Italy. However, the number of regular Catalan speakers is much lower since most of those 10 million persons have the Spanish as their first language. Nowadays Catalan is not being protected by the Spanish government and it is not even recognized by the European Union as an official language. So when my supervisor told me I could chose the language my application would be teaching I didn't think twice. I really think Catalan must be protected and any initiative to help it is welcome.

1.4 Structure of this document

The second chapter of this document aims to provide a broad vision of existing *CALL* and *ICALL* systems, as well as game engines, having a deep view on their functionalities. The third chapter discusses the design of the application and some important decisions made referring to it. The fourth chapter provides a deep view of the system itself and its implementation. It also speaks about relevant technical issues related to the development of the system. Finally, the fifth chapter provides a detailed look at the testing and evaluation of the system in relation to the goals described in this chapter and the sixth one contains the conclusions and some directions and ideas for future work.

2 Background and related work

In this section we'll discuss some related work previously done in this topic. We'll be discussing existing *CALL* and *ICALL* systems trying to classify them depending on their features and their approach to language learning. The languages and skill levels targeted, the *NLP* technology used and the teaching approaches differ widely from one to another. Additionally, we will also discuss some previous work done in other areas involved with the project such as game engines.

2.1 CALL and ICALL systems

The purpose of these systems varies from strictly research to strictly teaching passing through joint goals but this chapter is organized not in terms of purpose but in terms of type of learning approach, being either text-based tutors, text-based language games or graphics-based tutors.

2.1.1 Text-based tutors

This section presents some systems that have been influenced mostly by linguistic theory and that are mostly text-based with no images, both in terms of what the user types in and how the system prompts and reacts to it. They usually provide conventional kinds of exercises similar to those one can find in language workbooks, giving some grammatical feedback. Since such systems are not supported by visual aids, the emphasis is put on the parser and the lexicon.

Text-based *ICALL* systems vary widely in goals and methods, ranging in pedagogical methods from directed tutors to explanatory learning tools.

2.1.1.1 ALICE-chan – Japanese

ALICE is an acronym for *Automated Language-Instruction/Curriculum Environment*, and chan is a diminutive suffix in Japanese. It is used for first and second year Japanese instruction. *ALICE-chan* is a language-training environment for Japanese that uses *NLP* as a basis both for assisting instructors in preparing exercises and for evaluating student responses to exercises.

The *ALICE* project mission is to develop a multimedia foreign-language learning environment that supports the development and evaluation of many different types of language learning activities. The primary function of *ALICE* is to be a tool for research in second language acquisition, with the requirement that the research must be carried out in the context of normal language-learning activities, not in separate laboratory experiments.

Users need to have only basic linguistic knowledge such as the amount typically presented in beginning-level language textbooks.

ALICE-chan is modular so that it can easily be adapted to different languages and different levels of instruction. Changing the system involves changing the grammar rules and lexicons used by the *NLP* software. The rest of the software remains the same.

2.1.1.2 CALLE – Spanish

CALLE stands for *Computer-Assisted Language Learning Environment* and is a prototype language learning tool. *CALLE* supports foreign language learning by giving assistance in understanding and analyzing foreign language texts, such as documents and articles on a given topic. The system offers a window-based environment in which learners are presented a text for translation, analysis or some other practical task. Learners can query text at word or sentence level to morphological, syntactic and semantic information.

CALLE's original language is Spanish at the level of an intermediate college course. *CALLE* differs from many of the existing language tutoring systems in the sense that it focuses not on students' production, but in their comprehension.

2.1.1.3 BRIDGE – German and Arabic

This system was born in the post-cold war era when U.S. Soldiers had to be able to communicate with both their friends and their foes. Usually, military personnel needing foreign language skills are usually formally schooled in a classroom setting but later on it is up to the soldiers to maintain their language level during their off-duty hours. Training installations have classes for self-study but it was found out that soldier's skills, particularly production skills, got worse during this advanced training period.

The aim of the *BRIDGE* project was to develop a way to use *NLP* to analyze student input that is not in correct form, analyze multiple grammatical errors in the sentence and analyze the student's weaknesses and strengths to finally use this information to drive a lesson progression and remediation. This project was developed fully for German, with a parser extension to Arabic.

Main features

The *BRIDGE* is based in multimedia lessons which have to challenge the students in listening, reading and writing and adapt to student's individual capabilities. A variety of exercise types are available as building blocks for lessons. These include multiple choice, point-and-click on a graphic, fill-the-blanks, free responses and sorting text items under appropriate columns. Students receive feedback on the correctness of their exercise types. This feedback is produced as a result of those exercises sent to the natural language processor. Errors are classified into primary or secondary depending on their relevance. The progression from one exercise set to another can be in a fixed sequence or based on the student's performance.

There is also an on-line help which is available to the student from a drop-down menu item. This provides explanations of the tutor's capabilities and information on exercise procedures. For vocabulary help, there is a German-English dictionary that can be accessed either alphabetically or through search items.

2.1.2 Dialogue-based language games

This section covers the systems that link the text-based nature of systems discussed in the first section of this chapter and the graphic nature of the tutors discussed in the next section. These are dialogue-based language teaching games. Their main goal is to give students conversation-like practice in the target language. We will also discuss some system whose aim is not teaching but that can be used for that purpose too. They are all systems based mainly on text, like those in the first section, but some use the convention of a mistery scenario to constraint the dialogue, following the model of old commercial electronic adventure games, and the scenario exposed limits the language expected of students and simplifies the job of the natural language processor. Others, backed with powerful and compete databases, just offer general conversation about any topic. None has the linguistic theoretical considerations of the tools discussed earlier.

Notable in the dialogue-based tutors is the creative use of game conventions both to achieve pedagogical goals and to reduce computational complexity. For example most of those systems labels as *forbidden* to students those words that are not available in the lexicon and semantic network.

2.1.2.1 Spion – An AI Spy Game

Spion is a German-language adventure game for college language students, first programmed in 1981. Its designers intended to use its parser as a platform for the development of more extensive and more sophisticated tutorial German parsers. Although Spion and its daughter versions were indeed used by many college students of German, the program's phrase-structure-based syntactic element and its lexical-semantic element ultimately proved impractical for further development.

A syntactic parser and a semantic network are the basis of Spion, whose plot concerns spies in the East and West Berlin of pre-unification Germany. To play the game, students communicate by means of written commands or requests to the spy, Robotky, who describes events and objects through messages on the screen (with place names and locations of the real Berlin). The player must question Robotky for information, move him to various locations and tell him what actions to take. To win, the player must find secret information in a West Berlin chocolate shop, cross the border into East Berlin, exchange information with Frieda, a second spy, for other information, and return to the spy master, Max. Spion lacks a visual graphics context so that all the information the student can deal with is given by any of the characters as a text.

2.1.2.2 Herr Kommissar

Herr Kommissar ("Mr. Inspector") is a German-language intelligent computer-assisted language learning environment hiding a role-playing detective game. On entering Herr Kommissar's microworld, intermediate German students assume the identity of a visiting police inspector asked to solve a murder mistery by interrogating five simulated German-speaking suspects.

Immersed in this task, the learner has the experience of carrying on a natural dialogue, free from constraints on style or subject matter, entirely in the target language. Behind the scenes, however, Herr kommissar uses a full-functioned natural language processing (NLP) system to holding up the simulated conversation.

- **Lexical analysis:** The game looks up each word of the input in its on-line German lexicon and corrects most misspellings. When the word simply does not exist in the lexicon Herr Kommissar specifies the word was not found

- **Syntactic analysis:** The game performs a full-sentence parse on the learner's input. This parsing covers most aspects of basic and intermediate German grammar. If during the syntactic analysis one or more errors are detected, Herr Kommissar identifies the problem to the user and continues the conversation.

- **Semantic interpretation:** Herr Kommissar then maps the results of the lexical and syntactic analysis onto an internal model of the input's meaning. Semantic interpretations and error detection are done in this stage, such as the use of the verb *drink* with a non liquid object.

- **Response formulation:** The results of this check are then projected onto a new semantic structure and finally a response is generated.

2.1.2.3 A.L.I.C.E. Bot

The *A.L.I.C.E.* (Artificial Linguistic Internet Computer Entity, which is different from *ALICE-chan* discussed in section 2.1.1.1) is a free natural language artificial intelligence chat robot from the *A.L.I.C.E. Artificial Intelligence Foundation*. The software used to create *A.L.I.C.E.* is available as open source. *A.L.I.C.E.*'s Alicebot engine utilizes *AIML* (*Artificial Intelligence Markup Language*) to form responses to your questions and inputs. It is a project with hundreds of contributors worldwide.

Although it is not a teaching-purpose project, the Alicebot may also be used by English students wishing to practice English conversation.

2.1.3 Graphic-based tutors

This section presents systems that seek to integrate language with the physical, visual world that is its context. That world is represented mainly by computer graphics and digitized sound. Some of the earlier discussed systems used static graphics but they did not react to student's input. In contrast, we'll discuss now systems where graphics are an essential part of the learning experience and they are used to create micro worlds in which animated objects respond to requests, descriptions or actions by the student. These systems are shaped by theories of language learning that stress engagement in motivating, authentic communicative activity.

Those systems described in this section use various communicative approaches to language teaching, whose common premise is that language is best learned by using it to solve authentic and interesting problems. Moreover, most of the systems presented here are in an unfinished status, so that little information about their effectiveness is provided.

2.1.3.1 FLUENT

FLUENT is a complex conversational intelligent *CALL* system that presents many options for tutorial strategies. Its objective is to provide one essential form of foreign language learning experience. It offers conversation in the target language, uninterrupted by discussions of grammar, use of first language and difficulties of other students. Because conversation relates to situations, typically with visual components, the medium of language interaction in *FLUENT* is tightly integrated with a visual medium consisting of partially animated graphics under shared control of the tutor and student. A scene in fluent typically consists of a set of graphics which can be commented on states of objects by the user or also on relevant actions. For example, in the kitchen scene, the user can either type in sentences like *the cup is full* or *pick up the cup*.

2.1.3.2 Ling Worlds

The central problem addressed by this research project was to design a computer assisted language instruction system that could help beginning language learners develop their comprehension abilities. Ling Worlds is an instructional system that involves the student in solving communicative problems interactively with the system. The student participates in problem-solving simulations that allow the student to manipulate objects in a physical scenario or micro world. Information about the problem to be solved and information about the micro world are given orally in the second language. Metalevel commentary by the tutor is also in the second language. The teaching intervention in these simulations can vary from highly directed tutoring, where the tutor issues directions for action, to coaching, to purely student-controlled exploration. Ling Worlds uses dynamically generated speech from a digitized phrasal lexicon to produce its side of the tutor-student dialogue. The student's part of the dialogue consists of acts in the micro world.

The pedagogical model behind Ling Worlds requires many small problem-solving environments as well as many simple tutoring control strategies.

An example scene in Ling Worlds is called *Provisioning the lifeboat*. In this simulation the student is faced with the nonlinguistic task of provisioning a lifeboat before an ocean liner sinks. The computer displays an introductory animation showing an ocean liner colliding with an iceberg. A sample oral narrative accompanies the animation: “Your ship has hit an iceberg. It is sinking.” An on-deck scene of equipment and people near a lifeboat is then presented. At this point, the student interaction begins. In **tutoring** mode, the system directs the student actions through spoken language and the student responds to the instructions by pointing at or dragging objects with the mouse. In **exploratory** mode, the student can use the mouse to single-click objects and hear their names or double-click objects to hear a linguistic description of their locations in relation to another object. Finally, the **coaching** mode uses a game format with oral commands that direct the student to locations of provisions an equipment needed to stock the lifeboat successfully, but the system does not otherwise intervene.

2.1.3.3 Slime Forest Adventure

The Slime Forest Adventure is a classic role-playing game which is basically intended to learn the Japanese alphabet. Actually it can teach effectively both *Katakana* (Japanese use it for writing foreign words) and *Hiragana* (used for writing Japanese itself, in combination with *Kanji* chinese characters). Since it is not teaching Japanese grammar, this application does not contain any *NLP* module like most of the existing language-learning applications do. The teaching method is simple. The student must wander around the virtual world, trading and buying items as in any other role game but it is in the fights when the student practices and learns the Japanese alphabets.

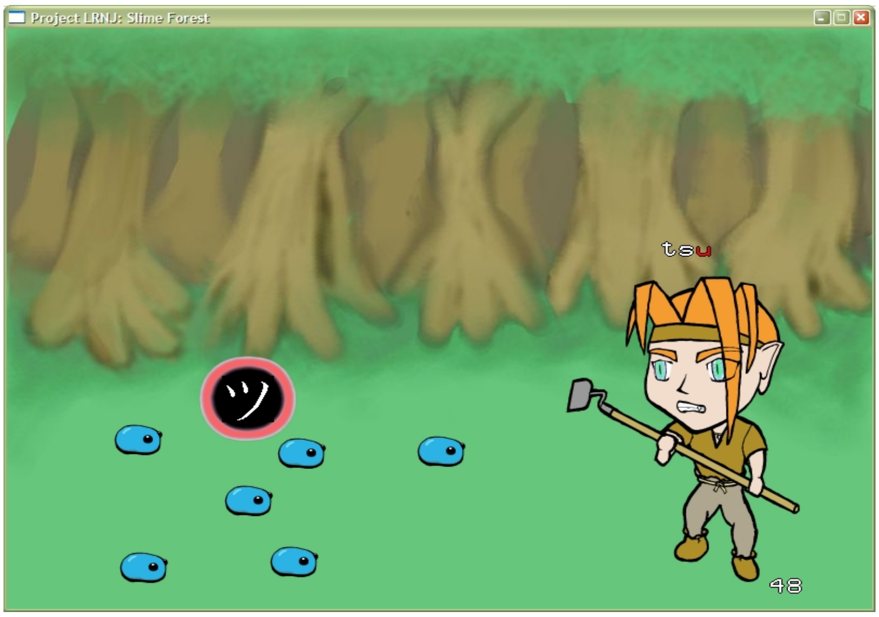


Figure 2: Slime Forest battle

The student must fight and beat the bad guys, the slimes, and to achieve that purpose the *Hiragana* and *Katakana* are used in a very ingenious way. Every time the character is to strike, a Japanese character appears over the target slime(see Figure 2). Then the user must write its pronunciation translation (usually no more than three letters) and hit the enter key. If the answer was correct, the character will strike the poor slime. Otherwise, the correct translation will appear over the character's head so that the user can type in the right letters. In battles time is crucial, since the slimes don't cease to attack the character so the quicker the user the lower the damage he receives.

2.2 Game engines

A game engine is the core component of any computer videogame or other interactive application with real-time graphics. It provides the underlying technology, simplifies development and often enables the game to run on multiple platforms such as game consoles and desktop operating systems such as *Linux*, *Mac OS X* and *Microsoft Windows*. The core functionality typically provided by a game engine includes the rendering engine (for rendering 2D or 3D graphics), the physics engine or collision detection, sound, scripting, animation, artificial intelligence, memory management and threading. The process of game development is usually economized by in large part reusing the same game engine to create several different games. In this section we could now start to speak about brand-new game engines[1] such as the *Quake III Arena* engine, but in relation to this project it would be pointless. However, we will shortly describe some libraries or game engines that are nearer to the scope of this project, and that offer some similar functionality that the one built during its development.

2.2.1 Allegro

Allegro is a game programming library for *C/C++* developers distributed freely, supporting the following platforms: *DOS*, *Unix* (*Linux*, *FreeBSD*, *Irix*, *Solaris*, *Darwin*), *Windows*, *QNX*, *BeOS* and *Mac OS X*. It provides many functions for graphics (including sprites, vector drawing, color palettes and text management), sounds (supports midi and wave formats), player input (keyboard, mouse and joystick) and timers. It also provides fixed and floating point mathematical functions, 3D functions, file management functions, compressed datafile and a *GUI*.

Allegro is a recursive acronym and stands for Allegro Low LEvel Game ROutines .

2.2.2 DarkBasic

DarkBasic is a game engine written in *C++* and that makes use of *DirectX*. It is basically a basic compiler with some fancy and useful add ons that make the task of programming a game easier. It supports both 2D and 3D graphics, it incorporates primitives to control animations and sounds and, although it has been used to create some commercial games, its most extended use is to create game prototypes.

1 Check www.devmaster.net/engines/ for wide information about the newest 3D engines available.

2.2.3 Easy Way Game Engine

The Easy Way Game Engine is a last generation open-source Java game engine. It is written in *Java* but, as its web claims, it can reach performances similar to 2D game engines written in *C*. It is based in the LightWeight Java Gaming Library (the same one used by the Computer-Aided Catalan Learning Application), which is a professional binding of *OpenGL*, *OpenAL* and other cross-platform libraries to *Java*.

It is a multi-platform game engine (can run on *Windows*, *Linux* and *Mac*), and its main principle is to keep it simple to use. It provides sprite management, collision detection and rendering and data loops.

3 Application design model

Since the Computer-Aided Catalan Learning Application is a complex system with several components interacting together, it is necessary to have an initial view on how the project works, without entering in too much detail. Each one of these components has its own architecture and purpose within the application. In this chapter we'll discuss the overall system architecture (what are the components and how they work together) and then we'll have a closer view at the design of each of the modules.

