Vrije Universiteit Brussel

FACULTY OF SCIENCE AND BIO-ENGINEERING SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

A Simulator for the Scenario Models Developed with ATTAC-L

Master thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in de Toegepaste Informatica

Dennis Geerts

Promoter: Prof. Dr. Olga De Troyer
Advisor: Frederik Van Broeckhoven

Academic year 2014-2015

Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN EN BIO-INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN

A Simulator for the Scenario Models Developed with ATTAC-L

Masterproef ingediend in gedeeltelijke vervulling van de eisen voor het behalen van de graad Master of Science in de Toegepaste Informatica

Dennis Geerts

Promotor:	Prof. Dr. Olga De Troyer
Begeleider:	Frederik Van Broeckhoven

VRIJE UNIVERSITEIT BRUSSEL • SCIENTIA VINCERE TENEBRAS •

Academiejaar 2014-2015

Abstract

Games have existed throughout the years in different forms, ranging from card games and board games to digital games, each having various subgenres and specific markets. Digital gaming is nowadays one of the biggest markets in the world. It has mostly been used in the context of entertainment. However, games induce effects that can also be useful for other purposes, such as training and learning. A person's concentration level is much higher when he is performing something he enjoys, like a game. Furthermore, video games can be so immersive that the player becomes unaware of its surroundings in real life. Therefore, there is an increasing interest in so-called serious games. The goal of serious games is to learn while playing. Failure in a virtual world does not have the same consequences as it would in real life. On top of that, even failure is a way to learn. However, the creation of digital games is time consuming and expensive. Therefore, in order to allow for the development of more serious games, tools that support and shorten this development process are needed. This thesis is a contribution to this goal.

The subject of the thesis is a simulator for the semi-automatic replay and verification of virtual scenario models, created with the ATTAC-L language specification. ATTAC-L is a Domain Specific Modeling Language (DSML) developed in the context of the Friendly ATTAC project, aiming to allow non-technical stakeholders to be involved in the modeling of scenarios for serious games. Virtual scenarios are modeled using the natural language based syntax of ATTAC-L and exported in a JSON format. The simulator we created takes this JSON as input and simulates the scenario in a 3D environment. A limited number of 3D environments are possible. Inconsistencies in the scenarios are detected and reported. The user can stepwise go through the scenario and explore different branches.

Keywords: Friendly ATTAC, ATTAC-L, games, serious games, Unity.

Samenvatting (in Dutch)

Spellen hebben bestaan door de jaren heen in verschillende vormen, variërend van kaartspellen tot bordspellen tot digitale spellen, met elk verschillende subgenres en specifieke markten. Digitaal spelen is tegenwoordig een van de grootste markten in de wereld. Het is meestal gebruikt in de context van amusement. Echter veroorzaken games effecten die van toepassingen kunnen zijn voor andere doeleinden, zoals opleiding en leren. Het concentratieniveau is veel hoger als een persoon iets doet waarvan hij geniet, zoals een spel. Bovendien, video games kunnen zo meeslepend zijn dat de speler zich onbewust wordt van zijn omgeving in het echte leven. Daarom is er een stijgende interesse in zogenoemde serious games. Het doel van serious games is om te leren tijdens het spelen. Falen in een virtuele wereld heeft niet dezelfde gevolgen dan falen in het echte leven. Daarbovenop, zelfs falen is een manier om te leren. Echter vergt het creëren van digitale spellen tijd en geld. Daarom zijn er middelen nodig die helpen met het ontwikkelen en verkorten van de ontwikkelings processen van serious games. Deze thesis is een contributie naar dit doel.

Het onderwerp van de thesis is een simulator voor de semi-automatische herhaling en verificatie van virtuele scenario modellen, gecreëerd met behulp van de ATTAC-L taal specificatie. ATTAC-L is een Domain Specific Modeling Language (DSML) ontwikkeld in de context van het Friendly ATTAC project. Het mikt op het engageren van niet-technische belanghebbenden in het modeleer proces van scenarios voor serious games. Virtuele scenarios worden gemodelleerd met behulp van de natuurlijke taal gebaseerde syntax van ATTAC-L en worden geëxporteerd naar een JSON formaat. De simulator die we gemaakt hebben neemt deze JSON als invoer en simuleert het scenario in een 3D omgeving. Een beperkt aantal 3D omgevingen zijn beschikbaar. Inconsistenties in de scenarios worden gedetecteerd en gerapporteerd. De gebruiker kan stapsgewijs door het scenario gaan en de verschillende takken verkennen.

Kernwoorden: Friendly ATTAC, ATTAC-L, games, serious games, Unity.

Declaration of Originality

Acknowledgements

I would like to express my gratitude towards my promoter Prof. Dr. Olga De Troyer and advisor Frederik Van Broeckhoven for the support, help, patience, and valuable feedback given.

I would also like to thank Roxana Rădulescu for supporting me throughout my studies, without her help and support I would not have made it up to where I am today.

Lastly, I would like to show my appreciation towards my family and friends. Their encouragements and support helped me to keep on going.

Thank you all.
Dennis Geerts

Contents

List of Figures

List of Tables

Listings

1 Introduction

In this first chapter, we start by describing the context of the thesis. Following, we explain the objective created and present an outline of the rest of the thesis.

1.1 Context

Games exist in different forms and in different genres. Digital gaming is one of the biggest markets in the world. It has mainly been used for entertainment purposes, however, new markets for games are expanding quickly. Serious gaming is one of these fields that is receiving increased interest. A serious game is a game where the main focus is not solely entertainment, but also has a different goal, which is mainly situated in learning or training. Serious games provide non-harmful environments where the player can train or educate himself. Learning experience can be gained both by achieving success or failure in these virtual scenarios. The advantage is that failure does not imply the same consequences as it would in real life.

Friendly ATTAC (2012) (Adaptive Technological Tools Against Cyberbullying) is an interdisciplinary research project aimed towards developing scenarios for serious games in the context of cyberbullying. The Friendly ATTAC project examines how serious games can serve a purpose in adjusting the behavioral aspect amongst youngsters related to cyberbullying.

ATTAC-L is developed in the context of Friendly ATTAC and is a graphical DSML (Domain Specific Modeling Language) that aims to ease the design of the serious game scenarios and allows involving non-technical people in the design process. Graphical DSMLs have an advantage over textual languages since the use of graphical notations makes it easier to grasp large amounts of information more quickly and ease the communication with non-technical people. Another benefit of the use of a DSML in the design process is that the specifications can be made unambiguous and this allows to semi-automatically generate the game based of the scenario. ATTAC-L uses a natural language-based syntax. Statements in the language read like sentences in natural language. A sentence is modeled by combining so-called bricks; each of these bricks contains one (or more) words. Combined bricks that form a sentence are called game moves. A typical game move consists of a subject, verb, and a direct object (e.g., player walks to Carl). Next to these bricks the language has three control structures: a choice, order independence, and concurrency. In each of these structures we can place game moves. A choice corresponds to an IF-statement or CASE-statement in programming languages. Order independence allows its encapsulated game moves to be executed in a random order. Concurrency allows all of its encapsulated game moves to be executed at the same time; this corresponds to parallel threads in programming languages.

1.2 Problem

Although, the use of natural language allows non-technical people, e.g., educators and scenario writers, to be involved in the design process, it is still difficult for that kind of people to imagine how the scenario could look in practice. In software engineering, fast prototyping is used to provide customers with a first system. Such a prototype allows them to understand, test, and verify the proposed system. This is what we also aim for: a fast prototyping tool that gives the opportunity to the scenario modeler to preview the different steps of a scenario before it is integrated in the actual game. In other words, we are looking for a tool that simulates the scenarios, hence the name of the tool: *ATTAC-L Simulator*. For this purpose we are faced with the challenge of translating scenarios modeled in ATTAC-L into scenarios executed in a game environment that are appealing and informative. As ATTAC-L only specifies scenarios, and not the game environment itself, it will be necessary to assume a number of aspects of the game environment in order to realize this simulation. This simulation includes: the simulation of game moves and control structures; providing errors when inconsistencies are detected in the scenarios.

1.3 Research Methodology

We use a research methodology based on Design Science (Peffers et al., 2007) to achieve our research goal. This research methodology provides principles, practices, and procedures to carry out research in disciplines oriented to the creation of successful artifacts. The methodology consists of 6 steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication.

Problem identification and motivation are explained in section 1.1 and section 1.2. The definition of the objectives for a solution are discussed in Chapter 4.

In section 4.2, the objectives are translated into a detailed list of requirements; these have to be accomplished to reach our solution. Next, the artifact (simulator) has been designed and developed. This is described in Chapter 5.

In order to demonstrate the simulator, a scenario is modeled in ATTAC-L and used as input for the simulator.

The user will be able to see his scenario simulated in a 3D environment and the choices made in the simulation (by the user) influence the outcome of the simulation. A preliminary evaluation was done by the WISE researchers involved in the Friendly ATTAC project. It is also available for others. Currently, the simulator was extensively tested for debugging purposes. Next, it was shared with the people involved. The simulator is able to simulate any scenario modeled in ATTAC-L containing the supported verbs. Communication is done by means of the thesis.

1.4 Structure of the Thesis

The first chapter is the *Introduction*, we explain the context of the thesis and continue by specifying the problem that has to be solved in this thesis as well as the research methodology followed. In the second chapter we look at *Related Work* regarding frameworks for story building. Chapter 3 provides the *Background* information, reviewing the major topics covered in this thesis: games, tools for implementing videogames and simulations, and ATTAC-L. In chapter 4 we discuss the *ATTAC-L Simulator* we made for the thesis. We start by explaining the aim and context, we then list the requirements necessary to run the simulator. After, we explain its features. We conclude this chapter by discussing the user interfaces. Chapter 5 discusses the *Implementation* of the simulator. We start this chapter by defining the key concepts in Unity, following we explain the system architecture and give the design. In chapter 6 we draw *Conclusions*; we start by giving a summary of the thesis and then explain the limitations and future work. We conclude with explaining the limitations in language support.

2 Related Work

2.1 Frameworks for Story Building

In this section we look at existing frameworks for story building. The resulting game (or simulation) will be automatically or semi-automatically created when the content –written or modeled in a specific format– is entered in these frameworks.

We describe five frameworks in this section. Based on their goals we split them in two groups; *Learning Programming* and *Prototyping Tools*.

2.1.1 Learning Programming

The main purpose of the frameworks discussed below is to teach the user programming. This is done via different methods like drag & drop and textual. To make it engaging and fun for the users, their results (games or simulations) can be shared with others.

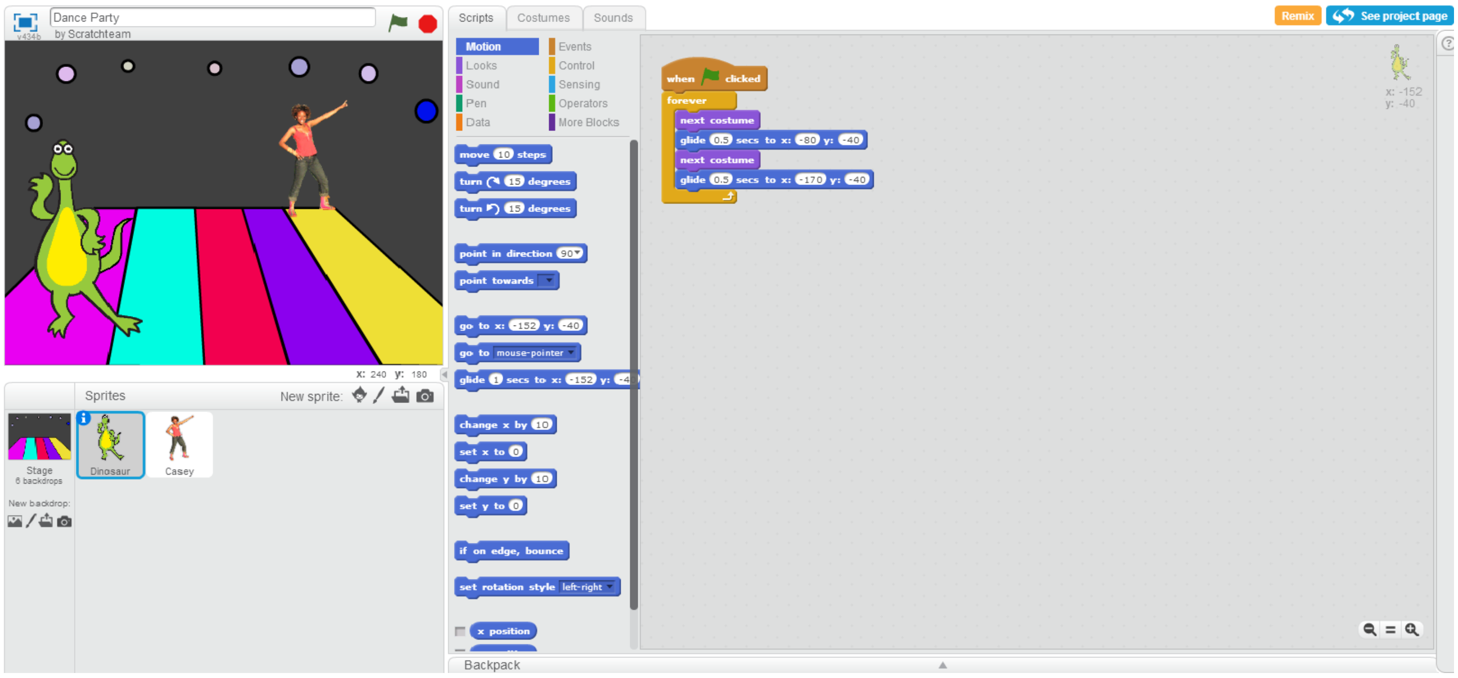
Scratch

Scratch (MIT Media Lab, 2015) is a "media-rich programming environment" which is created by the LifeLong Kindergarten research group at MIT's Media Lab. It allows you to create your own interactive stories, animations, and games. The main goal of Scratch is to give an introduction to programming for those who have no experience with programming at all. This is why they provide a drag & drop method of building blocks to code programs. One of the benefits of this way is that there are no possibilities for syntax errors.

