PROYECTO FIN DE CARRERA

Título:	Desarrollo de una interfaz gráfica de usuario para simulación social de planes de evacuación basada en BML
Título (inglés):	Development of a graphical user interface for social simulation of evacuation plans based on BML
Autor:	Antonio Maria Diaz Dominguez
Tutor:	Carlos A. Iglesias Fernández
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Mercedes Garijo Ayestarán
Vocal:	Tomás Robles Valladares
Secretario:	Carlos Ángel Iglesias Fernández
Suplente:	Francisco González Vidal

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

PROYECTO FIN DE CARRERA

Development of a Graphical User Interface for Social Simulation of Evacuation Plans Based on BML

Antonio Maria Diaz Dominguez

Marzo de 2016

Resumen

Esta documento es el resultado de un proyecto que tiene como objetivo realizar una interfaz gráfica para un simulador de evacuación de edificios, llamada SmartSim.

La interfaz permite realizar una simulación dadas una serie de posiciones iniciales de agentes y un escenario en donde tiene lugar la simulación. El programa calcula las rutas de evacuación del edificio propuesto y representa la evacuación.

Se han desarrollado varios casos de uso o simuladores para explotar las capacidades del programa. Cada uno de estos simuladores poseen varias opciones de configuración y de extracción de resultados. El objetivo de incluir estas opciones es facilitar el uso del programa por parte de un usuario no experimentado en tareas de programación.

La arquitectura del proyecto se ha estructurado en módulos para permitir el intercambio de cada uno de estos módulos en usos futuros.

Por último, se han presentado las conclusiones extraídas del trabajo, las posibles líneas de continuación del proyecto, así como los siguientes pasos en cuanto a desarrollo y aprovechamiento del programa.

Palabras clave: Social Simulation, UbikSim, SmartBody, Behavior Markup Language, BML, Python, Evacuation Plan, Ambient Intelligence, Multi-agent Based Simulation, Graphical User Interface

Abstract

This document is the result of a project which objective is to develop a graphic interface for an evacuation plans simulator, the SmartSim project.

The interface allows to create a simulation given the initial positions for agents and the scenario in which the simulation takes place. The program calculates the evacuation plan of the building and represents the simulation.

Some use cases or simulators have been developed to take advantage of the program capacities. Every one of these simulators can be configured with many options. Also many options of retrieving the results have been provided. The objective of including these options is easing the program usage by a non-experimented user.

The project architecture has been structured in modules in order to allow the change of any of these modules in future developments.

Finally, we gather the extracted conclusions plus some lessons learnt, the possible line of work regarding the continuance of the platform as well as the next step regarding development and exploitation of the service.

Keywords: Social Simulation, UbikSim, SmartBody, Behavior Markup Language, BML, Python, Evacuation Plan, Ambient Intelligence, Multi-agent Based Simulation, Graphical User Interface

Agradecimientos

Me gustaría aprovechar este espacio para agradecer a todo aquel que ha ayudado que hoy esté aquí. Intentaré no dejarme a nadie, aún sabiendo que es imposible que no lo haga.

En primer y especial lugar a mis padres, por ayudarme a llegar donde he llegado. Por darme lo que a ellos no les pudieron dar. Sin su apoyo, su ayuda y sus esfuerzos sé que jamás lo habría logrado.

A mis amigos, tanto dentro como fuera de la Escuela. Tanto dentro como fuera de Madrid. Tengo la grandísima suerte de haberme rodeado muy bien. No puedo nombrar a nadie en especial porque necesitaría otra memoria sólo para los nombres y el porqué. Ellos saben quiénes son.

A Carlos Ángel, por su paciencia y haberme ayudado en todo lo que le he pedido. Por haberme tendido una mano dándome la oportunidad de realizar este proyecto.

A Carmen, por creer en mí. Por creer en nosotros.

A todo aquel que lee esto y sabe que significa algo para mí.

Contents

List of Figures

List of Tables

List of Acronyms

MOSI-AGIL MOdelado Social de Inteligencia Ambiental aplicado a Grandes InstaLaciones . . . 3

BML Behaviour Markup Language . . . 4

AmI Ambient Intelligence . . . 10

MABS Multi-Agent Based Simulation . . . 10

XML eXtensible Markup Language . . . 15

ECA Embodied Conversational Agents . . . 15

DOM Document Object Model . . . 15

HTTP Hypertext Transfer Protocol . . . 11

HTTPS Hypertext Transfer Protocol Secure . . . 17

JSON JavaScript Object Notation . . . 12

sbgui SmartBody Graphical User Interface . . . 20

IDE Integrated Development Environment . . . 54

CHAPTER 1

Introduction

This chapter provides an introduction to the problem which will be approached in this project. It provides the context of our project, explaining some key social simulation concepts. Furthermore, a deeper description of the project and its environment is also given.

1.1 Context

1.1.1 Social simulation

Social simulation is a research field that applies computational methods to study issues in the social sciences.

In social simulation, computers supports human reasoning activities by executing processes, mechanisms and behaviors that build the reality. This approach allows to investigate some complex models that cannot be investigated through mathematical models.

Some authors as Robert Axelrod regards social simulation as a third way of doing science, differing from both the deductive and inductive approach [1]. The generated data can be analysed using induction, but it comes from a set rules obtained by a deductive method.

A kind of social simulation is agent based social simulation. Agent based simulation is the representation of social systems as societies of agents, executing and analysing their behaviour [2]. Agents are autonomous software systems that are intended to describe the behaviour of observed social entities such as individuals or groups.

1.1.2 The Mosi-Agil project

This project has been developed in the context of the MOdelado Social de Inteligencia Ambiental aplicado a Grandes InstaLaciones (MOSI-AGIL) project [3]. The MOSI-AGIL project is a four-year program funded by the Autonomous Region of Madrid through the program MOSI-AGIL-CM [1]. It aims at creating a body of knowledge and practical tools which are necessary to handle more effectively the behaviour of occupants of large facilities. Therefore, the project studies the development of ambient intelligence and intelligent environments supported by the use of Agent-Based Social Simulation.

The MOSI-AGIL program presents the following specific objectives:

1. Model of relevant social behaviours in intelligent spaces of big installations and their interfaces with the intelligent services (sensors and actuators)
2. Develop of services of control and monitoring in intelligent spaces
3. Platform to assist the development, usage and decision making in big intelligent spaces to achieve the last objectives

[1] S2013/ICE-3019

4. Demonstration of the results in the three use cases:
    - Simulation and deployment of a smart home
    - Simulation of control, monitoring and evacuation in Madrid Arena
    - Simulation and deployment in a university building
5. Coordination of the participants, integration and diffusion of the results

1.2 Project objectives

The main objective of this project is to develop a Graphical User Interface to an evacuation plans simulator making use of Behaviour Markup Language (BML) [4] . The project result will be named SmartSim. This project will be part of the MOSI-AGIL, as it is part of the use cases defined: simulation in a university building. It must be friendly for a not experimented user but also powerful for an user who want to use it in further developments.

To develop this main goal, we will need to achieve the following sub-objectives:

- Setting an indoor evacuation plans simulator
- Finding a suitable framework that allows us to serve a graphic interface to the social simulator
- Integrate the social simulator and the graphic engine
- Define a set of use cases that allows us to develop a realistic simulator able to take advantage of the functionality of the Graphical User Interface
- Make configurable the simulator for the user
- Testing the whole system

1.3 Structure of this document

In this section we will provide a brief overview of all the chapters of this document. It has been structured as follows:

Chapter 1 provides an introduction to the problem which will be approached in this project. It provides an overview of the context of the project. Furthermore, a deeper description of the project and its environment is also given.

Chapter 2 contains an overview of the existing technologies on which the development of the project will rely.

Chapter 3 describes the architecture of the system.

Chapter 4 describes the developed use cases. It is going to be explained the running of all the tools involved and its purpose. A collection of simulators with differents graphical users interfaces has been developed. We will analyse all them providing instructions to setting them up and explaining the expected results we can get of them.

Chapter 5 sums up the findings and conclusions found throughout the document and gives a hint about future development to continue the work done for this project.

CHAPTER 2

Enabling Technologies

This chapter introduces which technologies have made possible this project. We need UbikSim to create our paths and setting up our simulation. To render the simulation we need the SmartBody framework. We will operate this framework using BML to control movements and behaviours of the agents. The scene will be set using the Python language.

2.1 Overview

Agent based social simulation, has many possibilities of use, but maybe one of the most interesting is the possibility of predicting the behaviour of individual agents in complex environments.

A nice example of this is the possibility of simulate dangerous environments and test the results of acting according to simple rules. A common application is the simulation of a building in fire. The building is populated by different numbers of people, some doors are blocked, some people are handicapped... these common situations are simulated and the different agents act according to a few simple rules. The results of their actions can be measured and the best of the models can be picked without risking anything, avoiding dangerous situations.

An implementation of this type of simulator was developed in GSI. This tool is Ubiksim, a framework used for social simulations with a large sort of options, such as possibilities of editing and creating environment by users with an easy interface or set the number of agents from one to a huge amount of them. However, this tool presents some problems. It is implemented in Java and the user needs to have some knowledge about this language.

Because of this, the goal of this project is to create a more complex and easy to use graphical user interface to simulate evacuation plans. The selected environment will be the E.T.S.I. Telecomunicación, more complex than the scenarios commonly used in Ubiksim.

This framework will be operated using Python [5] and BML, friendlier modes of use to non-experienced users.

The use of BML is very interesting and fits the objectives of the project. It is a language that describes human non-verbal and verbal behaviour in a manner independent of the particular realization (animation) method used.

2.2 UbikSim

2.2.1 Overview

We will use a version of the social simulator UbikSim 2.0 [6], developed by Emilio Serrano and Carlos Ángel Iglesias to recreate the human behaviour inside a building. The map of our use cases will be modelled and represented on this tool. UbikSim will have the duty to send the initial positions of the agents involved in the simulation and the paths they will

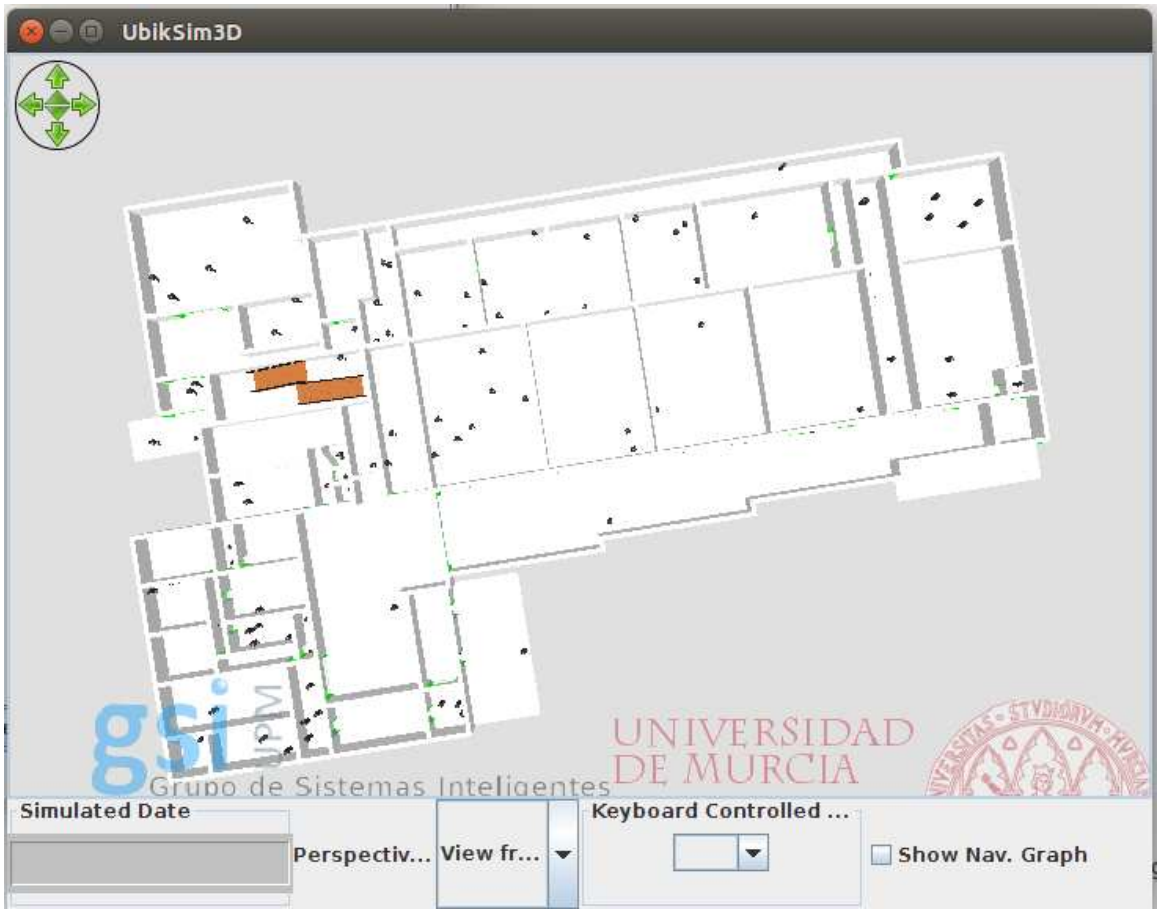

Figure 2.1: Graphic interface of UbikSim

follow.

UbikSim is a framework used to develop social simulation which emphasizes the construction of realistic indoor environments, the modelling of realistic human behaviours and the evaluation of Ubiquitous Computing and Ambient Intelligence systems. UbikSim is written in Java [7] and employs a number of third-party libraries such as Sweet Home 3D [8] and MASON [9]. Our implementation consists of a console that will let us launch the simulation as well as a map in 3D or 2D where we will able to see the position of all the agents involved in the simulation.

UbikSim is a tool for using Multi-Agent Based Simulation (MABS) in Ambient Intelligence (AmI) [10]. AmI is the development of computerized environments, sensitive to human and objects actions. MABS is the modelling of big environments with many agents taking every one of the subjects in the simulation as an agent as defined in agents based social simulation [11]. In other kinds of simulation, the entire environment is modelled as a mathematical model where the set of individuals is viewed as a structure that can be char-

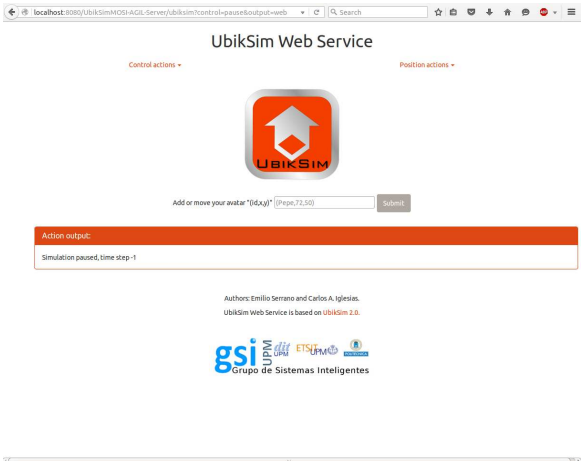

Figure 2.2: Web service of UbikSim

acterized by a number of variables. Traditionally, when AmI is applied to a large number of users, there is a point where the real tests are not feasible. However, UbikSim can test social behaviours of large users groups applying the MABS approach to AmI environments, proving as a solution for these problems.

We use a webapp version of UbikSim. This version has been developed by Emilio Serrano as part of the MOSI-AGIL project. It can be controlled via Hypertext Transfer Protocol (HTTP) requests. We will use these requests to communicate with it.

2.2.2 Simulation controls

UbikSim simulations are temporally divided in steps. We can advance the simulation clock and start our simulation from the step we prefer. The controls that allows us to control the simulation are:

- Pause: Pauses the simulation or plays it if it is already paused. We have to execute this control if the simulation has not started yet to set the simulator.
- Play: Advances one step of the simulation.
- Stop: Stops the simulation and destroys it.

2.2.3 UbikSim MOSI-Agil

Our UbikSim version can be controlled using HTTP requests. We use the API implemented in this version of UbikSim to interact with it.

Figure 2.3: UbikSim controls

Using this API we can retrieve the paths and positions of our agents. Using this API, UbikSim gives us the results in a JavaScript Object Notation (JSON). Parsing these JSON we can retrieve the steps of our agents and their initial positions.

2.2.4 UbikSim editor

UbikSim provides us of an Editor we can take advantage to export the environment to SmartBody. We can create the scenario in it using a very friendly interface. Thanks to the possibilities of this editor, we can create a very detailed scenario that we can use in SmartBody. The map creation is very friendly as we will see furtherly in Chapter 4.2.

Option	Effect
output=web	Displays the web graphic interface.
control=pause	Executes the pause control.
control=play	Executes the play control.
control=stop	Executes the stop control.
control=frames	Starts the displayers in the server side.
position=people	Returns the agents positions.
position=map	Returns the map coordinates and obstacles.
position=emergency	Returns the emergency position and room.
position="(id,x,y)"	Adds the agent to the position if there are no agents or obstacles in the position.

Table 2.1: UbikSim API

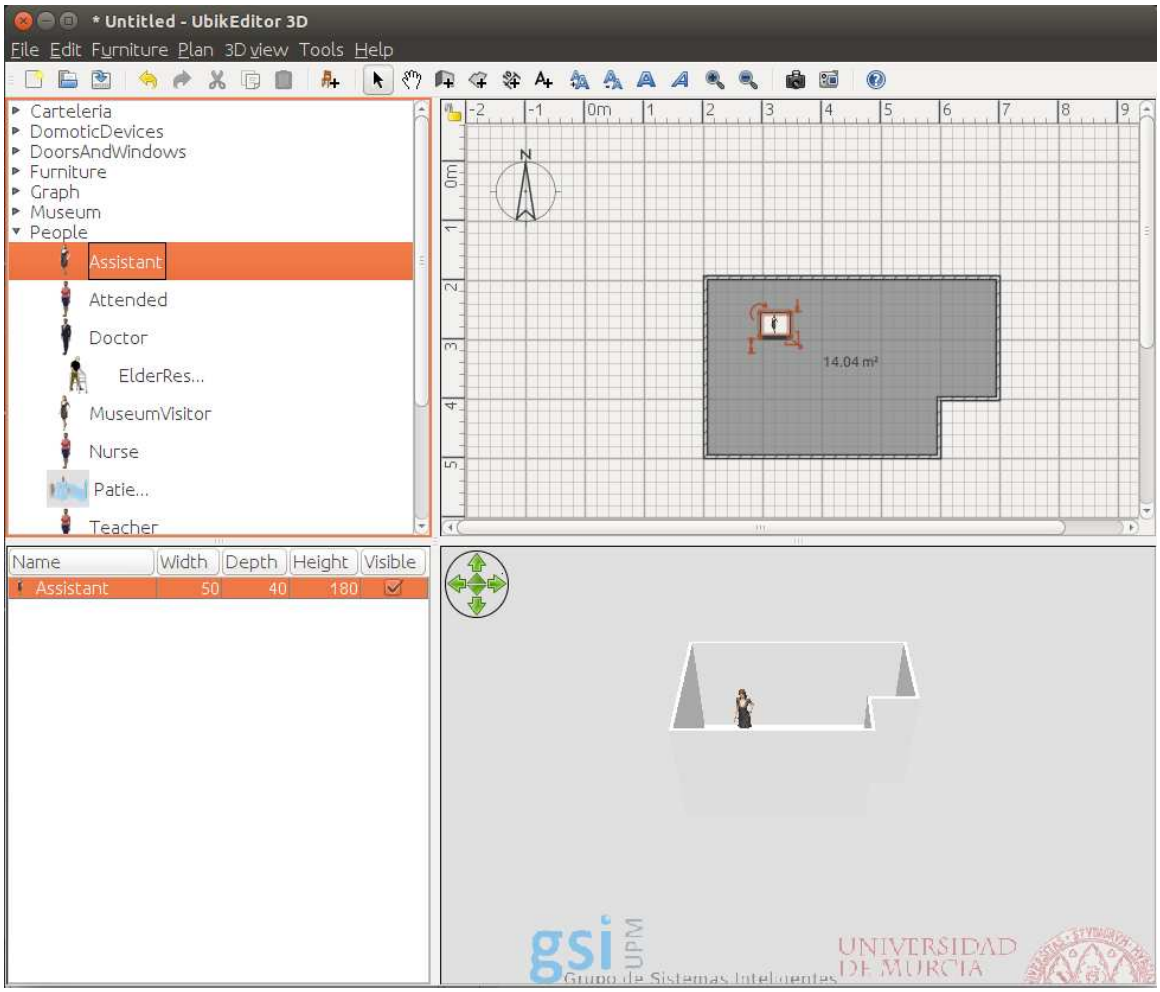

Figure 2.4: UbikSim editor interface.

2.3 BML

When we face the task of creating a social simulator we could have some difficulties on describing the behavior of our agents. In our project we will use BML, an eXtensible Markup Language (XML) [12] description for these tasks.

2.3.1 Introduction

BML is an XML description language for controlling the verbal and non verbal behaviour of Embodied Conversational Agents (ECA) [13]. A BML block describes the physical realization of behaviours (such as speech and gesture) and the synchronization constraints between these behaviours. The module that executes behaviours specified in BML on the embodiment of the ECA is called a BML Realizer (See Section 2.3.3).

2.3.2 BML Messaging Architecture

BML does not prescribe a specific message transport. Different architectures have drastically different notions of a message. A message may come in the form of a string, an XML document or Document Object Model (DOM) [14], a message object, or just a function call. However, no matter what message transport is used, the transport and routing layer should adhere to the following requirements:

- Messages must be received in sent order.
- Messages must contain specific contents that can be fully expressed as XML expressions in the format detailed in this document.

Currently, there are two types of messages:

- BML Requests.
  - Sent by the Behavior Planner to the Behavior Realizer.
  - BML requests are sent as <BML>blocks containing a number of behavior elements with synchronisation.
- Feedback Messages.
  - Sent by the Behavior Realizer.

– Used to inform the planner (and possibly other processes) of the progress of the realization process.

2.3.3 The BML Realizer

Conceptually, BML Realizers execute the behaviours described by a stream of incoming BML Requests. A BML Realizer is responsible for executing the behaviours specified in each BML request sent to it. Synchronization must be specified by the user.

Each BML Request represents a scheduling boundary. That is: if behaviours are in the same BML request, this means that the constraints between them are resolved before any of the behaviours in the request is executed.

2.3.4 BML Request Syntax

All BML behaviors must belong to a BML block. A BML block is formed by placing one or more BML behaviour elements. Unless synchronization is specified, it is assumed that all behaviors in a BML block start at the same time after arriving at the BML realizer. In Listing 2.1 we find a example of a BML command.

Listing 2.1: A BML command example

```
<bml id="bml1" xmlns="http://www.bml-initiative.org/bml/bml-1.0"
    characterId="Alice">
   <required>
     <gaze id="gaze1" target="PERSON1"/>
     <speech id="speech1"><text>Welcome to my humble abode</text></speech>
   </required>
   <head id="nod1" type="NOD"/>
 </bml>
```

2.4 Python

Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in other languages.

Python supports multiple programming paradigms, such as object-oriented, used in our

project. It features a dynamic type system and automatic memory management.

Python interpreters are available for installation on many operating systems, allowing Python code execution on a wide variety of systems. Python is free and open-source . Python development is managed by Python Software Foundation.

2.4.1 Python modules

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable name. Every Python modules has his own symbol table.

A module can contain executable statements as well as function definitions. These statements can be executed from a Python program that imports a module.

Modules can import other modules. It is customary to place all import statements at the beginning of a module. The imported module names are placed in the importing module's global symbol table.

In our project we use the following python modules to interact with Ubiksim:

- Module httplib [15]

  This module defines classes which implement the client side of the HTTP and Hypertext Transfer Protocol Secure (HTTPS) protocols.

  We use this module to interact with our web version of UbikSim. We send HTTP requests to it using its HTTP interface to retrieve the agents positions and routes.

- Module json [16]

  JSON [17] is a lightweight data interchange format inspired by JavaScript object literal syntax. It is widely used for web programming. This module allows Python to use data in JSON format.

  We use this module in order to control the data we retrieve from UbikSim.

- Module math [18]

  It provides access to the mathematical functions defined by the C standard.

  We use this module to perform some complex calculations we need to do.

- Module io [19]

The io module provides the Python interfaces to stream handling. SmartBody does not us allow to perform input/output actions using the Python 2 standard way, so we have to import this module in our project.

- Module config parser [20]

  This module defines the class ConfigParser. This module allows to create and parse a configuration file. This can be used to write Python programs which can be customized by end users easily.

  We use this module to handle a configuration file.

2.5 SmartBody

SmartBody is a character animation platform originally developed at the USC Institute for Creative Technologies. SmartBody provides locomotion, steering, object manipulation, lip syncing, gazing, nonverbal behavior and retargeting in real time.

SmartBody is written in C++ and can be incorporated into most game and simulation engines. SmartBody is a BML realization engine that transforms BML behavior descriptions into realtime animations. SmartBody runs on Windows, Linux, OSx as well as the iPhone and Android devices.

Smartbody provides a very complete rendering platform. It brings a variety of characters with predefined movements animations and behaviour sets. This allows us to simulate an environment without creating any character. Just in case we need to create one, the needed operations for creating it can be easily performed. We would only need exporting its geometry and provide it animations and behavior sets using SmartBody. It provides wizards to create them. If we prefer, we can use the ones included in the program.

2.5.1 Creating a Character

To create a character we need to create some files and import them to SmartBody. These files we would need are:

- The model. We have to provide the geometry of our character. We can get this model from a polygonal file. The supported formats are the Wavefront Format Files [21] (files with .obj extension) or a Collada Format File [22] (files with .dae extension). To add the geometry to our character, we would use the SmartBody Python API.