3.1 Functional requirements and use cases

As defined in requirements engineering, functional requirements specify specific behaviors of a system. They define the internal workings of the software, that is, the calculations, technical details, data manipulation and processing, and other specific functionality that show how the use cases are to be satisfied. In our case, since the system is mainly focused on teaching a language using a graphical game as a vehicle, the use case diagram is quite straightforward. Practically the whole system is hidden behind the graphic scene and one simple use case like "*type in sentence*" (see Figure 3 below) can actually be encapsulating a lot of functionality. Usually use cases are to match functionalities accessible through the *GUI*, and in our case the *GUI* is only destined to take in user inputs in form of text, to display the help and to show some error feedback.

- **Type in sentence** – This use case is actually hiding most of the system functionality. The user may type in any kind of sentence in Catalan in the input field. Then, either the graphical scene will react to what the student typed in or some error feedback will be produced by the system and shown in the error console. This use case is actually highly complex since when the user types in a sentence a long process starts, beginning by the parsing of the sentence. This sentence must go through the lexical analysis (divides the sentence into tokens, each one with its type), the syntactic analysis (applies a grammar to the outcome of the lexical analysis and build a tree) and finally to the semantic analysis, which parses the tree and looks for semantical incoherences. This process may have been produced some errors. If so, they are given to the error processing module which will list them to the user. Otherwise, the tree is sent to the action module in order to update the scene.

- **Check errors** – The user can always scroll up and down the error console in order to have a review of his grammar or spelling mistakes.

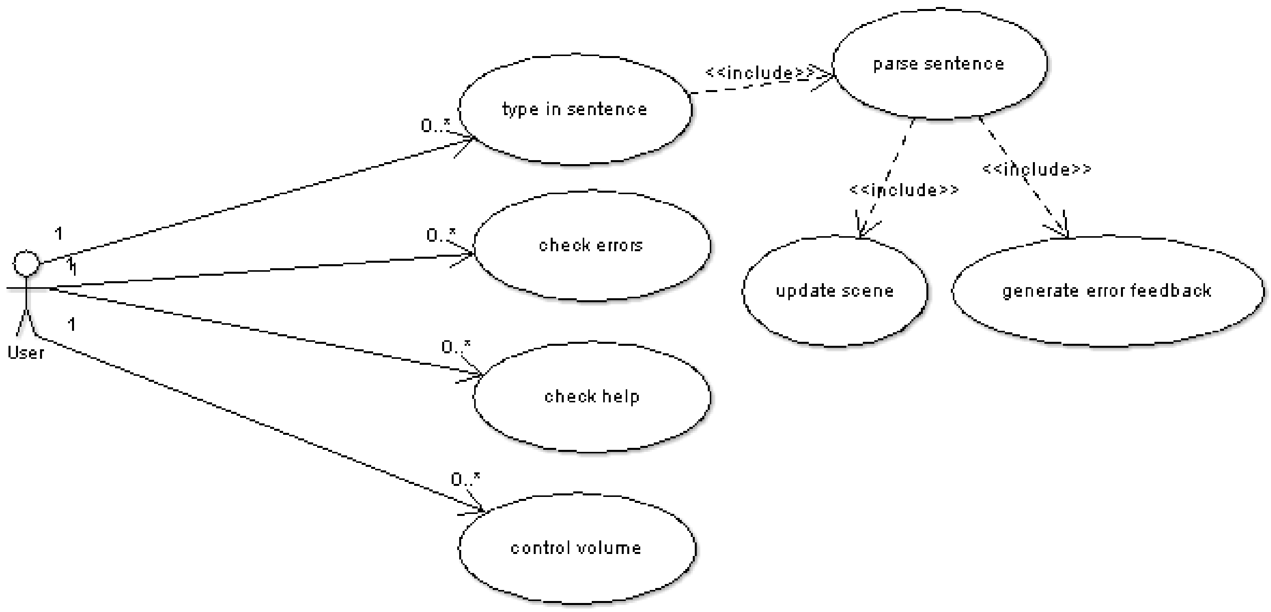


Figure 3: The use case diagram

- **Check help** – The user must be able to check the on-line help if he needs some kind of guidance on the Catalan language.

- **Control volume** – Finally, the user must also be able to control the volume of the voices that the application plays.

3.2 Non-functional requirements

The most important non-functional requirements that have been set are listed below:

- The on-line help must be short, simple and useful.
- The system must be able to run in both Windows and Linux systems.
- The system must be able to run in old machines, and 2D graphics will be used, trying to keep resource consuming at the minimum possible. See maintenance manual for more details on system requirements.
- The graphical user interface must also be very simple, and must not overshadow the game scene.
- The system response time (time the system needs to update the scene or produce error feedback after the user types in a command) must be the shortest possible. It must never be longer than 0.5 seconds. This requirement must be ensured at all costs. Otherwise, execution fluency may be affected and the overall feeling the user gets may weaken.
- Loading time between scenes must also be kept short if possible.
- It must be possible to change the target language (the language the application is actually teaching) easily. This must be achieved by building the application modularly and its objective is to keep the application easy to maintain at all costs.
- All the software resources used (libraries, programs -*IDE*-) must be freely available and, if possible, open source. This is a tough decision, but I think it is always better to rely on free open source software that can be checked and fixed if anything goes wrong. Another reason is that I don't really like to pay for something when I know

there is another similar solution which is as good as the proprietary one but free.

3.3 Overall system architecture

The application is composed by several modules that work together but that may also be replaced by other implementations intended for doing the same job. The Figure 4 below shows the main modules and their interaction. If you keep following the lines you'll trace the way that input string data takes to be converted into either an error output message or a scene update.

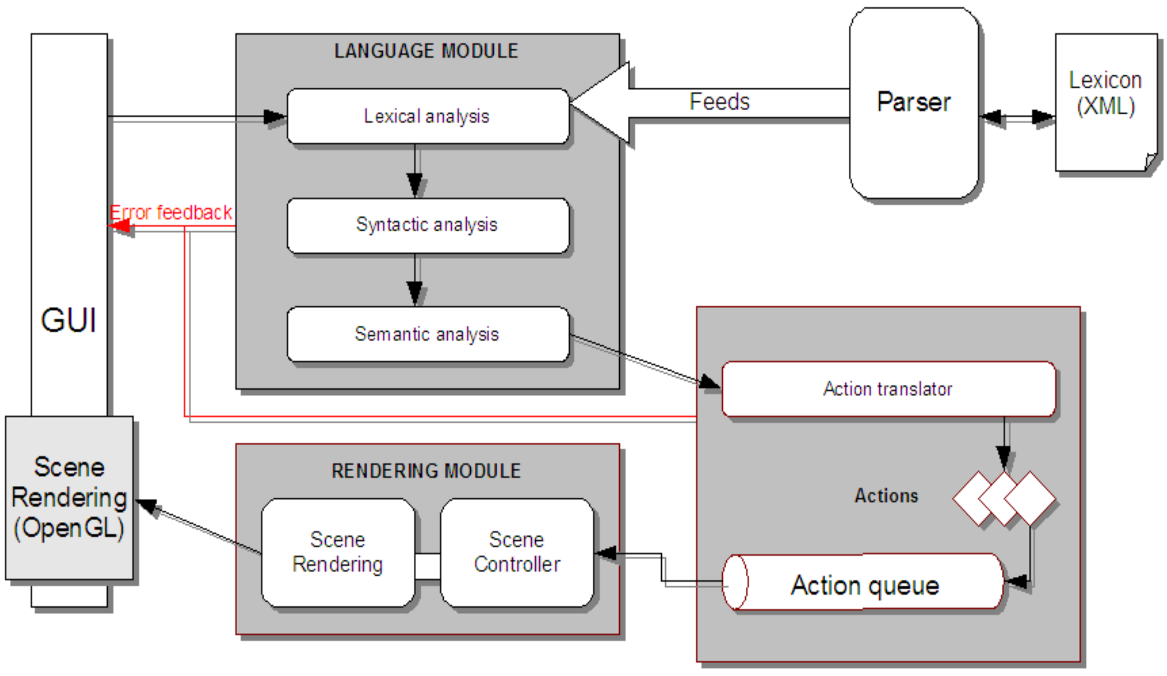


Figure 4: Architecture diagram

As you can see in the Figure 4 there are four main modules: the *GUI module*, the *Language module*, the *Action module* and the *Rendering module*. Finally, there is one last module which is quite small, the *Sound module*, but does not appear in the diagram.

I used the module separation approach which is based on component functionality (elements contributing to the main general functionality are in the same module) because this is a way to keep the whole system organized and structured, and it permits to build scalable systems (each module is scalable as a unit of functionality so that it is possible to add more features or functionalities to one module without the other ones noticing) and reduces further efforts in maintenance. Also, keeping the system structured and modular helps bug fixing be easier.

Now each module will be discussed separately in terms of how it communicates with the others so that the reader can understand the overall architecture and what are the units of information the modules are managing and passing forward.

- The **GUI module** is the *GUI* itself and is aimed to control all the necessary widgets in the main application window and the other dialogue shells such as the *volume* window or the *about* window. It is to be responsible for gathering the user input data and for passing it to the language module and for triggering its parsing and, at the end, it must also write in the dialogue console and shows error feedback in the error console. This module must ensure the usability of the system since it is a key point in any application designed for any kind of end-user.

- The **Language module** should contain the sentence parsing and the lexicon parsing. The sentence parsing should use the lexicon parsing as a tool for parsing user input strings. It gets the user input data from the *GUI module*, and takes it through a process that will produce the syntactical tree of the sentence as an outcome. Then it is passed to the *Action module*.

- The **Action module**, as said, should the tree and convert it into something "understandable" for it. This step would allow the system to be independent from the language module, so that if the target language must be changed, only some parts of the *Language module* and this last step should be modified. The Action module, basically, is the module that contains the queue of actions that is populated using the results of the conversion of the tree. Then, those actions are sent to a functional unit called *Scene Controller*, which is the link between the *Action module* and the *Rendering module*.

- The **Rendering module** is thought to control everything that is in the scene. As you can see in the Figure 4, there are two clearly differenced parts in this module, the *Scene Controller* and the *Scene Rendering*. The *Scene Controller* takes the output of the previous module as an input and uses it to "make things happen" in the scene. The *Scene Rendering* is basically in charge of drawing the scene in the canvas of the *GUI module*.
- Finally, the **Sound module** is a very small piece of software which contains the functionality needed to play both wave and midi files. It is in charge, obviously, for controlling the musics and sounds played during the application execution.

3.4 Module design

This section will discuss in depth each one of the modules that compose the system. All the class diagrams shown in this section are *UML* design diagrams. Specification diagrams are in all cases quite similar to the former ones, with some minor differences arising from the normalization process. Those differences are not substantial enough to show those diagrams in this document, and, I consider, attaching them would be a waste of space and time. Moreover, only the most important operations and attributes are shown in each object. Otherwise, some diagrams would not display correctly due to page size.

This section will also discuss and give arguments explaining the design decisions and issues taken, and why have they been taken.

3.4.1 GUI module design

This is a quite simple and straightforward module, whose objective is only to control and build the graphical user interface using the *SWT*[2] library.

The decision to include an Operating System driven Graphical User Interface was not easy. Actually, most current games do not use the Operating System libraries to build their interfaces, but they build their own customized ones. However, this is not just a game but also a learning tool, which has some special requirements. First, it must display large amounts of text in form of both dialogues and error feedback. Second, some kind of help must also be displayed in order to aid the student through its Catalan journey and, in addition, the student must be able to navigate freely through these help pages. It seems pretty obvious, then, that for achieving this purpose there is no better technology than *HTML*. For all these reasons I decided to embed the game scene in a window depending on the operating system "standard look'n'feel", having some useful and necessary widgets surrounding it.

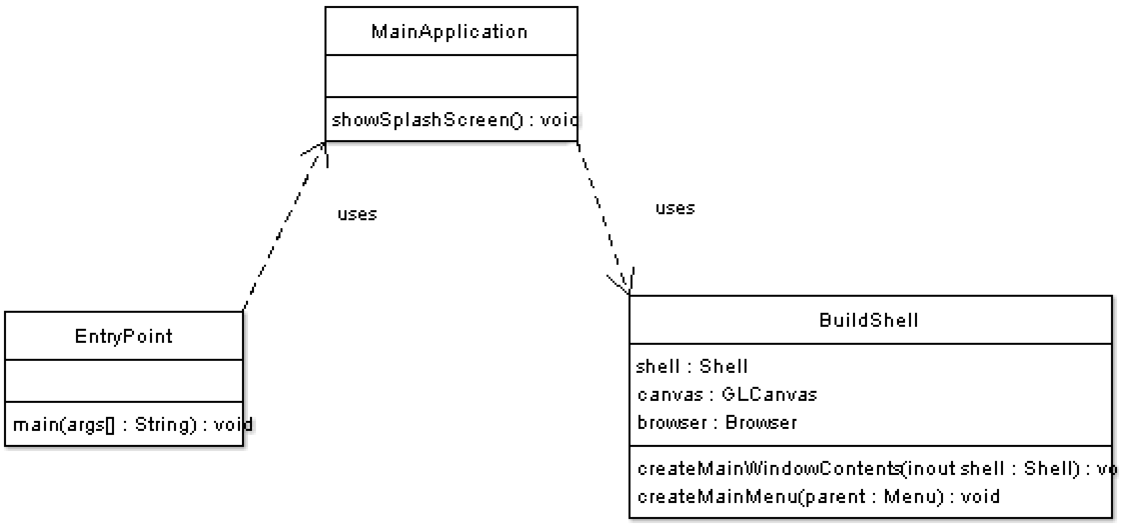


Figure 5: GUI Module class diagram

The **EntryPoint** class acts only as an entry point to the application. This one triggers the **MainApplication** which is responsible for coordinating the creation of the main window and the parsing of the lexicon[3]. The **BuildShell** builds the main window and should

2 Standard Widget Toolkit, from the Eclipse project (http://www.eclispe.org).
3 The lexicon is parsed just once at the application startup.

initialize all the widgets and menus. It would be also responsible for creating and initializing the help browser and for setting up the scene canvas. As you can see, the relation between the classes is only a usage relation since all of them are singletons.

The design idea in this module is one controller per view (usually a view is a window of the *GUI*). Actually, there is only one view in the application (the main window) and every functionality can be accessed from there. In this case the controller is the *BuildShell* and it deals with usability aspects, which are quite important in any application of this kind. I tried to keep the *GUI* simple, easy to use and effective.

3.4.2 Language module

Since the importance of this module in the whole project is quite high, this section is divided into two subsections. Firstly we'll be speaking about the functionalities that have been added to the module and those that have been discarded, explaining the reasons for doing so. Then, we'll have a close look at the final design itself, explaining some important issues concerning the planning and design of this module.

3.4.2.1 Functionalities

The main goal of the *Language module* is to provide the system with a reliable *NLP* system for a basic subset of the Catalan language. Its functionality has been split into three separate stages: The lexical analysis, the syntactic analysis and the semantic analysis. This has been done so because it's the way humans use to do syntactic analysis of a phrase and it is the approach used by linguistics and most *NLP* systems. However, the actual things the parser checks may vary from one system to another. Actually, this is the same approach used by compiler designers when they design and develop a new compiler. The process of compiling a source code to machine or intermediate code has a lot of similarities with the process carried out to “understand” a sentence and to update the scene.

The lexical analysis just splits the input sentence into a sequence of tokens and typifies them, this is, assigns a predefined type to each of them (noun, determiner, transitive verb...). The output of the lexical analysis is used as input to the syntactic analysis, which, using a grammar (more details on the grammar can be found in the chapter 4), builds an *AST* (*Abstract Syntax Tree*) which is the syntactical tree of the sentence. Thus, we get a tree which is similar to those that we were taught at school. Finally, and since the sentence may be grammatically correct (well built) but may not actually make sense, a final stage called semantic analysis is added. This basically go round the tree checking for some grammatical and semantic issues such as gender/number concordance, preposition usage or verb time. All of this stages may produce errors, which are to be then sent back to the user in the form of error feedback.

3.4.2.2 Design model

This is also a module that basically contributes with functionality. This means that classes are rather used by other classes (actually, most of the classes in this module are deemed to contain pieces of functionality that, joined with the others, produce a higher functionality unit which is the module itself) than contained in aggregations or inheritance hierarchies.