Scratch 2.0 is made in Flash (2015), which uses the ActionScript programming language. The back-end of Scratch 2.0 is written in Python. Python is a programming language that gets intepreted rather than compiled, meaning that it gets executed immediately rather than translating it to another language.

Figure 2.1 shows the graphical programming interface environment of Scratch where users can drag & drop blocks to create a game (or simulation). The example used is a starter project featured on the scratch website, it is called "Dance Party"[1].

Although game creation was not the first aim of this project, it can be used for this purpose.



```
when [flag] clicked
forever
    next costume
    glide 0.5 secs to x: -80 y: -40
    next costume
    glide 0.5 secs to x: -170 y: -40
```

Figure 2.1: Programming Interface Environment of Scratch with Dance Party Example

[1]https://scratch.mit.edu/projects/10128067/, accessed 10/04/2015.

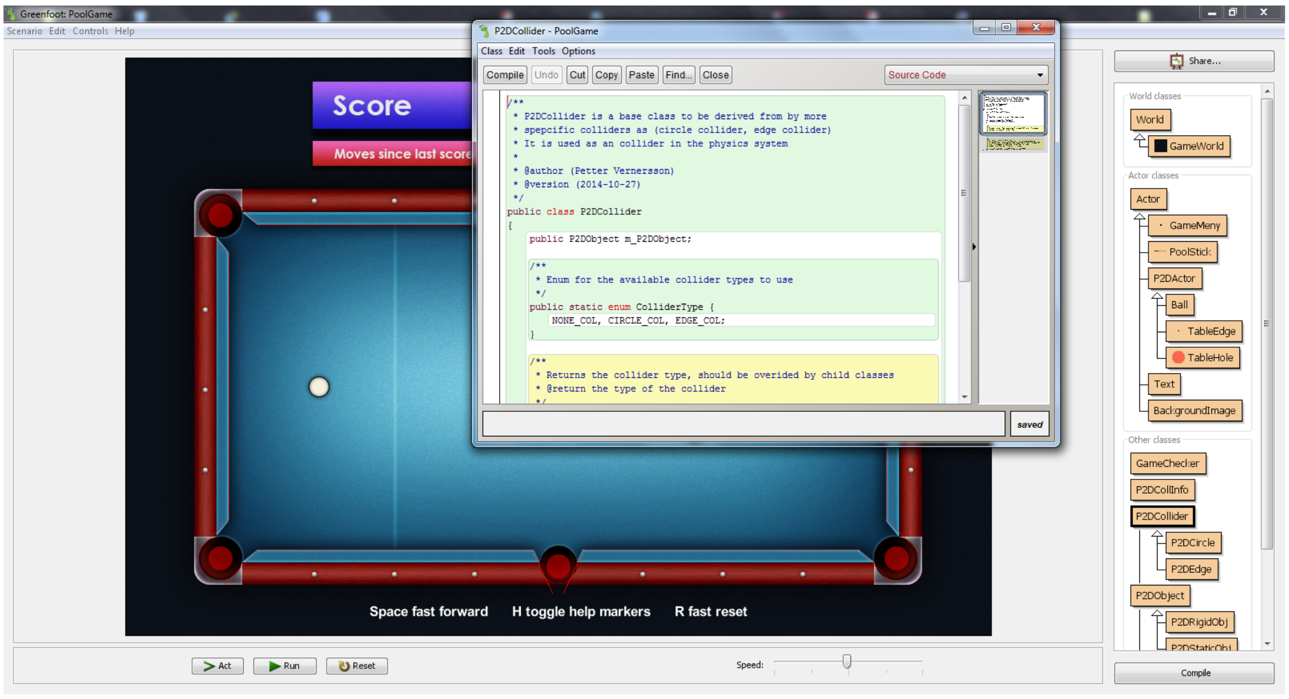
Greenfoot

Similar as Scratch. Greenfoot (2015) is an educational integrated development environment which has the purpose to teach its users programming. However, Greenfoot is for object oriented programming with Java. While Scratch provides a visual way for coding, Greenfoot uses the traditional way, textual, but combines this with a graphical, interactive output. The target age group is also older than the one Scratch aims at. It is created by the same developers who made BlueJ (2015).

Greenfoot is build in Java, as well as the games developed with it. Java is a language that has to be compiled to machine code for it to be executed.

Figure 2.2 shows the programming interface of Greenfoot. The example is "PoolGame", a popular scenario taken from their website[2].

Just like with Scratch, its main purpose is to teach the user how to program. However, the game creation aspect can be used to create (serious) games.



```
/**
 * P2DCollider is a base class to be derived from by more
 * spepcific colliders as (circle collider, edge collider)
 * It is used as an collider in the physics system
 *
 * @author (Petter Vernersson)
 * @version (2014-10-27)
 */
public class P2DCollider
{
    public P2DObject m_P2DObject;

    /**
     * Enum for the available collider types to use
     */
    public static enum ColliderType {
        NONE_COL, CIRCLE_COL, EDGE_COL;
    }

    /**
     * Returns the collider type, should be overided by child classes
     * @return the type of the collider
     */
```

Figure 2.2: Programming Interface Environment of Greenfoot with Pool-Game Example

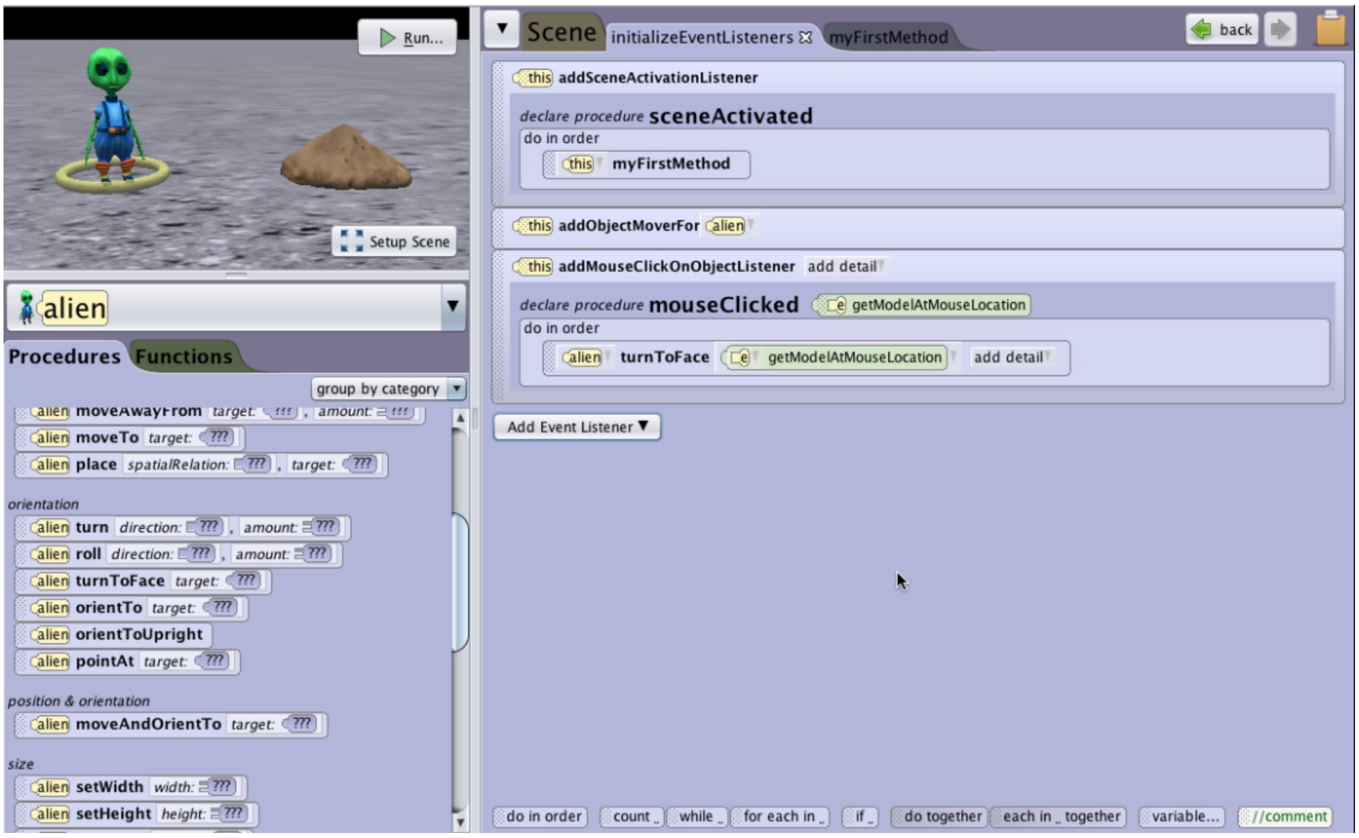
[2]http://www.greenfoot.org/scenarios/12394, accessed 10/04/2015.

Alice

Alice (2015) is a free 3D object-oriented programming environment. In the same way as Scratch, it has the objective to teach children an introduction to programming. It allows to create an interactive game or video. The user can put 3D objects in a virtual world, and animate them in Alice by creating a program. Programming in Alice is done via drag & drop of graphical elements to create a program, just like in Scratch.

The latest version of Alice is 3.1, it is different from its predecessors in the way that it is designed with the purpose in mind of making the users transition towards Java at the end. Alice is written in Java, just like Greenfoot.

Figure 2.3 shows the programming interface of Alice 3. The example is a video on "Collision Detection" taken from the tutorial videos[3] on the Alice website.



```
alien moveAwayFrom target: ??? , amount: ???
alien moveTo target: ???
alien place spatialRelation: ??? , target: ???
orientation
alien turn direction: ??? , amount: ???
alien roll direction: ??? , amount: ???
alien turnToFace target: ???
alien orientTo target: ???
alien orientToUpright
alien pointAt target: ???
position & orientation
alien moveAndOrientTo target: ???
size
alien setWidth width: ???
alien setHeight height: ???
```

```
this addSceneActivationListener
  declare procedure sceneActivated
  do in order
    this myFirstMethod
this addObjectMoverFor alien
this addMouseClickOnObjectListener add detail
  declare procedure mouseClicked  e getModelAtMouseLocation
  do in order
    alien turnToFace  e getModelAtMouseLocation  add detail
```

Figure 2.3: Programming Interface Environment of Alice 3

[3]http://www.alice.org/3.1/materials_videos.php, accessed 11/04/2015.

2.1.2 Prototyping Tools

In this section we describe frameworks where the focus is aimed towards creating the narrative. When creating the educational part in serious games, assistance from one or more field experts is necessary. The collaboration between field experts and game developers is difficult. The frameworks as discussed below aim to ease this collaboration.

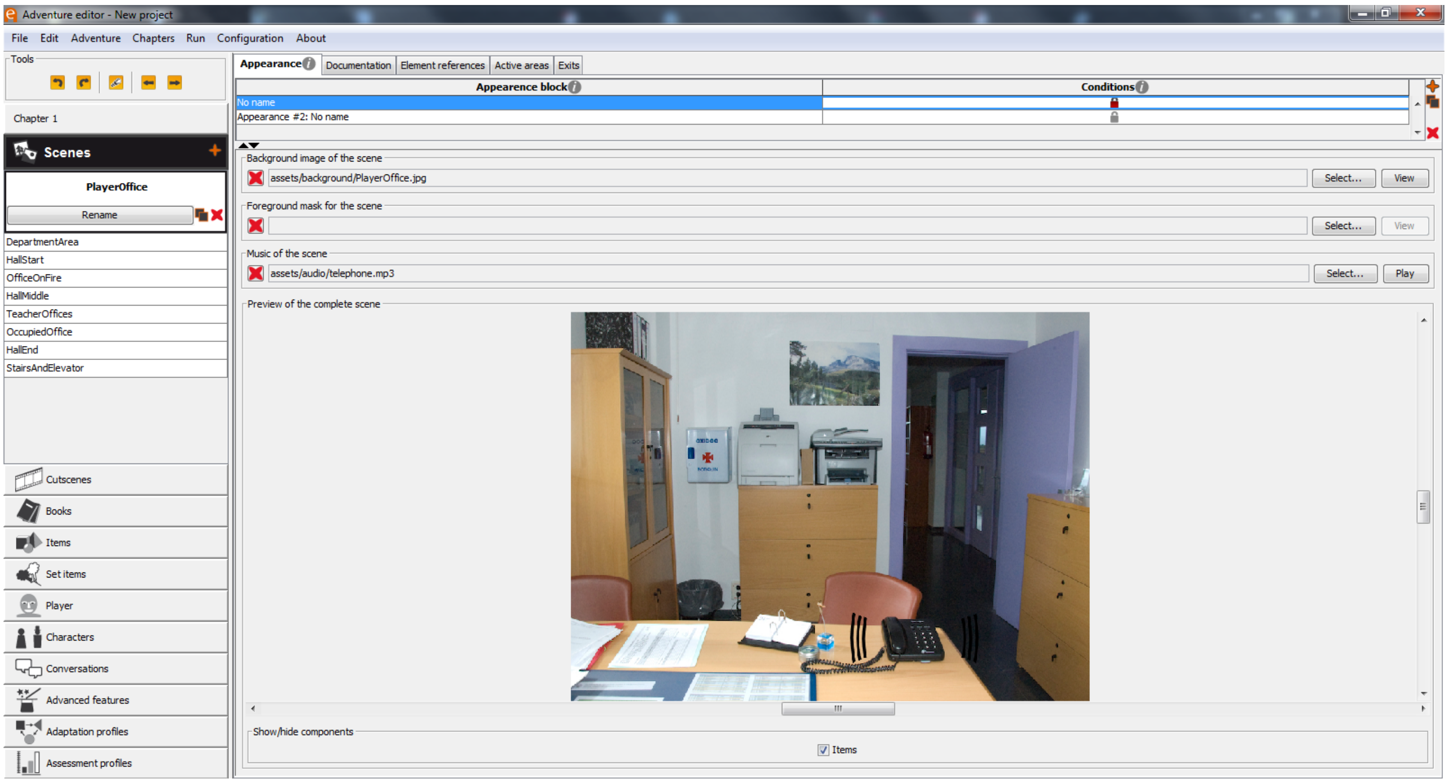
e-Adventure

e-Adventure (2012) is a research project that aims to integrate educational games and simulations in VLEs (virtual learning environments). Their goals are: development cost reduction, instructor involvement in the development process, and a white-box model aiding in the production of the games (Torrente et al., 2010).