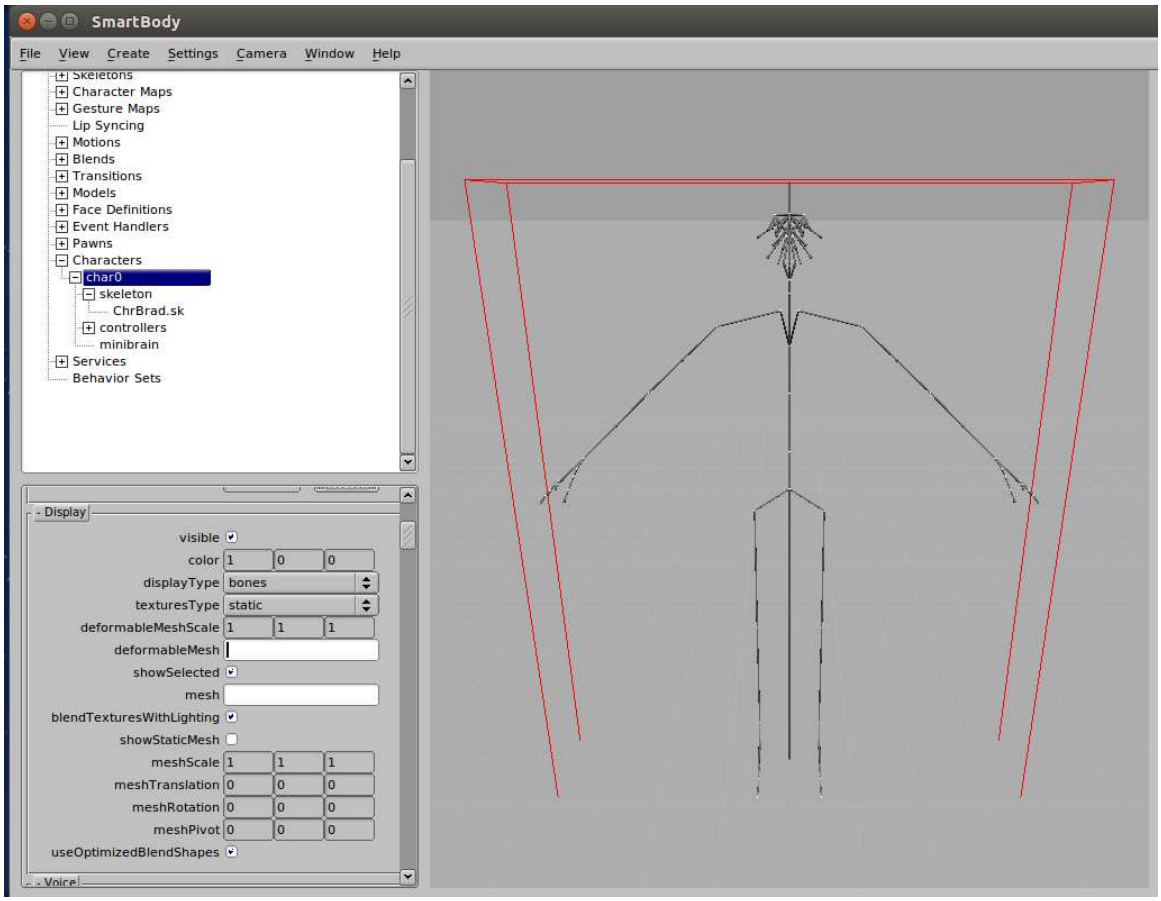

Figure 2.5: Adding the model to our character

- We have to provide the skeleton to our character. This skeleton provides the basic structure to our character.

In SmartBody we have to import the path of the file in our filesystem. After importing it we create a character and we add the model and the skeleton. We can perform these actions using the SmartBody editor or the Python API of SmartBody, as we will see later.

To create it in the editor we have to follow the next steps:

1. Import the folder by File ->Import folder.
2. Create a new character by Create ->Character. We add the skeleton at this step.
3. Add the model by using the menu that opens when we select the character.

2.5.2 Animating a Character

Now we have our character created, we have to animate it.

To animate it we have to add some files to our character. One more time, we can do it using the SmartBody Python API [23] or the SmartBody Graphical User Interface (sbgui) editor.

The files we have to import are:

- Behavior. The behavior is a python script that links the gestures of a character with the character. This gestures can be created and edited by some editors that SmartBody provides.
- The Steer Manager. The steer manager is a part of SmartBody. It is the component that manages the characters movements.

To add these to the character is necessary to make use of the Python API.

2.5.3 Python Scripts Usage

SmartBody can be controlled by Python. SmartBody has a Python API. We can control the simulations making use of this API via code.

There are many aspects of SmartBody that we can control making use of this API. From cameras, to scene parameters and characters settings, almost every little aspect of SmarBody can be control via Python.

Of course, this API provides us the instruments to use BML within our characters, as we will see later.

There are some ways that we can use to send Python commands to SmartBody. But we will make use of the Commands Window and the Scripts Loader mainly.

The commands window is part of the sbgui user interface. In this window (Figure 2.6) we can introduce many commands as we want, even full scripts. The Python interpreter will execute them. This tool is very useful if we want to test the effects of Python commands or short scripts.

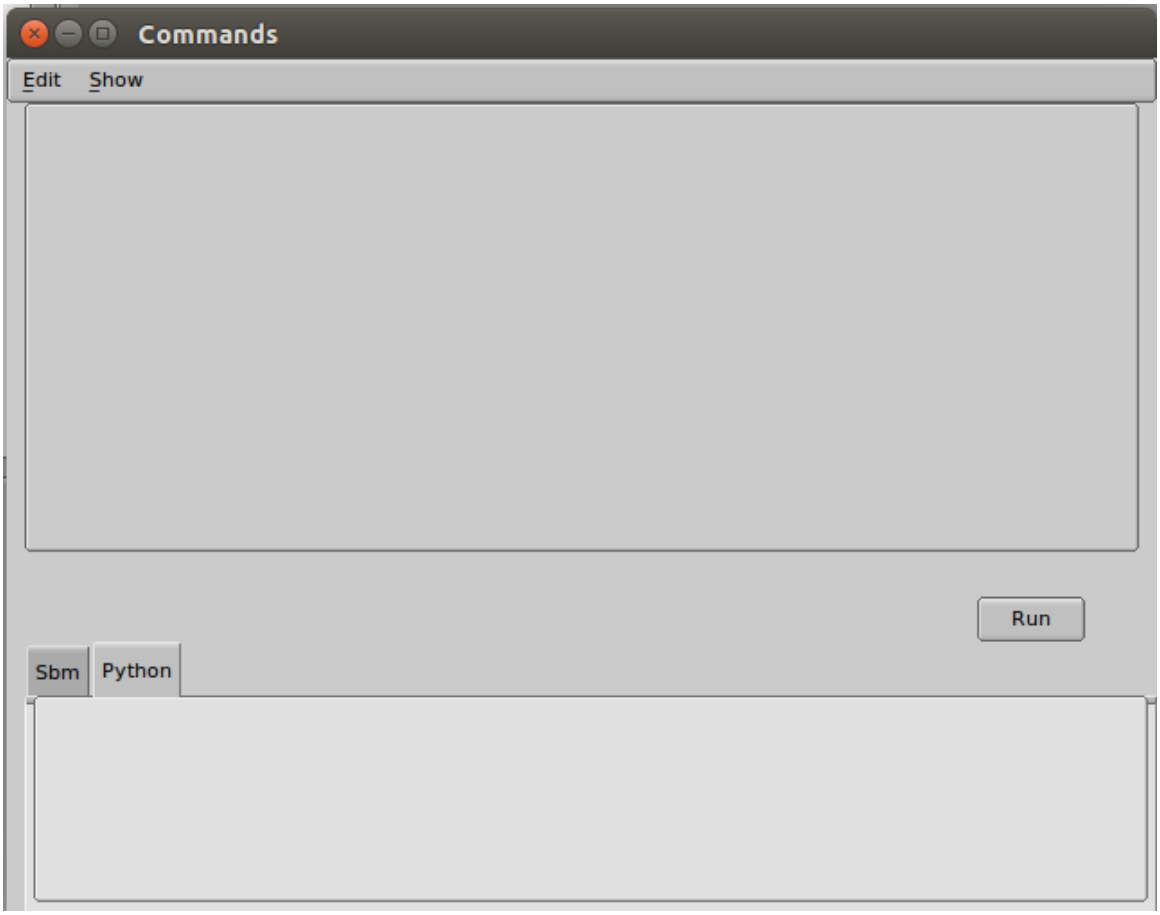

Figure 2.6: The Python command window of SmartBody

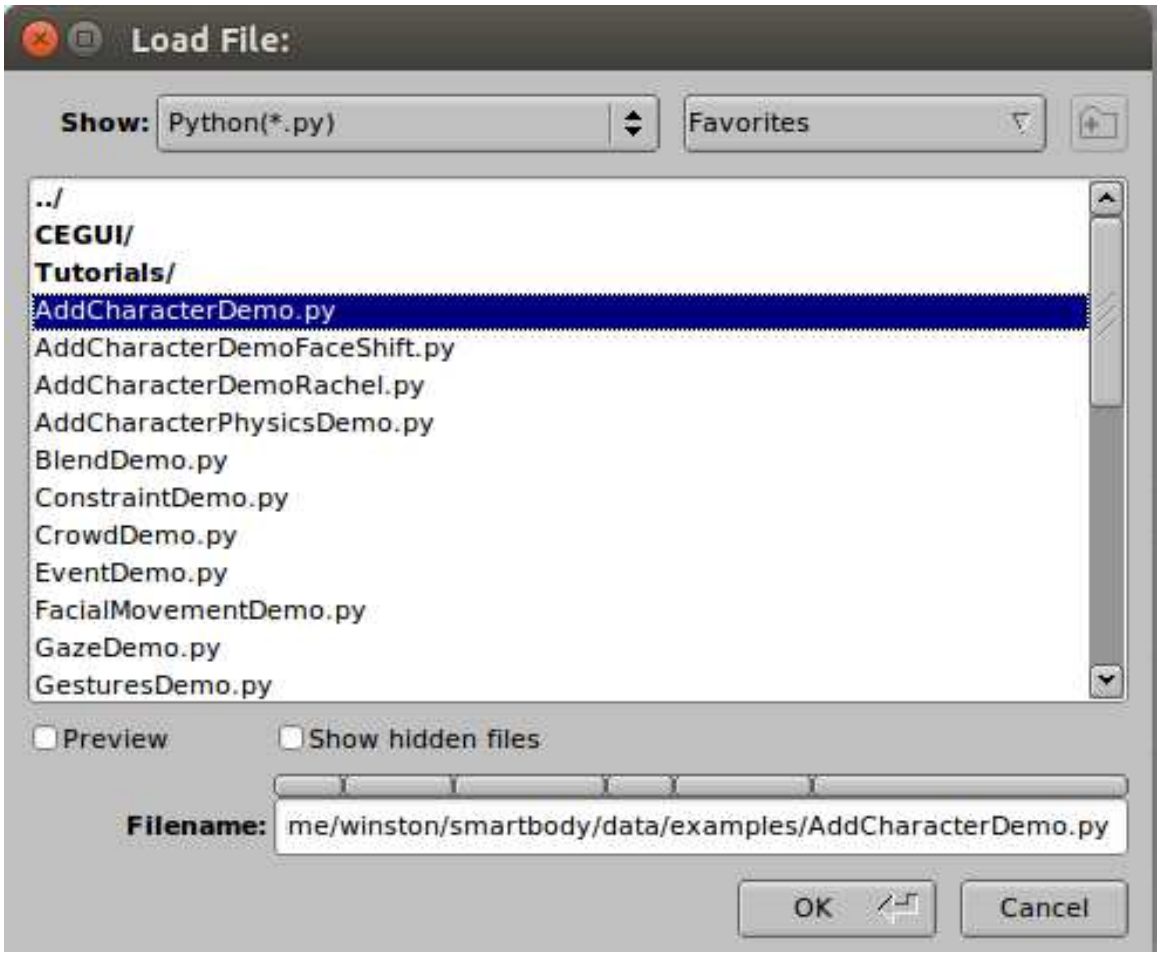

Figure 2.7: SmartBody python scripts loader

The Script Loader provides us the possibility of loading an existing Python script into our application. The Python interpreter will execute it. This tool is very useful for us to make use of larger scripts and to implement modules that add functionalities to SmartBody.

There are two ways to take advantage of this. We can use the graphic interface, depicted in Figure 2.7 or we can invoke a script executing sbgui from the terminal with the instruction -script and the location of the python script we want to execute in our filesystem.

```
/smartbody/bin/sbgui -scriptpath ../data/scripts -script SmartSim.py
```

2.5.4 BML Usage

SmartBody is a BML realization engine that transforms BML behavior descriptions into real time animations.

All the animations we have created for our characters can be triggered by BML instructions. The SmartBody Python API has some commands which make use of BML.

Using this, we can create Python scripts in order to have our agents performing actions and animations.

BML functionality is fully implemented. Even so, we can create new BML and associate them with new animations.

2.5.5 BML Creator

The BML Creator Window allows us to create BML commands easily. The interface shows the different BML commands available as tabs, as well as all the options that can be set for them. For each option, a description will appear if you mouse over the input box.

As you select the command option, an Python command will be generated that performs the same command, which can be copied and used in a script for later use. The 'Run' button will execute that BML command on the character selected in the 'Characters' dropdown, or '*' for all the characters.

For these features, we have used SmartBody to create the Graphic Interface of our evacuation plans simulator. SmartBody will be integrated with UbikSim for the first to use the powerful paths calculator of the second.

2.6 Conclusions

In this chapter we have introduced the technologies we are using in our project and why are we using them.

We have analyzed them in order to study how to connect them to our pivotal technology, the SmartBody framework. We will see in our architecture study how we have used them to create our graphical interface.

CHAPTER 3

Architecture

This chapter describes in depth how the system is structured in different modules and how the users interact with them. We will also see how the modules interact with other modules.

3.1 Introduction

The goal of our project is to develop a graphic interface for an evacuation plans simulator, the SmartSim project. In this chapter we are going to describe the architecture of the system we developed to accomplish this objective.

SmartSim is structured in two modules:

- Social simulator: In order to recreate the human behaviour, we use this tool to establish the destination of the agents and to edit the scene. We use UbikSim, described in the chapter 2.2, as our social simulator. It has been specifically built for evacuation plans, so it was perfect for our project.
- Graphic interface We will use this tool for the user to watch and control the simulation. We use SmartBody, described in the chapter 2.5. It allows us to create a graphic interface easily. It also makes an extensive use of BML, a language we wanted to make use of in our project.

A diagram of the SmartSim architecture is shown in Figure 3.1. Each module is detailed in the following sections.

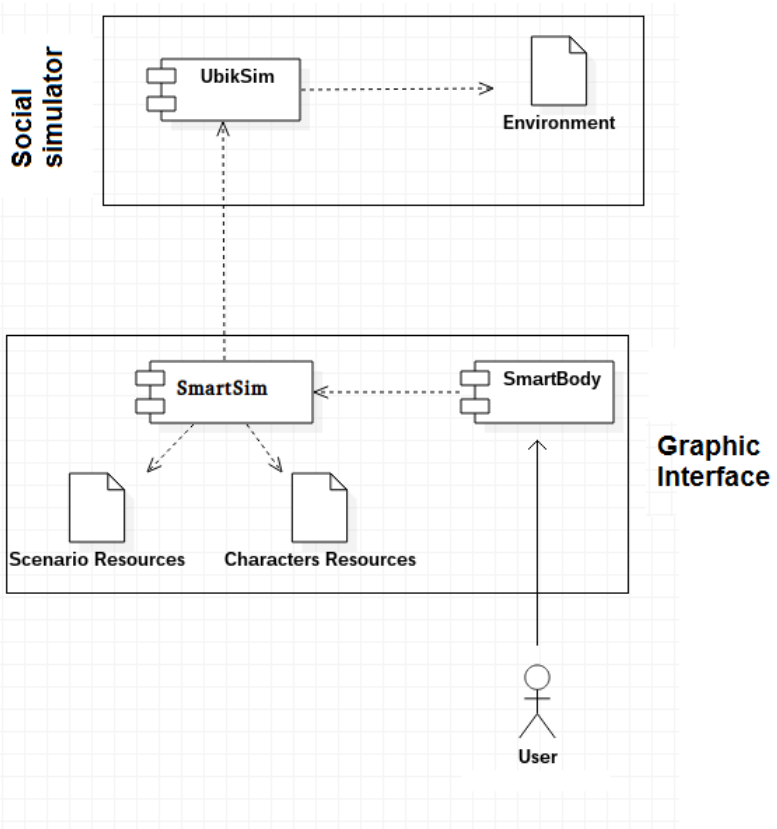

Figure 3.1: SmartSim Architecture

3.1.1 Social Simulator

We will use UbikSim as the base social simulator in our project.

As we saw in Chapter 2.2, UbikSim is a social simulator that can model human behaviours in an emergency situation. It is specially designed for modelling indoor environments and realistic human behaviours.

For these reasons, we use UbikSim as the creator of our own environments. We can set the scenario and the initial positions of our agents and add more agents and emergencies. Information about UbikSim editor have been provided in Chapter 2.2.4. How to use this tool to create the scene in SmartSim simulations will be treated in Chapter 4.2.

In Chapter 2.2.1 UbikSim is regarded as an exceptional tool for integrating AmI and MABS, specially designed for simulating evacuation plans. Because of this, the evacuation plans will be provided by UbikSim. These paths will be retrieved using HTTP, using the API described in Chapter 2.2.3.

3.1.2 Graphical User Interface

We will use SmartBody as the graphical user interface program in our project. SmartBody gives us many resources that we can use in our simulations. Nevertheless, SmartBody is not prepared to perform the simulation and calculate the paths that the agents have to follow in order to evacuate the building. For these reasons, we need to integrate it with UbikSim. UbikSim performs the pure simulation tasks and SmartBody retrieve the data from it. With this data, SmartBody creates a scene and allows the user to control it, pausing or advancing it.

The integration of SmartBody and UbikSim is provided by the SmartSim module. This module is described in Section 3.2.

3.2 SmartSim Architecture

This project has developed SmartSim, which extends SmartBody for Social Simulation, integrated with UbikSim. The architecture of SmartSim has been organised in modules as shown in Figure 3.2. This infrastructure is the core of this project.

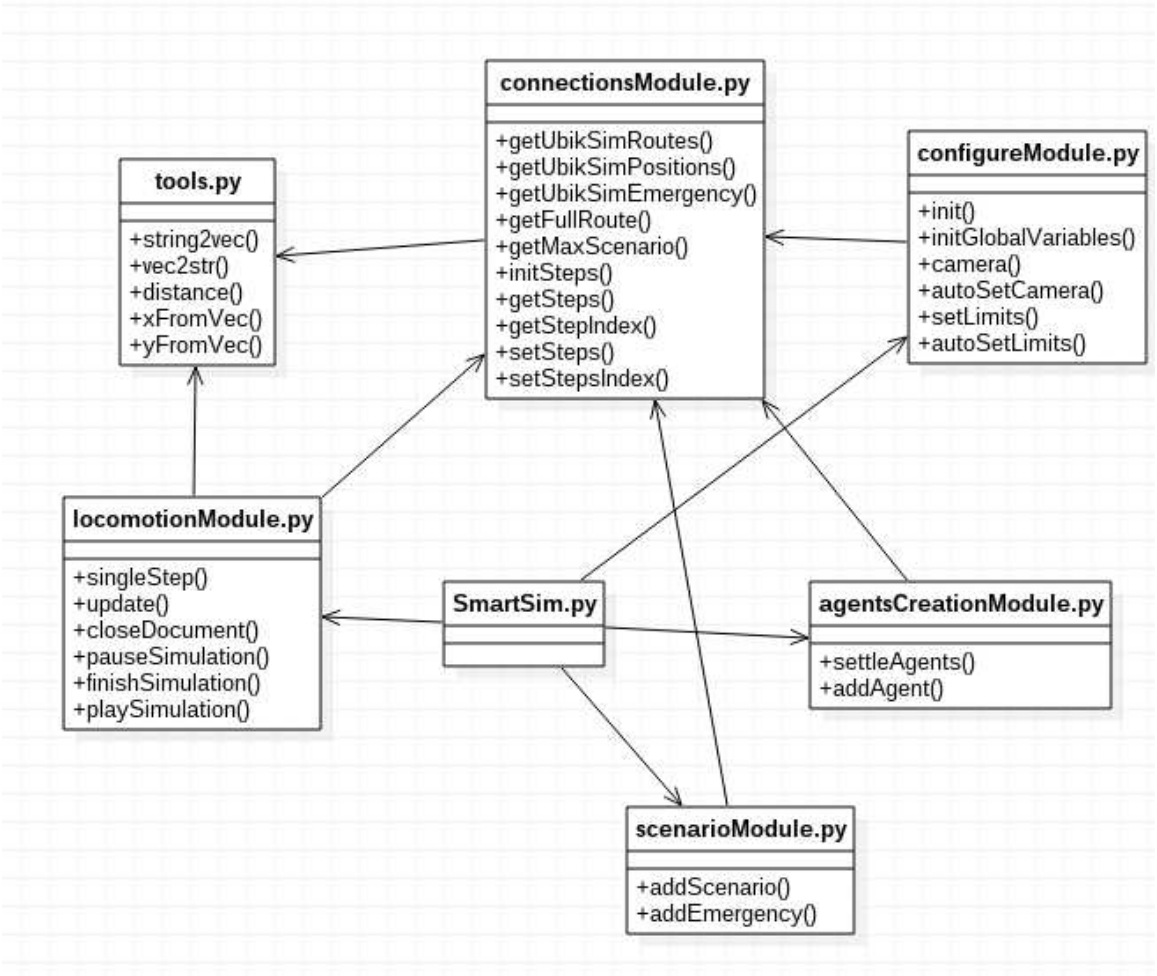

Figure 3.2: The module diagram of the Python infrastructure

3.2.0.1 Main Module - SmartSim.py

This is the main module of our Python infrastructure. We load this module from SmartBody and we control our simulation. As we will see further, we have created some case uses. The differences between those cases are the differences between their main modules. We will analyse the common functions to the main modules.

From this module we import the needed python modules.

Listing 3.1: Python modules import

```
#Importing Python modules
import io
import httplib
import time
```

```
import ConfigParser
import math
import json
```

We also load the other modules of our architecture. Loading our Python scripts in SmartBody is slightly different than importing scripts in Python. We have to use functions of the Python API of SmartBody in order to load them.

Listing 3.2: SmartSim modules import

```
#Importing the other modules
scene.run('configureModule.py')
scene.run('agentsCreationModule.py')
scene.run('tools.py')
scene.run('connectionsModule.py')
scene.run('scenarioModule.py')
scene.run('locomotionModule.py')
```

We load the configuration file. Not all the modules load the same options of this file, as we will see.

Listing 3.3: Configuration file parse

```
#Loading the configuration file
config = ConfigParser.RawConfigParser()
config.read('SmartSimSettings.cfg')

amountAgents = config.getint('Settings', 'amountAgents')
ubikSimServer = config.get('Settings', 'ubikSimServer')
meshScenario = config.get('Settings', 'meshScenario')
modeSimulation = config.get('Settings', 'modeSimulation')
```

SmartSim writes in a file the results of our simulation. In this file our program writes the time an agent spent to exit the building. Further descriptions of this file can be found in Chapter 4.7. This file is the results file of our simulation.

To perform input and output actions in this file, the file must be opened prior to any of these actions. In Listing 3.4 this action is performed.

Listing 3.4: Results file setup

```
#Opening the results file
results = io.open('SmartSimResults', 'wb')
```

SmartBody API contains a function that allows us to perform an infinite loop controlling the time and not blocking the framework. We use this procedure to run our simulation and steering the agents to the path they must follow to exit the building. To execute this module properly we have to invoke it in the main module. The needed python instructions are:

Listing 3.5: Update loop execution

```
scene.removeScript('locomotion')
locomotion = LocomotionModule()
scene.addScript('locomotion', locomotion)
```

3.2.0.2 Configuration module - configuremodule.py

This module initializes SmartBody. It makes some necessary configuration processes. There are some mandatory processes that must be performed after loading a script in SmartBody, so this module was created to cope with them. This module is also used to set the dimensions of the scenario and the default position of camera.

In SmartBody, we have to call the graphic interface of SmartBody and a default camera. Without these tasks, the graphic interface of SmartBody does not display, making for us impossible to have a graphic interface for our simulation (although it would run in a second plane). We also have to load the paths for our resources and set some resources as the skeleton of the characters.

These actions are performed in the init function, defined in Listing 3.6.

Listing 3.6: Init function

```
def init(self):

    ''' Inits SmartBody'''

    #Add asset paths
    print "Adding assets paths"
    scene.addAssetPath('mesh', 'mesh')
    scene.addAssetPath('motion', 'ChrBrad')
    scene.addAssetPath('motion', 'ChrRachel')
```

```
    scene.addAssetPath("script", "behaviorsets")
    scene.addAssetPath('script', 'scripts')
    scene.loadAssets()

    #Set scene scale
    print "Setting scene scale"
    scene.setScale(1.0)

    # Set joint map for agents
    print 'Setting up joint map for agents'
    scene.run('zebra2-map.py')
    zebra2Map = scene.getJointMapManager().getJointMap('zebra2')
    bradSkeleton = scene.getSkeleton('ChrBrad.sk')
    zebra2Map.applySkeleton(bradSkeleton)
    zebra2Map.applyMotionRecurse('ChrBrad')
    rachelSkeleton = scene.getSkeleton('ChrRachel.sk')
    zebra2Map.applySkeleton(rachelSkeleton)
    zebra2Map.applyMotionRecurse('ChrRachel')

    #Creates scene
    print "Creating scene"
    getViewer().show()
```

Apart from these crucial assignments, we also charge this module with the duty of initializing our scene in two more functions.

SmartBody scene is finite. We do not have an unlimited scenario, our agents are limited to a defined area. The default dimensions of this area are too small for the area our simulation will cover. For this, we have to change the area using a function defined in our module. The setLimits (Listing 3.7) function is used for applying the desired limits of the scene.

Listing 3.7: setLimits function

```
def setLimits(self, xLimitScene, yLimitScene):

  '''Sets limits of scene grid'''

  steerManager = scene.getSteerManager()

  steerManager.setDoubleAttribute('gridDatabaseOptions.gridSizeX',
      xLimitScene)
  steerManager.setDoubleAttribute('gridDatabaseOptions.gridSizeZ',
      yLimitScene)
```

How do we calculate these dimensions? We can calculate ourselves or letting the autoSetLimits (Listing 3.8) function to calculate them for us. This function makes use of the getMaxScenario function defined in Listing 3.19 to get the scenario borders. This interaction is detailed in Figure 3.3.

Listing 3.8: autoSetLimits function

```
def autoSetLimits(self, ubikSimServer):
  vectorLimits = ConnectionsModule().getMaxScenario(ubikSimServer)

  vectorLimits = [vectorLimits[0] + 100, vectorLimits[1] + 100]

  return vectorLimits
```

In this module we also adjust the camera position using their properties. We need to execute this function after loading our program. Without a defined camera, SmartBody would not display. The camera function, defined in Listing 3.9, sets a default camera and makes it point to a desired point.

Listing 3.9: camera function

```
def camera(self, cameraEye, cameraCenter, cameraPosition):

  ''' Configures camera'''

  #Gets camera
  getCamera().reset()
  scene.createPawn('camera')
  camera = getCamera()

  #Sets camera with provided values
  camera.setEye(cameraEye[0], cameraEye[1], cameraEye[2])
  camera.setCenter(cameraCenter[0], cameraCenter[1], cameraCenter[2])
  scene.getPawn('camera').setPosition(SrVec(cameraPosition[0],
      cameraPosition[1], cameraPosition[2]))
```

We have defined a function, the autoSetCamera function, defined in Listing 3.10 that calculates the medium point between our agents. In our simulators we have set this point as the default point for our camera to point. However, this function could be ignored.