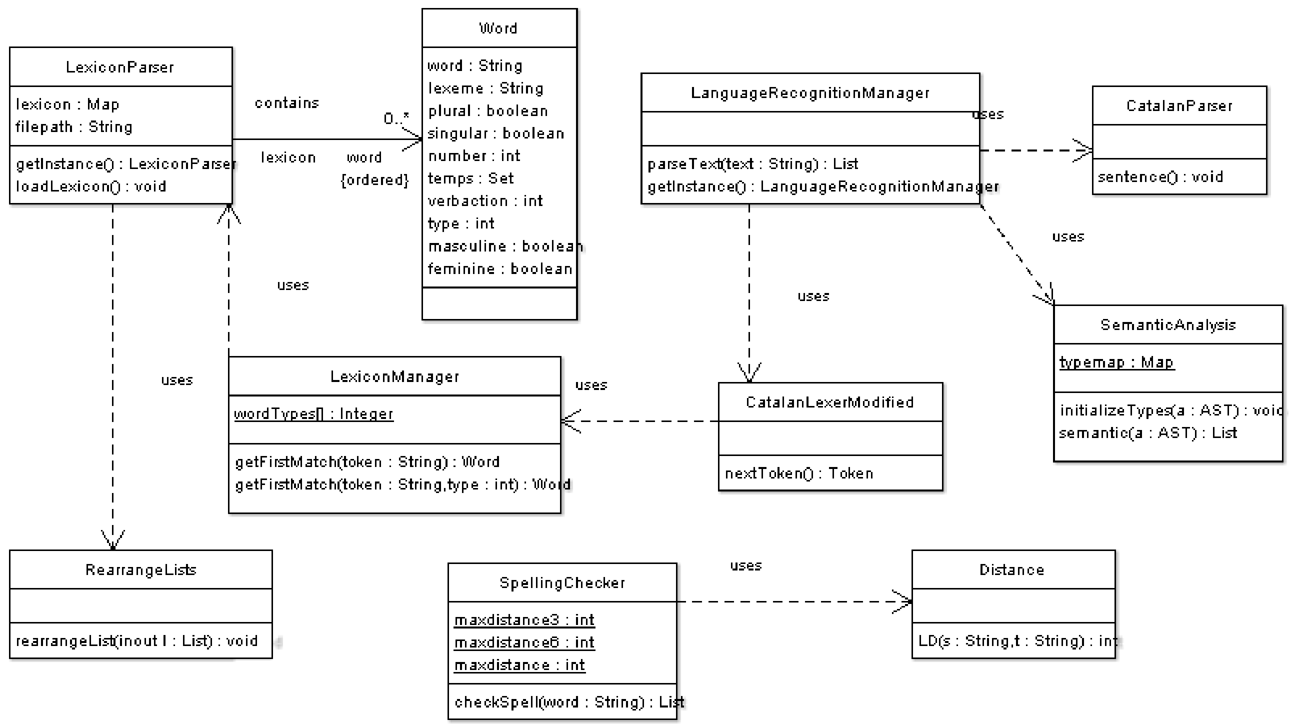


Figure 6: Language module class diagram

The main object of the module is the **LanguageRecognitionManager**, which is intended to act as an interface class through which the rest of the application should access the parsing facilities. It uses the classes **CatalanParser** (generated by the *ANTLR* from a grammar) and the **CatalanLexerModified**, which communicates with the **LexiconManager** in order to look up words in the lexicon. The **LexiconParser** parses the lexicon, and builds a map with **Word** objects, each one representing an entry in the lexicon. Finally, the two classes at the bottom provide the functionality of the spell checker.

As you can see most of the relations between classes are usages, which is normal due to the fact that most of those classes are actually singletons (classes with only one instance in the whole system) whose methods are static.

This module illustrates perfectly the design principle chosen for designing the responsibility allotment. As you can see, the functionality responsibilities of this module are shared between all the classes, owning each one of them a very particular and independent piece of “work to do” to achieve the final result. I chose this approach rather than putting it all in the same class or in two classes (i.e. Analysis and Parsing) because it helps keep the application structure modular, organized and built with replaceable pieces. You may want to change, for example, the way you parse the *XML* lexicon. This means you don't have to throw away the existing **LexiconParser** (which uses the commons-digester library from the Jakarta project) but provide another implementation for it. The same thing may happen with the other classes, you may want to change,for example, the semantic analysis but keep the lexical and syntactic the same. This is the main advantage this design approach provides. Moreover, it also helps in maintenance easiness and bug finding and fixing.

3.4.3 Action module design

The *Action module* is the one that controls everything that happens in the scene. It is in charge of translating the tree that comes from the *Language module* into a data structure understandable for the rest of the components of this module so that it can be treated and processed.

3.4.3.1 Functionalities

This module is maybe the cleverest in the whole system. Its basic functionality is to translate trees into sequences of actions. The principal decisions were made on determining what is translated into actions and what is not, since the language module is much more generic and can parse any kind of sentence whose words are in the lexicon. However, the step taken in this module implies that the scene must actually support the actions specified in the sentence in order to update correctly. So, the key point is that adding words to the lexicon to extend the range of language covered is a straightforward task, but adding action support to the new language may imply a lot of effort (means new actions, new graphics, new animations, new scene entity behaviours...). Hence, a set of actions were predefined at the planning stage of this module so that the system would allow anything to be parsed by the language module (as long as the lexicon covered it) but only a particular set of actions would actually be translated by the action module in order to be processed by the scene. Some of these actions are open, close, run, walk, speak, ask, look, behold, push and move. More information about technical issues and implementation aspects bout this can be found in the chapter 4.

3.4.3.2 Design model

The object **ActionModel** is a generic class that can represent any action to be taken in the scene. It has an action type (which actually bears the information of the action itself) and a set of modifiers and attributes. Most of these attributes are Identifiers. An **Identifier** represents a nominal syntagm, must have a noun (which is the core of the syntagm) and a set of adjectives or modifiers. As you can see, the object **Word** appears in this module again since it is one of the links between the *Language module* and the *Action module*. The class that does the translation job, from a tree to a list of ActionModels, is the **ActionTranslator**. This translation may eventually produce some **OutputMessage**s, which are messages that are to be displayed in one of the two consoles.

Finally, we've got the **SceneController**, which is the class that actually interprets and takes the actions. This class speaks to the entities of next module, the *Rendering module*, and contains a queue of **ActionModel**. These ActionModels must be processed one at a time by the **SceneController**, in order, so that the scene updates properly.

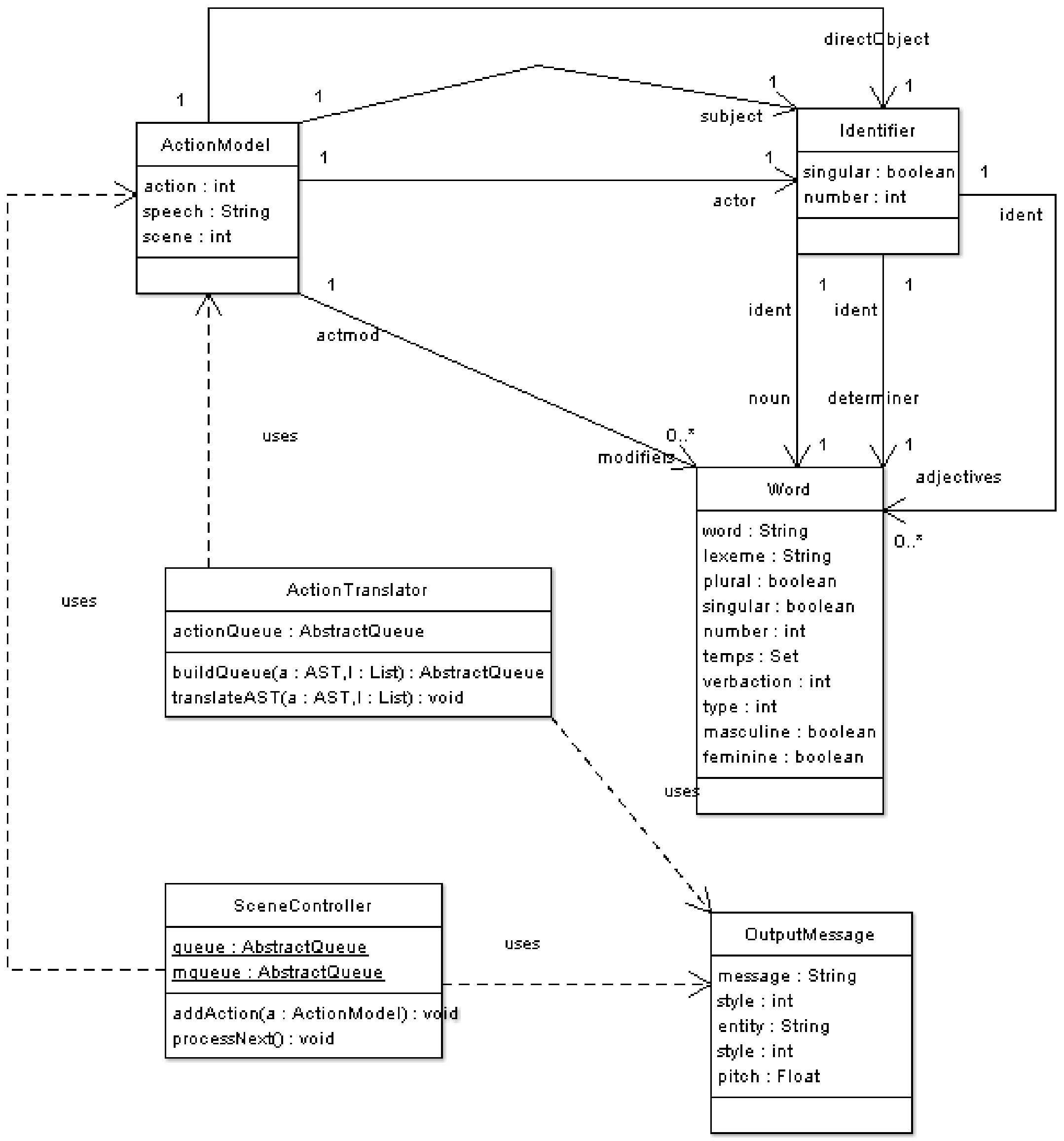


Figure 7: Action module class diagram

3.4.4 Rendering module design

The *Rendering module*, at its time, contains the necessary infrastructure to keep the scene up to date. It contains the classes that drive and steer all the entities appearing in the scene, and, in addition, it also holds the class which implements the main rendering loop. This loop is executed in a separate thread, so that it keeps executing without worrying about the other modules processing.

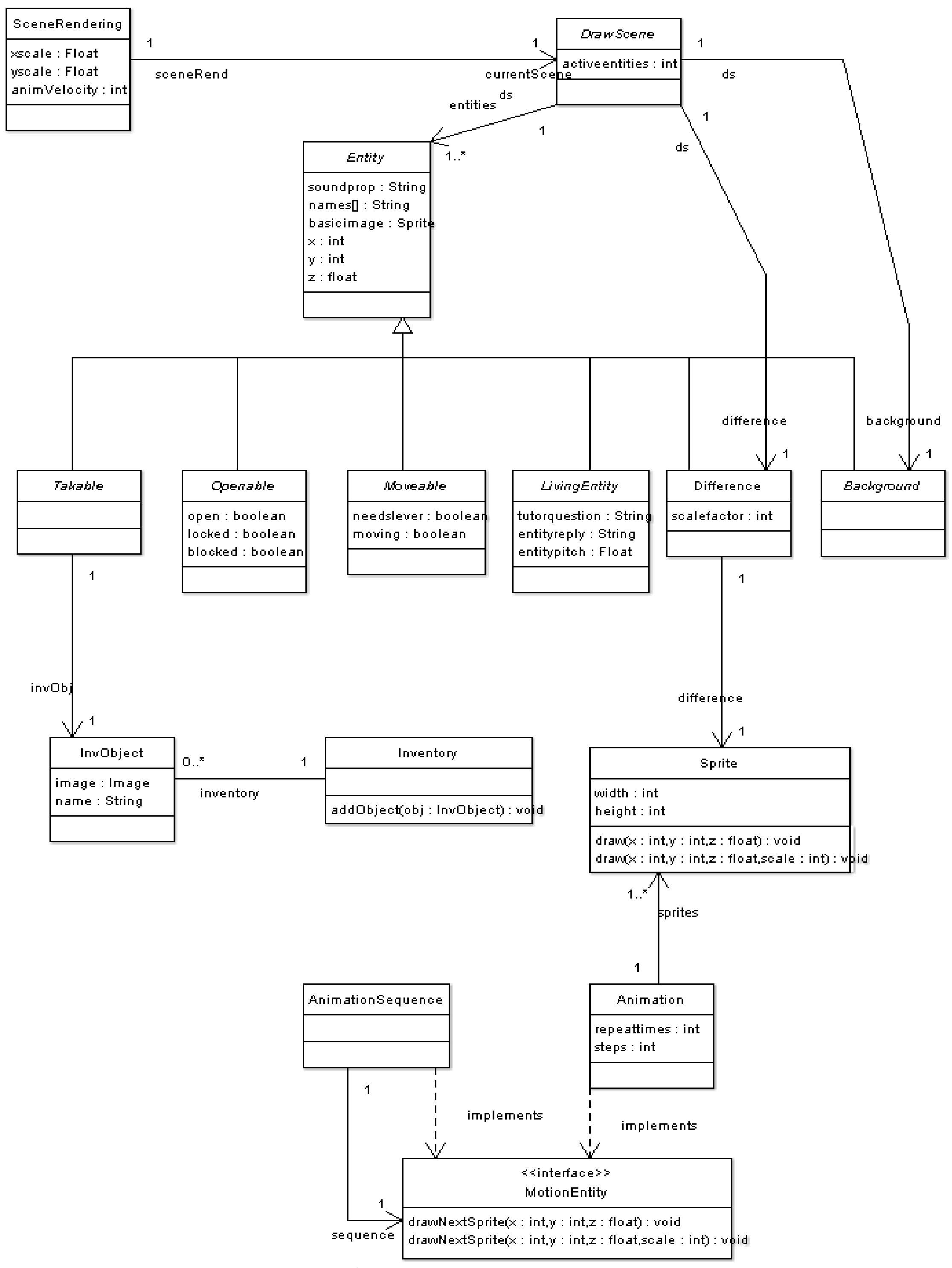


Figure 8: Rendering model class diagram

3.4.4.1 Functionalities

Actually, this module can be considered as the game engine. It is an own, personalized and totally adapted to the system needs game engine. It does exactly what the application requires, no more. This engine offers sprite, animation and animation sequence (sequence of animations that are to be displayed in a row) management, background and entity support, user collision support (means that there's a mechanism to define where the user can and where the user can not step on, defines “forbidden” zones in the screen. It is done via the **Difference** object, see chapter 4), and an automated mechanism to control the scenes. However, it does not support entity collision (usually a useless functionality in adventure games, where the character is not controlled directly by the player) or path finding (this would have been a very interesting feature to be added, but due to a lack of time the character's movement is based in straight vectors).

I chose to build my own engine because the existing engines usually provide too much functionality or are too much complicated to create a game of this scope. Moreover, writing my own engine allows me to adapt it totally to the application's requirements, and not vice versa.

3.4.4.2 Design model

The Rendering model class diagram shown in the Figure 8 has been highly simplified due to space reasons. The **DrawScene** is an abstract object that represents any scene. Each particular scene is represented by a class extending **DrawScene**. The **SceneRendering** class contains at all time a reference to a **DrawScene**. This reference may change every time the scene that is being displayed changes. It is the class **SceneRendering** that implements the main rendering loop as an asynchronous execution thread.

Every concrete **DrawScene** contains a vector of Entities, which are actual objects or things that appear in the scene. Anything which is to be drawn in the scene is an **Entity**. There is a class hierarchy hanging from entity, so that we can have **Openable** entities, **Takable** entities or **LivingEntities**.

The **Inventory** and **InvObject** objects are used to manage the inventory and the objects that are in it. The rest of the classes, **Sprite**, **Animation**, **AnimationSequence** and the interface **MotionEntity** are used as a low-level support to manage the graphical environment in the scene, to represent a graphic (sprite) an animation or a sequence or animations.

3.4.5 Sound module design

This module contains only two classes, so its design is quite straightforward. The class **WavPlayer** is used to play wave sound files, and the **MidiPlayer**, obviously, is used to synthesize midi sound files. Basically, the former is used for the speech voices and is to be triggered from the *Action module* and the latter is used for the background music and is triggered by a scene itself (class inheriting from **DrawScene**) when it is initialized. Actually it would usually be part of the game engine, but here it appears as a separate module because it was designed late in the development process as an “extra” feature.

4 Implementation and technical issues

The previous chapter provided an overview of the project, introducing the modules involved and their expected functionality. In this chapter we'll try to give the reader a depth and wide view on how the system was done, detailing procedures, specific design decisions taken at later stages and implementation issues and we will be arguing why some technical and implementation decisions were taken and what were the possible alternatives.

We will start discussing some important technical issues such as the programming language used and the existing libraries that helped achieve our purpose with this project. Then, we'll have a closer view at the system itself, detailing important classes, methods, functions and other relevant implementation decisions.

4.1 Technical decisions

This section argues some technical decisions made mostly at the beginning of the project, explaining the choice between several technology options.

4.1.1 Programming language

At the very beginning of the project, two alternatives were considered on this matter. Using an object-oriented programming language (such as *Java* or *C++*) or using the *Adobe*

Flash[4] technology. I chose to use a conventional programming language because the *Adobe Flash* is proprietary software and I had to pay if I wanted to use it. Moreover, I do not have any experience with it and I don't really know what it can offer.

Once decided the Computer-Aided Catalan Learning Application would be developed using a conventional programming language, the question of "which one?" came. The first indispensable requisite was that it had to be object-oriented. Amongst them, the two most important, *Java* and *C++*, were chosen. The main features that must have a good programming language to write a game are, basically, that it must allow the user to control threads of execution easily, it must have access to some rendering engine, it must have easy access to some sound library and finally, in our case, as I said, it must be object-oriented. If we look at Java and *C++* we see that both of them have the features described above. However, the main difference between them is that Java is a multi platform language (since it is compiled to an intermediate code which is then executed by the *JVM*), so that it should work correctly over any system. One of the non-functional requirements for this project is that it must run on both *Windows* and *Linux* systems, so the one that had to be chosen was clear. Moreover, the author has larger experience in Java programming than in *C++*.