In e-Adventure, everything is accomplished with a point & click method. There is no programming knowledge required to create a game or simulation. The user imports images or videos, selects (or imports) characters, creates conversations, etc. Every scene consists of one (or more) image(s), clickable areas can be defined with certain behavior added to it (e.g. going to a next scene, starting a conversation).

Figure 2.4 shows the programming interface of e-Adventure. The "PlayerOffice" scene in the editor is shown in the "Fire Protocol" game[4].

[4] `http://e-adventure.e-ucm.es/course/view.php?id=29\&topic=1`, accessed 11/04/2015.



Background image of the scene: assets/background/PlayerOffice.jpg

Foreground mask for the scene:

Music of the scene: assets/audio/telephone.mp3

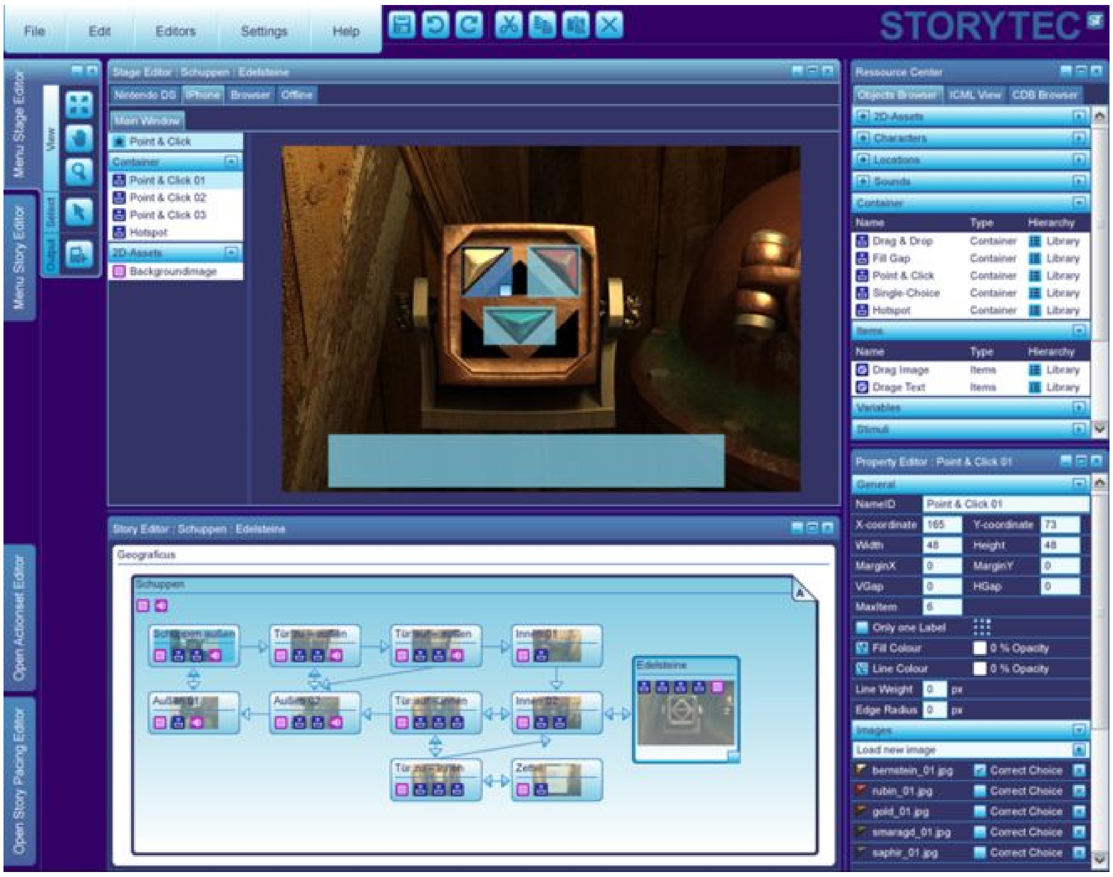
Figure 2.4: Programming Interface Environment of e-Adventure

80Days

The 80Days project (M. D. Kickmeier-Rust et al., 2008) is an European research project that aims to meld educational subjects with the immersiveness and entertainment of an attractive computer game. One game has been implemented in the project, it is called "Feon's Quest", and it is aimed towards teaching geography for 12 to 14 year olds (M. Kickmeier-Rust et al., 2011). Feon is an alien that needs to discover and learn about other planets, this time he landed on planet Earth. The player flies with the UFO and explores different locations on Earth together with Feon.

To create a story, they used their Story Editor tool. This allows for the creation of a story without writing a programming language. They decided to extend the Story Editor with an authoring tool which allows non-programmers to create entire games. This authoring tool is StoryTec (Gobel et al., 2008), it is a rapid prototyping environment for the authoring process of interactive applications like learning games. The 80Days project enhanced StoryTec so that it would be able to cope with the addition of adaptation and personalization in learning games. StoryTec can be used for example to structure the game's narrative, use videos or images, create interactive behaviour, and change parameters for the adaptivity.

Figure 2.5 shows the StoryTec and Story Editor framework.



Name	Type	Hierarchy
Drag & Drop	Container	Library
Fill Gap	Container	Library
Point & Click	Container	Library
Single-Choice	Container	Library
Hotspot	Container	Library

Name	Type	Hierarchy
Drag Image	Items	Library
Drage Text	Items	Library

NameID	Point & Click 01		
X-coordinate	165	Y-coordinate	73
Width	48	Height	48
MarginX	0	MarginY	0
VGap	0	HGap	0
MaxItem	6		
Only one Label			
Fill Colour	0 % Opacity		
Line Colour	0 % Opacity		
Line Weight	0	px	
Edge Radius	0	px	

bernstein_01.jpg	Correct Choice
rubin_01.jpg	Correct Choice
gold_01.jpg	Correct Choice
smaragd_01.jpg	Correct Choice
saphir_01.jpg	Correct Choice

Figure 2.5: StoryTec and Story Editor framework, retrieved from `http://storytec.de/index.php?id=49\&L=1` on 11/04/2015.

3 Background

In this chapter we describe the background of our work. Our goal is to develop a simulator for narrative-based scenarios for serious games. Therefore, we will start by introducing *Games*, on which serious games are based, and then we will give a more profound explanation of *Serious Games.* Continuing, we will explain how games are in general implemented. The most common way to implement a game is by using a game engine. Therefore, we will discuss *Game Engines* in more detail. We do so because this is also the obvious technology to implement our simulator. We discuss well-known game engines and compare them in order to decide which one to use in our work. Afterwards the language *ATTAC-L* and its associated tool for modeling scenarios are described, as ATTAC-L is the language for which the simulator will be developed.

3.1 Games

Gaming is one of the biggest markets in the world, generating a revenue of over \$100 billion in 2014[1]. Games have been used in various forms throughout the years as a source of entertainment. Major game types that immediately

[1]http://www.statista.com/statistics/278181/video-games-revenue-worldwide-from-2012-to-2015-by-source/, accessed 18/03/2015.

pop up to mind are computer games, board games, card games, mind games, puzzles, etc. This thesis focuses on computer games, also called digital games, so henceforth when referring to games, computer games are meant.

Since computers are part of our daily life, are becoming more and more accessible for everybody, and since people like to play, it does not come as a surprise that computer games are also starting to play an important role in everyday routine. Apart from the purpose of entertainment, games have the potential to offer much more. For example, in multiplayer games we can see that social (-and learning-) aspects play a huge role. The possibility to have a virtual life where someone can safely practice social skills is something only possible in multiplayer games (Kolo & Baur, 2004).

Gregory (2009a) define (most) 2D and 3D video games as *soft real-time interactive agent-based computer simulations.* We will explain this concept further:

Simulations
A game world is modeled mathematically so that computers can manipulate it. It would be too costly to include every small detail of the world, so it is simplified. These kind of models are called *simulations.* Mostly, these are simulations of real world situations, but also fictional worlds can be simulated.

Agent-based
Most 3D computer games have a lot of objects, some examples are: vehicles, weapons, NPCs (non-playable characters), etc. Agents are objects that can interact with each other in one way or another.

Interactive
A game world state can change over time when the story progresses or events occur. Different from a pure simulation, in a game the player provides input (i.e., interacts with the game) in order to change the game world state and the game has to respond in real-time, these are the *interactive temporal simulations.*

Soft Real-time
A real-time system has time requirements. An example of an important time requirement is the requirement of the screen to update at least 24 times per second. This will ensure that what the player perceives, is (an illusion of) fluid motion. When the system is able to deal with a missed time requirement and no severe consequences will occur, it is called a *soft real-time system.* An example of a soft real-time system is a game engine. An example of a *hard real-time system* is an aircraft control

system. In such a system, time requirements that are not respected could result in dangerous or unacceptable situations.

Juul (2010) gave a brief but good definition of a game using six aspects that are part of a game: rules, variable, quantifiable outcome, values assigned to possible outcomes, player effort, player attached to the outcome and the negotiable consequences. Klabbers (2003) gave a broader view on games in general, saying that games are social systems. They are also models of existing or imagined social systems, shaped by the players. He then made a distinction between games for fun and games for scientific purposes.

3.1.1 Game Genres

The genre of a game is based on its gameplay, as opposed to books for example, where the genre is based on their setting. Games can belong to one specific genre, or they can be a mixture of genres. What follows is a list of the most common game genres. We also specify per game genre some of their subgenres and include a famous game title.

Action games
This genre is characterized by physical activities such as accuracy, reaction time, and quick reflexes. Some subgenres of the action genre are: fighting games, pinball games, MOBAs (Multiplayer Online Battle Arenas), and platform games. League of Legends (2015) is an example of a MOBA action game.

Shooter games
The shooter genre focuses on combat with weapons. The most famous subgenres are: first person shooters and third person shooters. Counter Strike: Global Offensive (2015) is an example of a first person shooter game.

Adventure games
Adventure games are characterized by puzzle-solving and exploration. Examples of subgenres are: text adventures, graphic adventures and visual novels. Sherlock Holmes: Crimes & Punishments (2014) is an adventure game.

Role Playing games
Role Playing Games (RPG) are games where the player takes the role as one (or more) of the characters defined in a world. A common denominator between these kinds of games is leveling up by gaining experience

points. Some examples of subgenres are: MMORPGs (Massive Multiplayer Online RPG), Action RPGs, and Sandbox RPGs. An example of a MMORPG is World of Warcraft (2015).

Simulation games
Simulation video games aim to achieve the goal of simulating a real or fictional reality. Some subgenres are: Life Simulation, and Vehicle Simulation. An example of a Simulation game is The Sims (2014).

Strategy games
Strategy games allow the player to control units and create tactics in order to achieve victory. This type of game is probably the most closely related to board games. The trend in this genre has shifted from turn based systems to real-time systems. Some subgenres are: Real-time Strategy, Tower Defense, Turn-based Strategy. An example of a Strategy game is: StarCraft II (2014).

Sport games
The main subject of a Sports game is a real life sport. One player controls one person or an entire team and plays against another person or a team (controlled by artificial intelligent agents or another player). Some examples of subgenres are: Racing, (Physical) Sports. An example of a Sports game is: FIFA (2014).

3.1.2 Narrative-based Games

There is a notable difference between a narrative and a story. Corman (2013) defined a story to have "limited power" because they usually follow the same pattern (beginning, middle and end). There are limited engaging possibilities for the listeners because mostly the story is about the people involved. In case of a narrative, Corman notes two main differences, they are open-ended and they invite the listener to participate. We have different types of narratives: linear narrative form, interactive narrative, and emergent narrative (Lindley, 2005). When a *linear narrative form* is used, the narrative is told in a predefined order. When a user is able to change or choose the plot based on different decisions or actions it is called an *interactive narrative. Emergent narrative* studies how interactions among low level elements can lead to a well-defined high level narrative form that is not represented in the observed system.

Dickey (2006) notes that narrative plays a more prominent role in the following game genres: RPGs, action games, and adventure games. If we look at genres like shooters and sports, narrative is typically not that important.

He then continues by specifying two game mechanisms that are commonly used to convey narrative to the player. These are backstory and cut scenes. Backstory tells the player what happened before the game started. This can be presented to him in different ways (e.g. textual on the packaging or as an intro video before starting the game). Cut scenes typically happen at certain intervals during the game play when the player accomplished something. They are small intermezzos that explain more context regarding the narrative.