Listing 3.10: autoSetCamera function

```
def autoSetCamera(self, amountAgents):

  xCamera = 0
  yCamera = 0

  for i in range (amountAgents):

    character = scene.getCharacter(scene.getCharacterNames()[i])

    xCamera = xCamera + character.getPosition().getData(0)
    yCamera = yCamera + character.getPosition().getData(2)

  xCamera = xCamera/amountAgents
  yCamera = yCamera/amountAgents

  centerCamera = [xCamera, 0, yCamera]
  return centerCamera
```

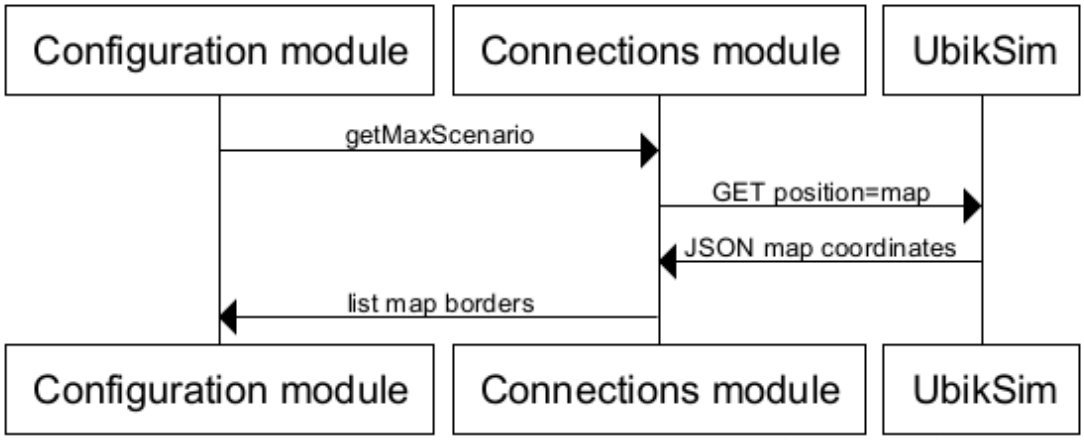

Figure 3.3: Connections module and configuration module interaction

3.2.0.3 Scenario module - scenarioModule.py

This module is created to cope with all the scenario related tasks. The scenario resources are loaded in a different way than the characters resources. This module loads the scenario resources and sets a mark for the emergency.

In the addScenario function, defined in Listing 3.11, the scenario polygonal model indicated in the configuration file is loaded and its dimensions are adjusted.

Listing 3.11: addScenario function

```
def addScenario(self, scenarioMesh):

  scene.loadAssetsFromPath("scene/" + scenarioMesh)
  scenario = scene.createPawn("scenario")
  scenario.setVec3Attribute("meshScale", .01, .01, .01)
  scenario.setStringAttribute("mesh", scenarioMesh)
```

We also get the emergency position and add the emergency mark in the addEmergency function, defined in Listing 3.12. The emergency position is given by UbikSim. As we saw in Table 2.1, UbikSim API allows us to make this request. We make this request using the getUbikSimEmergency function, defined in Listing 3.17. This interaction is detailed in Figure 3.4.

Listing 3.12: addEmergency function

```
def addEmergency(self, ubikSimServer):

  emergencyPosition = ConnectionsModule().getUbikSimEmergency(ubikSimServer
      )

  emergency = scene.createPawn("emergency")
  emergency.setStringAttribute('collisionShape','box')
  emergency.setVec3Attribute('collisionShapeScale',1.0,1.0,1.0)
  emergency.setPosition(emergencyPosition)
```

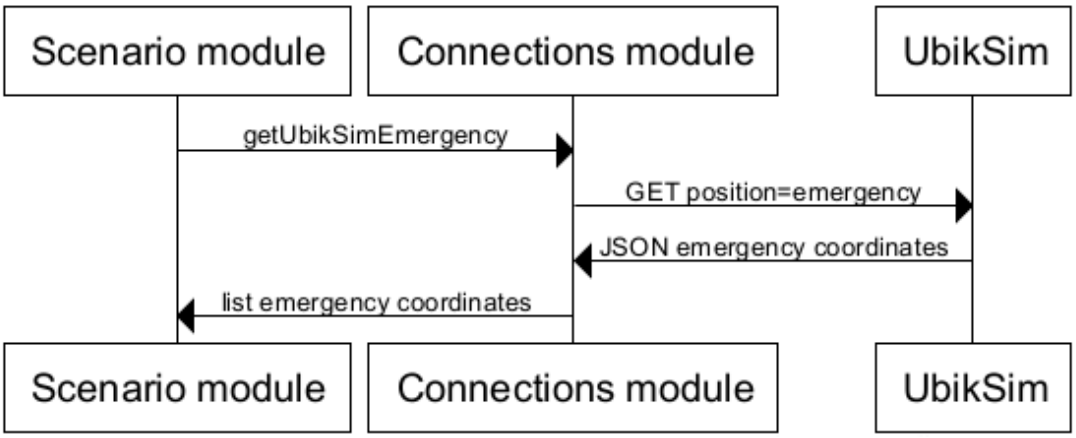

Figure 3.4: Connections module and Scenario module interaction

3.2.0.4 Agents creation module - agentsCreationModule.py

This module creates the agents and sets their positions with the data retrieved from UbikSim. We need this module to fill the simulation with agents. This module can be used both for getting agents from UbikSim and creating agents in a desired position.

The settleAgents function, defined in Listing 3.13, we interact with UbikSim to get the agents position. The interaction is described in Figure 3.5. In this function we call both the getUbikSimPositions (Listing 3.16) function, to get the agents positions, and addAgent function (Listing 3.14), to create the agents with the provided positions.

Listing 3.13: settleAgents function

```
def settleAgents(self, amountAgents, ubikSimServer):

    '''Adds the desired amount of agents to SmartSim'''

    #Getting list of initial positions from UbikSim
    listData = ConnectionsModule().getUbikSimPositions(ubikSimServer)

    #Creating the desired amount of agents
    for i in range(amountAgents):
      agentName = 'Agent%s' % i
      characterKind = 'default'
      posX = listData[i][1]["positionX"]
      posZ = listData[i][1]["positionY"]
      agentPosition = SrVec(posX, 0, posZ)
```

```
      for typeCharacter in typesCharacters:
        if (i in typeCharacterDict[typeCharacter]):
          characterKind = str(typeCharacter)
      self.addAgent(agentName, agentPosition, characterKind)

    # Set up list of Brads
    agentsList = []
    for name in scene.getCharacterNames():
      if 'Agent' in name:
        agentsList.append(scene.getCharacter(name))
```

The agents are created in the addAgent function (Listing 3.14).

The character is assigned a skeleton and a polygonal model. Two key elements for the character are provided: the behaviour set and the steer manager. Different models and behaviour sets are provided depending on the agent type. Agents types are further explained in Section 4.8.6.

Listing 3.14: addAgent function

```
def addAgent (self, agentName, agentPosition, *characterType):

    '''Adds an agent to SmartSim'''

    #Creation of the agent
    agent = scene.createCharacter(agentName, '')

    for character in characterType:
      characterType = character
    characterType = str(characterType)

    #Creation of the agent structure
    if (characterType == 'rachel'):
      print 'Hola'
      print characterType
      agentSkeleton = scene.createSkeleton('ChrRachel.sk')
    else:
      print 'Que te den'
      print characterType
      agentSkeleton = scene.createSkeleton('ChrBrad.sk')

    agent.setSkeleton(agentSkeleton)

    #Setting up agent position
```

```
agent.setPosition(agentPosition)

#Creation of standard controllers
agent.createStandardControllers()

#Adding deformable mesh to agent
agent.setVec3Attribute('deformableMeshScale', .01, .01, .01)
if (characterType == 'rachel'):
  agent.setStringAttribute('deformableMesh', 'ChrRachel.dae')
elif (characterType == 'maarten'):
  agent.setStringAttribute('deformableMesh', 'ChrMaarten.dae')
else:
  agent.setStringAttribute('deformableMesh', 'ChrBrad.dae')

if (characterType == 'maarten'):
  scene.run('BehaviorSetMaleMocapLocomotion.py')
  setupBehaviorSet()
  retargetBehaviorSet(agentName)
  scene.run('BehaviorSetGestures.py')
  setupBehaviorSet()
  retargetBehaviorSet(agentName)
else:
  scene.run('BehaviorSetMaleLocomotion.py')
  setupBehaviorSet()
  retargetBehaviorSet(agentName)
  scene.run('BehaviorSetGestures.py')
  setupBehaviorSet()
  retargetBehaviorSet(agentName)

#Adding Steer manager to agent
steerManager = scene.getSteerManager()
steerAgent = steerManager.createSteerAgent(agentName)
steerAgent.setSteerStateNamePrefix("all")
steerAgent.setSteerType("example")
agent.setBoolAttribute('steering.pathFollowingMode', False)

#Setting initial body posture
bml.execBML(agentName, '<body posture="ChrBrad@Idle01"/>')

scene.getCharacter(agentName).setStringAttribute("displayType", "mesh")
```

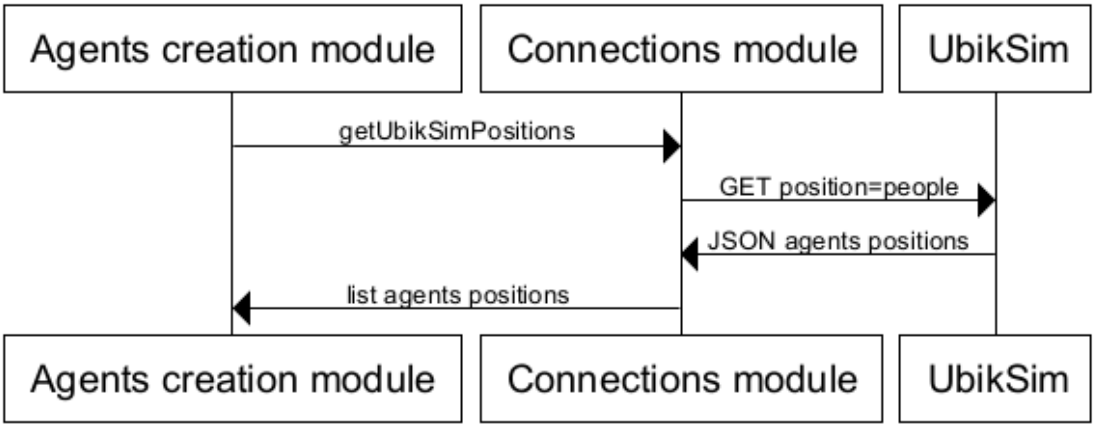

Figure 3.5: Connections module and Agents creation interaction

3.2.0.5 Connections Module - connectionsModule.py

This module makes the necessary connections between our version of UbikSim and SmartBody. As we have seen in Sections 3.2.0.3 and 3.2.0.4 and we will see in Section 3.2.0.7, we need to connect to UbikSim operating SmartSim. This module has been created to handle these connections.

The getUbikSimRoutes function, defined in Listing 3.15, is used to get the agents paths from UbikSim. Agent routes should be converted between UbikSim and Smartbody. This conversion is done by calling the functions xFromVec and yFromVec, defined in Listing 3.22.

Listing 3.15: getUbikSimRoutes function

```
def getUbikSimRoutes(self, ubikSimServer):

  '''Gets agents routes from UbikSim'''

  conn = httplib.HTTPConnection(ubikSimServer)
  conn.request('GET', '/UbikSimMOSI-AGIL-Server/ubiksim?position=goals')
  data = conn.getresponse()
  data = data.read()
  jsondata = json.loads(data)
  listdata = jsondata.items()

  step = []
  agentRoute = []
  routes = []
```

```
  for i in range (len(listdata)):
    for j in range (len(listdata[i][1]['goalPath'])):
      xStep = Tools().xFromVec(listdata[i][1]['goalPath'][j]) * 0.3
      yStep = Tools().yFromVec(listdata[i][1]['goalPath'][j]) * 0.3
      step.append(xStep)
      step.append(yStep)
      agentRoute.append(step)
      step = []
    routes.append(agentRoute)
    agentRoute = []

  return routes
```

The getUbikSimPositions function (Listing 3.16) is used to retrieve the initial positions of agents from UbikSim. This function is needed to get the positions to fill the agents as in settleAgents function defined in Listing 3.13.

Listing 3.16: getUbikSimPositions function

```
def getUbikSimPositions(self, ubikSimServer):

  '''Gets agents initial positions from UbikSim'''

  conn = httplib.HTTPConnection(ubikSimServer)
  conn.request('GET', '/UbikSimMOSI-AGIL-Server/ubiksim?position=people')
  data = conn.getresponse()
  data = data.read()
  jsondata = json.loads(data)
  listdata = jsondata.items()

  for i in range (len(listdata)):
    listdata[i][1]["positionX"] = listdata[i][1]["positionX"] * 0.3
    listdata[i][1]["positionY"] = listdata[i][1]["positionY"] * 0.3

  return listdata
```

The getUbikSimEmergency function, defined in Listing 3.17 we connect to UbikSim and return the emergency position. The xFromVec and yFromVec functions (Listing 3.22) are used in this one.

Listing 3.17: getUbikSimEmergency function

```
def getUbikSimEmergency(self, ubikSimServer):
```

```
  conn = httplib.HTTPConnection(ubikSimServer)
  conn.request('GET', '/UbikSimMOSI-AGIL-Server/ubiksim?position=emergency'
      )
  data = conn.getresponse()
  data = data.read()

  emergencyUbikSim = (data[data.index('('):data.index(')')+1])

  xVector = Tools().xFromVec(emergencyUbikSim) * 0.3
  yVector = Tools().yFromVec(emergencyUbikSim) * 0.3

  vectorEmergency = SrVec(xVector, 0, yVector)

  return vectorEmergency
```

The getFullRoute function (Listing 3.18) gets a string with the complete path for an agent.

Listing 3.18: getFullRoute function

```
def getFullRoute(self, ubikSimServer, agentIndex):

  agentsSteps = ConnectionsModule().getUbikSimRoutes(ubikSimServer)

  fullRoute = ''

  for i in range (len(agentsSteps[agentIndex])):

    xRoute = str(agentsSteps[agentIndex][i][0])
    yRoute = str(agentsSteps[agentIndex][i][1])

    fullRoute = fullRoute + xRoute + ' ' + yRoute + ' '

  return fullRoute
```

The getMaxScenario function (Listing 3.19) gets the maximum dimensions of the map. It is used in the limits calculation of the scene.

Listing 3.19: getMaxScenario function

```
def getMaxScenario(self, ubikSimServer):
```

```
conn = httplib.HTTPConnection(ubikSimServer)
conn.request('GET', '/UbikSimMOSI-AGIL-Server/ubiksim?position=map')
data = conn.getresponse()
data = data.read()
jsondata = json.loads(data)
listdata = jsondata.items()

maxX = listdata[0][1] * 0.3
maxY = listdata[2][1] * 0.3

vectorMax = [maxX, maxY]

return vectorMax
```

3.2.0.6 Tools Module - toolsModule.py

This module allows us to make some calculations and conversions between the UbikSim format and SmartBody standards.

The distance function, defined in Listing 3.20, calculates the distance between two points. The distance between our agents position and the next position they have to move into should be calculated. Using this method we control that our agents are given a new position when they have arrived to the last one. We check this in the singleStep function defined in Listing 3.23.

Listing 3.20: distance function

```
def distance (self, p1, p2):

  '''Calculates the distance between two points'''

  x = (p1.getData(0) - p2.getData(0))
  x = x*x
  y = (p1.getData(2) - p2.getData(2))
  y = y*y

  return math.sqrt(x+y)
```

We provide two methods we will need to get data from vectors in SmartBody and in UbikSim standards. This function is defined in the Listing 3.21.

Listing 3.21: string2vec and vec2str functions

```
  def string2vec (self, stvec):

    '''Converts String to SrVec'''

    posX = int (stvec[stvec.index('(')+1:stvec.index(',')])
    posZ = int (stvec[stvec.index(',')+1:stvec.index(')')])

    return SrVec(posX, 0, posZ)

  def vec2str(self, vec):

    ''' Converts SrVec to string '''

    x = vec.getData(0)
    y = vec.getData(2)
    z = vec.getData(2)
    if -0.0001 < x < 0.0001: x = 0
    if -0.0001 < y < 0.0001: y = 0
    if -0.0001 < z < 0.0001: z = 0

    return "" + str(x) + " " + str(y) + ""
```

We also need to retrieve the coordinates from the UbikSim positions. We use these two functions defined in Listing 3.22.

Listing 3.22: xFromVec and yFromVec functions

```
def xFromVec (self, vec):

  return int (vec[vec.index('(')+1:vec.index(',')])

def yFromVec (self, vec):

  return int (vec[vec.index(',')+1:vec.index(')')])
```

3.2.0.7 Locomotion Module - locomotionModule.py

This module is used to control our simulation and to make our agents follow their path. We control every step of our simulation and makes it advance, pause or stop it. We also define

how we save the results of our simulation, writing them in a document.

We have defined how to manage every step of our simulation in a function. This function is called every time the loop we described in Section 3.2.0.1 is executed (if the simulation is not stopped or paused).

We have implemented two modes of simulation. The difference between them is how we send the path to our agents. In Section 4.5, the differences are further explained.

With every loop we check if our character is near of his final position. If he has reached it, we write his statistics in the results file.

In this function we use the getUbikSimRoutes we describe in Listing 3.15. We also use the distance function we defined in Listing 3.20.

In Figure 3.6 we describe the interaction between the connections module and the locomotion module.

Listing 3.23: singleStep function

```
def singleStep(self):

  '''Updates the simulation'''

  listdata = ConnectionsModule().getUbikSimRoutes(ubikSimServer)

  #Charges the status of every agent in the simulation
  agentStep = ConnectionsModule().getSteps()

  for i in range (amountAgents):
    #Gets one character of the simulation
    agent = scene.getCharacter('Agent%s' % i)
    positionVec = agent.getPosition()
    finalPosition = SrVec(listdata[i][len(listdata[i])-1][0], 0, listdata
        [i][len(listdata[i])-1][1])

    if (modeSimulation == 'gettingSteps'):

      if (behaviors and firstStep[i]):
        bml.execBML(agent.getName(), '<body posture="ChrBrad@126_fear"/>'
            )
        firstStep[i]

      if (agentStep[i]+1 < len(listdata[i])):
```

```
        #Checks if the character has reached the position =
        reachPosition = SrVec(listdata[i][agentStep[i]][0], 0, listdata[i
            ][agentStep[i]][1])

        if (Tools().distance (positionVec, reachPosition) <=1.8):

          #Directs the agent to the next position
          nextPosition = SrVec(listdata[i][agentStep[i]+1][0], 0,
              listdata[i][agentStep[i]+1][1])

          if (behaviors and Tools().distance(positionVec,
              positionEmergency)):
            bml.execBML(agent.getName(), '<locomotion speed="10" target="
                ' + Tools().vec2str(nextPosition) + '"/>')
            bml.execBML(agent.getName(), '<body posture="
                ChrBrad@Idle01_BeatHighBt01"/>')
            #bml.execBML(agent.getName(), 'gritar Help')
          else:
            bml.execBML(agent.getName(), '<locomotion speed="20" target="
                ' + Tools().vec2str(nextPosition) + '"/>')
          agentStep[i] = agentStep[i]+1

    elif (modeSimulation == 'bmlRoute'):

      if (firstStep[i]):

        route = ConnectionsModule().getFullRoute(ubikSimServer, i)
        agent = scene.getCharacter('Agent%s' % i)

        agent.setBoolAttribute("steering.pathFollowingMode", True)

        bml.execBML(agent.getName(), '<locomotion target="'+route+'"/>')

        firstStep[i] = False

        if amountLeaders is not None:
          leaderName = ('Agent%s' % i)
          for j in range (amountFollowers):
            follower = scene.getCharacter(leaderName + '%s' % j)

            follower.setBoolAttribute("steering.pathFollowingMode", True)

            bml.execBML(follower.getName(), '<locomotion target="'+route+
                '"/>')
```

```
      if (Tools().distance (positionVec, finalPosition) <=1.5 and inside[i
          ]):

        #if (behaviors):
          #BML aliviarse

        timeExit = time.time() - timeStart[0]

        if (len(timeStart) > 1):

          timePaused = 0

          for i in range (len(timeStart) - 1):

            timePaused = timePaused + (timePause[i] - timeStart[i])

            timeExit = timeExit - timePaused

        print ('Agent%s has reached his final position' % i)
        print ('Agent%s has left the building' % i)
        results.write('Agent%s succesfully exited the building\n' % i)
        results.write('Agent exited the building in %s seconds\n' %
            timeExit)
        if (behaviors):
          bml.execBML(agent.getName(), '<body posture="ChrBrad@112_happy"/>
              ')

        inside[i] = False

    #Sets the step of the route in which the agent is
    ConnectionsModule().setSteps(agentStep)
```

We have implemented in this module the controls of the simulation. We can play, pause or close the document for it to be generated. These functions are defined in Listing 3.24.

Listing 3.24: Controls functions

```
def closeDocument(self):
  results.close()

def pauseSimulation(self):
  timeStop.append(time.time())
  simulationStarted = False
```

```
def finishSimulation(self):
  LocomotionModule().pauseSimulation()
  LocomotionModule().closeDocument()

def playSimulation(self):
  timeStart.append(time.time())
  global simulationStarted
  simulationStarted = True
```

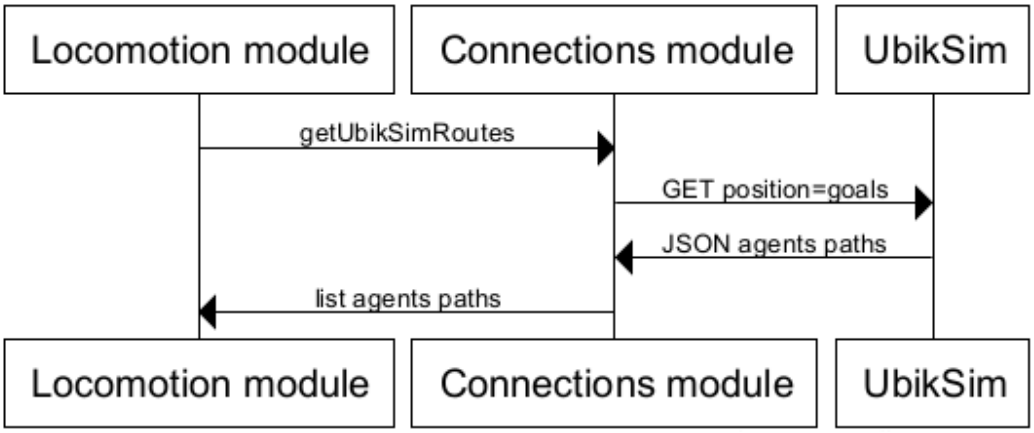

Figure 3.6: Connections module and Locomotion module interaction

3.3 Conclusions

We have seen the architecture of our project. As we see, the entire project has been implemented with a high modularity, making possible interchanging components with little effort and changing any of them without affecting all the system.

UbikSim allows us creating a scene easily and SmartBody make possible for us to implement a very complete and detailed graphic interface. Integrating and extending these two tools allowed us to create SmartSim.

The Python interface can be used to implement a high variety of projects, using parts of it or full. In this project we have made a demonstration of this implementing various use cases, changing only the main module of the infrastructure.

CHAPTER 4

Prototype and example usage

In this chapter we are going to describe the developed use cases. We will explain the running of all the tools involved and its purpose. We will analyze the outcome of each case and how we developed it.

4.1 Problem and scenario

As we explained in previous chapters of this document, we are facing the task of creating a social simulator that allows us to simulate evacuation plans. The goal of this project should be to provide the user the easiest and painless way to adapt his own case and run the simulations in it.

To test our infrastructure we have used a model based on the B building of the ETSI Telecomunicación of the Universidad Politécnica de Madrid.

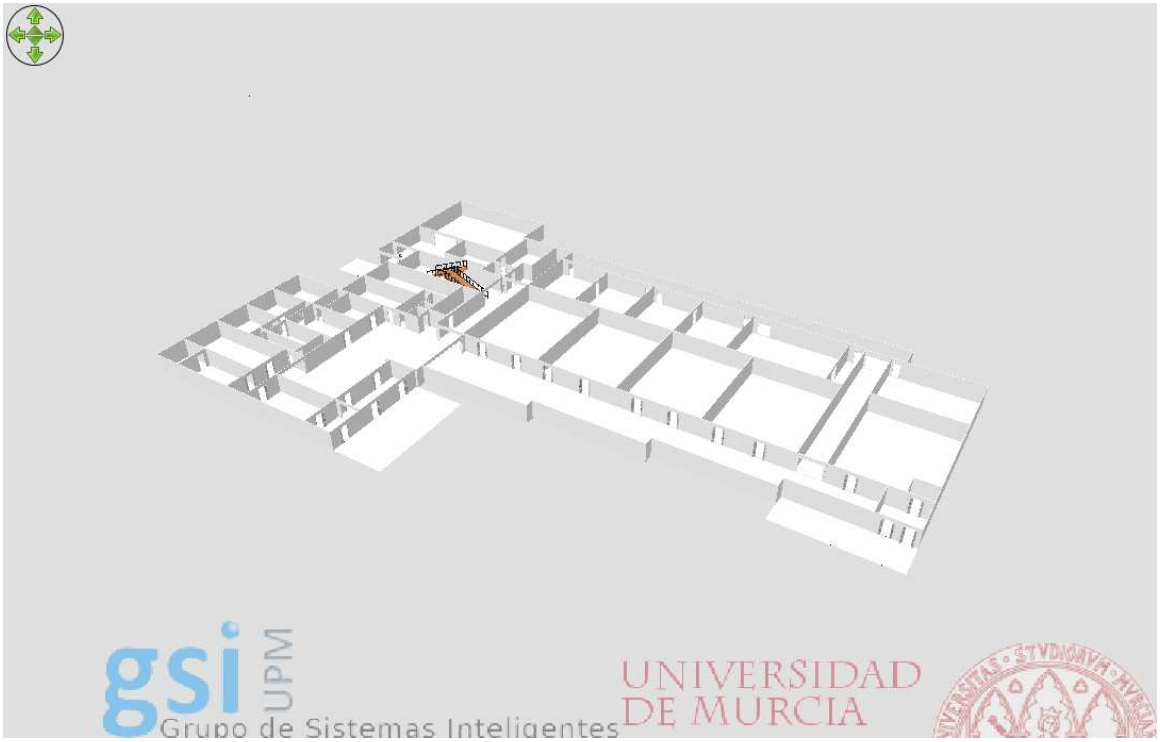

Figure 4.1: The model used for our case study.

4.2 Map creation

We can use any .obj file and export it to SmartBody to use it as the scenario in our simulation. Any polygonal model generated with 3D modelling programs such as Blender could fit. However, UbikSim provides us with its own editor, which eases so much the map creation task.

UbikSim editor is based in Sweet Home 3D. Sweet Home 3D is a free interior design application. We can draw the plan of our scenario, arrange furniture on it and visit the

results in 3D. It is so easy to create a scenario as drawing the walls and rooms over an imported plan of it. Several objects libraries of objects has been released and they can be imported to the editor, which can add completion and detail to our scenario, enhancing the simulator experience.