4.1.2 Rendering engine

The rendering engine is the piece of software (usually a library) which provides the functionality of drawing graphics to a particular section of the screen. In our application this is a crucial element, since a bad choice may become a performance bottleneck for the rest of the system functions. Actually, since the beginning, only one rendering engine was considered. *OpenGL* is a cross-platform high performance 2D/3D *API* and it is used in several commercial applications and games, so that its reliability and performance are largely ensured. It was initially written for *C++*, but the *LWJG*L[5] provides Java developers access to high performance cross-platform libraries, among them, *OpenGL*.

4 http://www.adobe.com/products/flash/
5 Stands for *Lightweight Java Gaming Library*. For more information visit http://www.lwjgl.org.

OpenGL competes with *Direct3D*, which is part of Microsoft's *DirectX*[6] *API*. *Direct3D* is only available for Microsoft's various *Windows* operating systems, so it was ruled out since the beginning. However, performance tests comparing the two solutions are always hard to be reliable, since loads of other elements are involved in the process, such as the graphic card or the system memory.

Referring to the game engine, after considering various solutions, I decided I didn't really need them, since the features they offered, although great, were too much for the Computer-Aided Catalan Learning Application, since it just needs animation management and some other particular features useful for adventure games. Moreover, writing my own engine would allow me to design it to adapt completely to my purposes. I didn't even consider using some of the most used engines nowadays such as the *Quake III* engine (released under the *GPL* recently) or the *Unreal Tournament* one.

4.1.3 Language recognition tool

The two most important language recognition tools at the moment for the Java platform are *ANTLR*[7] (*Another Tool for Language Recognition*) and *JavaCC*[8] (*Java Compiler Compiler*). As long as I know, both of them are quite similar in functionality, both of them are parser generators and are extensively used in the creation of compilers for programming languages. As I had been using *PCCTS*[9] *(Purdue Compiler Construction Toolset)*, the earlier *C* version of *ANTLR*, previously, I decided to choose it only because of my previous knowledge, which would presumably save me time on research and learning.

4.1.4 Lexicon format

There is not much to say about the choice of the format that I would use for the lexicon. *XML*[10] (*eXtensible Markup Language*) was always the first and only option. It is great to

6 http://www.microsoft.com/windows/directx/default.mspx
7 http://www.antlr.org
8 https://javacc.dev.java.net/
9 http://www.antlr.org/pccts133.html
10 http://en.wikipedia.org/wiki/XML

store structured data, and the format may be completely customized. In addition, there is a broad variety of solutions to parse *XML* files which make the task of writing a parser to load a file quicker and easier. Actually, my previous experience with the commons-digester library from the Jakarta project added another good reason to the choice.

4.2 Implementation

This section will discuss how the technologies mentioned above were used to realize the design and satisfy the project requirements. Also, the problems encountered while implementing and the possible solutions, as well as the decisions made, will be covered.

Documenting the implementation of a software system in detail is always a tough work that must be faced carefully. This section, however, is not intended to be a detailed description of the software developed, but it should highlight the most important issues concerning this topic.

4.2.1 Graphical User Interface implementation

In order to build the *Graphical User Interface* for the application, the most serious problem I had to face was to find a way to embed an *OpenGL* scene from the binding to *Java* offered by the Lightweight Java Gaming Library into an *SWT*[11] widget. *SWT* is an open source widget toolkit for Java designed, unlike *Java Swing*[12], to provide efficient, portable access to the user interface facilities of the operating systems on which it is implemented.

So, back to the problem, I could not find any example anywhere of an *OpenGL* scene embedded in an *SWT* application, so that I decided to explore it myself. Finally I found out that I could actually achieve it by creating an object *GLCanvas* and then setting the context to the *OpenGL* binding to this canvas.

11 http://www.eclipse.org/swt/
12 http://java.sun.com/docs/books/tutorial/uiswing/index.html

```
Composite comp = new Composite(shell, SWT.NONE);
comp.setLayout(new FillLayout());
GLData data = new GLData ();
//We use double buffering strategy
data.doubleBuffer = true;

canvas = new GLCanvas(comp, SWT.NONE, data);

canvas.setCurrent();
try {
        GLContext.useContext(canvas);
} catch(LWJGLException e){
        e.printStackTrace();
}
```

Code 1: Embedding OpenGL into SWT

For the rest of the *GUI*, all widgets are organized into groups, having 3 groups in total: two in the right pane, the *Hints and Help* and the *Inventory*, and one at the bottom, the *Dialogues/Output console*. The *Hints and Help* group contains a Browser object, which creates a browser window using the Mozilla rendering engine. The user can navigate freely through pages (the typical 'back' and 'forward' buttons are provided), but every time the displaying scene changes, the content of the help also changes in order to adapt better to the vocabulary and actions involved in the current scene.

Finally, the class **BuildShell** contains a set of public methods other than the one used to initialize the main window, that are intended to be used as external widget-state modifiers. For example, there is a method that sets the browser page, or another one that writes something in one of the two consoles in a particular format (usually dialogues and error feedback are the two elements that are to be written in consoles). This way, the rest of the application can access those methods and interact with the few elements that are in the *GUI*.

4.2.2 Natural Language Processing implementation

The *Natural Language Processing* system (*Language module*) is, together with the action module, the core of the application. In order to build this module, as explained before, the *ANTLR* library has been used, and it turned out that it worked pretty fine.

4.2.2.1 Using ANTLR for Natural Language Processing

The decision of using a parser generator such *ANTLR* for a *NLP* and not developing it from scratch was a tough decision, but at last, and due to time restrictions, I was forced to use one of these tools. Probably, if I had not, I would still be programming this module at this time. The main problem is that all these parser generators like *ANTLR* or *JavaCC* are intended to create programming language compilers. Programming languages are slightly different from natural languages in several senses. The most important, probably, is that they are deterministic, while the natural languages are not. This means that programming languages can be recognized and parsed by a computer without possible interpretation mistake, leading to only one translation[13] for the same input. However, natural languages are indeterministic, so that the same single sentence can be interpreted in many different ways depending on the context, the emotions, the dialect and a large set of other factors. This presents a real and challenging problem for the *NLP* developer. *ANTLR*, of course, is designed for parsing deterministic languages only, so that the solution taken was to use a subset of the Catalan language where no possible alternative interpretation could lead the parser to a mistake. Actually, for our purposes, and since the application teaches only some very basic Catalan to English speakers, this solution was deemed to be the right one.

However, Catalan's got another problem if you intend to write it with an English keyboard. At the beginning me and my supervisor decided that the user had to be able to write the input text using an English keyboard (rule that otherwise is pretty obvious given the application target students are English speakers). This meant that accents (`´) or other characters or other signs used widely in Catalan had to be banned. This presented another severe problem because now, two words that must actually be written differently (such as *sòl* -ground- and *sol* -sun-, or *mes* -month- and *més* -more-) and with completely different meanings had to be written the same way. This time, the solution taken was to use synonyms for the words written with accents so that no spelling errors would ever be produced. Although for teaching basic Catalan it is not a bad solution, anyone aspiring to become fluent in such language should really learn the use of accents and other symbols and letters that the English alphabet does not have (ñ, ç, ¨, l·l).

13 Actually a compiler is no more than a translator from one language to another.

4.2.2.2 The grammar

The grammar is the core of any *NLP* system. It defines what subset of language the system actually recognizes and is able to translate. Trying to write a grammar covering the whole Catalan language would be a pretty difficult task, and it is out of the scope of this project. However, a simplified version is perfect for this project and helps us achieve our purpose.

```
sentence: bsentence (CONJUNCTION^ bsentence)*
                   {
                            //Here we parse the tree and initialize the types
                            //for each node
                            SemanticAnalysis.initializeTypes(#sentence);
                   }
                   ;

bsentence: (
          (subject)?
          (vt:VERBT^ (ADVERB)? directObject (indirectObject)*
          | vi:VERBI^ (ADVERB)? (directObject|indirectObject)*)
          );

indirectObject: PREPOSITION^  (PREPOSITION|)  nominalSyntagm
          ;

directObject: nominalSyntagm
          {
                   ActionTranslator.addASTType(#directObject,
                                             ActionTranslator.DO);
          }
          ;

subject: nominalSyntagm
          {
                   ActionTranslator.addASTType(#subject,
                                             ActionTranslator.SUBJ);
          }
          ;

nominalSyntagm: nominalGroup
          ;

nominalGroup: (DETERMINER)? NOUN^ (complements)?
          ;

complements: ADJECTIVE (CONJUNCTION^ ADJECTIVE)*
          ;
```

Code 2: The Catalan grammar

Elements in lower case are nonterminal symbols that must have derivation rules. Elements written in upper case are tokens, or terminal symbols, and determine a type of word. As you can see, a symbol may derivate in symbols or tokens, all of them put together using the normal *CFG*[14] notation. So, a question mark, '?', after a symbol means that the element may or may not appear. A star '*' means that the element may not appear or may appear one, two or several times. The '^' symbol is used to define that the token it accompanies will be the root of the current tree, and the tree built by the elements at its right and left will

14 Context Free Grammar

be children. When there's something between {} means that this code will be directly ported to the generated parser file.

As you can see, a sentence is a conjunction of basic sentences. A basic sentence is a subject (optional, remember imperative sentences do not have explicit subject), and then a transitive verb followed by (possibly) an adverb, a direct object (this is a must) and a possible list of indirect objects, or an intransitive verb followed by (possibly) an adverb and a set of complements. The only difference between the direct and the indirect object is the preposition so that a direct object is just a nominal syntagm and an indirect object is a nominal syntagm preceded by a preposition (actually, a set of prepositions because in Catalan there are prepositions formed by groups of prepositions, as we will find out later).

Finally, a nominal syntagm is just a nominal group, which is just a determiner (optional), a noun and a set of optional complements. A complement is just a conjuncted list of adjectives.

As you can see, the grammar is pretty simple but will serve us to achieve our purposes. Subordinate sentences are not recognized.

But, how do you know a word is actually a preposition, a noun or an adverb? This leads us to the next section, where we will explain the problem faced with the lexical analysis provided by these automatic parser generators such as *ANTLR* or JavaCC and the solution found.

4.2.2.3 The lexical analysis problem

Automatic parser generators usually expect the tokens to be recognized as words as a whole or using a regular language. So that, for example, if you are building a compiler for pascal you could use the lexical analysis provided by *ANTLR*, having a token definition similar to this:

```
#token IF            "IF"
#token THEN          "THEN"
#token ELSE          "ELSE"
#token ENDIF         "ENDIF"
#token WHILE         "WHILE"
#token DO            "DO"
#token ENDWHILE      "ENDWHILE"
#token PROCEDURE     "PROCEDURE"
[...]
#token IDENT         "[a-zA-Z][a-zA-Z0-9]*"
#token STRING        "\"~[\"]*\""
#token INTCONST      "[0-9]+"
#token REALCONST     "[0-9]*.[0-9]+"
#token COMMENT       "\{~[\}]*\}"
```

Code 3: Possible token definition for Pascal in ANTLR

As you can see in Code 3 above, tokens are both complete words or patterns defined by a regular language. So that "*WHILE*" is a token for the reserved word 'WHILE' in Pascal, or any string starting by a letter (in upper or lower case) is a token for any identifier. A comment is, then, anything between '{' and '}'. As you can see, in this case, token definition is pretty straightforward.

However, when we try to use the same system for recognizing any word and typify it, the whole thing goes down. In Catalan, as well as in any other language, words are formed by concatenating several elements: the lexeme and the morphemes. The first clear thing we see is that if we want to put the whole of words the system is to recognize, the grammar file will grow up too much. Moreover, each time we want a new word to be recognized we will have to modify the grammar file, then use *ANTLR* to generate the Java files and then compile the project again. This is neither scalable nor handy. In addition, to create a lexical analysis for a natural language using the lexical analysis provided by *ANTLR* is a very tough and complex work, because you need to group words splitting them depending on their starting. With an example it will be clearer.

```
#token VERB    ('o'('b'('rir'|'scurar'|'turar')
                   |'m'('plir'|'etre)
                   )
               );
```

Code 4: Example of lexical analysis for Catalan using ANTLR

The code shown in Code 4 recognizes the verbs '*obrir*', '*obscurar*', '*obturar*', '*omplir*' and '*ometre*'. As you can see, having to do this for any word may become an extremely boring and unnecessarily complex task. The solution was to write the lexical analysis myself, using the same format used by the one *ANTLR* generates, so that I could use mine instead of their, but still use their syntactic analysis (generated from the grammar) at the same time.

4.2.2.4 The creation of the lexicon

So then I decided to create a lexicon in an *XML* file where anyone could add new words in an easy way. This lexicon should also provide grammatical information about the word such as its gender, its number or its grammatical type. Once having the lexicon, to write the lexical analysis would only be a matter of splitting the input string into words (blank spaces delimit words) and then look the particular word up in the lexicon.

4.2.2.5 The spell checker

The lexicon described in the previous section, an extract of which you can find in the Code 5 also simplifies the creation of the spell checker, in the sense that if the lexical analysis throws a “token not recognized” exception, then catching it we can look for possible candidates using the Levenshtein distance algorithm and display them to the student to help him correct his spelling. The distance required between the written word and the lexicon word, for the latter to be considered a possible candidate varies depending on the length of the word. For example, for words with less than 5 characters the distance must be 2 or less. For words between five and eight characters, the distance should be between less than 4. For larger words distance should be less than 5.

```
<determiner>
          <lexeme>un</lexeme>
          <masc>
                    <sing></sing>
                    <plur>s</plur>
          </masc>
          <fem>
                    <sing>a</sing>
                    <plur>es</plur>
          </fem>
</determiner>

<noun>
          <lexeme>plat</lexeme>
          <fem>
                    <sing>ja</sing>
                    <plur>ges</plur>
          </fem>
</noun>

<transitive>
          <lexeme>to</lexeme>
          <present>
                    <sing>ca</sing>
                    <plur>quen</plur>
          </present>
          <imperative>
                    <reflexive>
                              <sing>ca't</sing>
                              <plur>queu-vos</plur>
                    </reflexive>
                    <sing>ca</sing>
                    <plur>queu</plur>
          </imperative>
          <action>17</action>
</transitive>
```

Code 5: Sample code from the lexicon

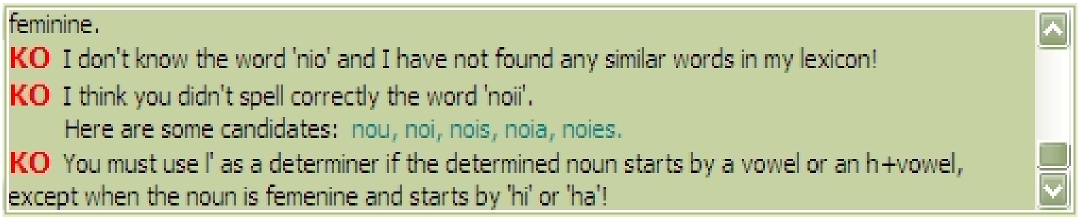
As you can see, there are several types of word (determiners, nouns, adjectives, prepositions, adverbs, transitive verbs and intransitive verbs). Each one has its own format in the lexicon, since different information is required for each type. The formation of each word is built concatenating the lexeme with the rest of terminations. For example, in the noun whose lexeme is 'plat' we can form two different words, '*platja*' -beach- and '*platges*' -beaches-. In addition, the lexicon tells us that '*platja*' is a feminine singular noun and '*platges*' is a feminine plural one. This will allow us to check the concordance later in the semantic analysis, as explained in the next section.

4.2.2.6 Semantic analysis

The semantic analysis is a piece of code that parses the tree the syntactic analysis outputs and checks basically determiner-noun concordance, noun-verb concordance, preposition usage and construction and determiner usage (for example, the "l'" rule). It does not actually check the action the sentence can be executed, but it ensures the sentence to be totally correct from the grammatical point of view. Semantic issues (such as "*open the dog does not make sense!"*) , are tested later in the action module.

4.2.2.7 Error feedback

Error feedback may be produced at any stage of the *NLP* system. The lexical analysis, as we discussed, produces spelling error feedback, the grammar or syntactic analysis produces phrase construction error feedback, and we capture it by catching the exceptions thrown by the generated parser. Finally, the semantic analysis also provides high level error feedback in some grammatical issues that have been discussed in the previous section. All these errors are captured by the *NLP* manager, the class which manages it all. It is actually told to parse a sentence and it returns a list of parsing errors. If this list is not empty then the errors are printed in in the error console and no further action is taken (the action module is not triggered until the sentence is correct). The error console, as you can see in the Figure 9, displays errors preceded by a red symbol, and correctly typed inputs with a green one. It improves usability.



```
feminine.
KO I don't know the word 'nio' and I have not found any similar words in my lexicon!
KO I think you didn't spell correctly the word 'noii'.
   Here are some candidates: nou, noi, nois, noia, noies.
KO You must use l' as a determiner if the determined noun starts by a vowel or an h+vowel,
except when the noun is femenine and starts by 'hi' or 'ha'!
```

Figure 9: View of the error console showing some error feedback

4.2.3 Action module implementation

This module basically translates the tree from the previous module into an action model and then adds it to the queue of actions. It is responsible also to process the actions in the queue.