An example of a game where the narrative plays a big role is the Mass Effect (2012) trilogy. It is an action third person shooter RPG. The game consists of a lot of interactions (see Fig. 3.1 for an example of an interactive conversation) between the player and NPCs. Every choice made will influence the rest of the game. When the player has completed Mass Effect 2 and wants to play Mass Effect 3, he can choose to import a file. This file will set over 1000 variables so the story continues from the decisions made in Mass Effect 2[2]. This is an example of a game which we previously defined as an interactive narrative.

In the next section we will explain what serious games are and discuss a classification system.

3.1.3 Serious Games

"The only source of knowledge is experience."

(Albert Einstein)

Certain skills can only be acquired by performing them. This is where serious games come into play, learning while playing. Small children learn by playing. Kids are much more motivated when doing activities they deem to be fun (e.g. playing a game), instead of doing work for school. Shute et al. (2009) describe a solution to the methodological obstacles (e.g. keeping a game engaging while making it educative) which are present when trying to combine learning (schoolwork) and having fun (playing a game). They use a two-stage approach for their solution. The first stage is used to collect academically relevant data from students by using engaging games. This data can then be used to approve the claim that important knowledge and skills can be learned while playing. The second stage consists of adapting existing, or designing new, engaging games that will monitor and support the students' acquisition of academically relevant skills.

[2]`http://www.engadget.com/2010/06/15/interview-bioware-casey-hudson-on-the-making-of-mass-effect-2/`, accessed 04/04/2015.

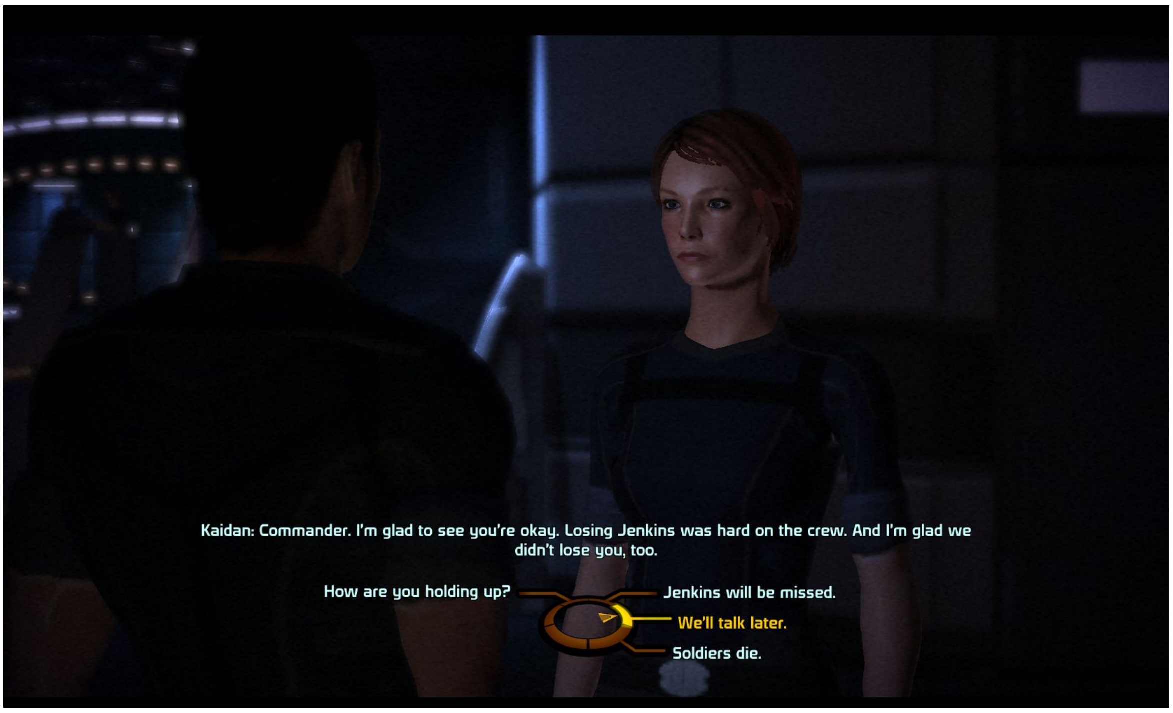

Kaidan: Commander. I'm glad to see you're okay. Losing Jenkins was hard on the crew. And I'm glad we didn't lose you, too.
How are you holding up?
Jenkins will be missed.
We'll talk later.
Soldiers die.

Figure 3.1: Example of an interactive conversation in Mass Effect 3 (retrieved from `http://www.giantbomb.com/profile/dalai/blog/when-a-dialogue-tree-has-too-many-leaves/76768/`).

Note that serious games are not solely targeting kids or youngsters, they can also play an important role in a business environment (Corti, 2006). Therefore, serious games are defined as games used for purposes other than entertainment alone (Susi et al., 2007). Serious games can be used to replace or complement traditional class-based learning, but they can also be used for situations where learning or training in real life would be too dangerous or too expensive. In serious games, the player can train himself in a non-harmful environment. These environments can be made as close as possible to real life scenarios. The advantage is that failure in such a virtual environment does not have the same consequences as it would have in real life, and in addition, failure is also a way to learn. Typical examples of situations that would have detrimental consequences when even the smallest mistake is made can be found in, but not limited to, the military domain and the medical field (surgery).

An example of a serious game is *Pulse!!* (2007), see Figure 3.2. Before performing operations, the learner has to go through training scenarios which teach him the skills to perform a successful operation. The operation scenarios in the game are real-life cases, which need to be solved by using current medical techniques.

Michael & Chen (2005, Part 2) mention six markets for serious games and provide detailed information about each of them. The markets mentioned are military games; government games; educational games; corporate games; healthcare games; political, religious, and art games.

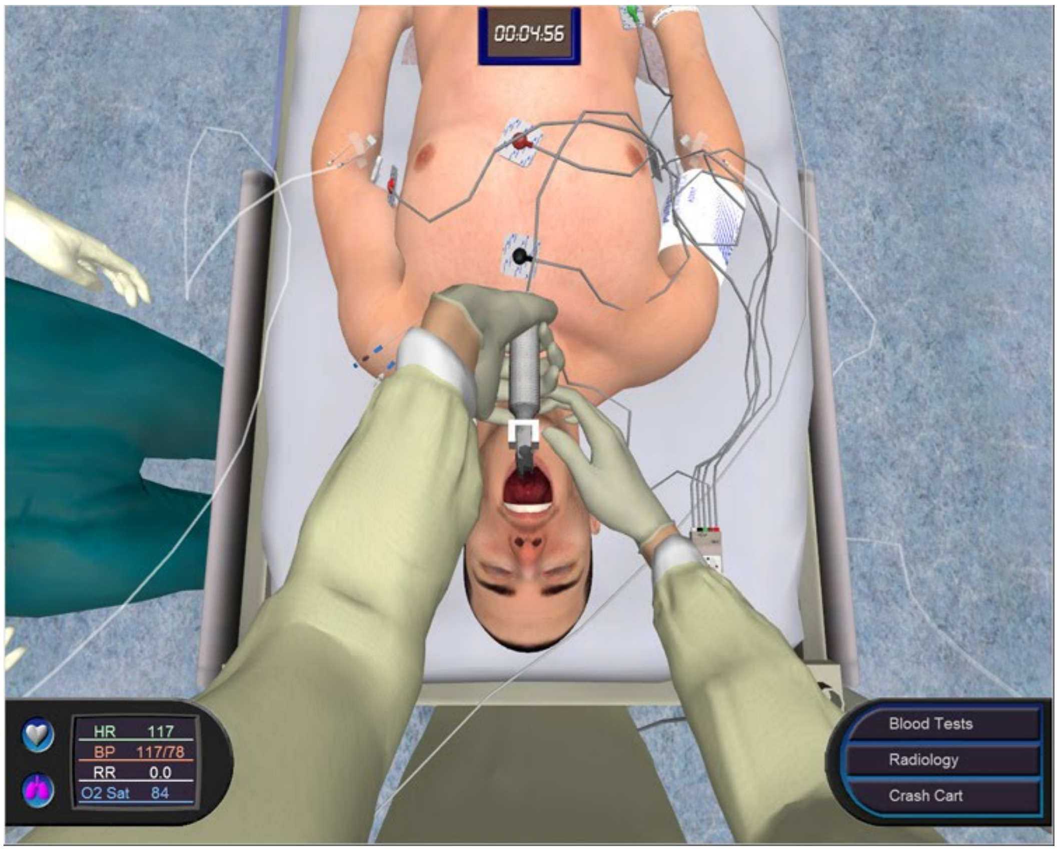


Figure 3.2: Scene from the serious game Pulse!! (retrieved from `http://www.paristechreview.com/2013/03/26/qui-peur-serious-games/`).

3.1.4 Classification for Serious Games

Djaouti et al. (2011) propose a classification system, the G/P/S (Gameplay, Purpose, and Scope) model, which classifies serious games based on their "seriousness" and "game-related" characteristics. We explain these three components briefly:

- The *gameplay component* gives information regarding the kind of gameplay used in the game. This is the *'how it is played'*.
- The *purpose component* gives information regarding the intended usage of the scenario when the game was developed. This is the *'for what'*.
- The *scope component* refers to whom the game is intended. This is the *'who uses it'*.

We will give an example of the use of this classification, by using the game *Pulse!!*. We will first give a short description of the game and then discuss the G/P/S model.

Pulse!!

Pulse!! was the first of a kind virtual game for training health care professionals in clinical skills. 3D graphics create an interactive virtual environment where civilian or military health care professionals practice their skills in order to respond better when those incidents occur.

- Gameplay:
  - Type: Game-based.
  - Goals: Avoid, Match.
  - Means: Manage, Select.
- Purpose:
  - Purposes: Training.
- Scope:
  - Markets: Healthcare.
  - Target audience: 25-35 / 35-60, Professionals.

Djaouti et al. (2011) provide more examples on how the G/P/S classification is used on other serious game titles.

Figure 3.3 shows the relationship between video games, serious games and serious gaming given by Djaouti et al. We will start with the three main components. Video games have already been discussed in section 3.1, we can say that video games contain only a *"game"* dimension. Serious games are the topic of this section. These games contain a *"serious"* dimension on top of a *"game"* dimension. The last one, serious gaming is a much larger area. This area covers all the serious games as well as the games that have been "purpose shifted".

A regular video game can become a serious game in two different ways, these ways are called *Purpose Shifting* and *Modding*, as can be seen in Figure 3.3. An explanation of both is provided:

Purpose Shifting

There is no reason why entertainment games cannot be used in a serious gaming context. If this happens we say that the game has been *purpose shifted* and the serious role of the game is determined relative to the perspective of the player. An example is *Surgeon Simulator* (2013). The game was released in 2014 where the player takes on the role of a surgeon, having to perform operations. In this game every finger has to be controlled separately, when picking up tools or performing manoeuvres. It has to be noted that this mechanism is intentionally implemented to be very difficult to handle, this with the intent to provide a dark humor to the game. Surgeon Simulator is not created with a serious purpose in mind; the hospital is just the stage where an entertaining scenario is held in. Of course, Surgeon Simulator can be used with a serious purpose in mind. This is what is defined as *Purpose Shifting*: the process of creating specific scenarios, which are used to serve a purpose different from entertainment, or a purpose the developers envisioned for the game.

Modding

When a game is purpose shifted, no software modification is involved. When software modification is possible, the person is able to create his own levels, scenarios, weapons, etc. The result is called a *Modification*, or simply a "mod". A famous example is *Day-Z* (2015) which is a mod from *ARMA 2* (2009), this mod turned out to be so successful it was later transformed into a standalone game.

ARMA 2 is a first person shooter while Day-Z is a survival horror game in a MMO (Massive Multiplayer Online) environment. This shows the power of modding, it can transform the entire genre of a game. It is not far-fetched that this way, any games (which allows modding) can be modded to include a "serious" purpose.

3.2 Tools for Implementing Videogames and Simulations

Games can be created in different ways. In general, a game is build using a game engine. A game engine provides the core functionality typically needed in games. Developing a game using a game engine requires good programming skills and knowledge about game developing. There also exist frameworks that allow composing games in an easier way, usually using drag

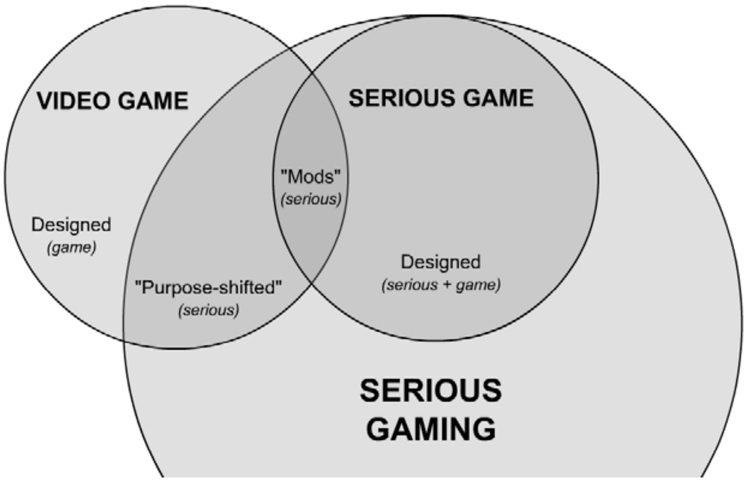


Figure 3.3: The relationship diagram of Video, Serious Games and Serious Gaming (Djaouti et al., 2011)

& drop functionality. The type of games and their complexity that can be created with these frameworks is usually limited.

Because of the flexibility that game engines offer, we decided to use a game engine to implement our simulation. For this reason, we discuss game engines into more details. We conclude this section with a comparison of well-known game engines and our choice for the implementation of our simulation.