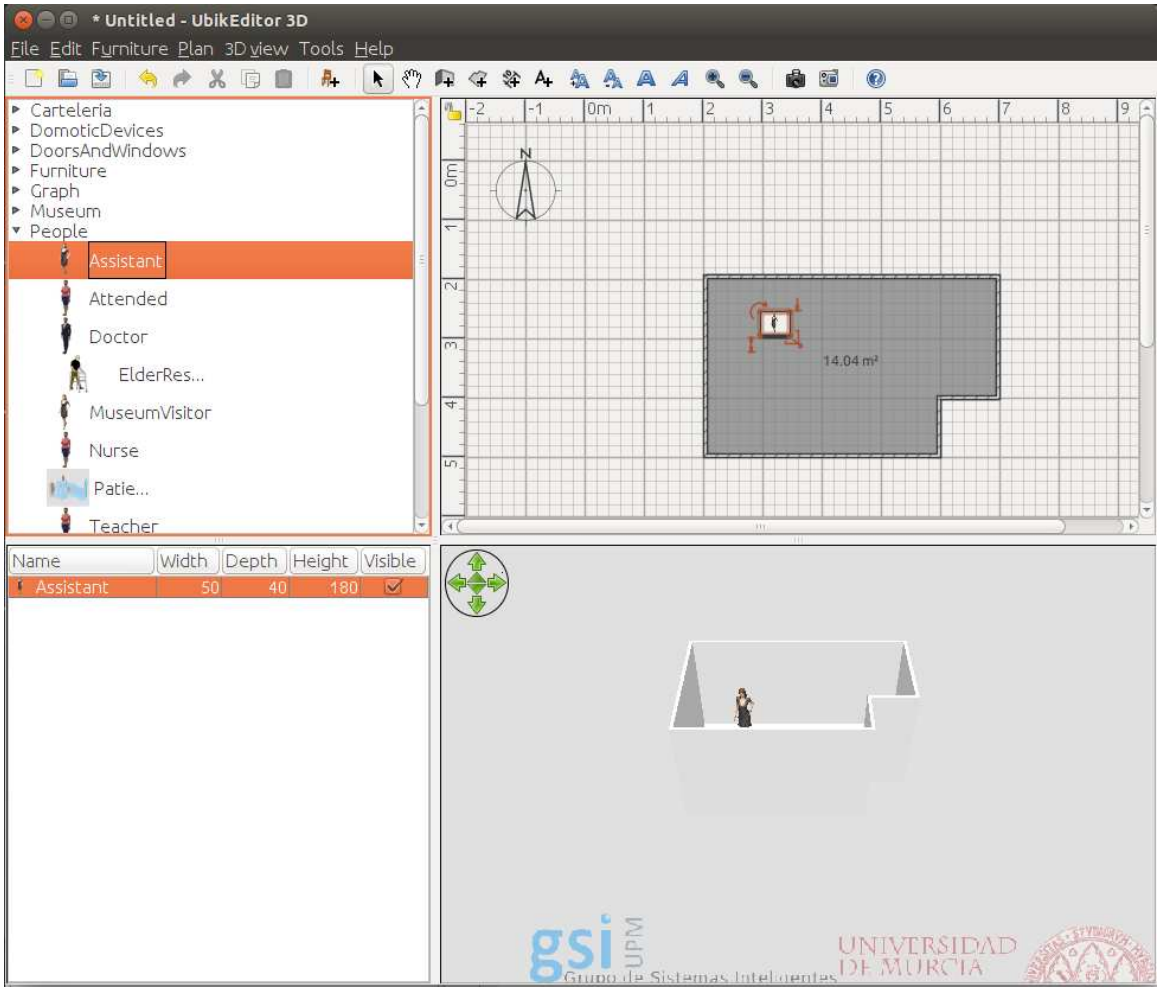

Figure 4.2: UbikSim editor interface

We can start UbikSim editor using the Integrated Development Environment (IDE) in where we load UbikSim project. To start it we have to run the StartUbikEditor.java class. The editor will start and we can use its interface to draw our scene. We can even import a plan and draw over it, easing the task.

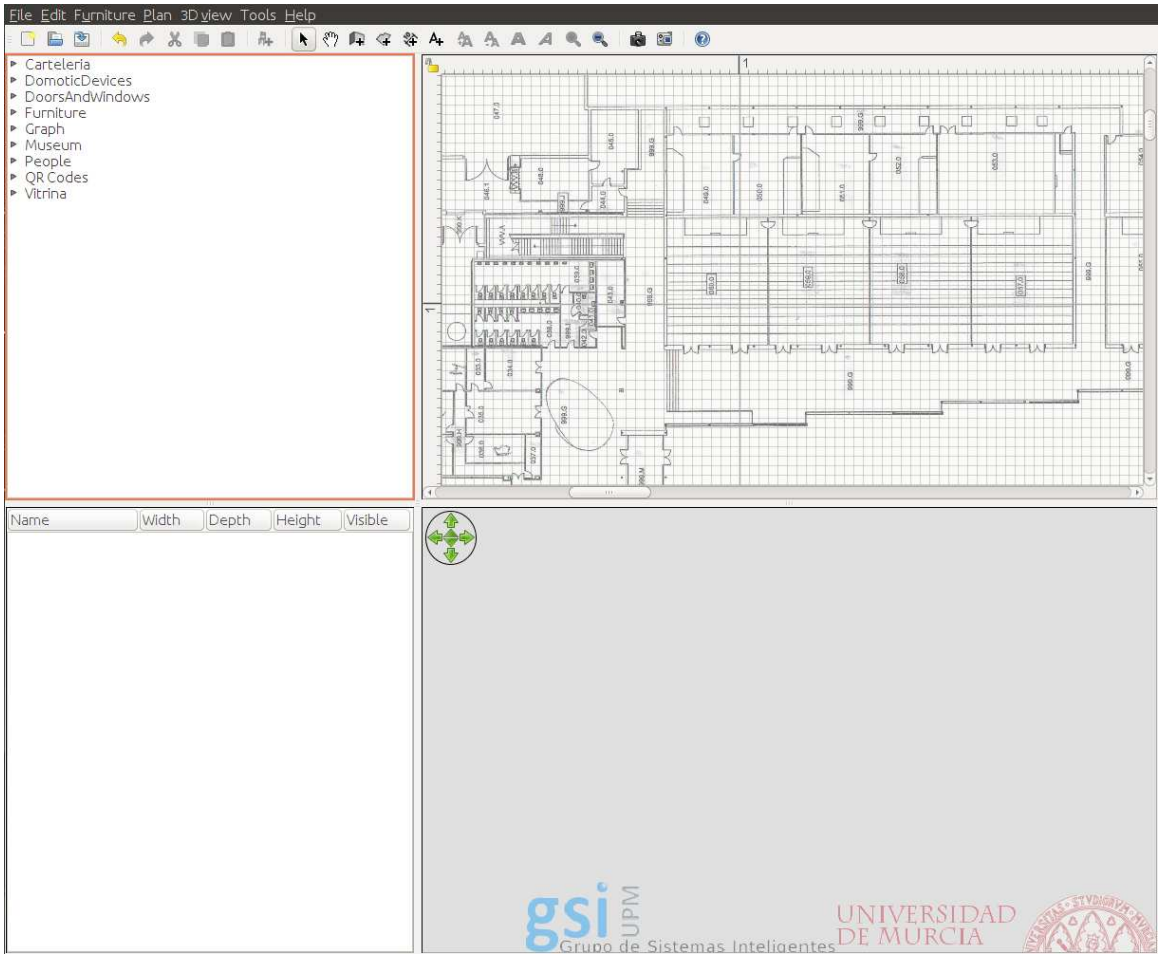

Figure 4.3: UbikSim editor with imported plan

When we have finished editing the scenario we can export it to the environments folder in UbikSim. UbikSim uses its own extension for the environments. UbikSim editor exports the files to this extension.

We have to export now the scenario to SmartBody. We can do it in the editor.

When we draw our scene, the editor creates a 3D model of it. We can export this model to an .obj file. This file can be used in SmartBody as our scenario. This procedure allows us to use exactly the same scenario in both SmartBody and UbikSim.

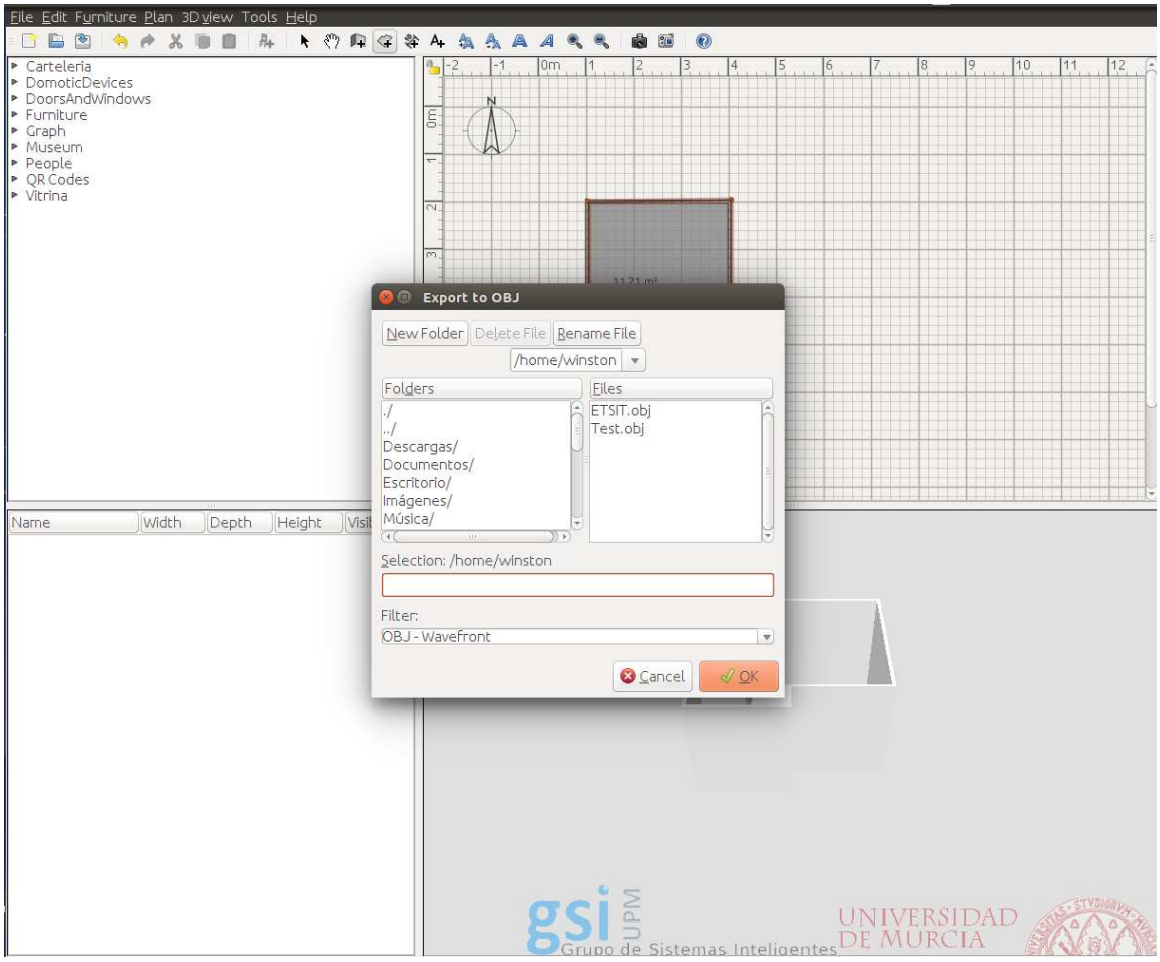

Figure 4.4: Exporting an .obj file

4.3 Setting the simulation

To start SmartSim we have to run the webapp UbikSim version. SmartBody retrieves the paths and positions from it, so it has to be started and in the step of simulation from which we want to run our graphic interface.

We have to configure our scene in the UbikSim editor. There we will set the positions of our agents, the amount of them, our scenario and the position of the emergency. All these elements will be exported to SmartBody using http requests.

As we saw in the architecture definition, SmartSim is configured using a configuration file. We can set up some important aspects in it. An example of this file and its sctructure is:

Listing 4.1: Configuration file example

```
[Settings]
amountAgents= 10
ubikSimServer= localhost:8080
meshScenario= ETSIT.obj
modeSimulation= gettingSteps
```

In Table 4.1 we find a descriptions of the options we can configure in SmartSim using this file:

Option	Description
amountAgents	The number of agents in our simulation
amountLeaders	The number of leaders in our simulation
ubikSimServer	The URL of the server where UbikSim is hosted
meshScenario	The name of the file which the scenario of our simulation is modelled after. The file must be located in the path smartbody/data/scene
modeSimulation	The two possible simulation modes we have are 'gettingSteps' and 'bmlRoute'
Types	Sets which agents will not be of the standard type and the type of them. This is explained in Chapter 4.8.6.2

Table 4.1: Configuration options

4.4 Running our simulation

There are two ways of running SmartSim scripts in SmartBody. We can run sbgui and load the main module script of our simulation or run the main module script directly using execution arguments of sbgui.

After loading the main module, sbgui will then start and load our scene. It will display the agents, the emergency and the scenario.

In Figure 4.5 we can see our scene loaded and ready for the simulation.

When we load our scene we can control the progress of it. As we saw in Chapter 3.2.0.7, we have play and stop controls defined in our code.

We can also add agents to the scene. SmartBody provides us with controls to control them. We can select a character and move it to a point using the secondary button, as in many video games.

In 4.6 we can see a character created by the user in SmartSim

Figure 4.5: The model loaded in SmartBody GUI.

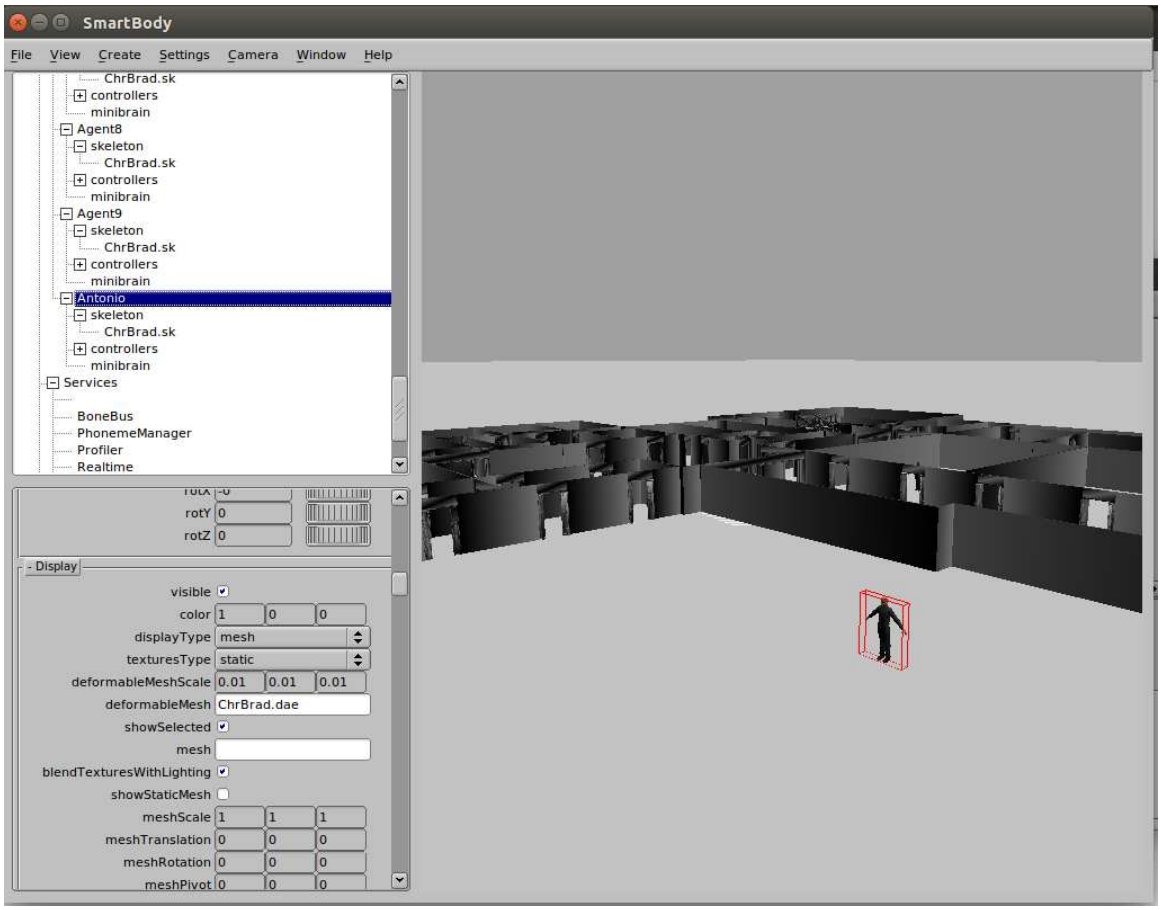

Figure 4.6: A character created by the user in SmartSim

4.5 Simulation Modes

In SmartSim, we can run the simulation in two different simulation modes.

The differences are in how the paths are assigned to the agents.

- 'gettingSteps' : The program checks periodically the agent position. If the agent is near to the next step in the path, he is directed to the next one.
- 'bmlRoute' : The complete path is given to the agent in a BML command. The agent follows it until he reaches the final step.

Each mode has its own advantages. In some simulations, the user can set in which mode he wants the simulation to be executed. Nevertheless, in certain simulations only one of the modes is available.

The unavailabilities of one the simulation modes are motivated by how the simulation cases are designed. In some cases the two modes cannot be applied.

We will see in which cases we can use both or only one of them in Section 4.8.

4.6 Controlling the scene

SmartSim allows us to control the camera. The camera points to the center position when we load the scene. We can rotate the camera around this point and zoom onto it.

Action	**Effect**
ALT + Right Click	Controls the zoom of the camera.
ALT + Middle Click	Controls the point which the camera points to.
ALT + Left Click	Rotates the camera towards the point it is pointing.

Table 4.2: Camera options

We can also change the point which the camera points to and its position using some python commands.

Listing 4.2: Camera control commands

```
camera.setEye(X,Y,Z) #Location of the camera
camera.setCenter(X,Y,Z) #Target at which the camera is pointed
scene.getPawn(`camera').setPosition(X,Y,Z) #The camera object position
```

4.7 Results

We can retrieve some interesting statistics from SmartSim. As we saw in the architecture chapter, SmartSim generates a results file. In this file we find the time that our agents spent exiting the building.

Listing 4.3: Results file example

```
Agent8 succesfully exited the building
Agent exited the building in 50.7330379486 seconds
Agent0 succesfully exited the building
Agent exited the building in 53.5379369259 seconds
Agent6 succesfully exited the building
Agent exited the building in 72.3267347813 seconds
Agent9 succesfully exited the building
Agent exited the building in 79.9186098576 seconds
```

The file is not autogenerated. Python makes possible to auto generate a file without closing it, but, due to SmartBody limitations, it has to be closed for SmartSim to generate it. If we close it during a simulation, it cannot be opened anymore, so it should be closed when all the data we want to retrieve has been generated.

4.8 Use cases

4.8.1 Introduction

We have developed some use cases to successively improving our system in order to achieve the objectives and to take advantage of the SmartBody and BML capacities. We developed a series of simulators increasing their complexity to achieve the full understanding of our base architecture. We have developed these for the user to understand our system and how to develop his own simulator using his own case effortlessly.

4.8.2 Agent Escaping

4.8.2.1 Overview

Our first case is based in the simplest possible case. We have only one agent in our building. He will escape of the building following the path given by UbikSim. We display the emergency and the character escaping the building.

This is the basic case which we will use to develop our system. We will explain how we build it and how to get the results of the simulation.

4.8.2.2 Setting the simulation

We use the standard SmartSim configuration file. We will use almost all the options except the amountAgents option, as it will be set to one by default.

The characters types are disabled also, as the only character type available is the default character.

Our scene must be in the path we declare in the configuration file and also in the environments path of UbikSim.

An example of how to configure our simulation would be:

Listing 4.4: One agent case configuration example

```
[Settings]
ubikSimServer= localhost:8080
meshScenario= ETSIT.obj
modeSimulation= gettingSteps
```

4.8.2.3 Main module

Our main module is the main difference between our cases, as the other modules are not changed. Changing the main module we change the interactions between our modules, getting the desired outcome for our simulation.

Listing 4.5: One agent case main module

```
agentStep = []
```

```
inside = []
firstStep = []
timeStart = []
timeStop = []

simulationStarted = False

config = ConfigParser.RawConfigParser()
config.read('SmartSimSettings.cfg')

amountAgents = 1
ubikSimServer = config.get('Settings', 'ubikSimServer')
meshScenario = config.get('Settings', 'meshScenario')
modeSimulation = config.get('Settings', 'modeSimulation')

print amountAgents
print ubikSimServer
print meshScenario
print modeSimulation

print "|-----------------------------------------|"
print "|            Starting SmartSim            |"
print "|-----------------------------------------|"
print ""

scene.addAssetPath('script', 'scripts')
scene.loadAssets()

print "Initiating Scene"
ConfigureModule().init()

print "Initiating global variables"
ConfigureModule().initGlobalVariables(amountAgents)

print "Setting scene limits"
vectorLimits = ConfigureModule().autoSetLimits(ubikSimServer)
xLimitScene = vectorLimits[0]
yLimitScene = vectorLimits[1]
ConfigureModule().setLimits(xLimitScene, yLimitScene)

print "Setting scenario"
ScenarioModule().addScenario(meshScenario)
ScenarioModule().addEmergency(ubikSimServer)

print "Creating agents"
```

```
AgentsCreationModule().settleAgents(amountAgents, ubikSimServer)

print "Configuring camera settings"
cameraCenter=ConfigureModule().autoSetCamera(amountAgents)
cameraEye = [cameraCenter[0]+5, 5, cameraCenter[2]+2]
cameraPosition = cameraEye
ConfigureModule().camera(cameraEye, cameraCenter, cameraPosition)

print "Getting routes"
ConnectionsModule().initSteps(amountAgents)

steerManager = scene.getSteerManager()
steerManager.setEnable(False)
steerManager.setEnable(True)

print "Scene settled"

scene.removeScript('locomotion')
locomotion = LocomotionModule()
scene.addScript('locomotion', locomotion)

def play():
  LocomotionModule().playSimulation()

def stop():
  LocomotionModule().finishSimulation()
```

We set the amountAgents option to 1 by default. We load the other options and run the simulation.

SmartBody is set up. We set the limits of the scenario, load the scenario and create the agents. The camera is set up according with the agents positions. We get the paths of the agents and run the update function of the locomotion module.

We have defined some functions to simplify the user interface. With executing play() or stop() in the Commands Window we are executing functions with longer names.

4.8.2.4 Simulation

We run SmartBody and load our script SmartSimOneCharacter.py. We can play the simulation using only the play() function. Our agent escapes from his initial position until one exit. He remains in the safe position. When we get until this point, we can stop the simulation. Doing this we can retrieve the results of it. In the results document we can get

the time that our agent has spent exiting the building.

Listing 4.6: One agent case results file example

```
Agent0 succesfully exited the building
Agent exited the building in 28.1872520447 seconds
```

The figures 4.7, 4.8, 4.9, 4.10 and 4.11 show some important moments of our simulation.