4.2.3.1 Tree translation

The translation from the tree into an action model takes advantage of some information stored in the lexicon for both transitive and intransitive verbs. As you can appreciate at the bottom of the Code 5, verbs have a tag called *action* (<action>17</action>) which is used for the action translator to determine which is the action the particular verb implies. This number matches a number in a constants file (*Actions.java*) which bears the information of the actual action.

```
[...]
public static final int GO = 0;
public static final int TALK = 1;
public static final int LOOKAT = 2;
public static final int ASKFOR = 3;
public static final int OPEN = 4;
public static final int CLOSE = 5;
[...]
```

Code 6: Extract from the actions file

It is used by the action translator to decide what action is it dealing with and to decide what elements the sentence may have, what of them are mandatory and what of them can not appear, following the below. This table contains more actions than the ones that are actually recognized and hence translated and interpreted by the *Rendering module* in order to update the scene. It is because the table was done before anything and it has the whole set of actions that could possibly be recognized. It sets the restrictions to simplify the translation process so that anything that is out of the table will be recognized as semantically incorrect and hence the appropriate error message will be displayed.

'X' means that can not have this element, 'V' means that must have it and '?' means that it is optional.

Action	Direct object	To	Through	From	Quantity	With
GO	X	V	?	?	?	X
TALK	X	V	X	X	X	V
LOOK AT	V	X	X	X	X	?
ASK FOR	V	V	X	X	X	X
OPEN	V	X	X	X	X	?
CLOSE	V	X	X	X	X	?
STOP	V	X	X	X	X	?
RUN	X	V	?	?	X	?
TAKE	V	V	X	X	X	X
DROP	V	?	?	?	X	X
MOVE	V	?	?	?	X	?
JUMP	X	X	X	X	?	X
CLIMB	V	X	X	X	X	?
PLAY	X	X	X	X	X	V
SWIM	X	X	X	X	X	X
EAT	V	X	X	X	X	X
TOUCH	V	X	X	X	X	?
USE	V	?	?	X	X	?
LISTEN	V	X	X	X	X	?
SAY	V	?	X	X	X	?
TRY	V	?	X	X	X	?
BREAK	V	X	X	X	X	?
SCREAM	?	?	X	X	X	X
BRING	V	V	X	X	X	X
PUSH	V	X	X	X	X	X
PULL	V	X	X	X	X	X

Table 1: Action properties

Finally, as mentioned in the third chapter, note that a tree can be translated into more than one action model. Usually, any sentence that specifies the interaction of the character with an entity implies two actions to be created. The *GO* action to the entity, and then the proper action.

4.2.3.2 Action interpretation and action queue

The action queue and the message queue are simply *FIFO*[15] queues that are used to organize the actions to be taken in the scene and the messages to be displayed.

The actions usually refer to entities in the scene. The matching between the entity and the sentence is done by name (every entity has a list of names by which it can be referred to). The action models are then interpreted and the pertinent actions are taken and output messages are created and put in the message queue. Usually, when an action is put out from the queue and interpreted, the message queue is populated with the messages that need to be displayed to the user. The interpretation of actions is done in the *scene controller*, class that triggers directly some parameters or methods of the entities appearing in the current scene, so, if there's an action "GO TO THE DOOR" to be translated, the *scene controlle*r will look for an entity named "DOOR" in the current scene, and then it will tell the main character (the Tutor) to move to the position of the door (executes a method similar to walk(door.x, door.y);). Once an entity has finished an action, it notifies the *scene controller* so that the next action can be taken.

To control when an entity is active and when it is not, a state variable is maintained. The state specifies if the entity is in an animation or not. A scene may receive a new sentence to parse only if all of its entities are not active, this means, if there's an animation going on in the scene, the user must wait for it to finish until being allowed to type in another sentence.

4.2.4 Game engine implementation

This module is responsible for drawing and updating the scene, as well as implementing the entity behaviour and infrastructure. The module has two different parts: the low-level

15 First In First Out

engine, in charge of controlling images, animations and sequences of animations, and the high-level engine, in charge of controlling scenes and objects that appear in them.

4.2.4.1 Sprites and animations

In order to load and represent sprites I found a sample game in the Internet with two useful classes. One was intended to load textures (image fies) and the other one to represent a texture itself. I needed to modify them, in order to add the capability to load textures from absolute paths. From there, I took another very simple class, Sprite, and extensively completed it in order to be able to draw the sprite in my environment. Additionally, I also added the capability to draw sprites flipped vertically or horizontally, and to draw them scaled using a float scale factor. This is the base object of the whole engine.

Then, some higher level functionalities were added, such as animations (sequences of sprites that can be repeated several times until the finish flag is triggered) and sequences of animations (in order to be able to play together existing animations).

Both *animations* and *animation sequences* implement the interface defined by the *motion entity*, which defines methods for, for example, drawing the next sprite (possibly flipping and scaling it) in the sequence or knowing if the animation is over.

4.2.4.2 Entities and scenes

However, the core of this module is the abstract class *entity*. Any object which displays in the scene is an entity. The *entity* defines the graphics, animations and behaviour of any object. There can be *living* entities (the user can talk to them), *movable* entities (can be moved or pushed), *'takable'* entities (can be taken), *openable* entities (can be open or close) and so on. The background is also an entity which defines a list of *background* entities, that represent each one an object in the background (mountains, path, city...).

A scene is represented by an object which extends a *draw scene*, which is the responsible for sorting out the entities and drawing them. The depth of an entity is controlled by the z attribute, and before drawing the entities of a scene, they are sorted so that the ones with a greater depth are drawn first so that the nearer entities cover the further ones. Also, in some scenes the user character (the tutor) is scaled depending on his depth, so that when he's moving away his sprite gets smaller and when he's moving to us his sprite gets bigger. This is a nice effect that helps to produce the perspective and depth illusion.

All this stuff is very technical and boring, but the global purpose of this module was to create an attractive scene with some fancy features so that the student feels motivated to keep on trying.

To go into a bit more detail we will discuss the movement of the main character. The movement of the main character, called tutor, is based in vectors. When he's required to move to a particular point (from x_0, y_0 to x, y), the vector between the two points is worked out ($x-x_0$, $y-y_0$), made unitary (dividing by its length) and its coordinates are put in the state of the tutor entity, as a target for the current movement. Then, at each step of the rendering loop, when the tutor is updated, the unitary vector we got is added to the current position of the tutor and then multiplied for the current velocity (which is different if the user walks or runs). This allows us to work out at each step of the loop the next position the tutor must take.

4.2.4.3 The rendering loop

An entity is not driven by a separate thread of execution. This means that everything is synchronized and executed sequentially. Hence, we need a rendering system where everything must be very carefully designed so that it all works fine. There is a main rendering loop which is responsible for actually “talk” to the *OpenGL* machine in order to draw things. At each step of this loop all the entities are updated, sequentially, and then sorted and drawn. It is quite straightforward, as you can see in the pseudo code below:

```
DrawScene ds = new HouseScene();
while(true){
         ds.updateEntities();

         [PREPARE OPENGL]

         ds.sortEntities();
         ds.drawEntities();

         sleep(animationSpeed*1000);
}
```

Code 7: Pseudo code of the rendering loop

4.2.4.4 The difference object

In order to determine where the user can and where he can't move to, a very simple method has been used. For each scene a difference image was created. A difference image is an image composed of only two colors, red and black. The red parts determine where the user can move, and the black parts determine 'forbidden' zones. This image is actually scaled down to ¼ of the actual scene size, so that to work out if a zone is allowed, you just need to have its coordinates, scale them and then check the colour in the difference image. If this colour is red, the movement is allowed. Otherwise, the character must stop.

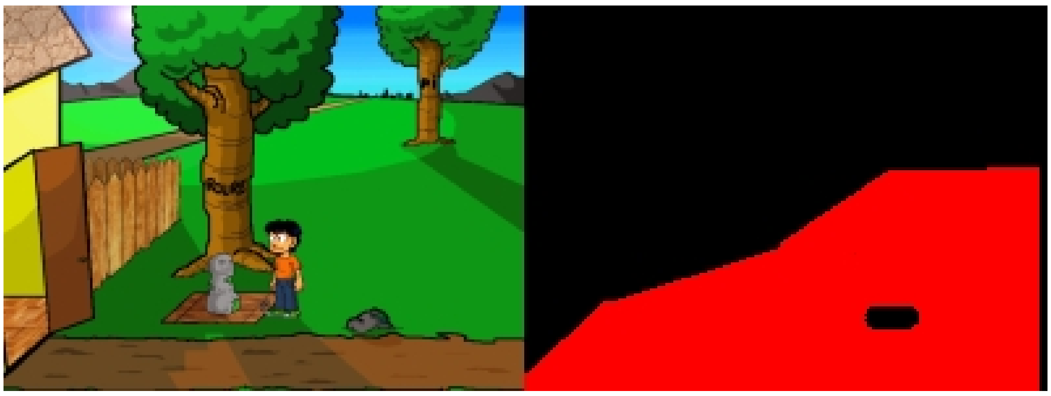


Figure 10: Outside scene background and its difference image

4.2.4.5 *Drawing the graphics*

Finally, I'd like to add some notes on the graphics. Absolutely all the graphics displayed in the scene but the camera that appears when an animation is going on are original. I used the *GIMP*[16] and my mouse to draw them. This allowed me to practice computer-aided

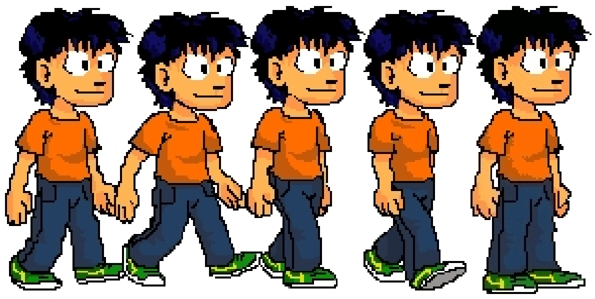


Figure 11: Tutor's animation sequence

drawing. However, drawing the animations was a tedious and boring work but at last the results were not that bad.

4.2.5 Music and sound

In order to play the wave files the *OpenAL*[17] binding for Java offered by the *LWJGL* has been used. It works pretty good and there are loads of examples about how to use it. For the midi background music the *javax.sound* library was used. The implementation of those two players does not have any secret, they are quite small classes and most of their code comes from sample examples I found in the *LWJGL* website. However, there's a little story behind it I would like to tell.

4.2.5.1 *Text-to-speech*

Having voice sound is a very good way to improve listening, so we decided it would be one of the first secondary goals to be added to the application.

For playing the voices there were initially two options. Using a text-to-speech in Catalan to

16 GNU Image Manipulation Program. http://www.gimp.org/.
17 Open Audio Library. http://www.openal.org/.

create the voices from a text automatically in execution time or recording the voices myself. I soon discarded the latter option, since I could not waste my time trying to record voices for the application. We had a quick look at the Internet and found that there's a *NLP* group in *La Salle* university, in Barcelona, that has a Catalan text-to-speech accessible via web (you actually write your sentence in a web form and in a few seconds the wave file is created and ready to download). I sent them an email telling them about my project and asked them if it would be possible to use their system in my application. They answered me very kindly that if the application did not have any commercial purpose, they could indeed lend me the software. However, they didn't have a Java version, so the solution they game me was to connect to their web server via *HTTP* protocol and then parse the response to be able to fetch the file from their server. I also discarded this option, since by the time the file had been created, published and downloaded the characters of my application would have finished their speech. Otherwise, if I tried to synchronize the graphics with the speech it would slow down the application critically.

Finally, the solution adopted was to pre-create the wave files using their system and include them in the application so that they can be accessed locally. This is not as cool as using a text-to-speech software, but the effects achieved are pretty similar.

However, and since the speech was added after the whole graphics engine was programmed and running, there's sometimes synchronization problems between the voice and the character's movement of the mouth. This is because the animations aren't actually waiting for the sound to finish to stop. It could have certainly be done, but I had not much time left and I preferred to focus my efforts in other aspects.

4.2.6 Final implementation notes

Every text in the application is in a properties file (*catalanlearning.properties*), so that the application can easily be internationalized and ported to several other languages. This file contains all the speech produced by entities, all the *GUI* text, image and file locations and error messages.

Also, the application has been built with logging capabilities, using the *Log4j*[18] library, so that you can create log files with everything the application writes as output (errors, speech...) by modifying the *log4j.properties* file. The default log output is the console but it is easily modifiable.

Finally, the lexicon parsed by the application at startup is not compressed or packed as a data archive. It is accessible and modifiable by any user, so that any user or student can complete it adding new words.

18 Stands for Log For Java.

5 Evaluation and testing

This chapter will discuss how the system has been tested. It is usually a difficult task to test the reliability of such systems and, specially, to test “how good they are”. The principal objective of our application is clear, teach Catalan basics to novices with some notions of English. However, it is always a good practice with such systems to have some people trying to do things and see what they achieve or what is their feedback referring to the user experience.

Also, this chapter will evaluate the whole application in terms of in what grade the requirements set at the beginning have been met.

5.1 Testing

The following subsections describe the various tests that have been carried out.

5.1.1 Reliability testing

The key point that needs to be tested is the *NLP* system, or the *Language module*, to be sure that it works as expected and the correct error feedback is produced. This was achieved via feeding the *language module* with a battery of inputs together with the expected error feedback. Since the *language module* is a complex system, where several stages take place, even defined in automatically generated files, the best way to locate errors and thus fix them was to use the debugging tool of the Eclipse *IDE*, which allows the developer to stop the execution of any thread (remember our application splits its execution line in several threads when it is running -GUI thread, Rendering thread, Music thread...-) and inspect a complete picture of the state of any variable at that moment. This feature was particularly useful for testing in depth the behaviour of the lexical analysis, which, as explained in the design and implementation chapters, had to be developed producing its output in a way the rest of *ANTLR* auto-generated files understood it.

These tests were also carried out to prove reliability of other components such as the *action translator* or the *lexicon parser*. However, since it is a game, most of its output is represented in the graphical scene and, eventually, in the consoles, so that you can find out the system works correctly for a particular set of inputs just by typing them in and check if the system's behaviour is the expected.

5.1.2 User testing

This test is much more interesting and useful than the previous one. It consisted in putting non-Catalan speakers in front of the application and see what they did, what they tried to do, what they did understand and what they did not, and, in general, what they expected of the application and what they could actually learn at the end.

The tests were done with three Galicians and a guy from Madrid. One of them has a good Catalan level. All the others, except the guy from Madrid, barely know how to say the basic sentences such as “good morning” or “see you”. The Madrid guy knew practically nothing about Catalan. All of them showed some interest in the application and felt open to learn some more Catalan. Their first impression was quite good, in the sense they actually liked the look'n'feel of the graphics and the whole application. Some of them told that the sensations they got were similar to those of being in a foreign country where you have to actually make an effort to understand anything or speak a word.

However, they found the error feedback very useful, specially the spell checker, since they knew some words in Catalan but they didn't actually know how to write them properly. In addition, they could also learn some basic grammar issues such as the use of l', and some vocabulary. However, the adventure is kind of short and they could reach the end easily.

The one whose Catalan level is quite good eventually tried to type in complex sentences the system didn't respond to, which indicated that maybe the Catalan level the application is intended to teach was not suitable for her.

Also, they produced a lot of useful feedback about the on-line help. For example, they suggested that it could actually adapt to the current scene, showing vocabulary and tips on how to behave or what actions to take. Also, they suggested to add some grammar pages on prepositions, articles and actions. Mostly, the final version of the help was based in their comments and feelings. Moreover, I am really convinced the final version of the help pages is really useful to the student and may provide hints on how to proceed in the adventure, which is also quite helpful if someone gets stuck.

5.2 Evaluation

Our evaluation will be based in discussing how good the requirements stated at the beginning of the document have been met. It also discusses how effective the current implementation described in the chapter 4 has been in fulfilling the goals.

5.2.1 Sentence parsing and error feedback

Obviously this is the most basic functional requirement and it has been achieved. However, the error feedback actually locates and reports some errors but not all the possible errors. It is because a natural language is often a complex and pretending to catch all the errors in a project of this scope would be a suicide. However, the current system has set the base for future improvements that could lead to a more powerful grammar or semantic analysis.

5.2.2 Error and dialogues console

Both dialogue and error consoles are located right under the input field, below the scene. Moreover, the dialogues console shows, for each piece of text, the character that produces it printed just before the text in a different color. The error console shows errors preceded by a red mark and right inputs preceded by a green one. Both of the consoles are scrollable.

5.2.3 The on-line help

As we have said in this same chapter, the on-line help has been largely based on user comments. It displays in a browser window embedded in the application window itself, in the right of the scene. It is colorful and easy to navigate from one page to another, since every page contains links to all the rest. However, could have been improved by, for example, allowing it to connect to the Internet for looking up words in a dictionary or searching for extra help.

5.2.4 Platform-independent system

I have tried my best to submit equally functional Linux and Windows versions. However, and given the application has been developed in a Windows environment, at the end I can't guarantee that all the functionalities work correctly under *Linux*. In the *Ubuntu Linux* I have installed in my laptop at the moment, for example, the help didn't display correctly due to a problem with the *Mozilla* engine, and it could not locate the *OpenAL* library to play voices. However, the Windows version works perfectly.