3.2.1 Game Engines

A game engine exists of an extensive collection of tools that help with building a game. Some of these tools are: level/scene editor, game asset import tools, animation system, scripting languages or an API to program the game with.

Gregory (2009a) notes that a data-driven architecture is what makes the difference between a game and a game engine. He proposes to use the term game engine for software which can be extended and used as a basis for a wide variety of other games without the need of major adjustments. However, as he states, game engines are fine-tuned for a specific game genre. This used to be more the case in the earlier days of game engines. Nowadays, this phenomenon is not as clear-cut anymore as it used to be, but small benefits (e.g. performance) will still be gained by using an engine for the specific game genre it was originally designed for. Figure 3.4 shows some Game Engines with regards to their reusability. Gregory (2009a) also talks about the technological requirements needed for some of the most frequent game genres.

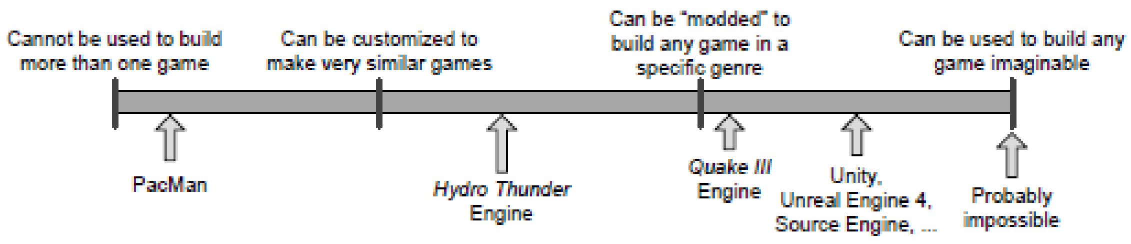

Cannot be used to build more than one game
Can be customized to make very similar games
Can be "modded" to build any game in a specific genre
Can be used to build any game imaginable
PacMan
Hydro Thunder Engine
Quake III Engine
Unity, Unreal Engine 4, Source Engine, ...
Probably impossible

Figure 3.4: Reusability of Game Engines (Gregory, 2009a)

Creating a game with a game engine will require coding, however it is just the game logic that needs to be coded. This is because game engines are pieces of software which bundle a collection of underlying technology, some important ones are: rendering, physics, networking, audio, etc. Rendering deals with the image generation of 2D or 3D models in a scene. All these technologies and more will be discussed below.

Runtime Game Engine Architecture

Figure 3.5 depicts all the major architectural runtime components of a typical modern 3D game engine (according to Gregory (2009a)). In what follows, we will explain only those components that are of interest for this thesis. Gregory (2009a) explains all of these components in more detail.

Hardware

This layer is a representation of the system on which the game will run. Some platforms, but not limited to, are Windows, Linux, Mac, PS4, XboxOne, and iOS.

Platform Independence Layer

Most game engines provide the option to publish the game to multiple platforms. The platform independence layer ensures that the game's behavior remains consistent across all the possible platforms.

Resource Manager

The resource manager provides a uniform interface to access game assets, textures, materials, fonts, etc.

Rendering

There is not a single accepted way to design a rendering engine. However, a layered architectural design that exists out of four components is common. These are: low-level renderer, scene graph/culling optimizations, visual effects, and front end.

The goal of the low-level render is to render geometric primitives as fast and as rich as possible. Nothing else is taken into account (e.g. viewpoints). This is what the scene graph/culling optimizations takes into account, it provides what primitives are visible and pass it on for rendering.

Game engines support a lot of different visual effects. Some examples of visual effects used in the thesis are: particle effects (for a water fountain), light mapping (pre-computed light on –mostly static– objects like benches), and dynamic shadows.

Front end is important in our thesis since it is used for the in-game main menu and other graphical overlays.

Collision and Physics

The term physics is also known as rigid body dynamics in the game industry. Collision detection is usually tightly coupled with physics. Most game engines use 3$^{\text{rd}}$ party SDKs (Software Development Kit) as physics engine.

Skeletal Animation

Skeletal animation deals with 3D character meshes which exist of bones, represented by vectors. When the characters (bones) move, the vertices are correspondingly updated.

Human Interface Device (HID)

The input of the player can be obtained from different devices: keyboard and mouse, joystick, controllers, etc. The HID is often made so that it can separate the low-level details of the input interface on a specific platform from the high-level game controls.

Gameplay Foundations

Gameplay is a concept that covers the action in game, the rules, abilities, objects, objectives, etc. The purpose of this layer is to form a bridge between the low-level systems in the engine and the code.

The games object model is the collection of all different objects that compose it, including: the player, NPCs, weapons, vehicles, etc.

The scripting system is one (or more) scripting language(s) the game engine supports to code specific rules and content. Thanks to the scripting system, recompiling and relinking the game executable is not necessary. It will allow changing the game logic and content by modifying or reloading the code.

Game-Specific Subsystems

In this layer the features of the game itself are implemented. There are a lot more game-specific subsystems than those shown in Figure 3.5, but some worth mentioning are: game-specific rendering, player mechanics, game cameras, and AI (artificial intelligence).

Game-specific rendering deals with the rendering of e.g. the terrain and water. Player mechanics deal with animation and movement. Game cameras deal with the cameras; this is what the player will see on the screen, most of the game engines take care of this by providing scripts which do most of the work. And last but not least, AI is used to implement realistic behaviors and decision making. In the thesis, it is used for path finding within a navigational mesh, in order to allow the characters to automatically find their own path to the desired destination.

For people, object collision is something we do not think about. It is just natural, e.g. two cars cannot magically pass through each other, they will collide against each other. However, in the digital world, it has to be explicitly stated that objects cannot pass through other objects.

Physics in games is used to compute for example the movement speed of (large) objects under the influence of gravity. When a ball is kicked up, it has to come down eventually.

Networking deals with the multiplayer aspect on games, how players can connect to each other worldwide.

Last but not least, audio deals with all the sounds the game produces at certain intervals. Detailed information on all these technologies can be found in (Gregory, 2009b, Part 3), (Eberly, 2010), (Caltagirone et al., 2002).

In what follows, we will list and briefly discuss a couple of the well-known game engines.

Well-known Game Engines

To compare different available game engines, we defined criteria to evaluate the different game engines and which are important to consider in our decision. These criteria are: licensing, platform, engine and coding (i.e., support for programming). We present the overview in a table. For each engine considered, the table also contains some general information. Table 3.1 provides the overview.

We considered the following game engines: Unreal Engine (2015) developed by Epic Games[3], CryEngine (2015) developed by Crytek[4], Source developed by Valve[5], and Unity developed by Unity Technologies[6]. All of these engines are well known and well documented. Unreal Engine and Unity are known to have an excellent (community) support.

As far as it concerns licensing, all the listed engines are free for non-commercial use. Things get more complicated when the engine will be used for commercial use. Unreal Engine handles a 5% royalty on the gross revenue per game, only if that revenue per calendar quarter is above $3000. CryEngine launched EaaS (Engine as a Service) which is a subscription to the CryEngine. It costs $9.90 per month and there are no royalties involved. Other options with other terms are available, including purchasing a license with the full source code[7]. Source engine handles the no royalty method but it forces the developers to release their game on Steam[8] (not limited to Steam only). Unity offers a lot of different plans, dependent on choosing the personal edition or the professional edition[9], however a royalty bonus is never to be paid.

All of these game engines support multi-platform. All of them are able to publish to Windows, Mac (exception of CryEngine), Linux, iOS, PS4, and Xbox One. Unreal Engine and Unity also offer support for Virtual Reality (Oculus Rift, Gear VR). It has to be noted that Unity has more deployment platforms than listed in the table[10].

Next we talk about the different technologies supported by the engine. All of the listed game engines support the same graphic engines, DirectX[11] and OpenGL[12]. OpenGL is the graphics API which is usable on Windows, Linux, and Mac, unlike DirectX, which is Windows exclusive. Collision detection and rigid body dynamics are in the game development community known as

[3]`http://epicgames.com/`, accessed 22/03/2015.

[4]`http://www.crytek.com/`, accessed 22/03/2015.

[5]`http://www.valvesoftware.com/`, accessed 23/03/2015.

[6]`http://unity3d.com/`, accessed 25/03/2015.

[7]`http://cryengine.com/get-cryengine/ce-games`, accessed 22/03/2015

[8]`http://store.steampowered.com/`, accessed 26/03/2015.

[9]`http://unity3d.com/get-unity`, accessed 26/03/2015.

[10]`http://unity3d.com/unity/multiplatform`, accessed 25/03/2015.

[11]`https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx`, accessed 26/03/2015.

[12]`https://www.opengl.org/`, accessed 26/03/2015.

physics. Unreal Engine, Source, and Unity support PhysX[13], a physics engine by Nvidia[14]. CryEngine has its own implementation of a physics engine. As far as it concerns audio, they all support the FMOD audio engine. Each of these game engines has their own implementation regarding networking.

Last but not least, we consider the support for programming. Unity is the only game engine which does not allow (natively) to program in C++, but rather in C#, JavaScript or Boo. Also visual scripting is supported. Visual scripting is the concept of letting developers manipulate program elements graphically instead of textually writing code. Previously, we gave some examples of visual programming languages (see 2.1). In Unreal Engine it is called Blueprints, CryEngine calls it Flow Graph, and Unity does not support this natively but this can be acquired by using third party assets from the Unity Asset Store[15].

(Petridis et al., 2010) compared over 100 game engines using a framework they suggested for high fidelity serious games. They went into more detail on some of the big players as discussed above. They note that one of the most important elements in the creation of serious games is the graphical quality.

Unity is the game engine chosen for the implementation of the thesis. The low hardware requirements, the excellent documentation & (community) support, the multi-platform deployment possibilities and the fact that it has been used in the context of serious games[16] multiple times, led to this decision.

3.3 ATTAC-L

In this section we describe ATTAC-L. ATTAC-L is a domain specific modeling language for specifying narrative-based serious games. This section consists of three parts. Firstly we will explain the purpose of ATTAC-L. Secondly we describe the language itself. Thirdly we talk about the interpretation of game moves. An example at the end of this section will illustrate how ATTAC-L is used to specify a complete narrative.

3.3.1 Purpose

ATTAC-L is a DSML (Domain Specific Modeling Language) (Luoma et al., 2004) for specifying narrative-based serious games. It is developed in the

[13]`http://www.geforce.com/hardware/technology/physx`, accessed 26/03/2015.
[14]`http://www.nvidia.com/content/global/global.php`, accessed 26/03/2015.
[15]`https://www.assetstore.unity3d.com/`, accessed on 26/03/2015.
[16]`https://unity3d.com/industries/sim`, accessed on 26/03/2015.

context of cyber bullying to specify scenarios about cyber bullying situations for serious games against cyber bullying. However, the language can also be applied in other domains than cyber bullying. DSMLs are graphical languages that use the vocabulary of the application domain. Abstractions are provided to make the solution specification easier and more accessible for domain experts and end users. A graphical representation has a clear advantage over a textual representation in the fact that a large quantity of visual information can be understood much faster than its textual counterpart.

Using a DSML will offer great benefits to non-technical people, as they can be involved in the design process of the serious game. For instance, understanding the causes and impacts of cyber bullying require knowledge from different fields. Therefore there is a need to include social scientists, health psychologists, computer scientists, domain experts, and game designers (Van Broeckhoven & De Troyer, 2013) into the development process. Since these persons are not always technically schooled, the DSML should abstract from technical and implementation details and be easily understandable by all parties involved. A DSML has the advantage over natural language that it is unambiguous and it will allow for the (semi-)automatic creation of the actual game.

More information on DSMLs suggested in the context of game development can be found in (Marchiori et al., 2011; A. W. Furtado & Santos, 2006; A. W. B. Furtado & de Medeiros Santos, 2006).

3.3.2 Language Description

ATTAC-L makes use of a natural language like syntax; the sentences used will appear as natural language but they have a strict syntax. The sentence structure will usually exist of: subject, verb, and a passive object. There is also a possibility to add an indirect object. Examples of these will be given while describing the language.

Bricks

An action or a step defined in ATTAC-L is called a *game move*. Bricks are the basic building blocks to define game moves. They contain a word (object, verb, noun, etc.). Figure 3.6 shows some examples of bricks.

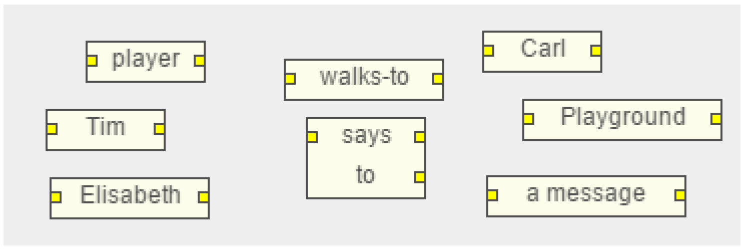


Figure 3.6: Bricks

A game move is composed by connecting bricks to each other in such a way that they form a non-ambiguous sentence. The structure of a game move has to follow strict grammatical rules. It always has to contain a subject, verb, and a passive object. An indirect object can also be introduced in a game move, although only when the others are present as well. An example of a game move containing a subject ("player"), verb ("walks-to"), and a passive object ("Carl") is given in Figure 3.7 and a game move with an indirect object ("Elisabeth") in Figure 3.8.