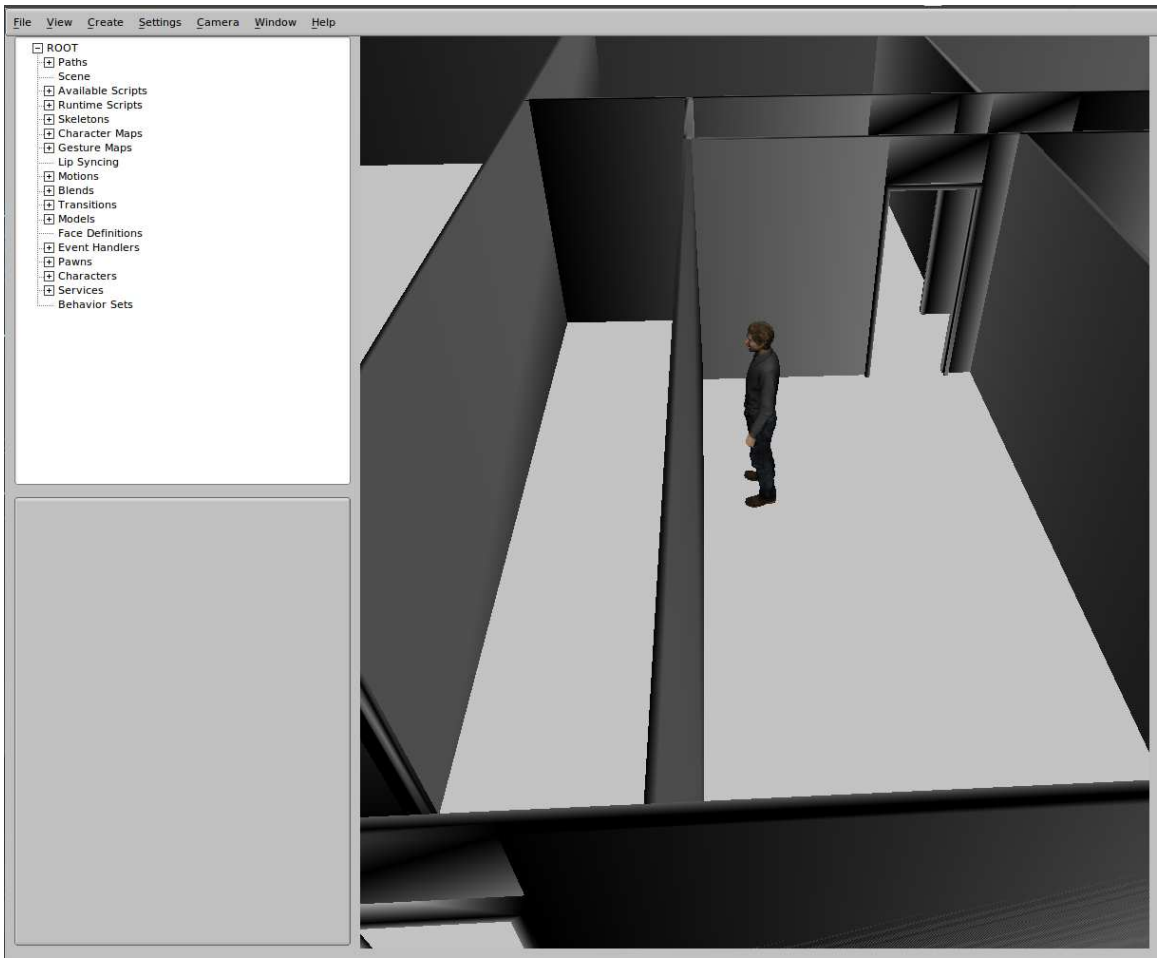

Figure 4.7: The simulation loaded

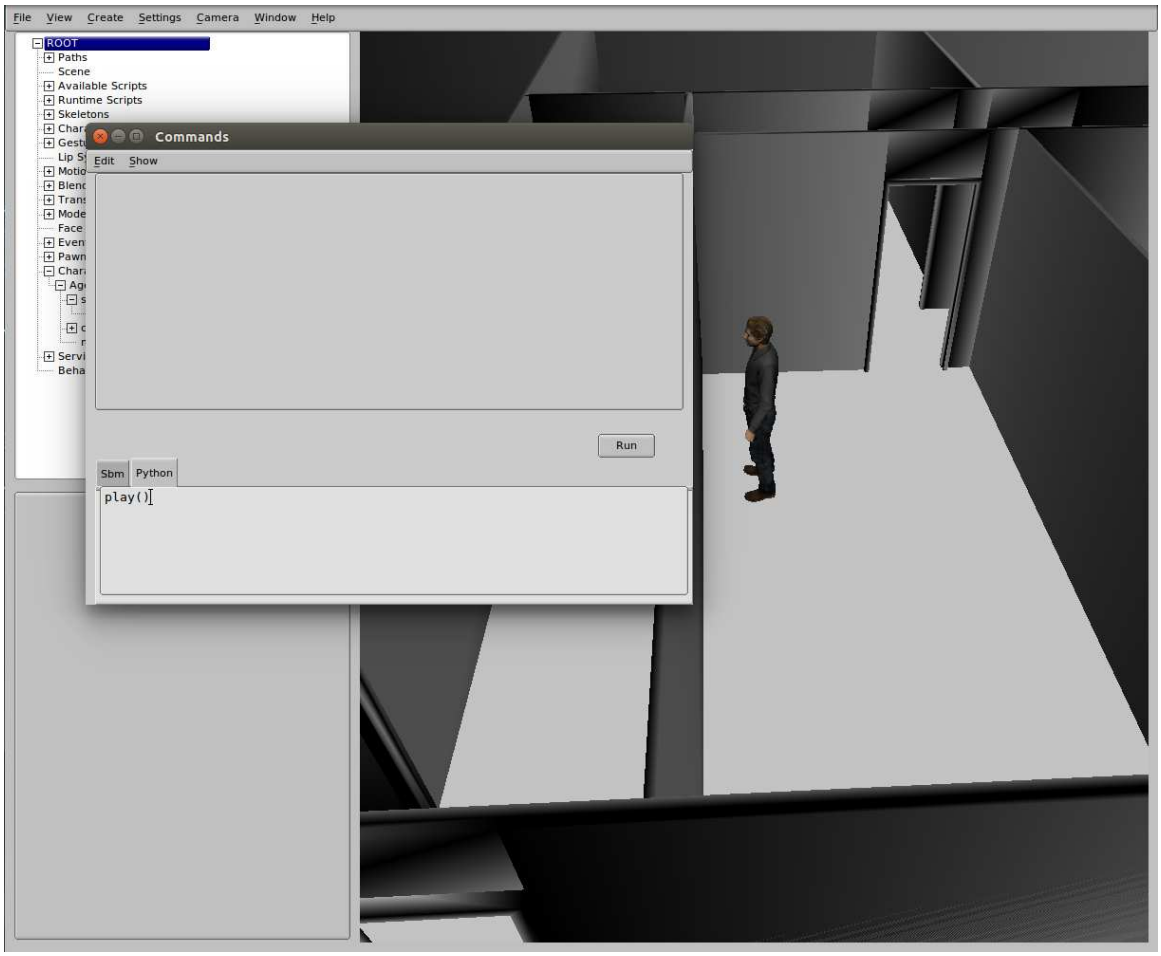

Figure 4.8: Starting the simulation

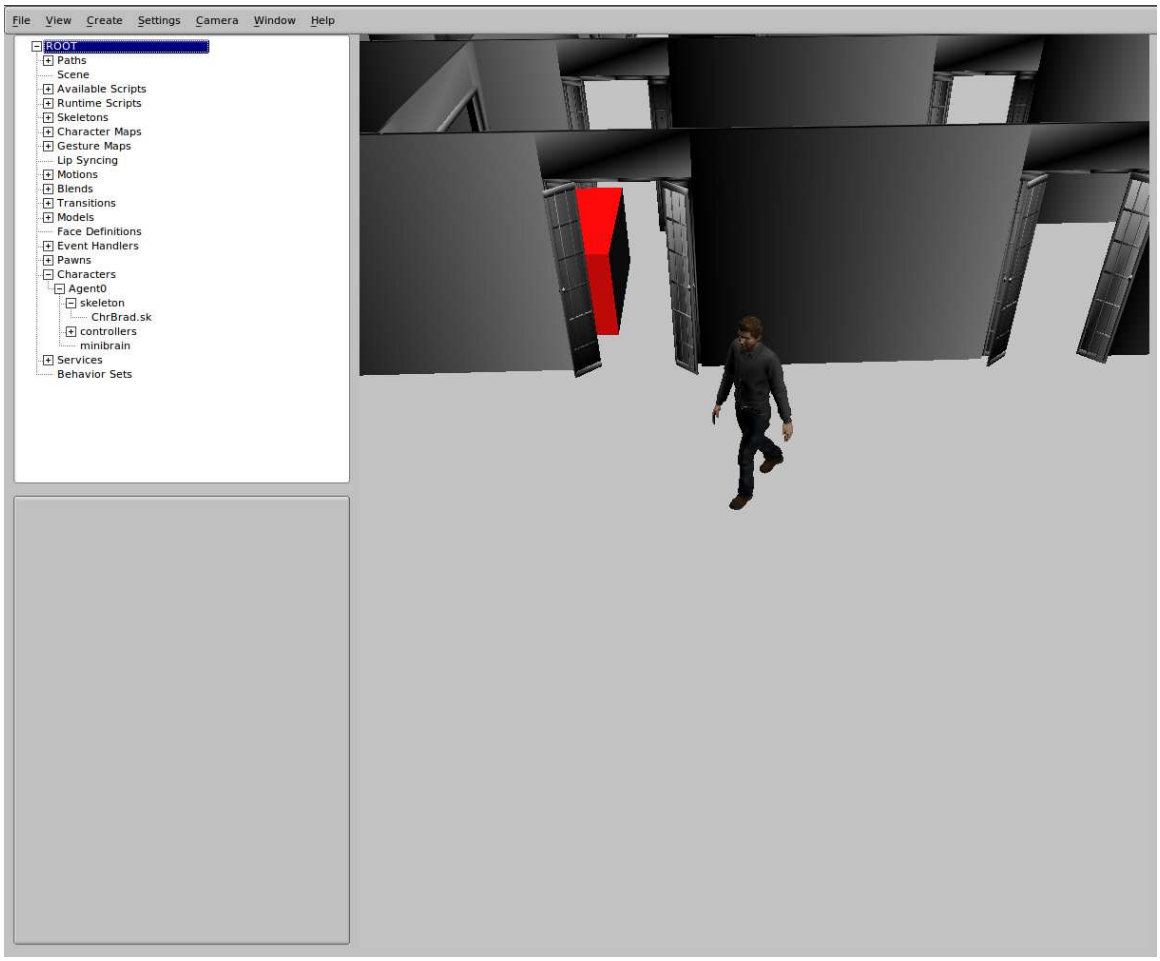

Figure 4.9: The agent escaping

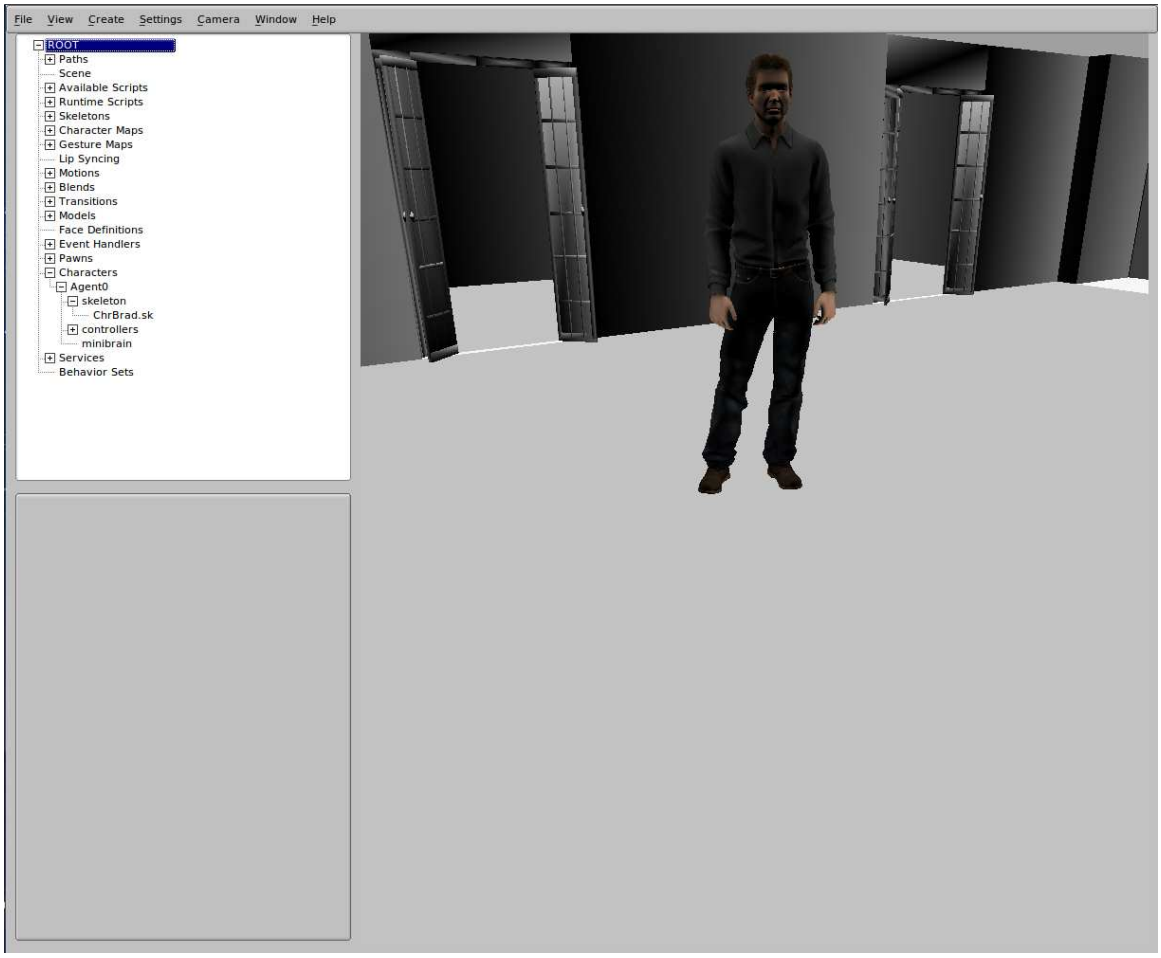

Figure 4.10: The agent has reached the final position

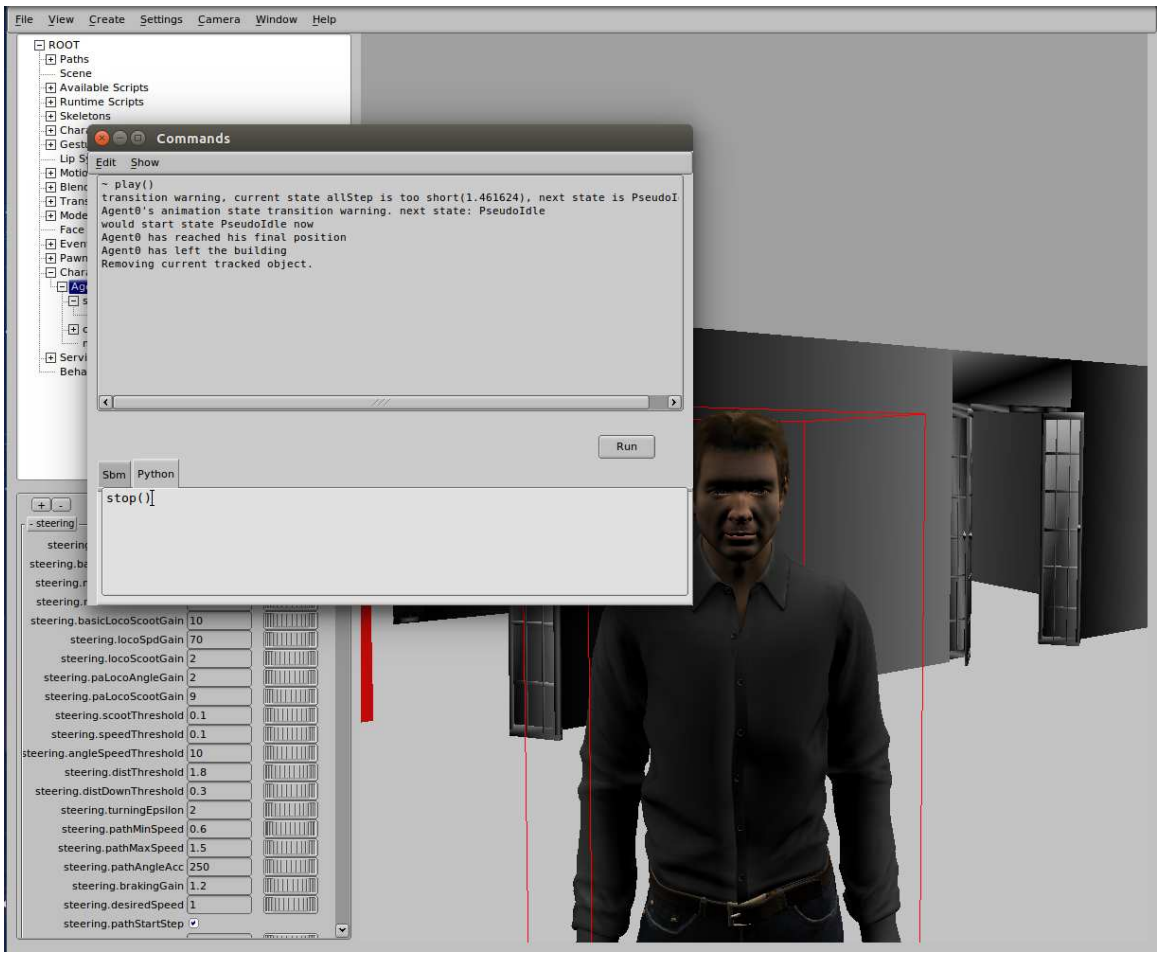

Figure 4.11: Stopping the simulation

4.8.3 Crowd Escaping

4.8.3.1 Overview

We build this case over the last case. As we stated before, we are building our project brick by brick, so after having a functional simulator with only one agent, the next logical step is developing a simulator with more agents.

In this simulation we design a number of characters and one of these agents will be the leader. The other characters will follow the leader from their initial point to the exit.

This case will help us to improve our previous approach without bothering processing multiple paths and characters informations. Also, following some designed leaders is a very common behavior in evacuation plans.

4.8.3.2 Setting the scene

For this simulation we will use the amountAgents value to set how many agents will follow each leader. We will also use the amountLeaders option to set up how many characters will be leaders.

Our configuration file could be:

Listing 4.7: Crowd case configuration file example

```
[Settings]
ubikSimServer= localhost:8080
meshScenario= ETSIT.obj
modeSimulation= bmlRoute
amountLeaders=1
amountAgents=3
```

4.8.3.3 Main module

To execute this simulation we have to run the main module of this simulation. As we saw previously, the other modules are common to every simulation.

Listing 4.8: Crowd case main module

```
import io
```

```
import httplib
import time
import ConfigParser
import math
import json

agentStep = []
inside = []
firstStep = []
timeStart = []
timeStop = []

results = io.open('SmartSimResults', 'wb')

simulationStarted = False

config = ConfigParser.RawConfigParser()
config.read('SmartSimSettings.cfg')

amountLeaders = config.getint('Settings', 'amountLeaders')
amountAgents = config.getint('Settings', 'amountAgents')
ubikSimServer = config.get('Settings', 'ubikSimServer')
meshScenario = config.get('Settings', 'meshScenario')
modeSimulation = 'bmlRoute'

print "|-------------------------------------------|"
print "|            Starting SmartSim              |"
print "|-------------------------------------------|"
print ""

scene.addAssetPath('script', 'scripts')
scene.loadAssets()

scene.run('configureModule.py')
scene.run('agentsCreationModule.py')
scene.run('tools.py')
scene.run('connectionsModule.py')
scene.run('scenarioModule.py')
scene.run('locomotionModule.py')

print "Initiating Scene"
ConfigureModule().init()

print "Initiating global variables"
ConfigureModule().initGlobalVariables(amountAgents)
```

```
print "Setting scene limits"
vectorLimits = ConfigureModule().autoSetLimits(ubikSimServer)
xLimitScene = vectorLimits[0]
yLimitScene = vectorLimits[1]
ConfigureModule().setLimits(xLimitScene, yLimitScene)

print "Setting scenario"
ScenarioModule().addScenario(meshScenario)
ScenarioModule().addEmergency(ubikSimServer)

print "Creating agents"
AgentsCreationModule().settleAgents(amountLeaders, ubikSimServer)

for i in range (amountLeaders):
  leaderName = ('Agent%s' % i)
  leader = scene.getCharacter(leaderName)
  leaderPosition = leader.getPosition()

  for j in range (amountAgents):
    xLeader = leaderPosition.getData(0)
    yLeader = leaderPosition.getData(2)
    followerPosition = SrVec(xLeader + (j+1)*0.5, 0, yLeader + (j)*0.5)
    AgentsCreationModule().addAgent((leaderName+'%s' %j), followerPosition)

print "Configuring camera settings"
cameraCenter=ConfigureModule().autoSetCamera(amountLeaders)
cameraEye = [cameraCenter[0], 10, cameraCenter[2]+10]
cameraPosition = cameraEye
ConfigureModule().camera(cameraEye, cameraCenter, cameraPosition)

print "Getting routes"
ConnectionsModule().initSteps(amountAgents)

steerManager = scene.getSteerManager()
steerManager.setEnable(False)
steerManager.setEnable(True)

amountFollowers = amountAgents
amountAgents = amountLeaders

print "Scene settled"

scene.removeScript('locomotion')
locomotion = LocomotionModule()
```

```
scene.addScript('locomotion', locomotion)

def play():
  LocomotionModule().playSimulation()

def stop():
  LocomotionModule().finishSimulation()
```

The main module is pretty similar to the last case. However, some notable changes had been applied. We use another variable, amountLeaders, that sets the number of leaders that steer the groups. The variable amountAgents is used to set the number of agents that every group has. In this module the agents that conform the groups are created using the functions of the Agents Creation Module.

4.8.3.4 Simulation

Our agents follow their leader to the exit. They will follow the same path, but only the leader is given a path from UbikSim and the others will follow him.

After stopping the simulation we will get a results file. We only get the results for the leader.

Listing 4.9: Crowd case results file example

```
Agent0 succesfully exited the building
Agent exited the building in 43.6237959862 seconds
```

The figures 4.12, 4.13, 4.14, 4.15 and 4.16 show us some important moments of our simulation progress.