6 Conclusions and future work

This last chapter will discuss potential future work that could be done on the project. Then, the final conclusions are presented.

6.1 Directions for future work

This section details the improvements that could be carried out in order to enhance the current Computer-Aided Catalan Learning Application.

6.1.1 Improve the grammar

Obviously, it is easy to say but not that easy to carry out. The current grammar does not cover subordinate sentences, weak pronouns and other constructions. A good starting for future work would be improving the grammar so that it covers a greater set of Catalan. Actually, this is not a hard task indeed, but it is kind of worthless if the action and graphical support is not added.

6.1.2 Adding context to sentence interpretation

Another interesting improvement would be to add a context to the dialogues the application and the student maintain. For example, if the user writes:

```
Entra a la casa (Get in the house)
```

The system's response would probably be:

```
TUTOR: Per on he d'entrar? (Where should I get in through?)
```

Then, in the current system, you must write the complete correct sentence again:

```
Entra a la casa per la finestra (Get in the house through the window)
```

It would be great if the system just kept some information of the last sentences the user typed in, so that instead of writing the whole phrase above, the user could only write:

```
Per la finestra (Through the window)
```

6.1.3 Extending the lexicon

This work can be done by anyone, it is just a matter of editing an *XML* file. The idea is just to add more nouns, verbs, adjectives and adverbs. Determiners and prepositions are all already there. This work could be done along with the addition of new scenes.

6.1.4 Referring to entities depending on their position

The main idea of this improvement is that every entity would have a relative position to any other entity in the scene (right, left, behind, in front...). This can be done by just making some processing over the x, y and z attributes of a couple of entities (the reference and the referred one). For example, the user should be able to write sentences like “open the door that is on your left” or “take the paper under the table”. Obviously, these kind of sentences make sense when there's more than one door and more than one paper, because the subordinate sentence is actually determining what paper and what door the character must interact with.

6.1.5 Adding more actions

This is a very generic improvement, since you can spend the rest of your life adding actions or improving the current ones. The ideal goal is to have a game which reacts to anything the user types in, but this is a lot of work indeed, and since the natural language is indeterministic, a much more powerful and intelligent *NLP* system would be required. However, some more actions may be added such as 'eat', 'play', 'break', 'smash'... Actually, there are infinite possibilities.

6.1.6 Path finding

A very fancy improvement would be to implement a path finding algorithm for the user's character movements. It would result in the character being able to dodge objects in the scene, so that the movements should not be only driven by straight vectors as they are currently. This would be particularly useful with bigger scenes.

6.1.7 Adding scroll

This is the solution to having greater scenes. Currently, scenes are static, in the way that they can not be scrolled right or left depending on the character's movements. However, it would be a very good enhance to add the support for scrollable backgrounds in the game engine so that bigger scenes could be used.

6.2 Final conclusions

Having the possibility to take on, develop and finish a project like the Computer-Aided Catalan Learning Application has given me a lot of satisfaction. However, it is still work in progress and, regarding to its current state, we could say it is a prototype of a possible future real Catalan learning application that might be used by Catalan students. Moreover, and given its graphic-based nature, each step I took in the project produced immediate and tangible results which helped keeping me motivated to go on.

I really think that converting a game, and specially a graphic game, into a learning tool is a very good way to keep students 'on the work' due to the intrinsically motivating nature of games. Moreover, this game is designed to exercise listening, reading and writing and could be used, as stated in the introduction, for both preventing student's language skills from decaying and for pure teaching purposes.

Finally, this project has given me loads of new knowledge. Although I had done something in game programming before, I never designed and developed a game engine almost from scratch, and I think I did pretty well in this sense. Also, I never ever used a tool for creating computer language compilers to build a natural language processing system, and I had to face with several problems that gave me an idea of how complex this area of computing science can turn out to be.

Appendix A - User Manual

A.I Introduction

This User Manual is intended to help the user get started with the Intelligent Catalan Tutor application. Since the application is quite self-explanatory and simple to use, one should be able to proceed without this document, having it only as a reference to check if some problem arises. The User Manual is neither a Catalan help document nor a maintenance one. A first steps tutorial is included at the end of this document though. Once said this, you can find a proper on-line Catalan help in the application itself and a maintenance document in the *doc* folder of the software package you just got. In this document we will refer to the application base folder as *$APP_HOM*E.

A.II Installation and Uninstallation

For the information about how to install or uninstall the application please refer to the Maintenance Manual, chapter 3.

A.III The Intelligent Catalan Tutor

A.III.I Run the application

The Intelligent Catalan Tutor is an application intended to help people learn basic Catalan in a friendly and motivating environment such is that of an adventure game. To run it you just need to execute the *CatalanLearningTool.bat* file in Windows or to execute the *CatalanLearningTool-linux* script provided in any Linux flavour. You may also need to give execution permissions to it. If so, just proceed as explained before.

Since it is a game-like application with some minimal GUI (Graphical User Interface), its usage should not give the user too many headaches. However, here below is a short guide to explain most of the functions of the application.

A.III.II The main screen

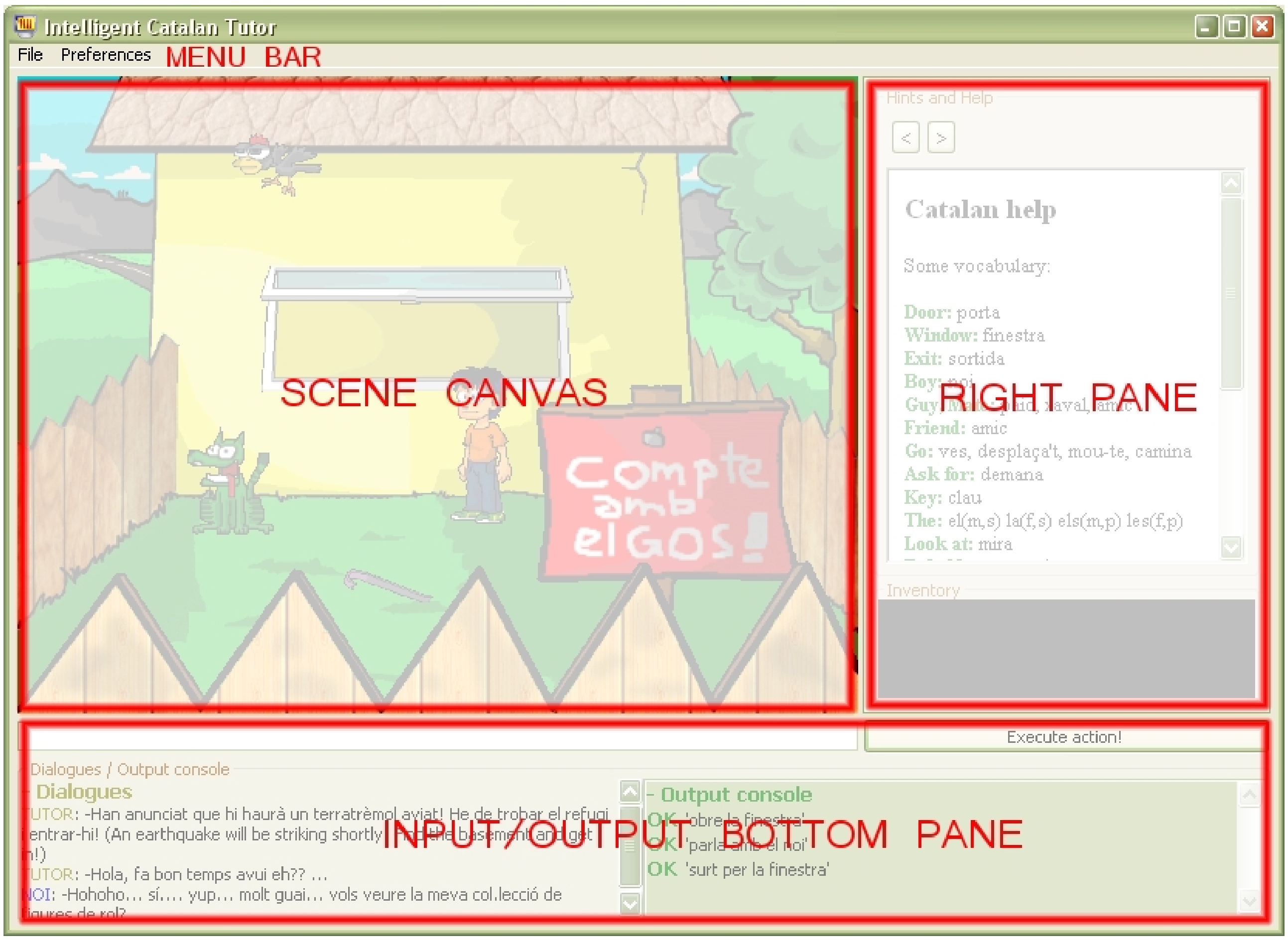
The main screen is composed by four main zones: the scene canvas, the right pane, the input/output pane at the bottom and the menu bar.

The **scene canvas** is the zone used to display the graphics. Everything happens there, so that when you type in some text in the input field and this text contains no errors, the feedback is given through the scene canvas.

The **right pane** contains two elements. The on-line help, which is in HTML so that the user is able to navigate forward and back through links and look for the information he needs about various useful language issues. It also contains the inventory, which is a list of objects the user has and can use at the moment.

The **input/output bottom pane** includes the *input form* (where the user types in the text in Catalan), the *dialogues console* and the *output console* are intended to give text feedback to the user. The *dialogues console* shows the written speech the characters produce. The *output console* shows some error feedback and the spell checking outputs.

Finally, the **menu bar** is a menu intended to control the user preferences and perform some minor actions.



Catalan help

Some vocabulary:

Door: porta
Window: finestra
Exit: sortida
Boy: noi
Guy, Man: paio, xaval, amic
Friend: amic
Go: ves, desplaça't, mou-te, camina
Ask for: demana
Key: clau
The: el(m,s) la(f,s) els(m,p) les(f,p)
Look at: mira

- Dialogues
UTOR: -Han anunciat que hi haurà un terratrèmol aviat! He de trobar el refugi i entrar-hi! (An earthquake will be striking shortly! Find the basement and get in!)
UTOR: -Hola, fa bon temps avui eh?? ...
OI: -Hohoho... sí.... yup... molt guai... vols veure la meva col.lecció de figures de rol?

- Output console
OK 'obre la finestra'
OK 'parla amb el noi'
OK 'surt per la finestra'

Figure 12: Main window parts

A.III.III Preferences menu – adjust volume

There are two types of sound output. The background music, which is in a midi format and the output speech. To control the volume of the output speech, just click on *Adjust sound* in the menu Preferences. This will prompt a new window with a scale which controls the volume. To do the same with the background music you've got to use your operating system volume control centre and adjust the Midi synthesizer scale.

A.III.IV Help menu

The help menu has two menu items: *User Manual* and *About*. By clicking on *User Manual* you'll be able to check this document, the User Manual. If you click on *About* a message window with some information about the author will display.

A.IV Tutorial

First of all you should know that the aim -goal- of the game itself is to find the shelter of the house and figure out how to get in since they forecasted an earthquake for today. However, the purpose of the application is to help the English-speaker user learn some Catalan basics in an entertaining and motivating way. With the help provided, the user should be able to sort it out and make his way to the shelter. If you got stuck with the Catalan, just think of what would you write in English and then check the grammar section in the help pane.

We'll start off with some grammar basics and then we'll introduce the basic actions that will allow you to move around and interact with things. All the sentences the user can type in and that make something happen in the scene are in imperative. All sentences are processed and, if they're wrong, error feedback is displayed in the output console. The idea is that you give orders to your character (the guy with the orange shirt). You can also write other types of sentences, but the effect in the scene will be reduced to the character advising you to write something in imperative. However, if you type in a proper sentence (Subject + Verb + Complements) the system still will give you the error feedback.

The normal sentence structure in Catalan is the following:

> Transitive verb (imperative) + Direct Object + Complements

> Intransitive verb (imp) + Complements

For a complete list of actions please refer to the on-line help displayed in the right pane in the application main window. You can have all sorts of complements in any order, but to get into more detail just look at the examples below. They're a very good starting point for any novice in Catalan.

A.IV.I Basic actions

The first main action you should learn is to **move around**. You can make your character to walk or run right or left, or to walk or run to entities in the scene. Usually, everything you see in the scene is an entity and you can tell the character to interact with. For example, in the first scene, the house, you can type in:

```
Ves a la porta
```

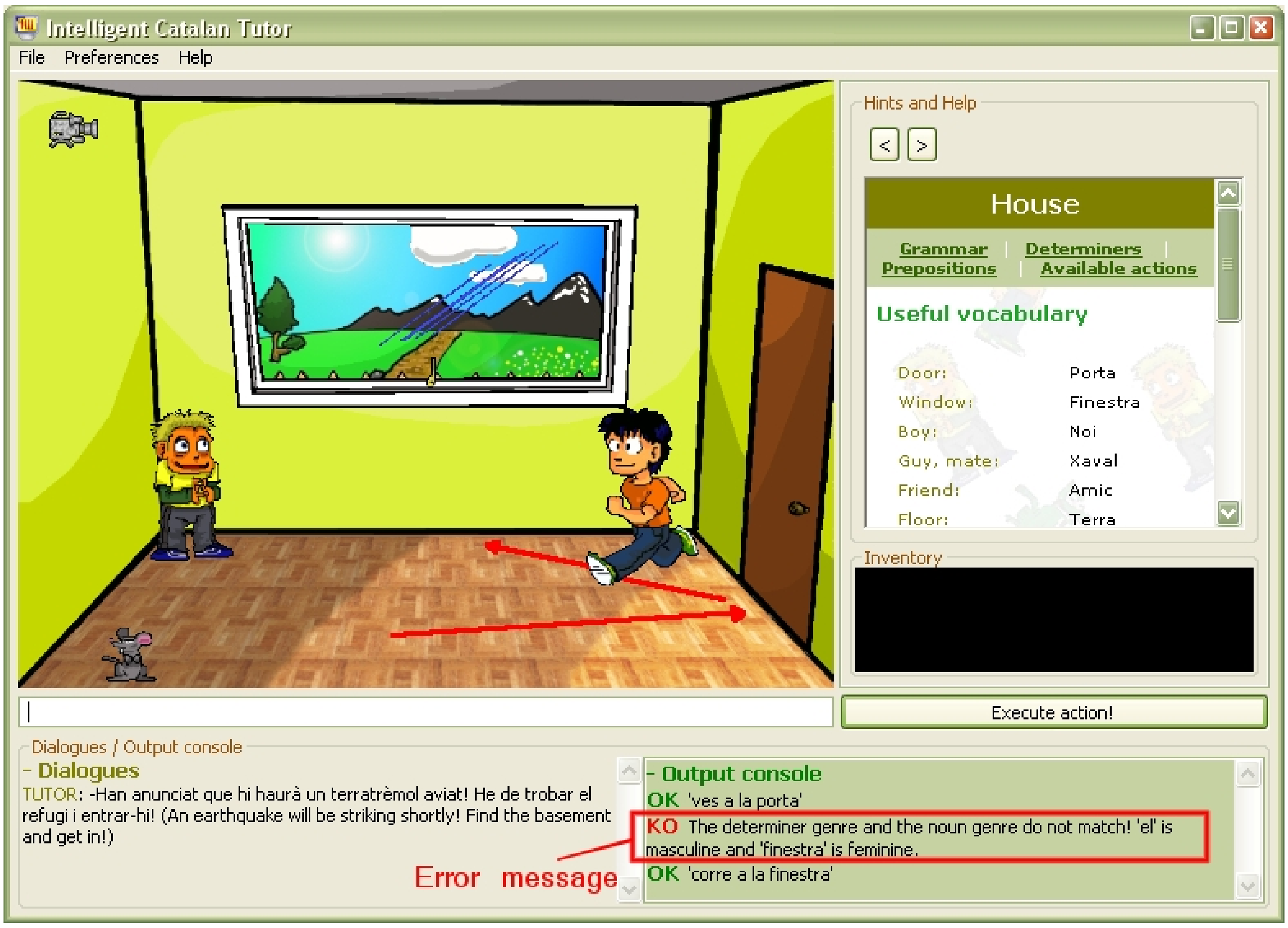
Which means literally “Go to the door”. This will result, after you press the Intro key, in the character walking to the door. If you now type:

```
Corre a la finestra
```

You'll see the user move to the window running. *Corre a la finestra* means “Run to the window”. Notice that if you type in:

```
Corre a el finestra
```

The bottom left panel shows an error message informing that the determiner genre does not match that of the noun (since *“el”* is the article for masculine singular nouns and *“finestra”* is feminine). You can find a whole vocabulary list for each scene in the on-line help in the right pane. If you have any problem just check it out.



House

Grammar | Determiners | Prepositions | Available actions

Useful vocabulary

Door:	Porta
Window:	Finestra
Boy:	Noi
Guy, mate:	Xaval
Friend:	Amic
Floor:	Terra

- Dialogues

TUTOR: -Han anunciat que hi haurà un terratrèmol aviat! He de trobar el refugi i entrar-hi! (An earthquake will be striking shortly! Find the basement and get in!)