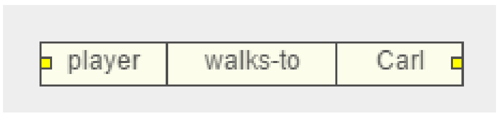


Figure 3.7: Regular Game Move

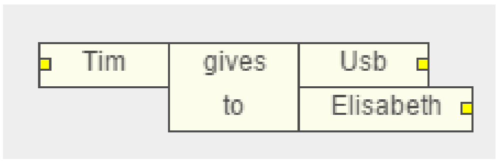


Figure 3.8: Regular Game Move with Indirect Object

Value Brick

Sometimes a passive object brick requires some more details to give it exact meaning. Consider the example given in Figure 3.9; there are no details given about what is exactly being said. To cope with this problem, a special "value brick" is used. This brick contains one (or more) property name(s) and its value(s). Figure 3.10 provides a game move for the action (verb) "says to", that also specifies what exactly is said. Note that a shorter version of Figure 3.10, by removing the passive object ("a message"), is also valid ATTAC-L. This is only the case when the sentence will not become ambiguous when the passive object is left out (as is not the case with "says to").

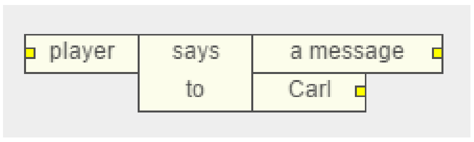


Figure 3.9: Unclear Game Move

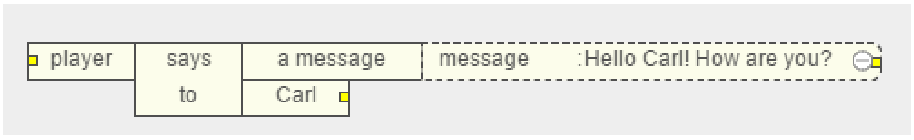


Figure 3.10: Clear Game Move by using a Value Brick

Passive Voice

It is also possible to model a game move using a passive voice sentence construction. Figure 3.11 shows a normal game move and Figure 3.12 shows the passive voice of the same game move. Both are interpreted in the same way.

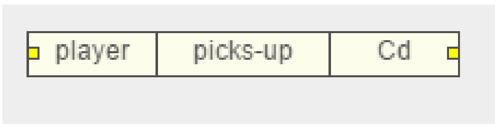


Figure 3.11: Normal Game Move

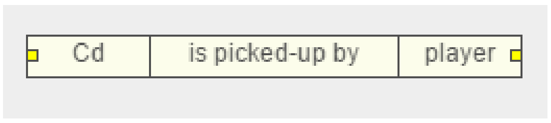


Figure 3.12: Game Move in Passive Voice

Inline Definition

NPCs can be dynamically assigned a name and used later to reference to the dynamically selected character. An example of this is given in Figure 3.13. This game move will assign "X1" to a random (available) NPC in the game world. In all the following game moves X1 can be used to refer to that character.

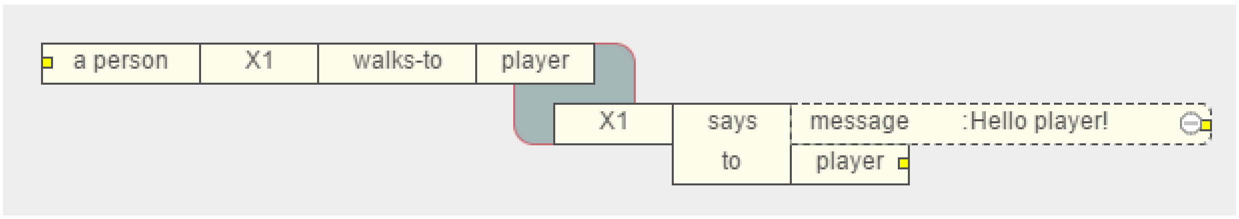


Figure 3.13: Inline Definition

Sequence Brick

A story consists of several game moves. These can be connected to eachother by using a sequence brick as illustrated in Figure 3.14. An example of two game moves connected is given in Figure 3.15.

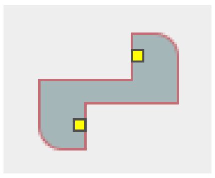


Figure 3.14: Sequence Brick

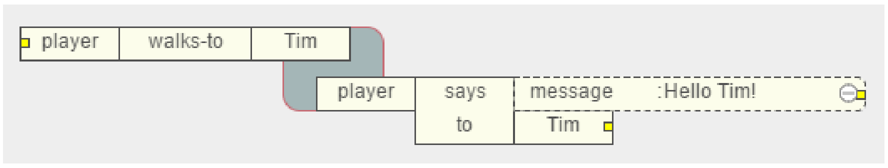


Figure 3.15: Two Connected Game Moves

Choice Brick

A choice brick as illustrated in Figure 3.16, corresponds to an IF-statement or CASE-statement in programming languages. Using a choice brick, the story can be split in two or more different game moves (or even entire plots). A decision has to be made (either by the player or by the game itself) which game move to follow.

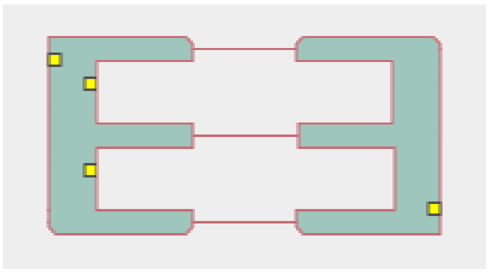


Figure 3.16: Empty Choice Brick

Figure 3.17 shows an example of a choice brick with three choices.

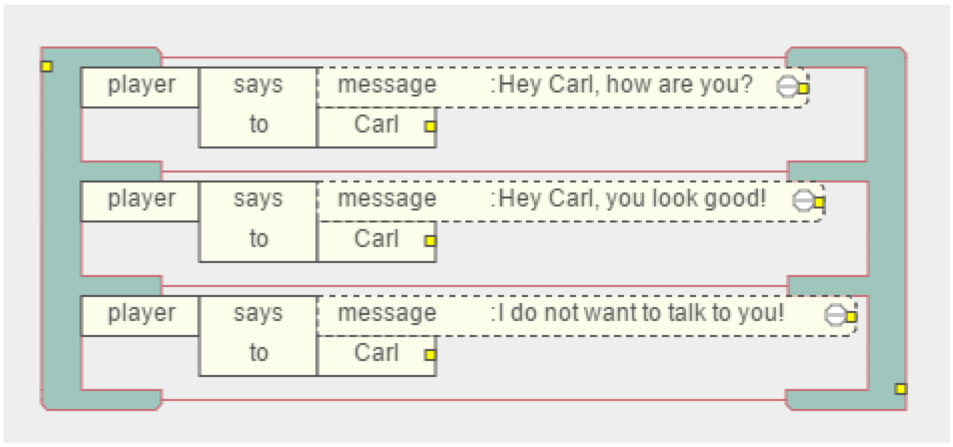


Figure 3.17: Choice Brick with Three Choices

Order Independence Brick

An order independence brick, as illustrated in Figure 3.18, encapsulates two or more different game moves. All of these game moves are executed one after the other but in a random order.

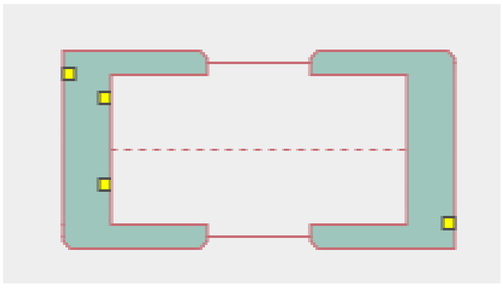


Figure 3.18: Empty Order Independence Brick

Figure 3.19 shows an example of an order independent brick with three game moves. Note that the order in which the player says something to Carl, Elisabeth, or Tim is not specified. However, he will say “Hello” to all three of them.

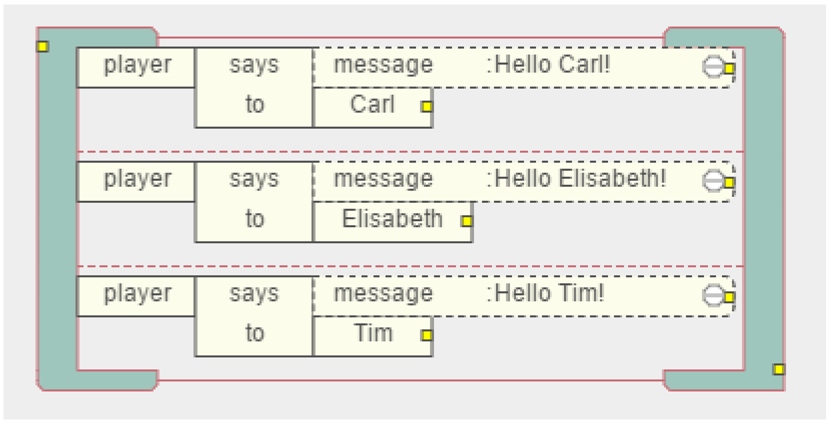


Figure 3.19: Order Independence Brick with Three Game Moves

Concurrency Brick

A concurrency brick, as illustrated in Figure 3.20, corresponds to parallel threads in programming languages. All the game moves encapsulated in a concurrency brick are executed at the same time (concurrent). However, practically this is not always feasible, this is why we opted to implement concurrency as order independence without the randomness factor (from first to last).

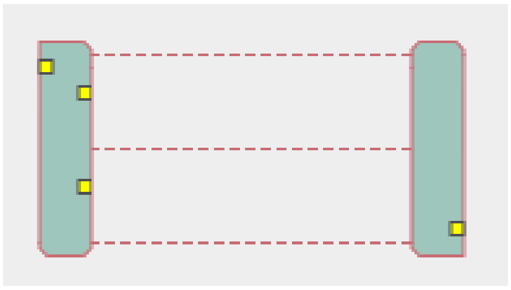


Figure 3.20: Empty Concurrency Independence Brick

Figure 3.21 shows an example of a concurrency brick with three game moves. Note that it is not always possible to visualize three people walking to (possibly) three different locations. This is why we opted for the order independent implementation. The order of execution will be from top to bottom, one after the other.

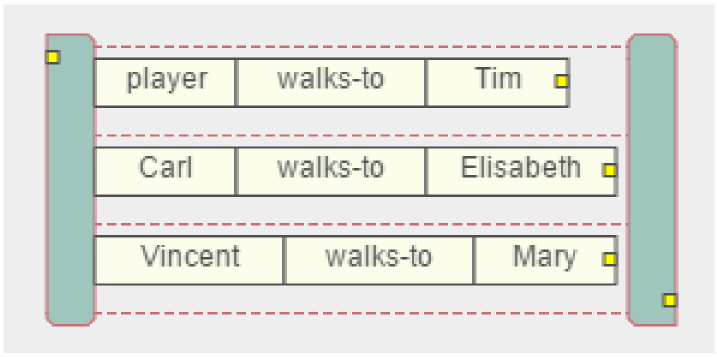


Figure 3.21: Concurrency Brick with Three Game Moves

3.3.3 Language Export

To allow for the (semi-)automatic creation of the actual game, the narrative must be converted into a data structure that is readable and understandable by a computer. ATTAC-L models can be exported to JSON (JavaScript Object Notation) format. Such a JSON file will be the input for our simulator, which will visualize the narrative in a visual way. Therefore, we explain the structure of such a JSON file.

Let us start by explaining the basic JSON structure with a simple story, shown in Figure 3.22.

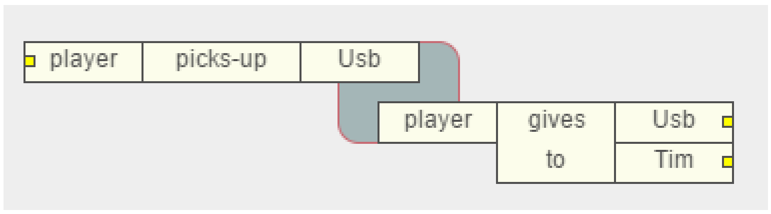


Figure 3.22: Simple Story Export Example

The translation of the story shown in Figure 3.22 is translated to:

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g1": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "gives",
                            "_next": [
                                {
                                    "word_": "Usb"
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Tim"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "gives to",
                    "_direct": "Usb",
                    "_indirect": "Tim"
                }
            }
        },
        "g0": {
            "_next": "g1",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "picks-up",
                            "_next": [
                                {
                                    "word_": "Usb"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "picks-up",
                    "_direct": "Usb"
                }
            }
        }
    }
}
```

Listing 3.1: Basic JSON Structure of a Simple Story

As can be seen in Listing 3.1 the JSON consists of two main parts. Before going into detail on these two, it is important to know that every game move is labeled as “gx”, with the x being its unique number. We will call this the game move label.

The first part is the "_entries" array. This contains references to all the game move labels where the story starts. In most cases this will only contain one label. If in rare occasions it has more than one, they are executed one after the other. In our example it contains only “g0”.

The second part of the JSON is the "_flow" object. This object contains all game moves which are present in the story. As said earlier, every game move is identified by its unique label. This label contains another JSON object which consists of three parts. The first part is optional and contains the next game move label to be executed if this one is completed. In our example, g0 has a next (being g1) but g1 has no next since the story ends there, so it is omitted. The second part contains the "_expr" (expression) array, where every brick is translated into its word and its next connecting brick. The third part contains the interpretation of the game move and will be explained in great detail in the section below.

When we add control structures (choice, order independence, concurrency) to our story this also needs to be translated to JSON. These kinds of game moves also have a unique label, specified in the same way. However, the difference is that these structures do not have an "_expr" and "_interpr" part; the only thing they have in common is the "_next" . Control structures have a "_type" field that tells what kind of structure it is (possibilities are: cho, oin, con) and a "_paths" field. The latter one contains all the game move labels which are defined in the control structure.

The translation of Figure 3.17 is given in Listing 3.2, note that the details of the game moves inside the choice are not included in the listing.