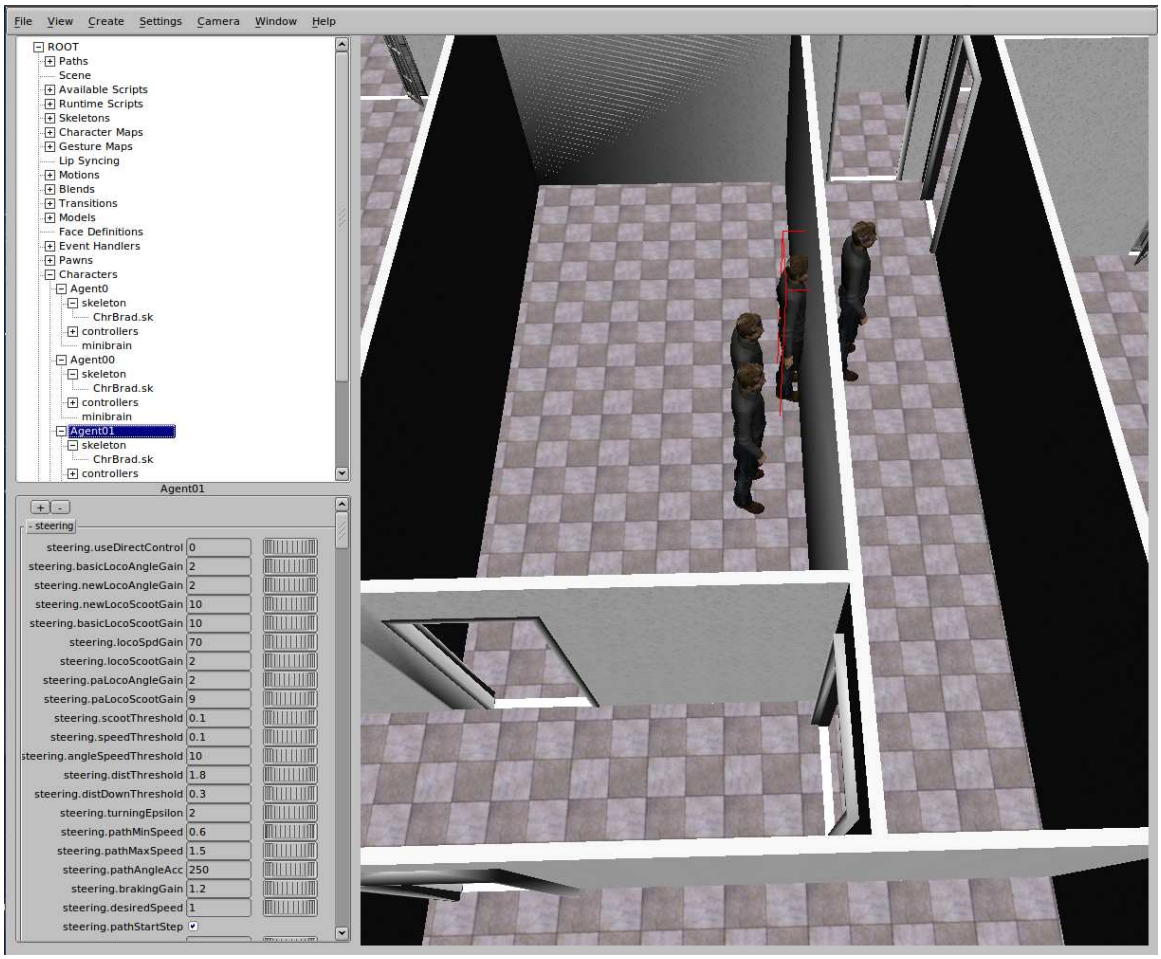

Figure 4.12: The simulation loaded

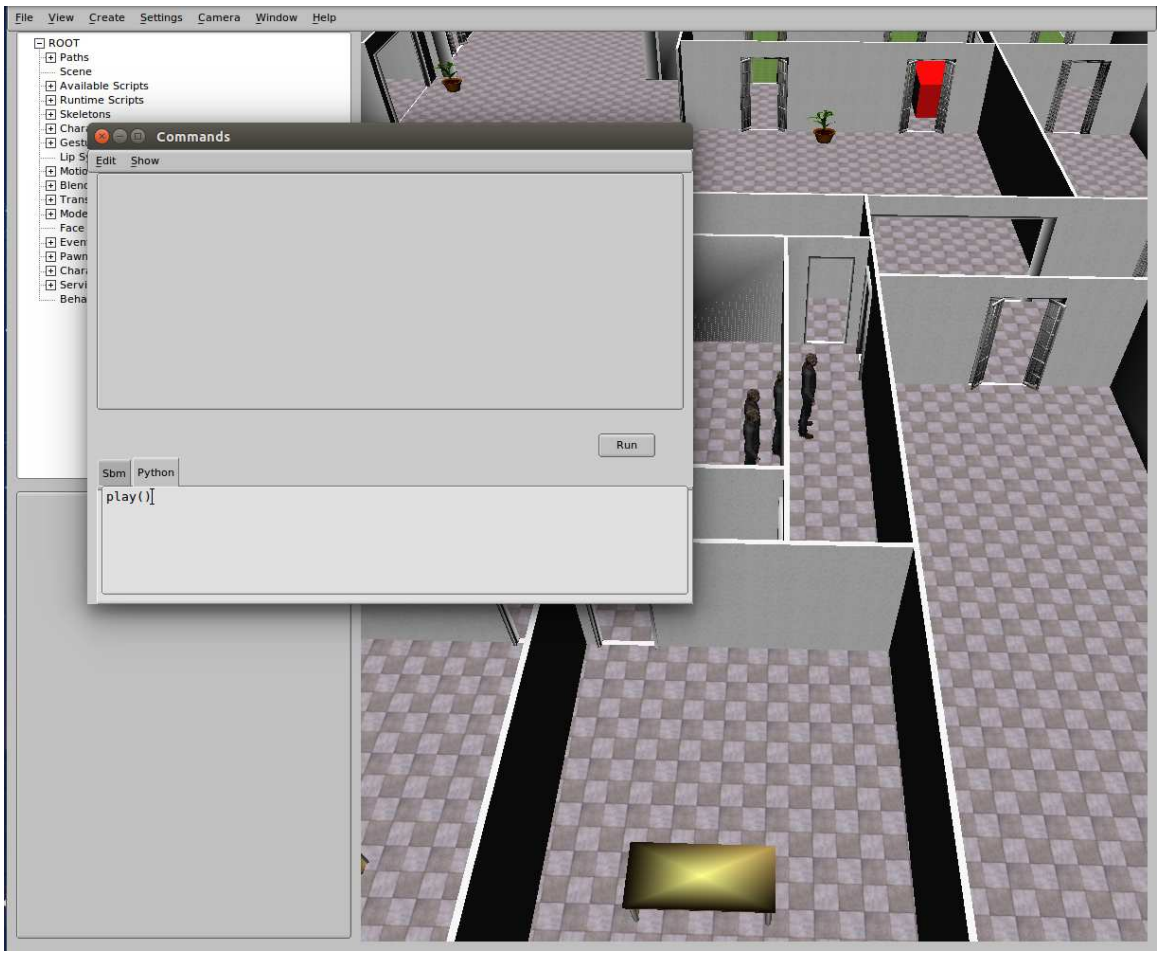

Figure 4.13: Starting the simulation

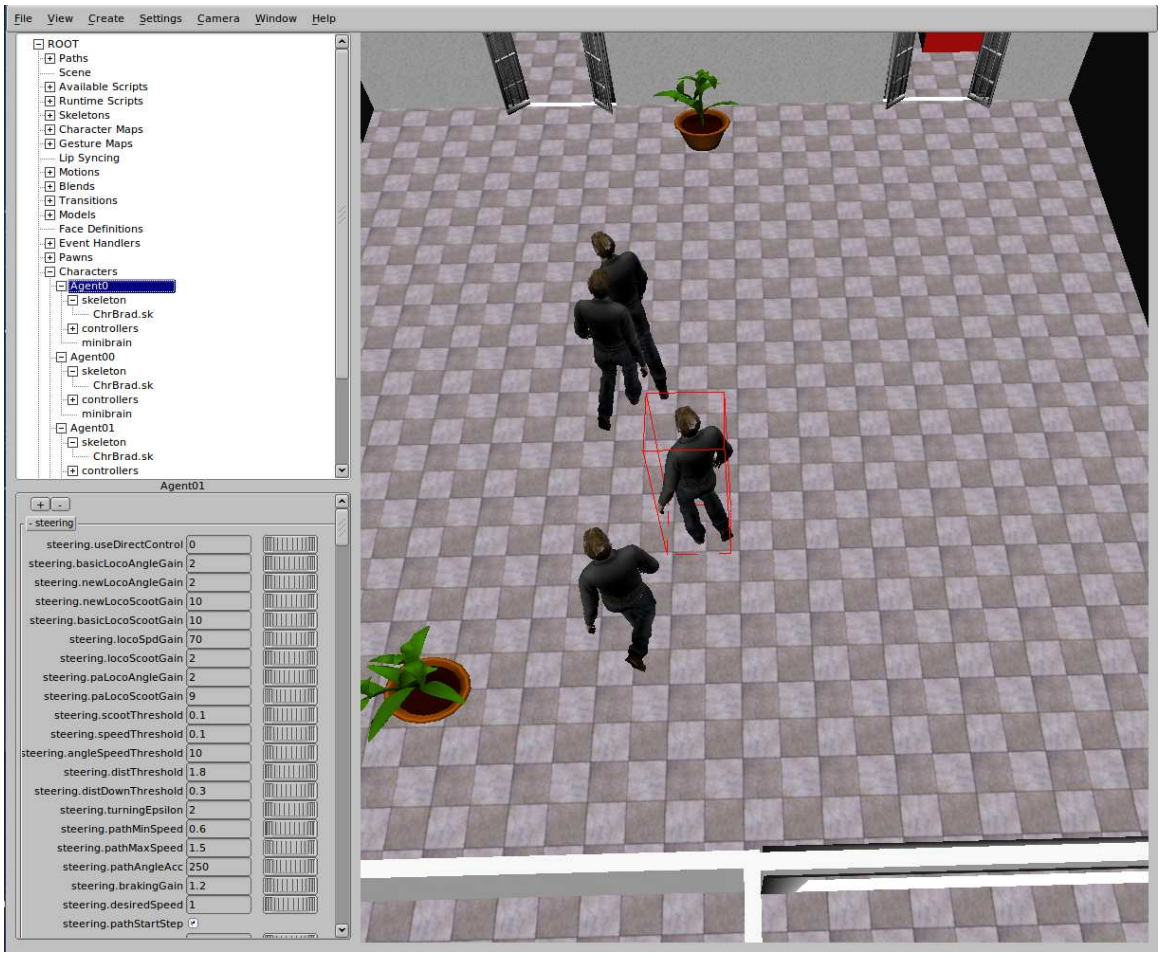

Figure 4.14: The agents escaping

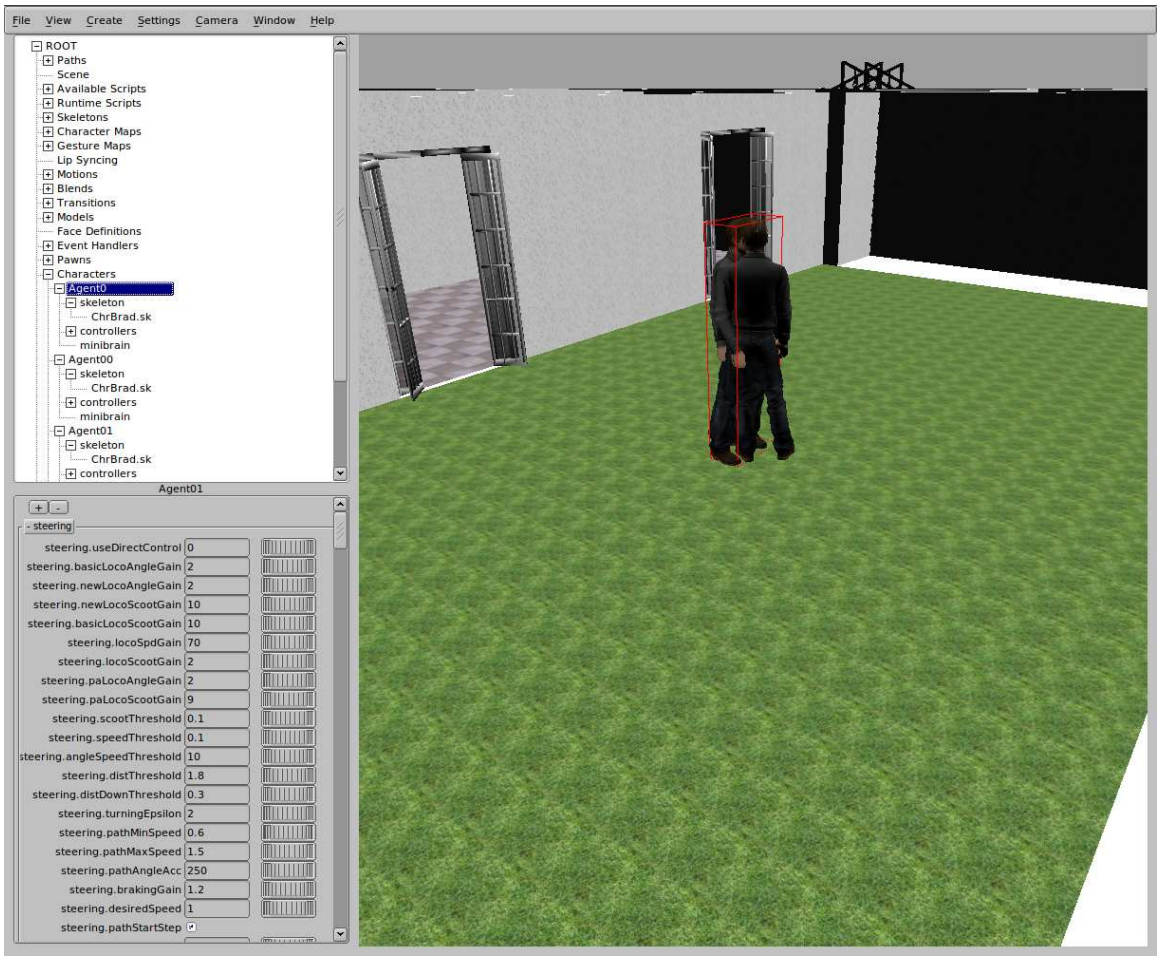

Figure 4.15: The agents have reached the final position

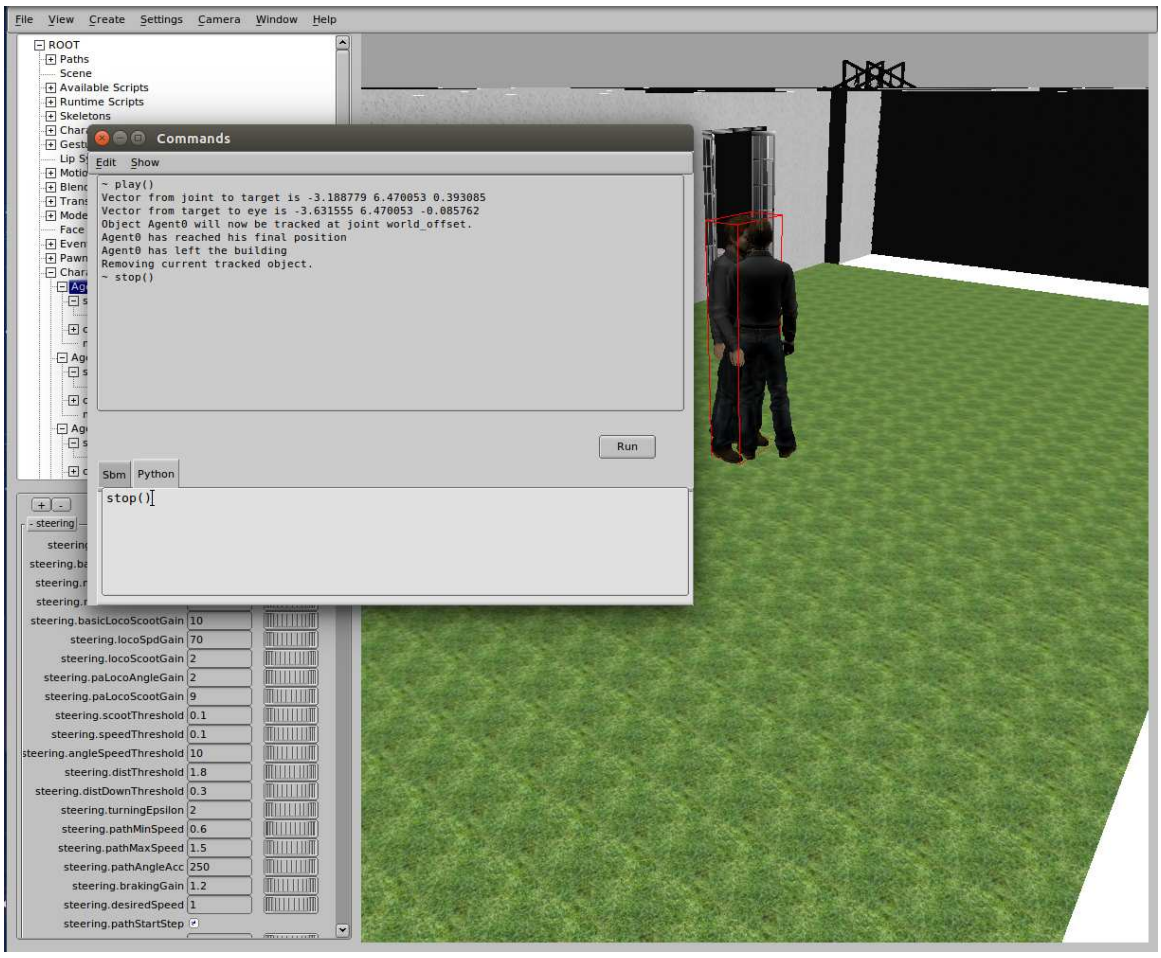

Figure 4.16: Stopping the simulation

4.8.4 Social simulator

4.8.4.1 Overview

We have developed a simulator with multiple characters. So it is time for us to develop a more complex simulator. We have managed a few characters paths and many agents followed these paths. So, introducing more paths and managing them seems to be the next logical step in developing our simulator.

This case fulfills the objectives of the MOSI-AGIL project. We retrieve our paths from UbikSim and use SmartBody as our graphic interface.

4.8.4.2 Setting the scene

As in the previous cases, we implement the configuration options in the configuration file. We disable the amountLeaders option we used in the previous case, as we have abandoned the leader role we used in it.

Nevertheless, we have not used the type character options we can configure in our file. This option is not implemented in this case yet. All our agents are modelled after the default character we used in the previous cases.

We must set the characters files and the scenario in their paths, as we do in the previous cases.

The configuration file we used is similar to:

Listing 4.10: Social simulator case main module

```
[Settings]
ubikSimServer= localhost:8080
meshScenario= ETSIT.obj
modeSimulation= gettingSteps
amountAgents=15
```

4.8.4.3 Main module

```
import io
import httplib
import time
import ConfigParser
```

```
import math
import json

agentStep = []
inside = []
firstStep = []
timeStart = []
timeStop = []

results = io.open('SmartSimResults', 'wb')

simulationStarted = False

config = ConfigParser.RawConfigParser()
config.read('SmartSimSettings.cfg')

amountAgents = config.getint('Settings', 'amountAgents')
ubikSimServer = config.get('Settings', 'ubikSimServer')
meshScenario = config.get('Settings', 'meshScenario')
modeSimulation = config.get('Settings', 'modeSimulation')

print amountAgents
print ubikSimServer
print meshScenario
print modeSimulation

print "|-------------------------------------------|"
print "|            Starting SmartSim              |"
print "|-------------------------------------------|"
print ""

scene.addAssetPath('script', 'scripts')
scene.loadAssets()

scene.run('configureModule.py')
scene.run('agentsCreationModule.py')
scene.run('tools.py')
scene.run('connectionsModule.py')
scene.run('scenarioModule.py')
scene.run('locomotionModule.py')

print "Initiating Scene"
ConfigureModule().init()

print "Initiating global variables"
```

```
ConfigureModule().initGlobalVariables(amountAgents)

print "Setting scene limits"
vectorLimits = ConfigureModule().autoSetLimits(ubikSimServer)
xLimitScene = vectorLimits[0]
yLimitScene = vectorLimits[1]
ConfigureModule().setLimits(xLimitScene, yLimitScene)

print "Setting scenario"
ScenarioModule().addScenario(meshScenario)
ScenarioModule().addEmergency(ubikSimServer)

print "Creating agents"
AgentsCreationModule().settleAgents(amountAgents, ubikSimServer)

print "Configuring camera settings"
cameraCenter=ConfigureModule().autoSetCamera(amountAgents)
cameraEye = [cameraCenter[0], 30, cameraCenter[2]+40]
cameraPosition = cameraEye
ConfigureModule().camera(cameraEye, cameraCenter, cameraPosition)

print "Getting routes"
ConnectionsModule().initSteps(amountAgents)

print 'Steering'

steerManager = scene.getSteerManager()
steerManager.setEnable(False)
steerManager.setEnable(True)

print "Scene settled"

def play():
  LocomotionModule().playSimulation()

def stop():
  LocomotionModule().finishSimulation()

def createAgent(name, x, y):
  AgentsCreationModule().addAgent(name, SrVec(x, 0, y))

scene.removeScript('locomotion')
locomotion = LocomotionModule()
scene.addScript('locomotion', locomotion)
```

The main module is similar to previous ones. However, there are notable differences. The agents creation is similar to the one character case, nevertheless, the amount of agents is retrieved from the configuration file. A function to ease the agents creation process is created.

4.8.4.4 Simulation

Every character follows his own path and has his own initial position. When the character arrives at its final position, he stays there.

We can retrieve the results in a file as in the previous cases.

Listing 4.11: Social simulator results file example

```
Agent10 succesfully exited the building
Agent exited the building in 16.9189949036 seconds
Agent13 succesfully exited the building
Agent exited the building in 36.6269879341 seconds
Agent12 succesfully exited the building
Agent exited the building in 39.6204659939 seconds
Agent8 succesfully exited the building
Agent exited the building in 40.1318058968 seconds
Agent7 succesfully exited the building
Agent exited the building in 62.198641777 seconds
Agent6 succesfully exited the building
Agent exited the building in 85.1019399166 seconds
Agent3 succesfully exited the building
Agent exited the building in 89.0953888893 seconds
Agent5 succesfully exited the building
Agent exited the building in 97.0570399761 seconds
Agent4 succesfully exited the building
Agent exited the building in 116.99630785 seconds
Agent9 succesfully exited the building
Agent exited the building in 123.446609974 seconds
Agent2 succesfully exited the building
Agent exited the building in 138.223900795 seconds
Agent1 succesfully exited the building
Agent exited the building in 140.211885929 seconds
```

The Figures 4.17, 4.18, 4.19, 4.20 and 4.21 show us some important moments of our simulation. Figures 4.22 and 4.23 show us how to add a character to the simulation.