- Output console

```
OK 'ves a la porta'
KO The determiner genre and the noun genre do not match! 'el' is masculine and 'finestra' is feminine.
OK 'corre a la finestra'
```

Figure 13: Movement and errors

Then you can also make the character **behold** things or **look** at entities. For example, if we want the character to look at the landscape:

```
Mira el paisatge
```

Then, he'll move to the window and will have a look through it. *Mira el paisatge* means “Look at the landscape”. Then he'll say what he sees. You'll be able to hear him speaking and you'll also have the text written in the *Dialogues console*.

You can also **talk** to living entities (which means human beings and animals). To do so, just type something like:

```
Parla amb la rata
```

Which means “Talk to the mouse”. Then some animations and speech will trigger. To ask someone for something:

```
Demana la clau al noi
```

With this, we'll have the character asking the guy for the key, since *Demana la clau al noi* means “Ask the guy for the key”.

You may also want to **open** or **close** things. For example, to open or close the window:

```
Obre la finestra
```

```
Tanca la finestra
```

Obre la finestra means “Open the window” and *Tanca la finestra* means “Close the window”.

And finally, to end this short tutorial, we'll show you how to **get in** places or **get out** from places. For example, to get out the house through the door:

```
Surt de la casa per la porta
```

And in order to get in the house through the window:

```
Entra a la casa per la finestra
```

Appendix B - Mainenance Manual

B.I Introduction

Every application may need to be improved, debugged or even studied. In order to help people do so the maintenance manual covers a wide range of technical aspects and particular details concerning the development of the Intelligent Catalan Tutor application. This document is intended to be used by anyone wishing to install, modify or debug the program. It is also suitable for people wanting to understand its internal structure. Although in this document there's an installation/uninstallation guide, most of the chapters of this manual go into much detail and some previous knowledge in computer science is advised.

B.II Previous requirements

In order to be able to install and use the J*ava SE Runtime Environment (JRE)* version 1.5 release 5 or above (if you also want to compile or debug the program, you should install the *Java SE Development Kit*, the *JDK*). The application may work in previous releases, but it has not been tested so it is not guaranteed it works well with them. The installer does NOT perform any system checking in this direction, so it will simply crash if you don't have it in your PC. To install the latest *JRE* or *JDK* just download it from the Java SE

Downloads page (http://java.sun.com/javase/downloads/index.jsp).

The Intelligent Catalan Tutor is a kind of 2D Adventure Game and although the rendering engine used is OpenGL, no 3D graphics are used at all, so old computers equipped with no brand-new graphic cards shouldn't have any problem in running it.

B.II.I Linux version notes – Important for Linux users

The system's been developed entirely in WindowsXP™ and every feature it contains has been tested to work properly in such system. However, it also works considerably well in any Linux or Unix-like platform with the X11 system installed on it. It needs the multi-platform toolkit *GTK* and the *OpenAL* Library (package libopenal). I tried my best to get it working in Linux but I faced some problems with the *Mozilla* HTML rendering engine for the help or with the speech sound.

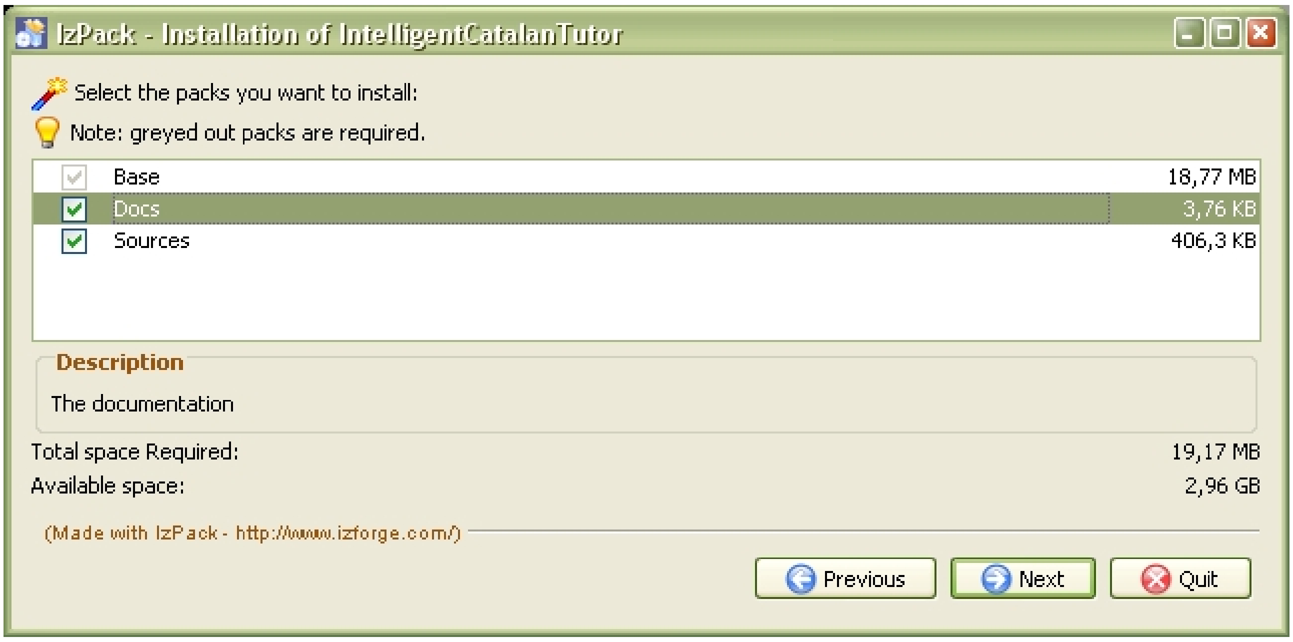
It's been tested in *Ubuntu 6.10* and the application executed properly but some features just didn't work. For example, it neither displayed the on-line help nor played the speech sounds. Although I didn't have enough time to fix all these bugs and decided to spend my time improving the application itself, a Linux version is also included since it executes correctly and you've got all the rest of features (complete interaction with the scene via text, *OpenGL* scene, background music, error feedback...) working, which makes it quite usable. Finally, to be able to launch this user manual from the application itself, you should have the *Evince Document Viewer* software installed.

B.III Installation and Uninstallation instructions

Here are the installation procedures for both Windows and Linux. Remember you must install the correct version of the software. Otherwise it wont work, logically.

B.III.I Windows installation

Just double click on the *Installation-win32.bat* file provided with the application package. This will make the installation wizard to start. Then choose the language (English or Catalan) and proceed following the instructions given. You will be prompted to select the installation folder and the packages. The only essential package is the *Base* package, which contains all the files and folders needed for the application to execute properly, but you can also tick the *Docs* package (contains some useful documentation) and the *Sources* package, which contains the source code. If you are not interested in extending the application or understanding how it was done, you shouldn't tick the *Sources* package.


IzPack - Installation of IntelligentCatalanTutor

Select the packs you want to install:

Note: greyed out packs are required.

✓	Base	18,77 MB
✓	Docs	3,76 KB
✓	Sources	406,3 KB

Description

The documentation

Total space Required: 19,17 MB

Available space: 2,96 GB

(Made with IzPack - http://www.izforge.com/)

Figure 14: Installation window

Alternatively, you can also open the Windows console (Start->Execute->”cmd”), change to the folder where you unzipped the application package and type the following:

```
$  java -jar CatalanTutorInstall.jar
```

B.III.II Linux installation

The process to install the application in a Linux system is quite similar. You just need to run the *Installation-linux* script provided in a shell and the installation wizard will show up. You may need to add execution permissions to the file to be able to run it:

```
$  chmod u+x Installation-linux
$  ./Installation-linux
```

B.III.III Uninstall the application

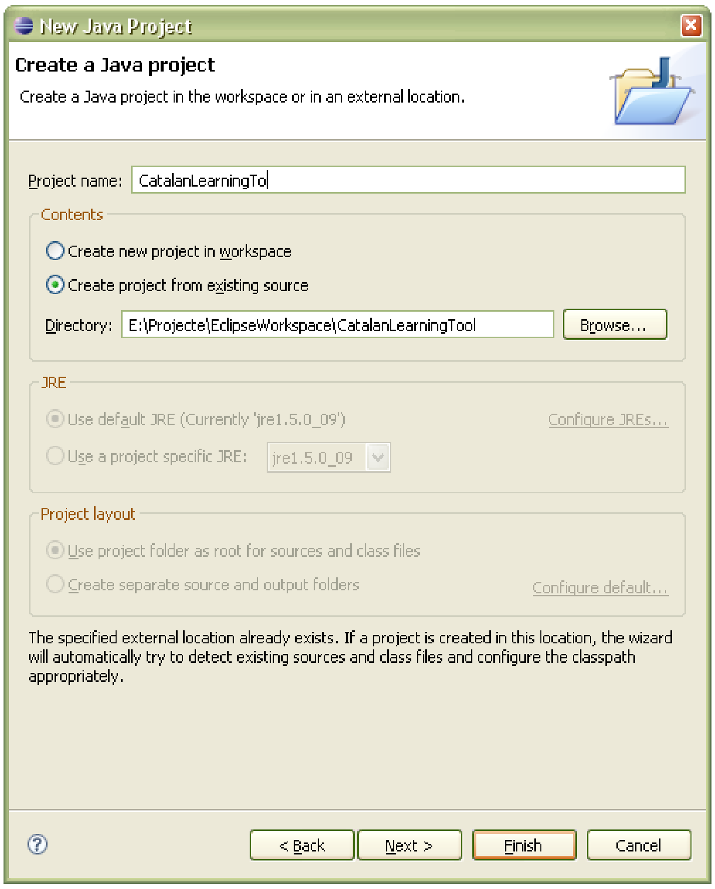
To uninstall the application just go to the *$APP_HOME/Uninstaller* folder and execute the *uninstaller.jar* file the same way you did with the installer.

```
$  java -jar uninstaller.jar
```

B.IV Building the application

To build and compile the system you'll need the Eclipse IDE 3.1 or greater (http://www.eclipse.org/downloads/). Firstly you need to download it and extract it in your local disk. If you need information about how to configure the Eclipse IDE just take a look at its documentation in the same web page.

Now we'll start by setting up the project into the IDE, so run the Eclipse and select a workspace location when prompted for it (any location in your disk should be fine). Now you need to create a new project from an existing source. Just go to the *File > New > Project...*, select *Java Project* and click Next. Write the project name (usually *CatalanLearningTool*), tick *Select project from existing source* and select the *CatalanLearningTool* folder which is inside the *EclipseWorkspace folder* from the project CD. Note that inside this folder there's an Eclipse project file called . Now, if everything went right, you should have a new project in the *Package Explorer* window in your Eclipse.


New Java Project

Create a Java project

Create a Java project in the workspace or in an external location.

Project name: CatalanLearningTo

Contents

Create new project in workspace

[✓ Create project from existing source]

Directory: E:\Projecte\EclipseWorkspace\CatalanLearningTool

JRE

Use default JRE (Currently 'jre1.5.0_09')

Use a project specific JRE: jre1.5.0_09

Project layout

Use project folder as root for sources and class files

Create separate source and output folders

The specified external location already exists. If a project is created in this location, the wizard will automatically try to detect existing sources and class files and configure the classpath appropriately.

Figure 15: Importing the project

Moreover, the project should be totally configured and you should even be able to run it. To do so, just open out the project folder and select the file *EntryPoint.java* in the package *com.abdn.project.start*. Remember that all sources are under the *src/* folder. Right click on it and then *Run as > SWT Application*. This should make the application to execute properly.

B.IV.I Building the application in Linux

If you are in Linux, you'll need to change a few parameters. First, go to the *Run* menu and then select *Run....* Then, select *EntryPoint* under the *SWT Application* group and in the *Argument*s tab, in *VM Arguments* change the line *-Djava.library.path=library/win32* for this one:

-Djava.library.path=library/linux

Once done this, you just need to change the current *SWT* package which is the win32 version for the *GTK* one intended for Linux. To do so, just remove the current SWT package (whose name is *org.eclipse.swt.win32.win32.x86_3.2.1.v3235.jar*) from the build path (right click > *Build path* > *Remove from build path*) and add the proper one for your system. Your Eclipse distribution should have it among its plugins, so just right click on your project, *Build path* > *Configure build path*... This will display the Build path window. Here go to the *Libraries* tab and click on *Add Variable*... button. Now double click on ECLIPSE_HOME and select the proper file under the plugins folder. The file should be something like *org.eclipse.swt.gtk.[version].jar*.

If you've reached this point you have configured the project correctly and you are now in position to start working with it. What you need now is to gain some base knowledge on the project itself, where to find the correct file to change something.

B.V Dependencies

The project package prepared to be imported into the Eclipse contains all the necessary libraries to execute the project properly. However, it is always a good practice to offer a complete list of software and hardware dependencies so that future developers can replace some of those libraries with newer versions.

B.V.I Software dependencies

The libraries used in the development of the application, in alphabetical order, are:

- ANTLR (www.antlr.org).
- From the project Jakarta-commons (http://jakarta.apache.org/commons/) of Apache:
    - BeanUtils (http://jakarta.apache.org/commons/beanutils/).
    - Digester (http://jakarta.apache.org/commons/digester/).
    - Logging (http://jakarta.apache.org/commons/logging/).

- Log4j (http://logging.apache.org/log4j/docs/).
- Lightweight Java Gaming Library (LWJGL, http://www.lwjgl.org/).
- Standard Widget Toolkit, SWT, from the Eclipse project (http://www.eclipse.org/swt/).

Briefly, the *ANTLR* library is used in the language parsing, to build the lexical analysis and the grammar. The *jakarta-commons* libraries are used as various resource libraries. Specifically, the *commons-digester* is used to parse the lexicon file which contains the words. Finally, the *LWJGL* allows access to cross platform libraries used in the application such as OpenGL and OpenAL from Java applications.

For the project to run without crashing you'll also need some operating system dependant libraries (basically needed for the *SWT* and the *LWJGL* modules to work). In the folder */library* -as explained below- are the libraries for both Windows and Linux.

B.V.II Hardware dependencies

In order to run the application properly you just need a PC with Windows XP or any Linux-like operating system (with a desktop environment and the GTK). Your graphic card must support OpenGL and you need a compatible sound card able to synthesize midi tracks, or, if not possible, any synthesizer emulation by software.

B.VI Hardware requirements

This section details the minimum hardware necessary to run the application.

B.VI.I Disk space

The installed base package requires 18,94 MB of disk space. If you choose to install the documentation and the sources, it will require about 20 MB more or less.

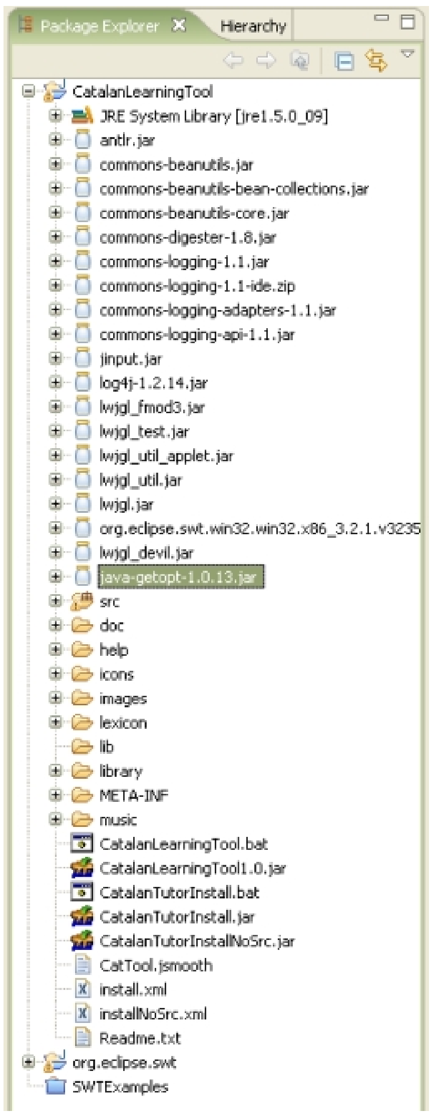
B.VI.II Memory requirements

The RAM memory required to run the application varies from one computer to another (depending on the type of graphic card, the sound card...), but having seen the results I got in my tests, it should fluctuate between 48 and 55MB, being 48 the minimum and 55 the maximum required. Once said this, it should work perfectly fine in any modern laptop or desktop computer, since mine is 4 years old -2003- and works pretty well on it.

I think the system should work in a computer with at least 128MB of RAM, but I have not tried. Anyway, the graphical stuff is the part that needs more computing power and any PC with a reasonably good graphic card (the scene is *OpenGL*) shouldn't have any problem.

B.VII Directory structure

The project files are organized into several subfolders inside the main project folder. Most of the folders have an intuitive name which clearly represents what they contain, but some may have at their turn more subfolders, leading us to a tree structure.



```
Package Explorer  Hierarchy
CatalanLearningTool
    JRE System Library [jre1.5.0_09]
    antlr.jar
    commons-beanutils.jar
    commons-beanutils-bean-collections.jar
    commons-beanutils-core.jar
    commons-digester-1.8.jar
    commons-logging-1.1.jar
    commons-logging-1.1-ide.zip
    commons-logging-adapters-1.1.jar
    commons-logging-api-1.1.jar
    jinput.jar
    log4j-1.2.14.jar
    lwjgl_fmod3.jar
    lwjgl_test.jar
    lwjgl_util_applet.jar
    lwjgl_util.jar
    lwjgl.jar
    org.eclipse.swt.win32.win32.x86_3.2.1.v3235
    lwjgl_devil.jar
    java-getopt-1.0.13.jar
    src
    doc
    help
    icons
    images
    lexicon
    lib
    library
    META-INF
    music
    CatalanLearningTool.bat
    CatalanLearningTool1.0.jar
    CatalanTutorInstall.bat
    CatalanTutorInstall.jar
    CatalanTutorInstallNoSrc.jar
    CatTool.jsmooth
    install.xml
    installNoSrc.xml
    Readme.txt
org.eclipse.swt
SWTExamples
```

- **/src**: Contains the source code of the whole application. It also contains the property files *log4j.properties* and the *catalanlearning.properties*. The former is related to the *log4j* library and describes how the logger outputs must be produced among other things. The latter is strongly bound to the application and contains image and sound files locations, output data, captions and a lot of information which must not be hard-coded. The packages and the source codes are explained deeply in subsequent sections.