```
"g0": {
        "_type": "cho",
        "_paths": [
                "g1",
                "g2",
                "g3"
        ]
}
```

Listing 3.2: JSON Structure of a Choice

The structure for order independence and concurrency is the exact same, with the only exception being the "_type" is respectively “oin” and “con”.

Game Move Interpretation

The interpretation object is found in every valid ATTAC-L game move. Note that if a game move does not contain an interpretation part, it is not valid (exceptions are the control structures: choice, order independence, and concurrency).

The "_interpr" object consists of three elements, namely: "_passive", "_subject", and "_predicate". This can be seen in Listing 3.3.

```
"_interpr": {
        "_passive": <true|false>,
        "_subject": <noun-structure>,
        "_predicate": {
                "_verb": <verb-structure>,
                "_direct": <noun-structure>,
                "_indirect": <noun-structure>
        }
}
```

Listing 3.3: Basic Structure of an Interpretation Object

The first element is "_passive"; this is optional. If the sentence is modeled in passive voice, the "_passive" boolean is set to true. If not, it is omitted.

Before defining the other two elements, we first must define what a noun-structure is. A noun can be expressed in different ways:

- By specifying "player", which is a predefined entity in ATTAC-L. This is directly translated to its string representation.
- By a string that starts with a capital letter, followed by numbers or non-capital letters (no spaces). These refer to (pre)defined entities in the story and are directly translated to its string representation.
- By an expression that refers to an entity in the story. In this case a simple string is no longer sufficient so we defined a sub-structure containing the following elements (note that all of these can be optional except for "_noun"):
  - "_det", this is the determiner, it says how the entity needs to be selected. Possibilities are "a", "an", "any", and "last".
  - "_noun", this is the type of the entity that needs to be selected. There are no interpretations made on the meaning of the word, but it is assumed to be a singular countable common noun. An example is "person".
  - "_nom", this is the nominator, it is only present during inline definition. This will contain the dynamically assigned name that can be referenced to in later game moves.
  - "_of", this occurs when the "of" proposition is used in a brick. It contains the string of the brick after it. An example where "_of" is "Tim" is when using "posts to" "the hl-page of" "Tim".

  - "_attr", this is only present when there is a value brick in the game move. This object will contain all the property names and its values.

Now that we defined what a noun-structure is we can explain the remaining two elements of the "_interpr" field.

The next element is "_subject", which is a noun-structure as explained above. This is the subject of the game move, the one who invokes the action.

The last element is "_predicate", which describes everything (the action) behind the subject. It consists of three parts:

- "_verb", this is the action of the game move itself, represented as a string.

- "_direct", this is optional but will nearly always be present. This is the primary passive object of the game move, which is directly influenced by the action.

- "_indirect", this is optional. If this is present, "_direct" has to be present as well. This is the secondary passive object of the game move, it is indirectly influenced by the action.

The game move g0 in Listing 3.1 shows an example where the noun-structure for subject is the "player" string representation. In the same listing, g1 shows an example of a predicate that contains an indirect.

The interpretation structure of a game move that contains a value brick, as in Figure 3.10, is shown in Listing 3.4.

```
"_interpr": {
        "_subject": "player",
        "_predicate": {
                "_verb": "says to",
                "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                                "message": "Hello Carl! How are you?"
                        }
                },
                "_indirect": "Carl"
        }
}
```

Listing 3.4: Interpretation of a Game Move with a Value Brick

The interpretation structure of a game move defined in the passive voice, as in Figure 3.12, is shown in Listing 3.5. We can see that the "_passive" boolean is set to true.

```
"_interpr": {
        "_passive": true,
        "_subject": "player",
        "_predicate": {
                "_verb": "picked-up",
                "_direct": "Cd"
        }
}
```

Listing 3.5: Interpretation of a Passive Game Move

The interpretation structure of a game move that contains an inline definition, as in Figure 3.13 (the first game move), is shown in Listing 3.6.