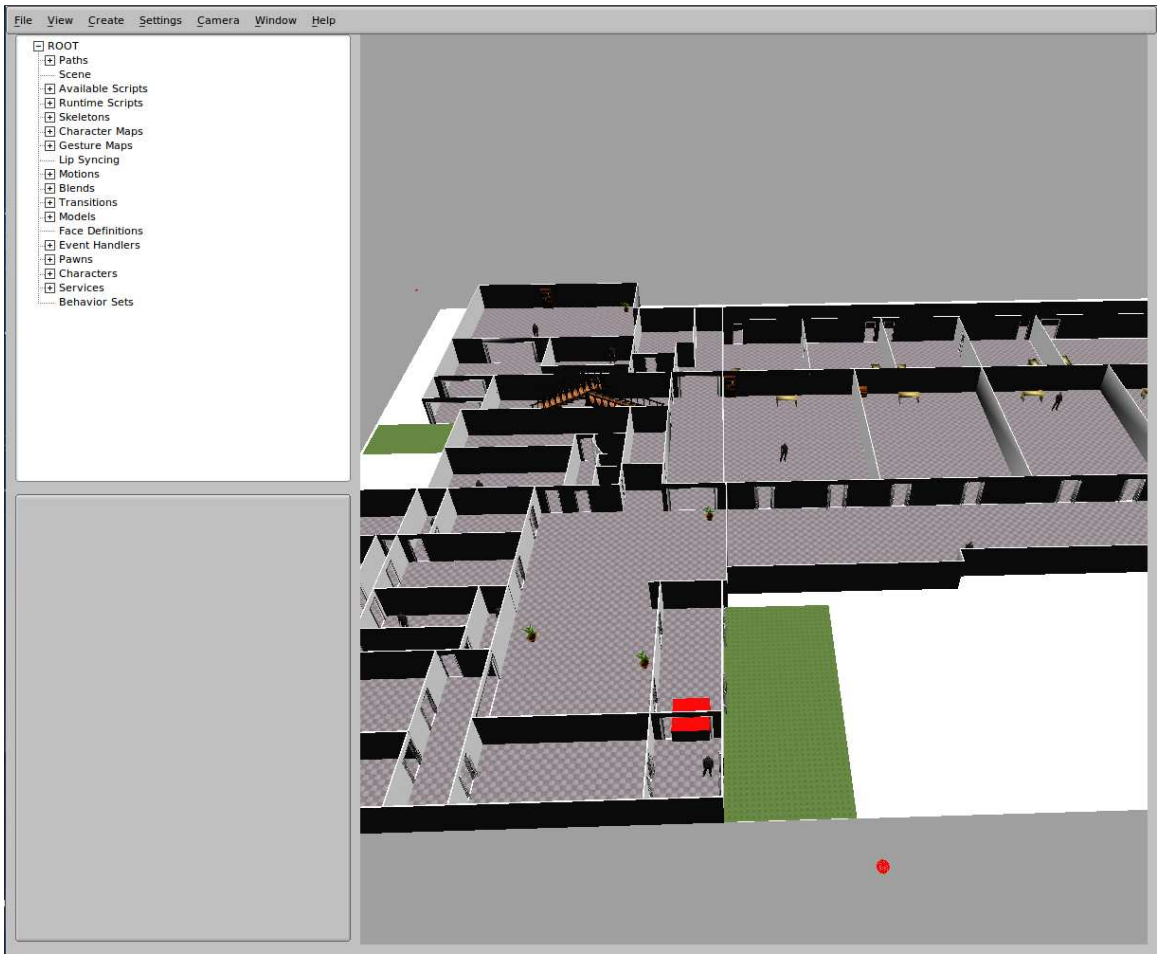

Figure 4.17: The simulation loaded

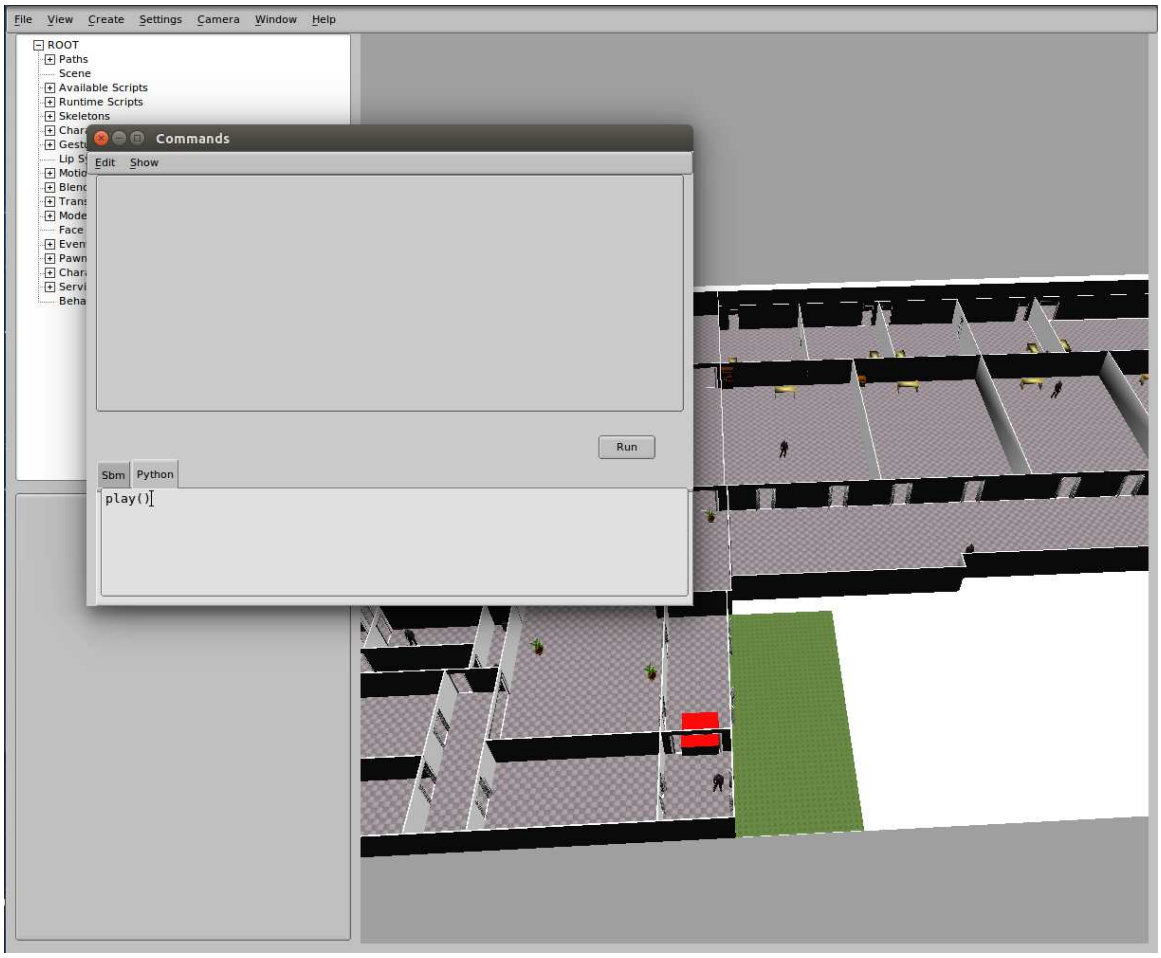

Figure 4.18: Starting the simulation

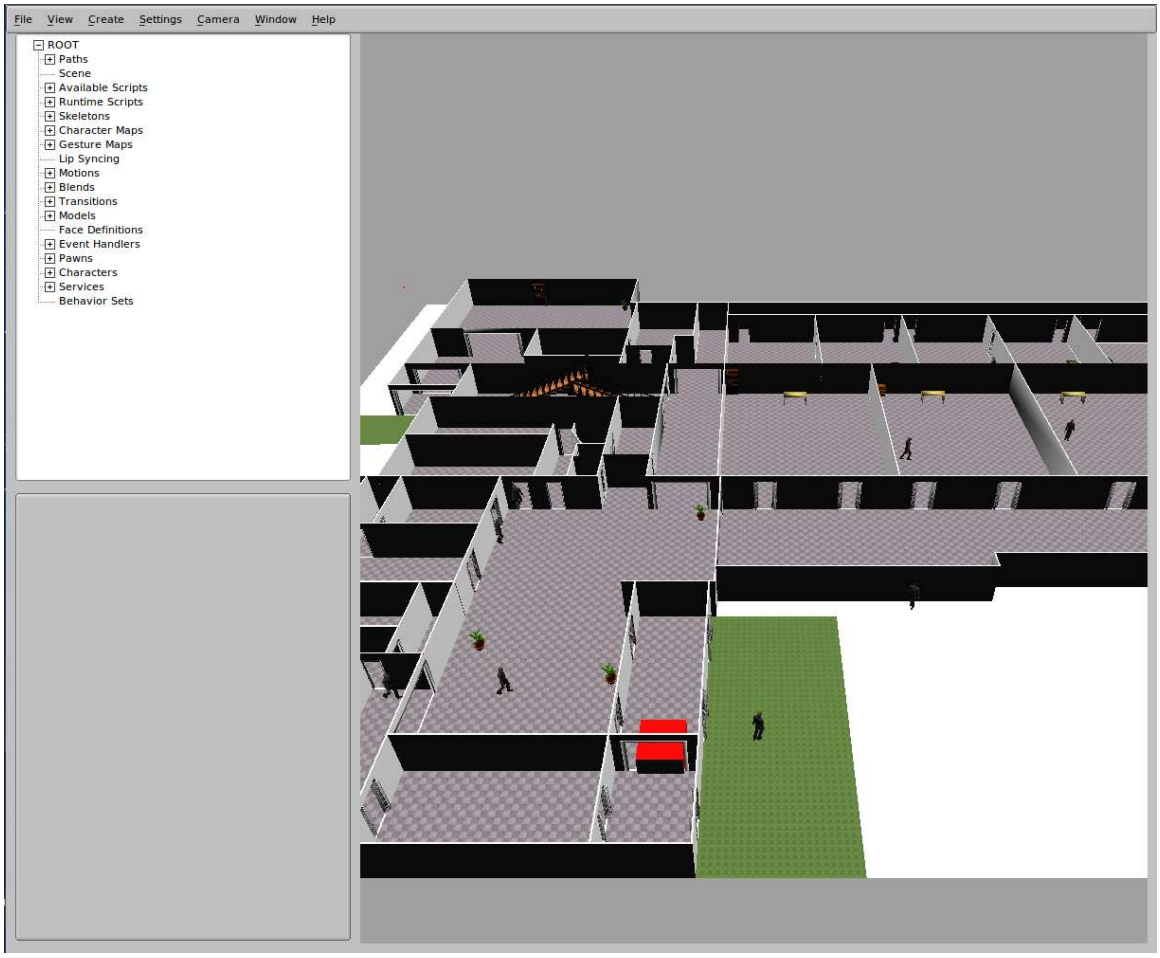

Figure 4.19: The agents escaping

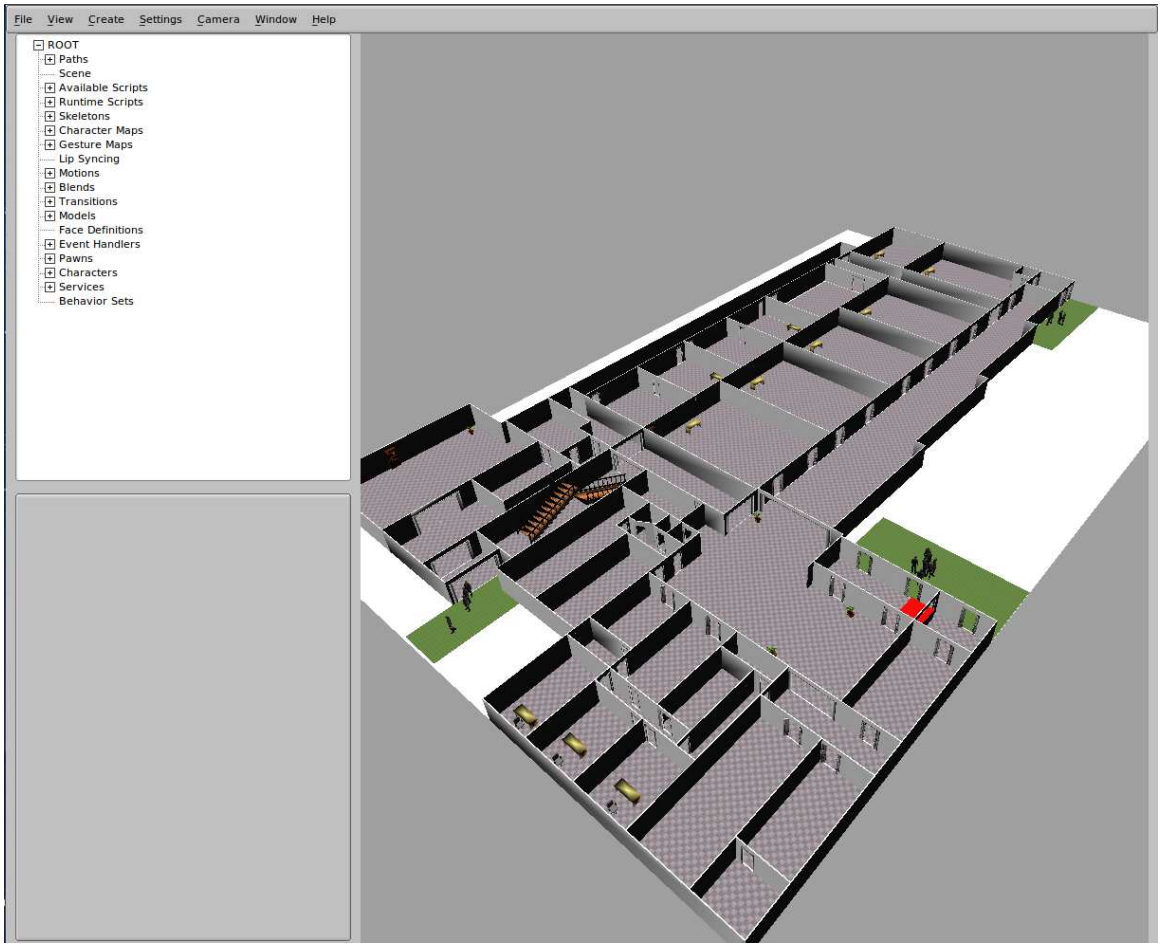

Figure 4.20: The agents have reached the final position

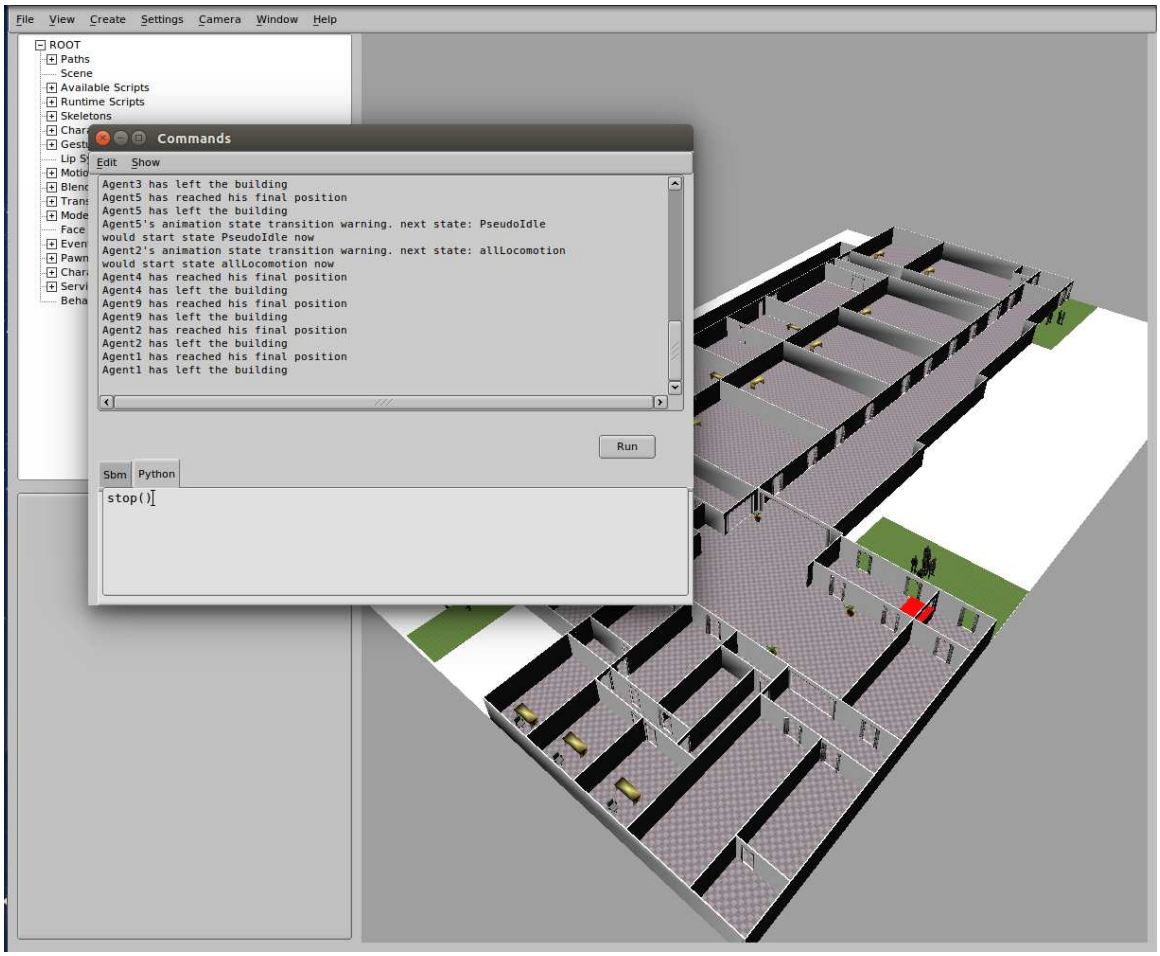

Figure 4.21: Stopping the simulation

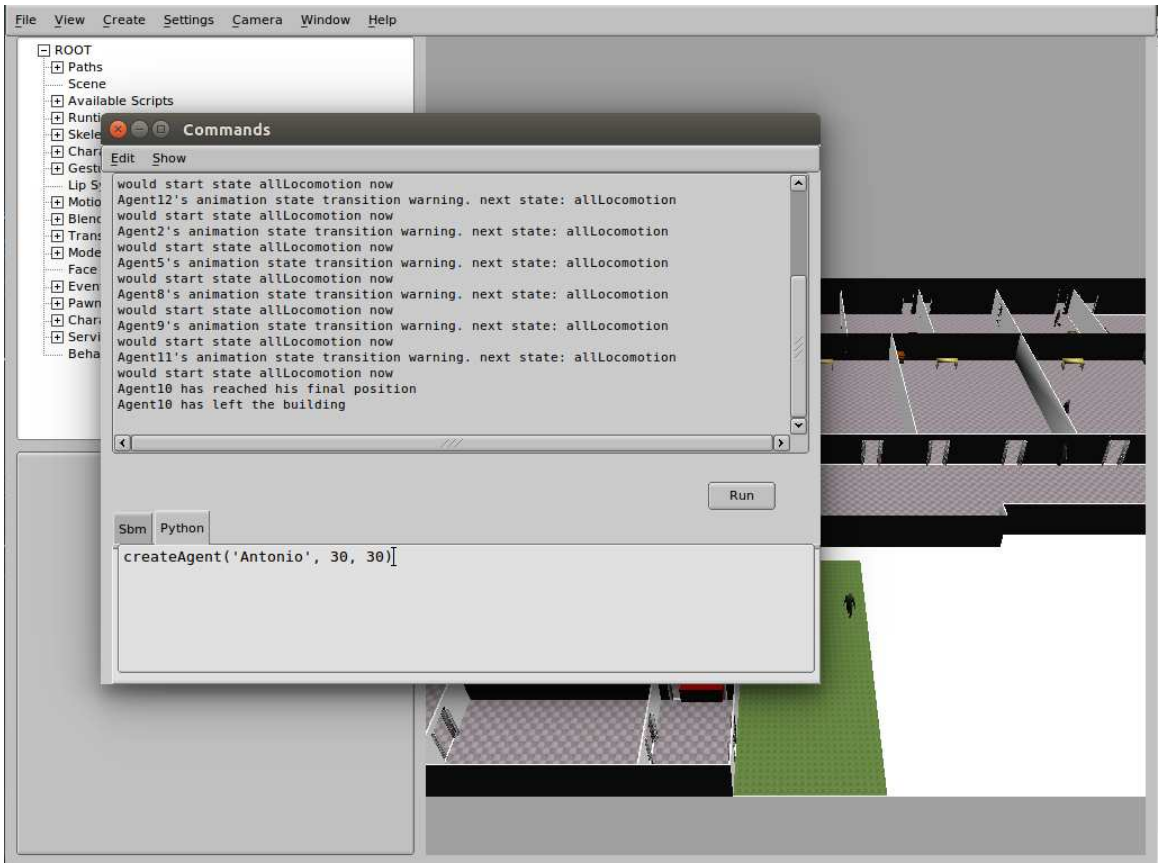

Figure 4.22: Creating an agent

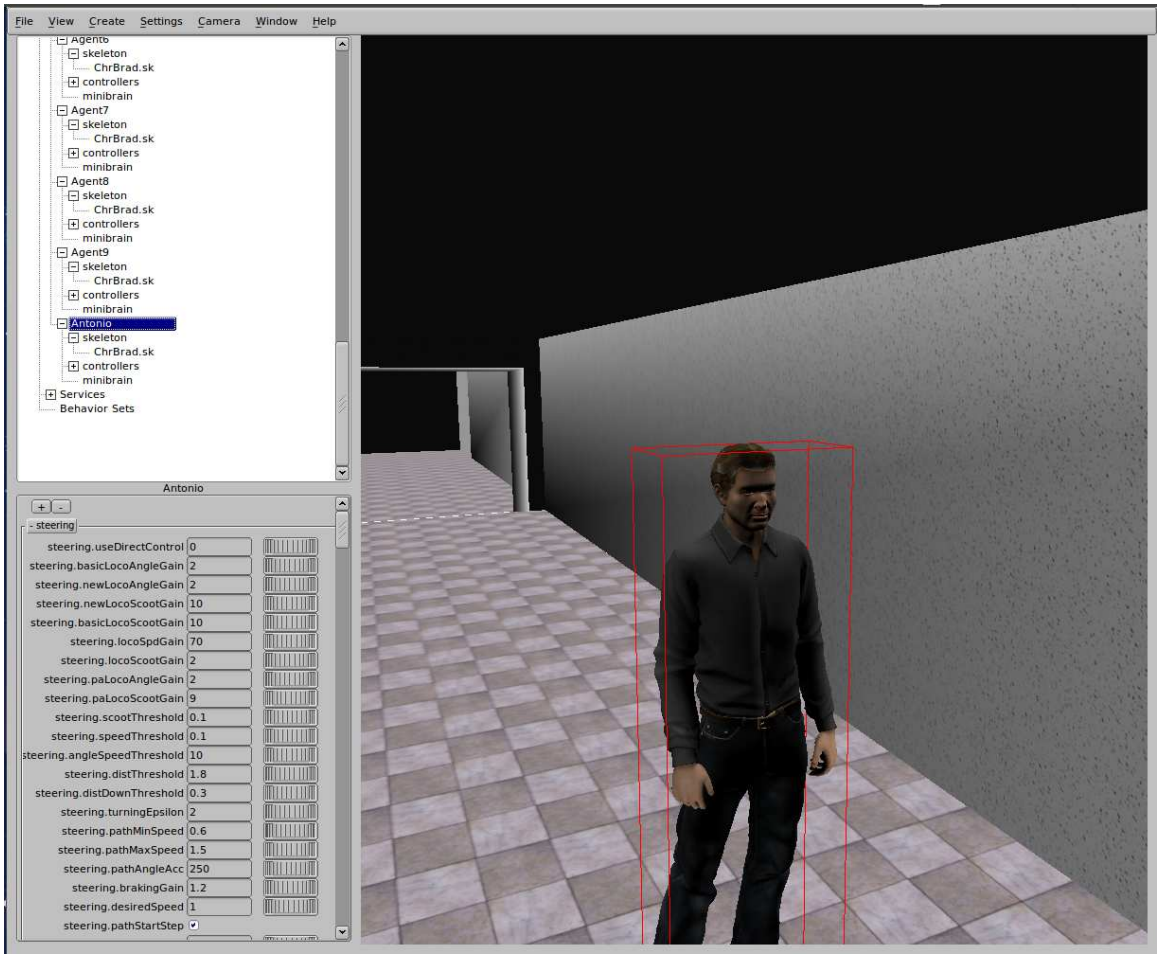

Figure 4.23: The agent created

4.8.5 Social simulator with emotions

4.8.5.1 Overview

We have used BML for moving our agents in the previous cases. However, using BML only for this would be a waste, as we can take advantage of several other functions of agents modelling. The big amount of emotions the agents can represent that we can trigger using BML can be used to model the human behavior in our simulator.

Giving our agents the ability to cry when they feel danger or to run when they are close to the emergency is a step ahead for our simulator. Using BML we can design behaviors as close to reality as we want.

So, it is clear that the evolution of our project is related to an extensive use of BML to represent with more realism the human behavior in an emergency situation. So, this stage of the simulator is focused in developing that.

4.8.5.2 Setting the scene

We use the same configuration file we used in the previous cases. The configuration step is the same than in the previous case, as the differences are managed in the main module.

As in previous cases, we have to set our files in their correct paths.

4.8.5.3 Main module

Listing 4.12: Social simulator with emotions case main module

```
import io
import httplib
import time
import ConfigParser
import math
import json

agentStep = []
inside = []
firstStep = []
timeStart = []
timeStop = []
```

```
typesCharacters=[]

results = io.open('SmartSimResults', 'wb')

behaviors = True
simulationStarted = False

config = ConfigParser.RawConfigParser()
config.read('SmartSimSettings.cfg')

amountAgents = config.getint('Settings', 'amountAgents')
ubikSimServer = config.get('Settings', 'ubikSimServer')
meshScenario = config.get('Settings', 'meshScenario')
modeSimulation = 'gettingSteps'

print amountAgents
print ubikSimServer
print meshScenario
print modeSimulation

print "|-------------------------------------------|"
print "|            Starting SmartSim              |"
print "|-------------------------------------------|"
print ""

scene.addAssetPath('script', 'scripts')
scene.loadAssets()

scene.run('configureModule.py')
scene.run('agentsCreationModule.py')
scene.run('tools.py')
scene.run('connectionsModule.py')
scene.run('scenarioModule.py')
scene.run('locomotionModule.py')

print "Initiating Scene"
ConfigureModule().init()

print "Initiating global variables"
ConfigureModule().initGlobalVariables(amountAgents)

print "Setting scene limits"
vectorLimits = ConfigureModule().autoSetLimits(ubikSimServer)
xLimitScene = vectorLimits[0]
yLimitScene = vectorLimits[1]
```

```
ConfigureModule().setLimits(xLimitScene, yLimitScene)

print "Setting scenario"
ScenarioModule().addScenario(meshScenario)
ScenarioModule().addEmergency(ubikSimServer)

positionEmergency=ConnectionsModule().getUbikSimEmergency(ubikSimServer)

print "Creating agents"
AgentsCreationModule().settleAgents(amountAgents, ubikSimServer)

print "Configuring camera settings"
cameraCenter=ConfigureModule().autoSetCamera(amountAgents)
cameraEye = [cameraCenter[0], 30, cameraCenter[2]+40]
cameraPosition = cameraEye
ConfigureModule().camera(cameraEye, cameraCenter, cameraPosition)

print "Getting routes"
ConnectionsModule().initSteps(amountAgents)

print 'Steering'

steerManager = scene.getSteerManager()
steerManager.setEnable(False)
steerManager.setEnable(True)

print "Scene settled"

def play():
  LocomotionModule().playSimulation()

def stop():
  LocomotionModule().finishSimulation()

def createAgent(name, x, y):
  AgentsCreationModule().addAgent(name, SrVec(x, 0, y))

scene.removeScript('locomotion')
locomotion = LocomotionModule()
scene.addScript('locomotion', locomotion)
```

This module is pretty similar to the simulator one. The main differences is setting true the 'behaviors' flag.

4.8.5.4 Simulation

In previous stages of our simulator, our agents acted as robots. They did not express any emotion or reacted to the situations they found in their paths. In this stage, we see our agents as living beings. They react to several stimulus, as danger or relieve when they arrive to a safe position.

The reactions we can find are such as expressing fear when in danger or happiness when they reach a safe place.

As in previous cases, we can retrieve the results.

The Figures 4.24 and 4.25 show us the emotions our agents can express.

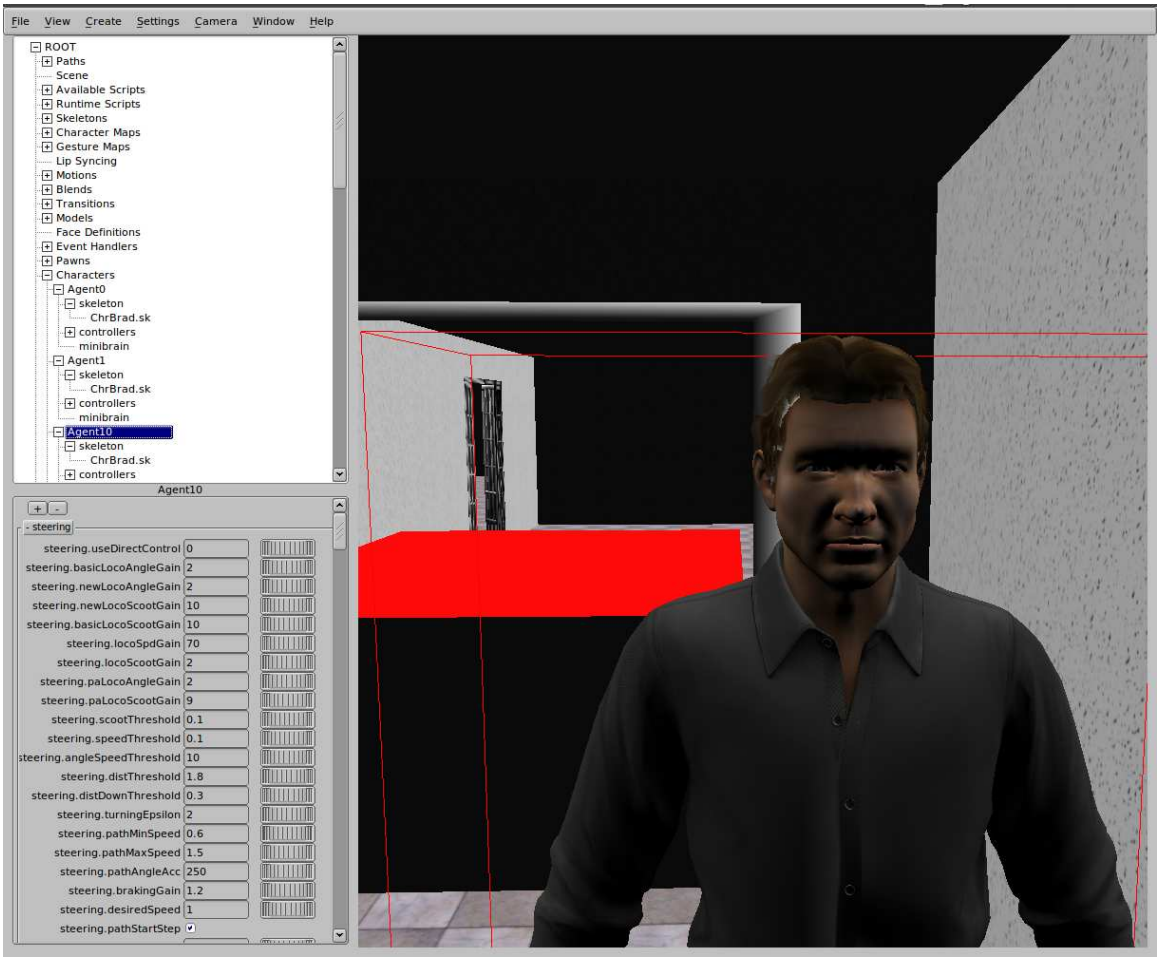

Figure 4.24: An agent expressing fear

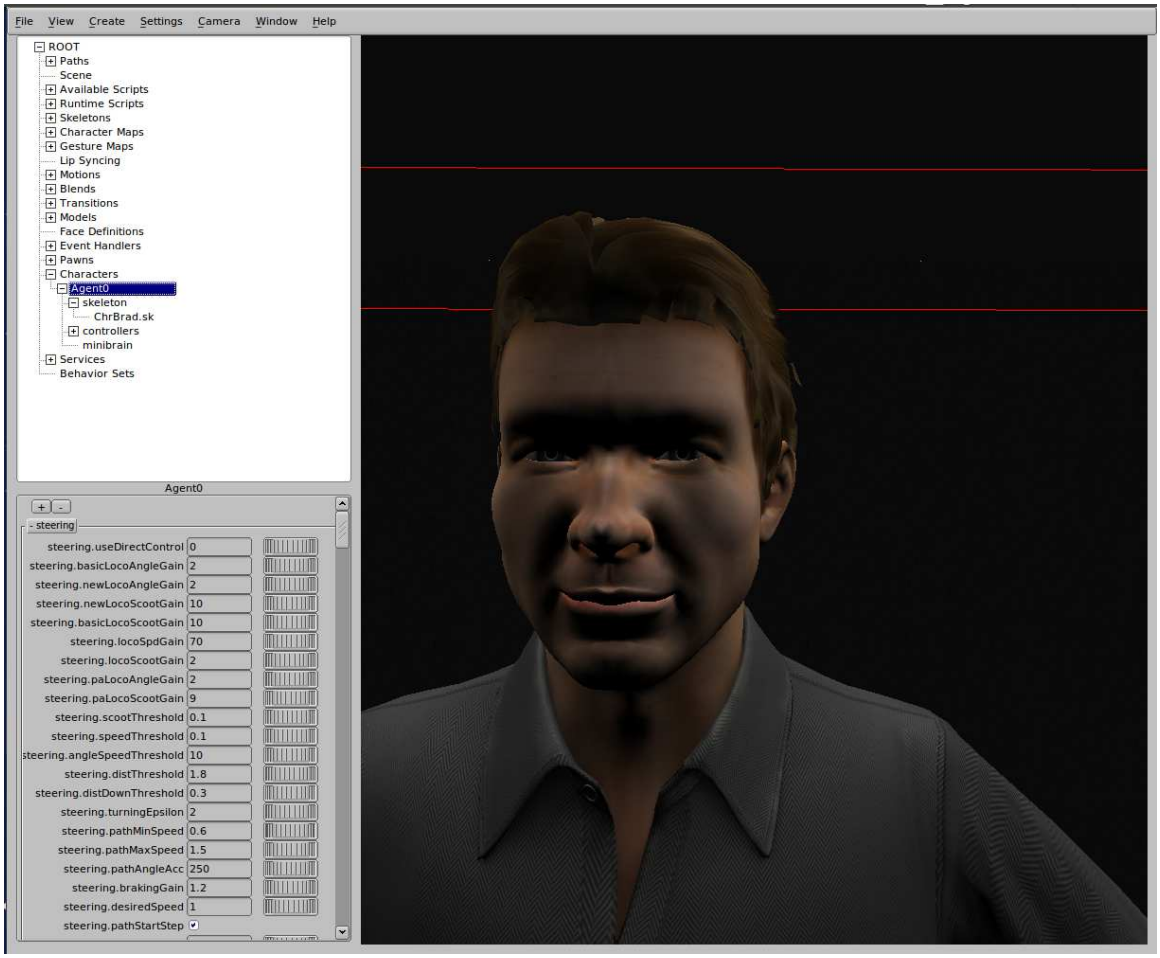

Figure 4.25: An agent expressing happiness

4.8.6 Social simulator with character types

4.8.6.1 Overview

In the previous stages of our simulator, we have only one character type. Every character is modelled after this kind and there are no differences between them. Everyone has the same behaviour and reactions to the situations.

However, we do not find this in a real situation. In the real world, everyone reacts in his own different way. Everyone has his own behaviour. So, in that way, our simulator does not seem really accurate.

So, a logical stage in our simulator development should be the creation of more profiles of characters. These characters should be selected by the users and should have their own behaviours.

4.8.6.2 Setting the scene

We set our scene using our configuration file. In this file we add a new option that was disabled in the previous versions of the simulator. We add a new section in where we introduce the types of characters we are going to use in our simulation. The main module reads the types and if they are defined in our model, they are created.

Our configuration files would be similar to

Listing 4.13: Social simulator with character types configuration file example

```
[Settings]
ubikSimServer= localhost:8080
meshScenario= ETSIT.obj
modeSimulation= gettingSteps
amountAgents=15

[Types]
rachel= 10,2,5
```

We are indicating that the agents 10, 2 and 5 are the 'rachel' type characters.

4.8.6.3 Main module

Listing 4.14: Social simulator with character types main module

```
import io
import httplib
import time
import ConfigParser
import math
import json

agentStep = []
inside = []
firstStep = []
timeStart = []
timeStop = []
typeCharacterDict = {}

results = io.open('SmartSimResults', 'wb')

simulationStarted = False

config = ConfigParser.RawConfigParser()
config.read('SmartSimSettings.cfg')

amountAgents = config.getint('Settings', 'amountAgents')
ubikSimServer = config.get('Settings', 'ubikSimServer')
meshScenario = config.get('Settings', 'meshScenario')
modeSimulation = config.get('Settings', 'modeSimulation')
typesCharacters = config.options('Types')

for typeCharacter in typesCharacters:
  charactersInType = config.get('Types',typeCharacter)
  charactersInType = [int(n) for n in charactersInType.split(',')]
  typeCharacterDict[typeCharacter] = charactersInType

print "|-------------------------------------------|"
print "|            Starting SmartSim              |"
print "|-------------------------------------------|"
print ""

scene.addAssetPath('script', 'scripts')
scene.loadAssets()

scene.run('configureModule.py')
scene.run('agentsCreationModule.py')
```

```
scene.run('tools.py')
scene.run('connectionsModule.py')
scene.run('scenarioModule.py')
scene.run('locomotionModule.py')

print "Initiating Scene"
ConfigureModule().init()

print "Initiating global variables"
ConfigureModule().initGlobalVariables(amountAgents)

print "Setting scene limits"
vectorLimits = ConfigureModule().autoSetLimits(ubikSimServer)
xLimitScene = vectorLimits[0]
yLimitScene = vectorLimits[1]
ConfigureModule().setLimits(xLimitScene, yLimitScene)

print "Setting scenario"
ScenarioModule().addScenario(meshScenario)
ScenarioModule().addEmergency(ubikSimServer)

print "Creating agents"
AgentsCreationModule().settleAgents(amountAgents, ubikSimServer)

print "Configuring camera settings"
cameraCenter=ConfigureModule().autoSetCamera(amountAgents)
cameraEye = [cameraCenter[0], 30, cameraCenter[2]+40]
cameraPosition = cameraEye
ConfigureModule().camera(cameraEye, cameraCenter, cameraPosition)

print "Getting routes"
ConnectionsModule().initSteps(amountAgents)

print 'Steering'

steerManager = scene.getSteerManager()
steerManager.setEnable(False)
steerManager.setEnable(True)

print "Scene settled"

def play():
  LocomotionModule().playSimulation()

def stop():
```

```
  LocomotionModule().finishSimulation()

def createAgent(name, x, y, *characterType):
  for character in characterType:
    characterType = character
  AgentsCreationModule().addAgent(name, SrVec(x, 0, y), characterType)

scene.removeScript('locomotion')
locomotion = LocomotionModule()
scene.addScript('locomotion', locomotion)
```

The module is similar to previous modules but in this case we read the Types options to create the characters.