- **/doc**: The doc folder contains, obviously, the documentation. It contains the User Manual and the current document.

- **/help**: Contains the HTML help that is displayed in the right pane, whose purpose is to help and guide the user through the learning of the Catalan language. It contains a file for each scene in the game and some other additional pages the user may look up to find more help in grammar or other issues.

- **/icons**: This folder contains some windows icons and images that are no longer used in the final version of the *CatalanLearningTool* but that are included in case someone wished to use them.

- **/images**: Here is located the bulk of the images used in the project.

    - **/images/icons**: Some windows icons.

    - **/images/scenes**: The game graphics, divided and organized by scenes. The inventory images are in the /images/scenes/inventory folder.

    - **/images/screenshots**: Some screenshots taken during the project development. You can see here the state of the application at earlier development stages.

    - **/images/shellImages**: The image that appears in the upper-left corner of the application window.

    - **/images/splashScreen**: A couple of images used as a splash screen.

- **/lexicon**: Contains the *lexicon.cat.xml* file containing the lexicon.

- **/lib**: All the libraries (*.jar* files) used in the project must be placed in this folder.

- **/library**: Operating System dependant libraries (*.dll* files in Windows, *.so* files in Linux). Those comprise OpenAL libraries, SWT libraries and some others.

- **/META-INF**: Contains the *Manifest.mf* file used to build the application *.jar* file.

- **/music**: Contains sound and music files.

    - **/music/speech**: Contains the wave files with the speech produced by the characters in the application.

    - **/music/backgroundmusic**: Contains the midi files with the background music.

The rest of the non-directory files that are located in the application base folder are various configuration files such as the *XML* file used to create the installation, the scripts that run the application, or the *readme* file.

B.VII.I Source files and packages

The source files in Java are usually grouped in packages. In this section all the packages and the source files of the project are described and summarized so that anyone quickly find the particular file responsible for something.

B.VII.I.I Package com.abdn.project.examples

This package was usually intended to hold general examples concerning any topic, but at last it was decided to put the specific examples inside the *test* package of every specific package.

- **Game.java** – Thats an example of a basic scene rendered in OpenGL. It displays a white square rotating over a black background. It is useful to study the main parts of a scene rendering: the main loop, the update and the rendering (OpenGL calls).

B.VII.I.II Package com.abdn.project.gui

This package holds the files responsible for building and controlling the graphical user interface. In the classes belonging to this packages we usually find calls and objects of the Standard Widget Toolkit (SWT), library we use to build the GUI.

- **BuildShell.java** – Builds the main application window and initializes all the widgets. It also controls option windows such as the *Speech volume control* window or the *About* window. That's the class responsible for controlling the *Dialogues* and the *Output* consoles.

- **MainApplication.java** – This class controls the rendering of the splash screen and launches the application afterwards. The time the splash screen is displayed is used to parse the lexicon and to initialize the main window.

B.VII.I.III Package com.abdn.project.language.model

The package model inside language holds all the classes (in this case just one) needed to represent a word in a sentence.

- **Word.java** – This class is used to represent directly a word in the lexicon. When the XML lexicon is parsed, each word in the lexicon becomes an instance of this class. Then, the instances are put in a list in order to have the lexicon in memory.

B.VII.I.IV Package com.abdn.project.language.module

This package contains classes that deal with the language module, the one that deals with the inputs the user type and produces the proper output feedback. It contains subpackages that are explained below.

- **LanguageRecognitionManager.java** – All the operations dealing with the language module called by the main program are supposed to pass through this class since it acts as an interface class.

- **ParsingError.java** – It just represents a parsing error and is produced by the syntactic analysis.

B.VII.I.V Package com.abdn.project.language.module.semantic

This package has the class that implements the semantic analysis and some other that have utilities.

- **AstUtil.java** – It contains a method that writes a given AST (Abstract Syntax Tree) into a String. It is only used for debugging purposes.

- **SemanticAnalysis.java** – As the name claims, this class implements the semantic analysis that parses the AST that comes out from the syntactic analysis. It checks the prepositions are used well and the concordance of noun-determiner-verb, amongst other things.

B.VII.I.VI Package com.abdn.project.language.module.syntactic

This has the classes needed for both the lexical analysis and the syntactic analysis. It also contains the grammar file, which will be translate into the syntactic analysis.

- **CatalanLexerModified.java** – This file implements the lexical analysis. Normally, it would be produced by ANTLR, but as I needed more features that those ANTLR offers, I had to write it myself.

- **CatalanParser.java** – This file is just the translation of the grammar contained in the *CatalanParser.g*. It's an automatically generated file.

- **CatalanParserTokenTypes.java** – Contains the token type constants. It is also generated automatically.

- **TestCatalanGrammar.java** – It is just a main class that tests the grammar itself atomically. Calls the *lexer* and the *parser*.

- **CatalanParser.g** – Contains the grammar in the ANTLR language. For more information about this language just check www.antlr.org.

B.VII.I.VII Package com.abdn.project.language.module.translation

This package contains the necessary files for converting the AST that produces the syntactic analysis into an Action Model suitable for updating the scene.

- **ActionModel.java** – Represents an action to be taken in the scene with all its complements, and it will be processed by the *SceneController* in order to update the scene.

- **Actions.java** – Constants representing the actions that can be carried out in the scene.

- **ActionTranslator.java** – This class contains only one public static function whose target is to split an AST (usually the outcome from the semantic analysis) into one or more actions describing only one action each and put them in the action queue in order to be processed.

- **Identifier.java** – The identifier class represents a nominal syntagm. It is used in the *ActionModel* to store the direct object, the actor, the subject...

B.VII.I.VIII Package com.abdn.project.language.module.xmlparser

This package contains the classes that perform the parser of the XML lexicon file.

- **LexiconManager.java** – This class is intended to be used as a manager for the lexicon so that all the operations over the lexicon should be done through this class.

- **LexiconParser.java** – The lexicon parser parses the lexicon. Thats quite obvious. It is done just once and it loads the lexicon contained in the XML file into a Map<Integer, List<Word>>, which is the data structure used to hold the lexicon during the execution.
- **ParsingTest.java** – This class loads the lexicon and halts. It is used for test and debugging purposes only.
- **RearrangeLists.java** – This class performs some useful operations over the parsed lexicon. It looks for entries with the same word and mixes their properties into one word, which adds to the lexicon. The two source words are deleted.

B.VII.I.IX Package com.abdn.project.language.spelling

Contains the necessary classes to perform the spell checking.

- **Distance.java** – This class contains the algorithm that implements the method that works out the distance between two words.
- **SpellingChecker.java** – Implements the spell checking. It takes a word as an input and uses the distance algorithm to find possible candidates amongst the lexicon that may be similar to the given word.

B.VII.I.X Package com.abdn.project.language.test

This package contains a couple of ANTLR examples with all the generated classes in the sub-packages *expr* and *token* respectively.

B.VII.I.XI Package com.abdn.project.rendering

This package contains classes and sub-packages whose purpose is to build the structure necessary to be able to render the scene and to animate the graphics.

- **SceneRendering.java** – This class builds the canvas where the scene will be displayed and calls the *DrawScene* class properly. Contains the main rendering loop and the resizing algorithm.

B.VII.I.XII Package com.abdn.project.rendering.graphics

This sub-package contains classes useful to deal with graphic stuff and animations.

- **Animation.java** – This file represents a basic animation usually composed by a sequence of sprites which may be displayed sequentially.
- **AnimationSequence.java** – This package is intended to group a sequence of *MotionEntities* and run them sequentially.
- **MotionEntity.java** – This may represent either an animation or a sprite. It's an abstract class.
- **Sprite.java** – Represents an image which has been loaded and is ready to be drawn on the scene.
- **Texture.java** – It also represents an image but without the necessary infrastructure to draw it. Just contains the image data.
- **TextureLoader.java** – This class loads any image file on disk into a Texture object.

B.VII.I.XIII Package com.abdn.project.scene.basic

This package contains some basic classes that will help us to manage the scene in a higher level. We've got a class hierarchy so that most of the functionalities of a concrete class are implemented in parent classes.

- **BackEnt.java** – Represents an entity that's in the background of the scene but does not have an image itself. However, it can be beheld by the user and referred to.
- **Background.java** – This class represents a background.

- **BasementBackground.java** – Represents the background of the shelter.
- **Bird.java** – Represents the bird flying from right to left and from left to right in the Garden scene.
- **Difference.java** – This class represents the difference image which every scene must have so that the system can work out where the user character may step on and where may not. More information about this class and how the system works can be found in the project report.
- **Dog.java** – Represents and defines the behaviour of the green dog standing by the house in the Garden scene.
- **Entity.java** – Represents ANY entity. This means that all the classes in this package inherit from *Entity*. This abstract class defines the methods any entity should implement.
- **Exit.java** – Represents the exit of the shelter in the Basement scene.
- **Fence.java** – Represents the fence that's on the Foreground in the Garden scene.
- **Finestra.java** – Represents the window of the house in the House scene.
- **GardenBackground.java** – Represents the background of the Garden scene.
- **GroundDoor.java** – Represents the door that leads to the basement in the Outside scene. It inherits form *Openable*.
- **GuardiaClau.java** – Represents the weird guy standing by the window in the House scene. This guy is the key holder.
- **HouseBackground.java** – The background of the House scene.
- **Lever.java** – Represents the lever the user can take from the ground of the Garden scene.
- **LivingEntity.java** – Represents any living entity, animals or persons. If a class inherits from *LivingEntity*, the user character will be able to speak to it.

- **Mouse.java** – Represents the running mouse which is inside the room in the House scene.
- **Moveable.java** – Any object that can be moved should inherit from this class.
- **Notice.java** – Represents the notice advising to beware the dog which is on the foreground in the Garden scene.
- **Openable.java** – Describes the methods and properties any entity that can be opened or closed should have.
- **OutsideBackground.java** – The background of the Outside scene.
- **OutsideDoor.java** – The door to enter the house in the Outside scene.
- **Porta.java** – The door to get out in the House scene.
- **ScConst.java** – Contains some important constants concerning the state of the entities.
- **Statue.java** – Represents the statue blocking the shelter door in the Outside scene.
- **Takeable.java** – Represents any entity that can be taken by the user character.
- **Tree.java** – Represents a tree.
- **Tutor.java** – Represents the user character, the tutor in the game.
- **WindowOut.java** – Represents the window to get in the house in the Garden scene.

B.VII.I.XIV Package com.abdn.project.scene.controller

This package contains some classes that control and update the scene.

- **Ents.java** – Contains a list for each scene with the entities that should be drawn when the scene is active. It also defines all of the entities (that otherwise are in any of the previous lists) so that they can be accessed directly.

- **OutputMessage.java** – Represents a message that should be displayed in one of the two consoles. The *OutputMessages* for a particular action are displayed just after that action has been triggered and processed.

- **SceneController.java** – This is one of the most important classes in the application. It contains a queue of *ActionModel*s and it is responsible for parsing this action model and triggering the proper animations in the scene so that it updates correctly.

B.VII.I.XV Package com.abdn.project.scene.draw

This package contains a class for each scene named *Draw[name-of-scene].java* and that are called when their scene must display. It updates the entities the scene contains and changes the background, the difference, the background music and some other parameters.

- **DrawBasement.java** – Implements the DrawScene for the basement scene.

- **DrawGarden.java** – Implements the DrawScene for the garden scene.

- **DrawHouse.java** – Implements the DrawScene for the house scene.

- **DrawOutside.java** – Implements the DrawScene for the outside scene.

- **DrawScene.java** – This is the main class, from which all others inherit, and defines the methods that shall be called at each iteration of the main rendering loop to update and draw the whole scene.

B.VII.I.XVI Package com.abdn.project.scene.inventory

Contains the classes that control the inventory.

- **Inventory.java** – Contains a list of *InvObject*s representing the objects that are currently in the inventory and the methods to add and delete items from the list.

- **InvObject.java** – Represents an object in the inventory. It has a name and a path to an image which will be displayed for this object in the inventory zone of the main application window.

B.VII.I.XVII Package com.abdn.project.sound.basic

Contains the two classes used to play both the background music and the speech sounds.

- **MidiPlayer.java** – This class creates a thread that plays a midi song until it is told to stop. It is responsible for playing the background music.
- **WavPlayer.java** – This class is able to play a wave file or more than one at once. It is used to play the speech sounds.

B.VII.I.XVIII Package com.abdn.project.sound.test

Contains some sound test classes.

B.VII.I.XIX Package com.abdn.project.start

Contains the entry point to the application.

- **EntryPoint.java** – This class contains a main method that must be called in order to run the application.

B.VII.I.XX Package com.abdn.project.util

The util package contains some utility classes that are used from any part in the application.

- **Constants.java** – This is a global constants file.
- **ResourceManager.java** – This class helps us deal with the properties file. It is able to retrieve a string from this file given a key.

B.VIII Known bugs

Here is the list of the known bugs of the system.

- In some rare occasions it may occur that the character, the tutor, is drawn in front of an entity that is closer to the camera. This is due to some entities have a drawing canvas bigger than others so that it may appear they are closer when they are actually further. Moreover, in the garden and outside scenes, the character is scaled down when he walks away and he's scaled up when he walks to the camera. This may produce this depth bug as well.

- Sometimes the speaking animation and the voice are not synchronized very well. This is due to the speech voice was added after building the rest of the system, and, specifically, after designing the animation system. So the mouth animation of characters should depend on the voice but it does not.

- In Linux the application may crash at startup or may not play any sound if the sound device is busy. This may happen when there's any music or video player running in the system. It is not a bug of the application itself but of the operating system.

- Finally, it has happened that sometimes the application gets stuck and everything gets slower, animations, rendering and even the *GUI*. It has only happened in my laptop, which is a bit old, when resizing the application. However, in my desktop computer works fine and this problem never occurs.

Appendix C - Glossary

Terminology	**Meaning**
CALL	Computer-Assisted Language Learning
ICALL	Intelligent CALL
XML	Extensible Markup Language
LWJGL	Lightweight Java Gaming Library
OpenGL	Open Graphics Library
OpenAL	Open Audio Library
SWT	Standard Widget Toolkit
FIFO	First In First Out
ALICE-chan	Automated Language-Instruction/Curriculum Environment
ALICE (bot)	Artificial Linguistic Internet Computer Entity
CALLE	Computer-Assisted Language Learning Environment

NLP	Natural Language Processing
ANTLR	Another Tool for Language Recognition
PCCTS	Purdue Compiler Construction Toolset
JavaCC	Java Compiler Compiler
Log4j	Log For Java
UML	Unified Modeling Language
GUI	Graphical User Interface
AST	Abstract Syntax Tree
GIMP	GNU Image Manipulation Program
GPL	General Public License

References

[1] Claude Frasson, Gilles Gauthier, Alan Lesgold (Eds.), *Intelligent Tutoring Systems, Third International Conference, ITS '96, Montréal, Canada, June 1996.* Springer.

[2] V. Melissa Holland, Jonathan D. Kaplan, Michelle R. Sams, *Intelligent Language Tutors – Theory Shaping Technology* – 1995, Lawrence Erlabaum Associates.

[3] Lightweight Java Gaming Library.
http://www.lwjgl.org

[4] OpenGL. Graphics library specification.
http://www.opengl.org

[5] OpenAL. Audio library specification.
http://www.openal.org

[6] Argo UML. UML modeling tool.
http://argouml.tigris.org/

[7] Standard Widget Toolkit. GUI toolkit from eclipse.
http://www.eclipse.org/swt/

[8] ANTLR. Language recognition tool specification.
http://www.antlr.org/

[9] The Digester component. XML to Java mapping component.
http://jakarta.apache.org/commons/digester/

[10] GIMP. Image manipulation program.
http://www.gimp.org/

[11] Computer-Assisted Language Learning. An introduction.

http://www.gse.uci.edu/faculty/markw/call.html

[12] Computer-Aided Language Learning at International Computer Science Institute. http://www.icsi.berkeley.edu/~gelbart/call/

[13] Slime Forest Adventure. Learning the Japanese alphabets in a game. http://lrnj.com/

[14] Tactical Language & Culture Training System. Advanced ICALL system. http://www.tacticallanguage.com/tacticaliraqi/

[15] Log4j. A logging library for java. http://logging.apache.org/log4j/docs/

[16] Catalan grammar. Comprehensive grammar of the Catalan language. http://www.sola-sole.com/gramat.htm

[17] Quake III Game Engine. http://ioquake3.org/, http://en.wikipedia.org/wiki/Quake_III_engine