4.8.6.4 Simulation

As we can see, we do not only create one type of characters. We add a female type character. This character has her own behaviour. We can select the type of the character using the configuration file or as we create it.

The Figures 4.26, 4.27 and 4.28 show us some important moments in our simulation. Figures 4.29 and 4.30 show us how to add characters to the simulation.

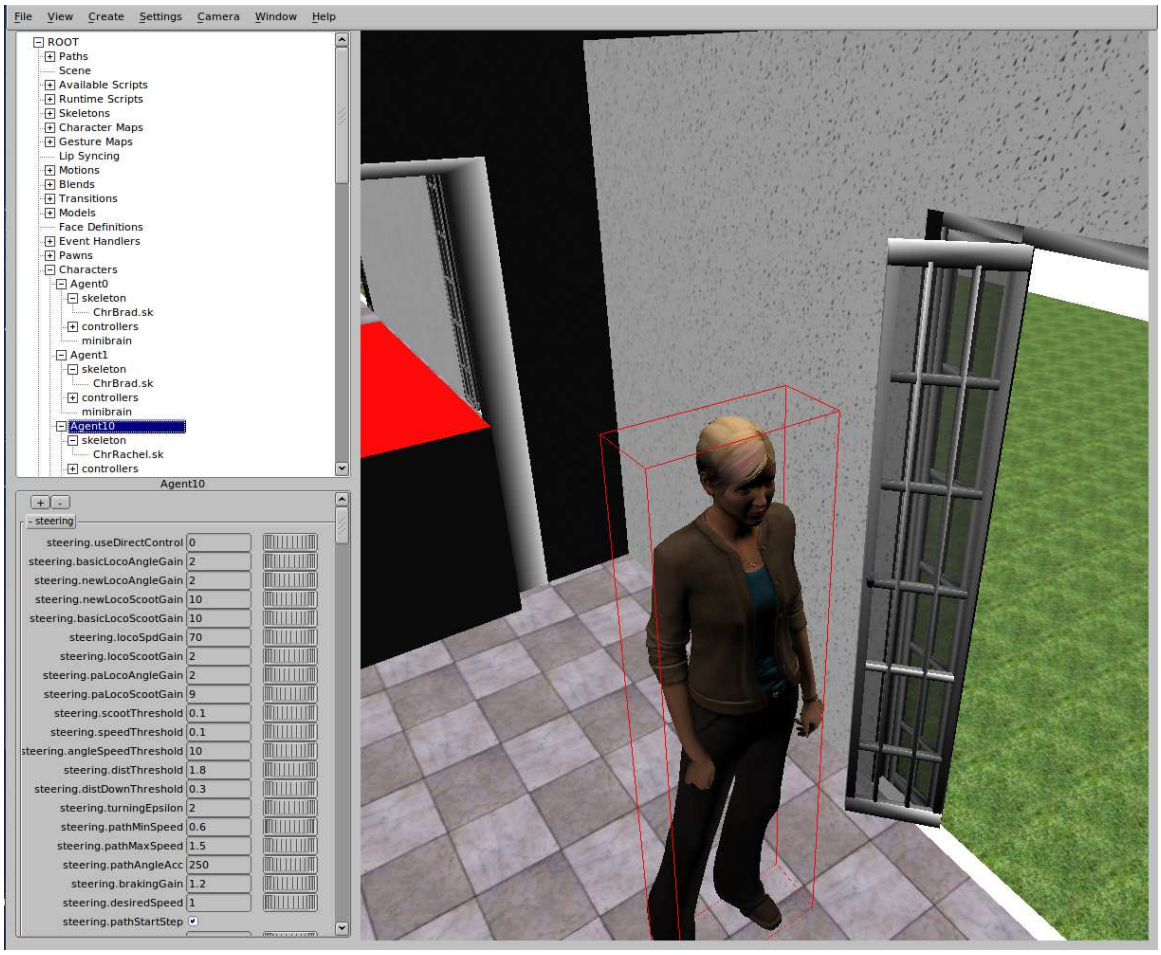

Figure 4.26: A 'rachel' type character

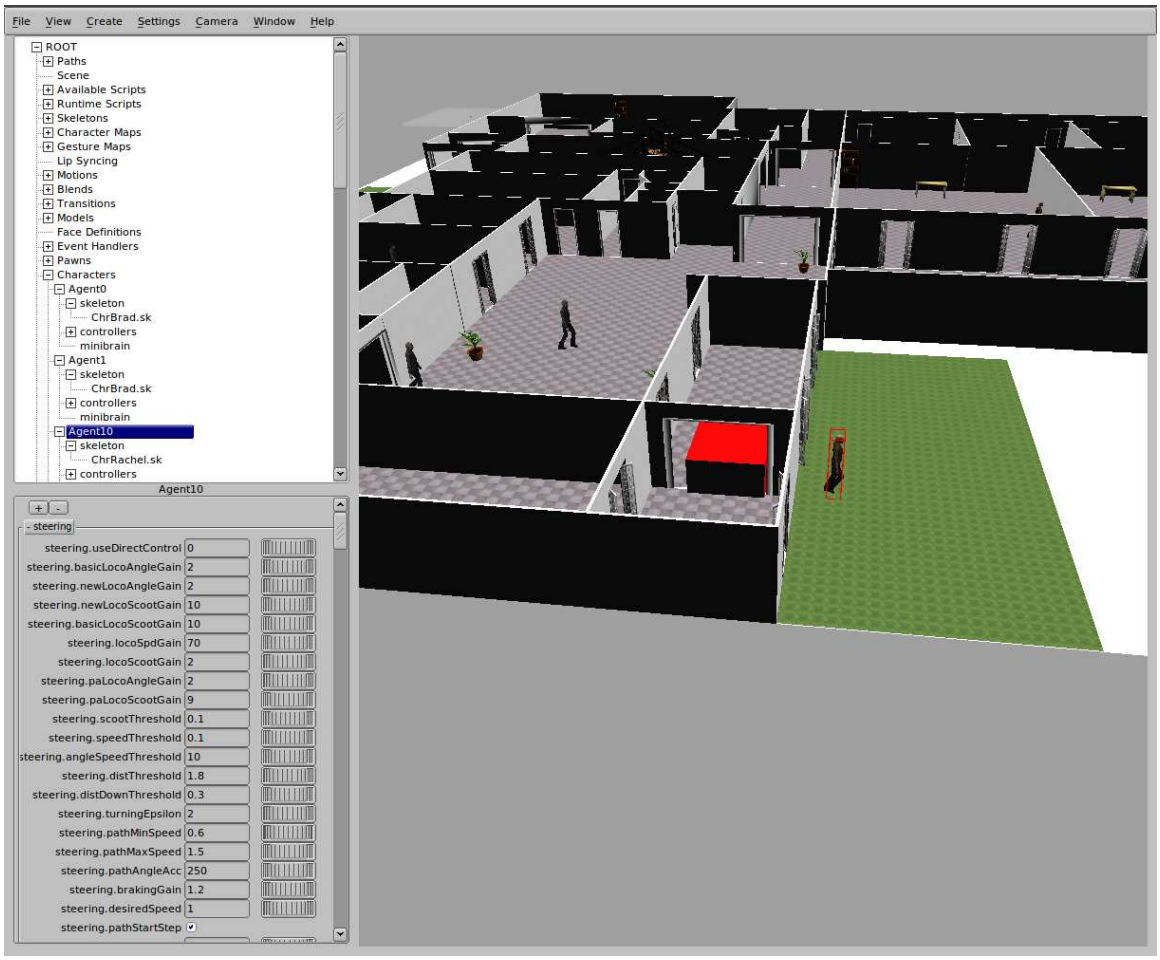

Figure 4.27: The agent escaping

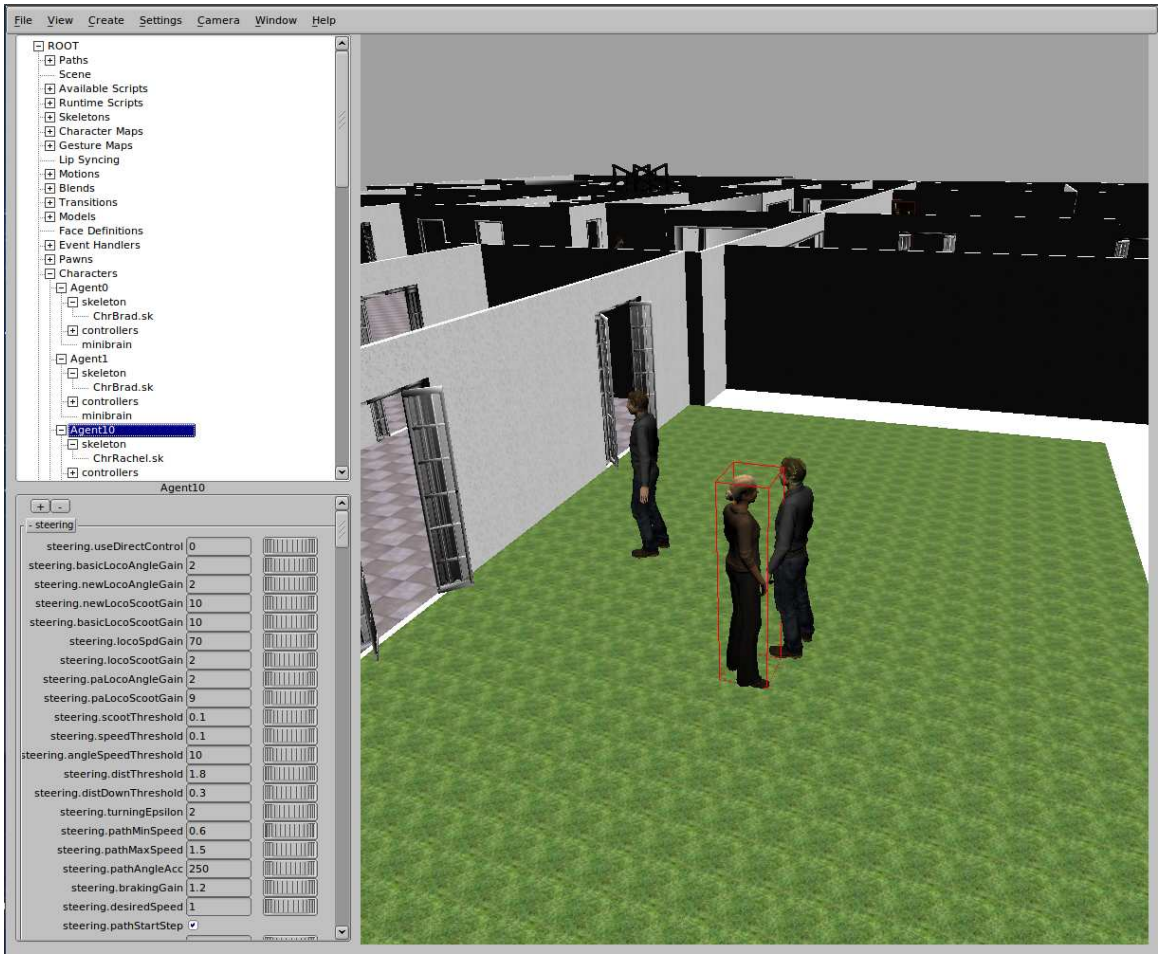

Figure 4.28: The agents have reached the final position

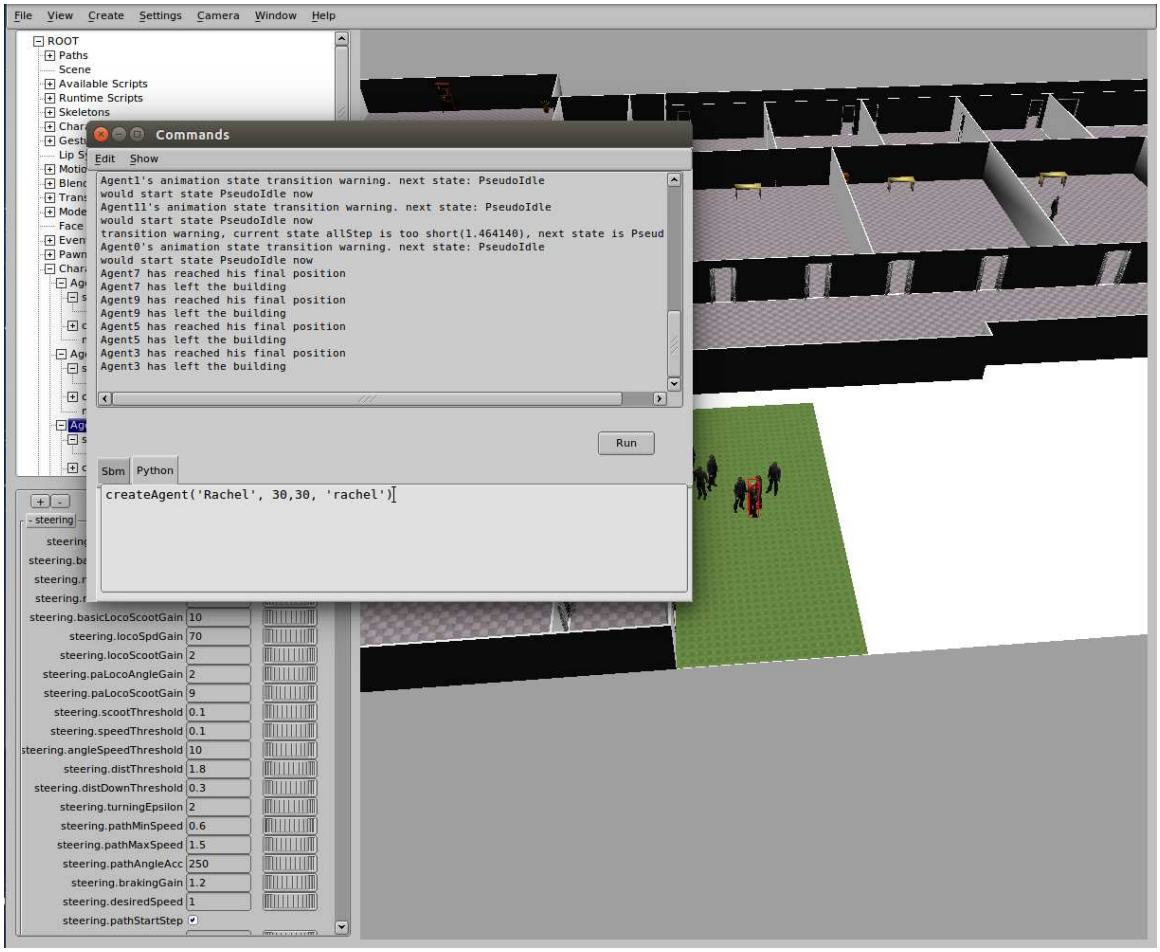

Figure 4.29: Creating an agent of type 'rachel'

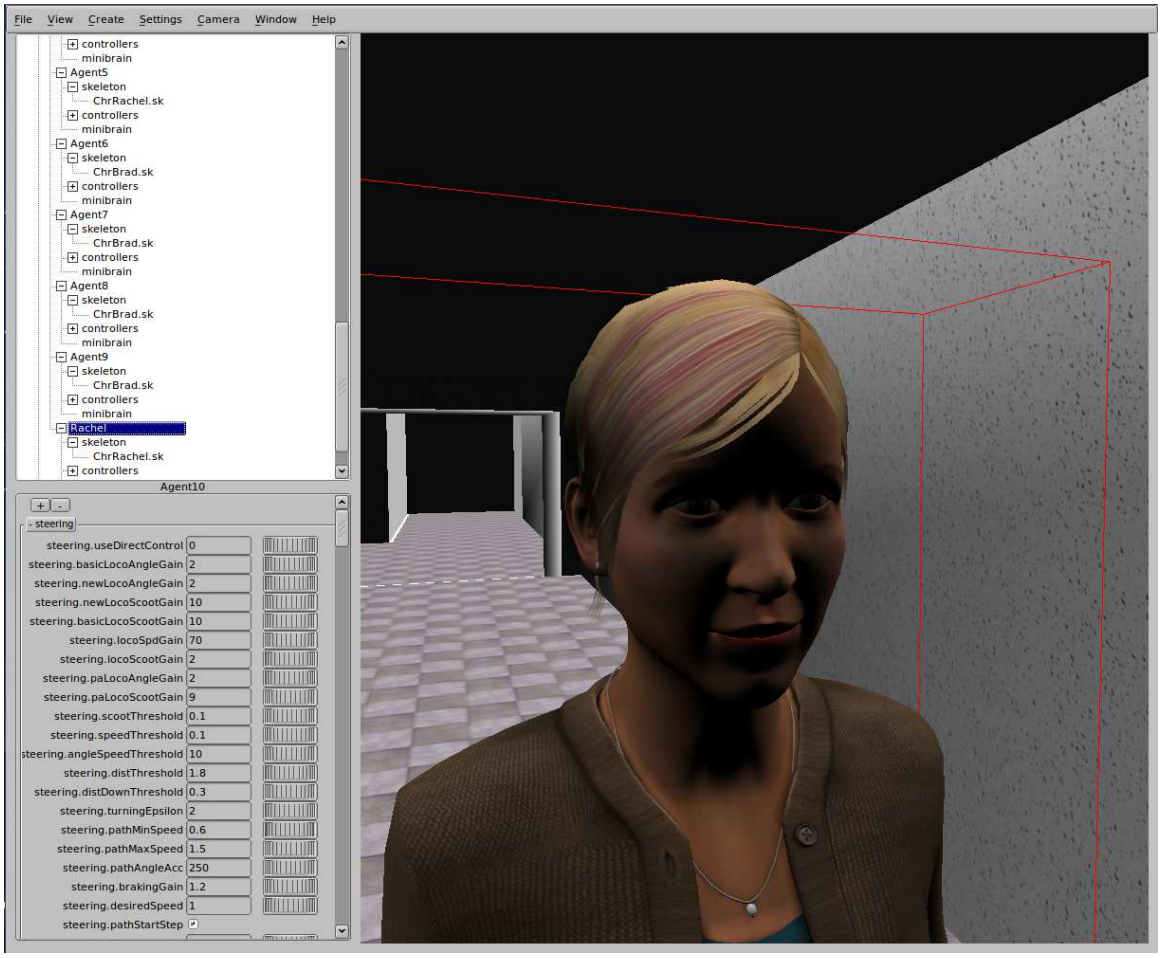

Figure 4.30: The agent created

4.9 Conclusions

In this chapter we have analysed the cases we have developed. Developing these cases has helped us in learning not only the SmartBody usage but BML also.

The modularity of our architecture allowed us to create a wide range of simulators making little changes in our main module, which enhances the user experience. We could evolve our simulator from the first stage to a more complex one in the last stage.

Taking these cases as an example, the user would be able to develop his own simulator with little effort.

CHAPTER 5

Conclusions and future lines

In this chapter we will describe the conclusions extracted from this project, the achievements and thinkings about future work.

5.1 Conclusions

Using UbikSim and SmartBody in a combined effort, we have created a powerful graphic interface for a social simulator of evacuation plans, SmartSim.

This project allowed us to interact with social simulators. We learned about their functionalities and how they operate. We needed to know they well in order to develop a graphic interface for one.

We learned to use tools like UbikSim and SmartBody creating the connections between them. They were very useful as they proved very accurate for the purposes we wanted them to accomplish. We learned how to take advantage of the most useful functionalities of them and we created an architecture that could make them interoperate.

We applied our knowledge in systems architecture to design our architecture. We created our project with a high modularity as parts of it could be used in future developments. This feature made us possible to develop a wide variety of simulators to highlight the possibilities of the system. The functionalities of the modules had been well defined in this documents and any user can take advantage of them to develop his own simulator without caring for most of the harshest parts.

5.2 Achieved goals

Create an scene This project has provided facilities for creating scenarios, that can be seamless exported to UbikSim and SmartBody. These scenarios can be designed highly detailed. The scenario creation process can be accomplished easily. We could also get the emergency position and represent it. This is described in Chapter 4.2.

Integrating UbikSim and SmartBody We can create an interface that allowed us the connections between SmartBody and UbikSim. In this interface we make the conversions between SmartBody and UbikSim standards, they do not use the same measurement units. This is detailed in Chapter 3.

Configuration file Our user can configure the simulation with editing a simple text file. The simulators created can be configured with a good variety of options. This allows us to test many cases without editing our simulators. This is explained in Chapter 4.3.

Create an agent Given its position, we can create an agent of our simulation. The agent

is very detailed both in appearance and behavior. A wide variety of animations can be performed by him. All of these can be performed using BML commands. This is described in Chapter 4.8.2.

Directing the agent to the exit We can retrieve the path to the exit of the building and steer the created agent to it. Using this feature we can create our social simulator of evacuation plans. The agent is steered using BML commands. This is detailed in Chapter 4.8.2.

Extending the number of agents We can extend the number of characters from our initial approach of only one character. We took the positions and paths and created our agents recursively. We only have to indicate the amount of agents we want to add to our simulation. This is further explained in Chapters 4.8.3 and 4.8.4.

Giving emotions to agents We managed to make our agents able to express emotions and a variety of behaviours. We achieved to give our agents a better approach to the human behaviour. We got a more realistic simulator. This is explained in Chapter 4.8.5.

Extending the types of agents We wanted to add some variety to our agents. We developed more types of characters with their own set of animations and behaviours. The type of the characters can be selected by the user. This is explained in Chapter 4.8.6.

5.3 Future work

There are several lines than can be followed to continue and extend features of this work.

In the following points some fields of study or improvement are presented to continue the development.

- We could develop a graphic interface for the scene control. Although the control has been simplified creating functions in the main module that execute functions in other modules, we still have to call python functions in the Commands Window of SmartBody Graphical User Interface. The user can make mistakes while introducing the commands. Pushing a button to play or stop the scene offers no possibilities to mistakes.
- Although SmartBody Graphic Interface offers a very good outcome, we could integrate SmartBody with a graphical engine such as Unity. The steps we must follow to

achieve the integration are described in the SmartBody manual. All the SmartBody functionalities could be exported, as our project would be.

- SmartBody has a mobile version. It is simpler than the desktop SmartBody, but using it could be useful for creating a simulator interacting with the position of the user, using the GPS of the mobile phone.
- SmartBody also has a web version in development. Currently, it seems to be broken as its development has been started recently. However, using a fully developed version would allow us to have a web infrastructure combining it with our UbikSim webapp.
- Although we have used SmartBody in a indoor case, it can be used in a wide range of applications. BML provides us a very friendly and powerful way to control a character. SmartBody can be used in many social simulators.

Bibliography

[1] R. Axelrod, "Advancing the art of simulation in the social sciences," *Japanese Journal for Management Information System*, vol. 12, no. 3, 2003.

[2] C. Sansores and J. Pavón, "Simulación social basada en agentes," *Revista Iberoamericana de Inteligencia Artificial*, no. 25, pp. 71–78, 2005.

[3] "Mosi-agil project website." `http://www.gsi.dit.upm.es/mosi/`.

[4] S. Kopp, B. Krenn, S. Marsella, A. N. Marshall, C. Pelachaud, H. Pirker, K. R. Thórisson, and H. Vilhjálmsson4, "Towards a common framework for multimodal generation: The behavior markup language," 2006.

[5] G. van Rossum, *The Python Language Reference*. Python Software Foundation.

[6] J. A. Botía, P. Campillo, F. Campuzano, and E. Serrano, "UbikSim website." `https://github.com/emilioserra/UbikSim/wiki`.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. Sun Microsystems.

[8] "Sweet home 3d website." `http://www.sweethome3d.com/`.

[9] S. Luke, G. C. Balan, and L. Panait, "Mason: A java multi-agent simulation library," 2003.

[10] E. Serrano and J. Botia, "Validating ambient intelligence based ubiquitous computing systems by means of artificial societies," *Information Sciences*, vol. 222, no. 0, pp. 3 – 24, 2013. Including Special Section on New Trends in Ambient Intelligence and Bio-inspired Systems.

[11] P. Davidsson, "Multi agent based simulation: Beyond socialsimulation," 2000.

[12] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan, "Extensible markup language (xml) 1.1 (second edition)," second edition of a recomendation, W3C, Sept. 2006. https://www.w3.org/TR/xml11/.

[13] J. Cassell, J. Sullivan, S. Prevost, and E. Churchill, *Embodied Conversational Agents*. The MIT Press, 2000.

[14] A. van Kesteren, A. Gregor, Ms2ger, A. Russell, and R. Berjon, "W3c dom4," first edition of a recomendation, W3C, Nov. 2015. https://www.w3.org/TR/dom/.

[15] "httplib module." `https://docs.python.org/2/library/httplib.html`.

[16] "json module." `https://docs.python.org/2/library/json.html`.

[17] "The javascript object notation (json) data interchange format, rfc7159," tech. rep., Internet Engineering Task Force, Mar. 2014. https://tools.ietf.org/html/rfc7159.

[18] "math module." `https://docs.python.org/2/library/math.html`.

[19] "io module." `https://docs.python.org/2/library/io.html`.

[20] "Configparser module." `https://docs.python.org/2/library/configparser.html`.

[21] "Wavefront files formats, version 4.0 rg-10-004," tech. rep., Wavefront Technologies Inc., 1993.

[22] M. Barnes and E. L. Finch, "Collada – digital asset schema release 1.5.0," specification, Sony Computer Entertainment and Khronos Group, Apr. 2008. https://www.khronos.org/files/collada_spec_1_5.pdf.

[23] "Smartbody python api." `http://smartbody.ict.usc.edu/HTML/smartbody.html`.

[24] P. Davidsson, "Agent based social simulation: A computer science view," *Journal of Artificial Societies and Social Simulation vol. 5, no. 1*, vol. 5, no. 1, 2002.

[25] E. Aarts and R. Wichert, *Technology Guide: Principles – Applications – Trends*, ch. Ambient intelligence, pp. 244–249. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[26] M. Thiebaux, S. Marsella, A. N. Marshall, and M. Kallman, "Smartbody: Behavior realization for embodied conversational agents," 2013.