```
"_interpr": {
        "_subject": {
                "_det": "a",
                "_noun": "person",
                "_nom": "X1"
        },
        "_predicate": {
                "_verb": "walks-to",
                "_direct": "player"
        }
}
```

Listing 3.6: Interpretation of a Game Move with Inline Definition

3.3.4 Example Story

Figure 3.23 shows a complete story modeled in ATTAC-L. The story is about Elisabeth losing her USB stick that contained her homework. She asks player to find it for her, but he cannot find it. Elisabeth then notices that Carl picked something up and asks if player can look into it. Carl found the USB but is unsure what to do with it. The player has the choice to demand Carl to give it back to Elisabeth or steal the homework on the USB and not tell Elisabeth about it.

This story contains three choices and two order independent game structures. Based on the player's choices, the story develops in a complete different direction.

Appendix A lists the complete JSON representation of this story.

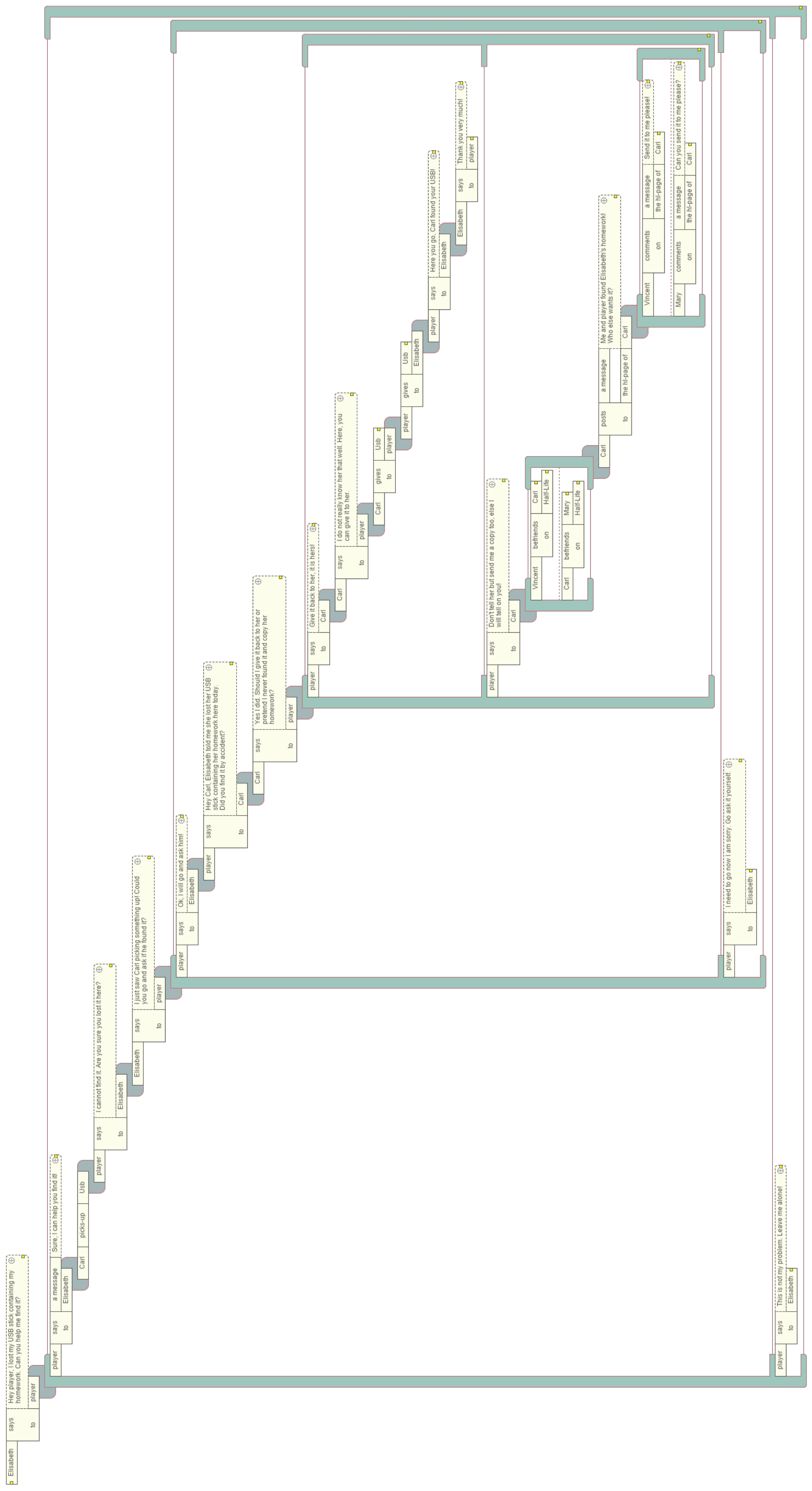

Elisabeth says to player Hey player, I lost my USB stick containing my homework. Can you help me find it?
player says to Elisabeth a message Sure, I can help you find it!
Carl picks-up Usb
player says to Elisabeth I cannot find it. Are you sure you lost it here?
Elisabeth says to player I just saw Carl picking something up! Could you go and ask if he found it?
player says to Elisabeth Ok, I will go and ask him!
player says to Carl Hey Carl, Elisabeth told me she lost her USB stick containing her homework here today. Did you find it by accident?
Carl says to player Yes I did. Should I give it back to her or pretend I never found it and copy her homework?
player says to Carl Give it back to her, it is hers!
Carl says to player I do not really know her that well. Here, you can give it to her.
Carl gives Usb to player
player gives Usb to Elisabeth
player says to Elisabeth Here you go, Carl found your USB!
Elisabeth says to player Thank you very much!
player says to Carl Don't tell her but send me a copy too, else I will tell on you!
Vincent befriends Carl on Half-Life
Carl befriends Mary on Half-Life
Carl posts a message on the hl-page of Carl Me and player found Elisabeth's homework! Who else wants it?
Vincent comments a message on the hl-page of Carl Send it to me please!
Mary comments a message on the hl-page of Carl Can you send it to me please?
player says to Elisabeth I need to go now I am sorry. Go ask it yourself!
player says to Elisabeth This is not my problem. Leave me alone!

Figure 3.23: ATTAC-L Example Story

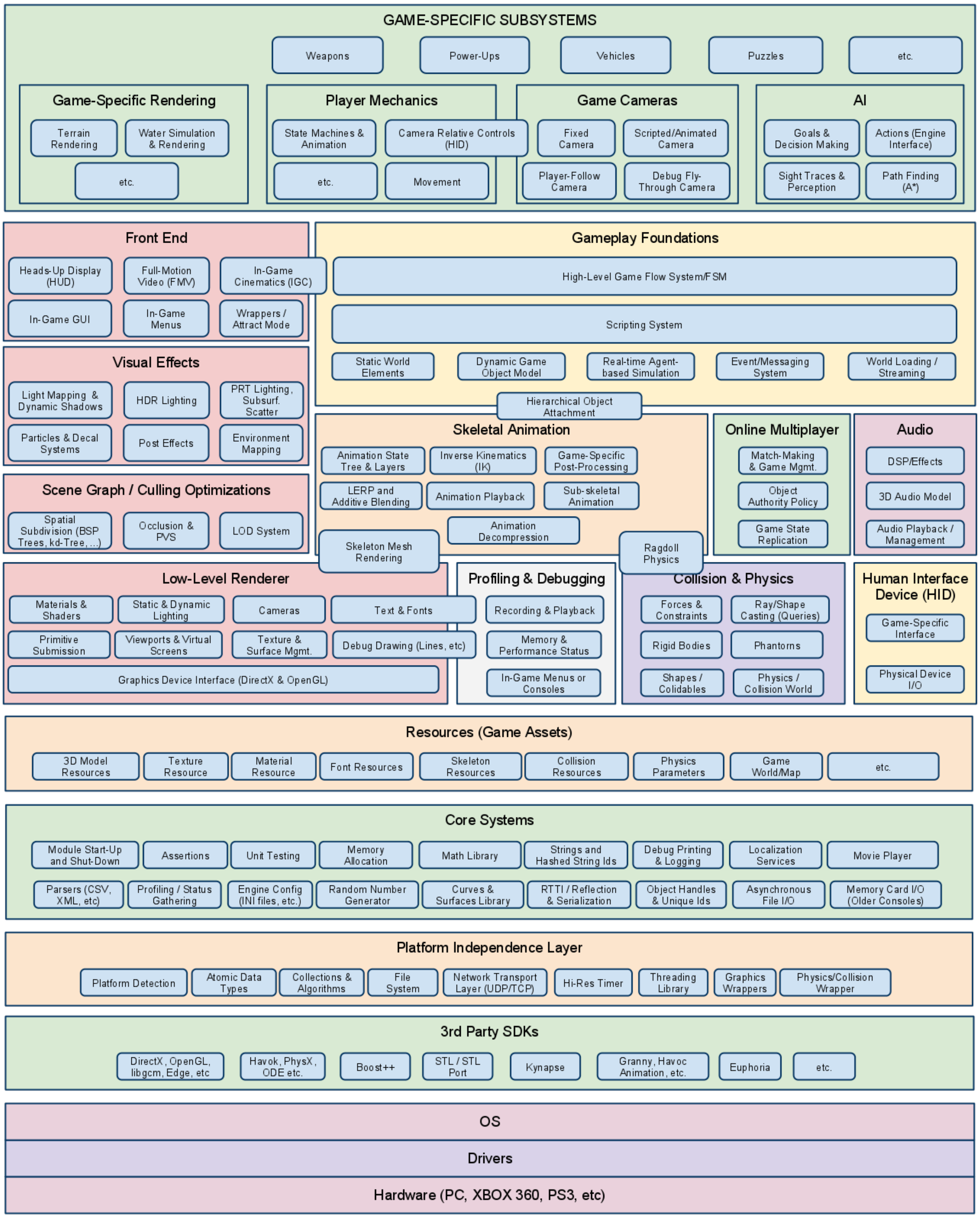


Figure 3.5: Game Engine Architecture (Gregory, 2009a)

	Unreal Engine	**CryEngine**	**Source**	**Unity**
Information				
Developer	Epic Games	Crytek	Valve	Unity Technologies
Latest Version	4	3	2	5
Documentation and Support	Excellent	Good	Good	Excellent
Licensing				
Non-commercial Use	Free	Free	Free	Free
Commercial Use	5% royalty (if revenue +$3000)	0% royalty $9.90/month (EaaS)	0% royalty (has to be released on Steam)	0% royalty Pers.Ed.: Free Prof. Ed.: $75/ month
Platform				
Deploys to	Windows, Mac, Linux, Android, iOS, PS4, Xbox One, HTML5, Oculus Rift, Gear VR, SteamOS	Windows, Android, Linux, iOS, PS4, Xbox One, Wii U	Windows, Android, Linux, iOS, PS4, Xbox One, Wii U	Windows, Mac, Linux, iOS, PS4, Xbox One, Wii U, Oculus Rift, Gear VR, ...
Engine				
Graphics	DirectX, OpenGL	DirectX, OpenGL	DirectX, OpenGL	DirectX, OpenGL
Physics	PhysX	Integrated physics engine	PhysX	PhysX
Audio	FMOD	FMOD	FMOD	FMOD
Networking	Unreal Engine Networking	CryNetwork, CryLobby	Source 2 Engine Networking	UNET
Coding				
Programming Languages	C++	C++, LUA	C++	C#, JavaScript, Boo
Visual Scripting	Blueprints	Flow Graph	N/A	Via third-party assets

Table 3.1: Game Engines Comparison

4 ATTAC-L Simulator

Based on the theoretical background of games, game engines, and ATTAC-L, introduced in the Background chapter, we can now elaborate in this chapter on the contributions made in this thesis. More in particular, we created an application that allows simulating scenarios developed in ATTAC-L, hence the name ATTAC-L Simulator. We start this chapter by repeating the aim and context. From this aim and context, we derive the requirements for the application. Next, we will describe the main features in the application used to satisfy the requirements. We conclude this chapter by describing the user interfaces available in the simulator.

4.1 Aim and Context

As explained in the introduction, the aim was to develop a fast prototyping tool that gives the opportunity to a scenario modeler to preview the different steps of a scenario after the scenario had been created with ATTAC-L and before it is integrated in the actual game. In order to realize this, the textual description given in the ATTAC-L story model needs to be converted into a playable story that can be invoked and observed in a step-wise manner by the modeler.

Although ATTAC-L is a general-purpose scenario modeling language, it is currently mainly used in the context of the Friendly ATTAC project for the development of scenarios for a serious game against cyber bullying.

4.2 Requirements

Before designing the application, we specified the requirements for the software. Some requirements are generally applicable but some of the requirements are derived from the current domain for which ATTAC-L is used, i.e., cyber bullying. As we mainly aimed for simulation scenarios related to cyber bullying, we opted for creating an environment that provides the means for this, e.g., different persons, social media like Half-Life and Twitter, mobile phones, email, etc. Also the (ATTAC-L) verbs supported by the simulator are specially targeted towards support for cyber bullying situations.

Below is the list of requirements divided into two categories: general requirements and requirements specific for the domain under consideration (cyber bullying). When needed requirements are also motivated briefly.

General requirements:

- R1. To start a simulation, the JSON representation of a story is required.
- R2. A simulation can be paused at any moment.

  The modeler should be able to inspect each step and reflect if needed.
- R3. The main menu can always be accessed while a simulation is running.

  The modeler should be able to stop a simulation or change options at any time during a simulation.
- R4. A scenario log is visible while a simulation is running.

  In order to follow the scenario, the modeler should be able to see the current and past game moves.
- R5. Detailed feedback, warnings, and error messages are presented during the simulation when an action finished, when an action makes no sense (e.g., walking to yourself), or when an unsupported action is defined.

  Since one of the purposes of the simulator is verification of the scenarios, the modeler should be notified when an error or a potential problem occurs.

- R6. The modeler can set several environment-related properties for the simulation.

  Allows changing the environment to provide a better match with the story that is being simulated.

- R7. The scene is changeable at any moment.

  The modeler should be able to choose between the Park, Office and House scene to select the one that is most suitable for the story (part) to be simulated.

- R8. The simulator provides a number of predefined persons (characters).

- R9. The simulator provides a number of predefined items.

  In many games, players and/or NPCs have to collect items.

- R10. The simulator provides an inventory for storing items.

  In many games, players are able to keep track of collected items in an inventory.

- R11. Every character is able to pick up items.

- R12. An item that was picked up is removed from the scene and if the player picked it up, it is added to the inventory.

- R13. Every character is able to give items to any other character. The player should be able to give any item in his inventory to any other character.

  This is also a common requirement for games.

- R14. Every character (NPC or player) can walk to certain locations in a scene.

- Every character (NPC or player) can talk to any other character.

Domain specific requirements:

- DR1. Every character (NPC or player) has its personal Half-Life page.
- DR2. A character is able to post, reply, and like a message or befriend a person with Half-Life.
- DR3. Every character (NPC or player) has its personal Twitter page.
- DR4. A character is able to tweet and re-tweet a message with Twitter.
- DR5. Every character (NPC or player) has its personal phone.
- DR6. A character is able to send and receive text messages.
- DR7. Every character (NPC or player) has its personal email inbox.
- DR8. A character is able to send and receive emails.
- DR9. Current verbs used in ATTAC-L for the cyber bullying domain are supported.

4.3 Features

In this section we explain the main features of the simulator. We start by describing the different persons (characters) available. Following this, we explain the inventory and the different types of items and how they are interacted with. Next, we discuss the different locations we defined in each scene. Then we explain the scenario log. Afterwards we explain how a conversation is started. We then give an overview of the different social media in the simulator. We also give an overview of the messaging system. We conclude this section by listing all the verbs supported in the simulator and give an example on how to use each of them.

4.3.1 Persons

Having persons are essential in many scenarios, including those for the domain of cyber bullying. Therefore, next to the player's character, there are 8 NPC models available in the simulator (R8). These are: Carl, Elisabeth, Joan, Justin, Kate, Mary, Tim, and Vincent. We opt for eight NPC characters so that a story can contain up to nine different characters (player included). This gives the flexibility to the modeler to create scenarios where multiple characters are involved. Figure 4.1 shows the player model and the 8 NPC models. When a game move contains the word "a person" (e.g., player goes-to a person), a random NPC or the player itself will be chosen (note

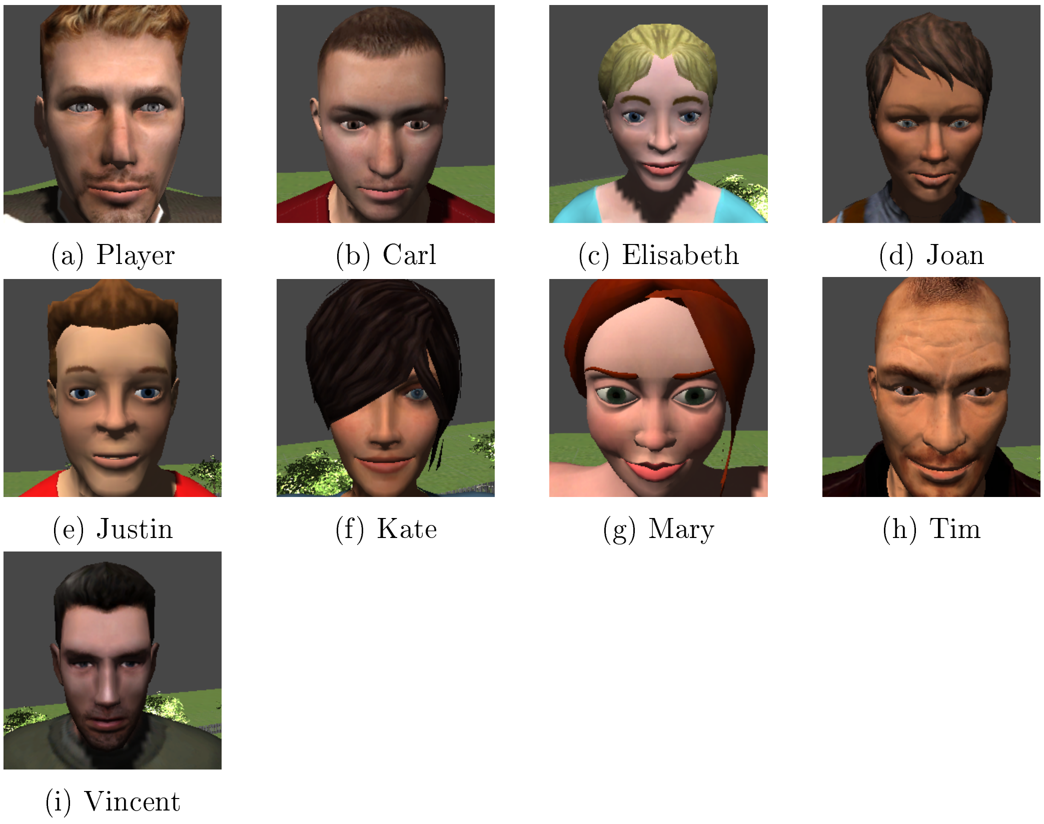


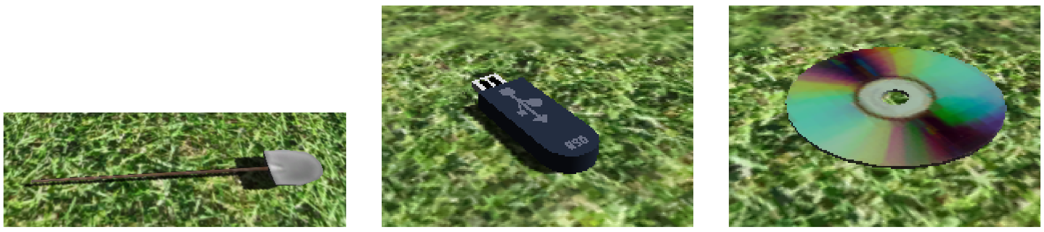
Figure 4.1: Player and NPC Models

in this example "a person" can never become player, checks are made that subject and direct will not become the same).

4.3.2 Items and Inventory

There are three types of items available in the simulator: a shovel, an USB, and a CD (Figure 4.2) (R9). We chose to only implement these items because they can be used in a meaningful manner in a cyberbullying scenario (e.g., a USB stick containing homework that has been lost and found). However, new items can be added in the simulator. This is discussed in future work (see section 6.3.4). Using the action (verb) "picks-up" with a direct object (shovel, USB or CD) will cause the item to be in the player's inventory after the game move is completed (R10, R12). More information about the inventory is given in section 4.4.3.

When using the action "gives to" an error message will be thrown if the player does not have the corresponding item in his inventory (R5). If he does have the item, it is removed from his inventory when the game move is completed (R12). Note that NPCs do not have an inventory, they can however pick up items (R11), but checking whether the NPC has the item or not is not done.



(a) Shovel (b) USB (c) CD

Figure 4.2: Available Items in Simulator

There is a finite amount of items that can be present inside each scene. The maximum number of items allowed per scene at the same time is:

- Park: 32
- House: 16
- Office: 8

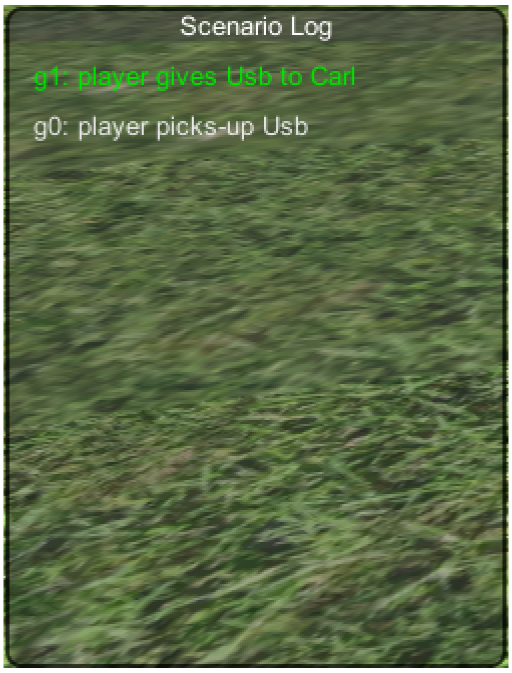
The amount of items placed in a simulation has to be between 0 and its maximum. The exact desired amount per item can be set in the configuration file. More details about the configuration file can be found in section 5.4.

4.3.3 Locations

Any character (NPC or player) can walk to certain *locations* in a scene (R14). Some examples of locations are "Playground" or "Fountain" for the park scene. All the defined locations per scene are listed in Appendix B.

4.3.4 Scenario Log

The *Scenario Log* panel always appears on the bottom left of the screen while a simulation is running (R4). It keeps track of every game move executed. When a game move is completed, it will turn white while the next one (if any) will appear on top of the log in green. Figure 4.3 shows the scenario log panel with a finished game move and the current game move running.



g1: player gives Usb to Carl

g0: player picks-up Usb

Figure 4.3: Scenario Log Panel

4.3.5 Conversation

Any character (either an NPC or the player) can talk to any other character in the environment (R15). In ATTAC-L, this is specified with the verb "says to". The condition for it to execute is that the two characters involved are facing each other. If not, the character invoking the game move (the subject) will walk towards the other character involved. When they are close enough a conversation overlay is shown (see section 4.4.4).

4.3.6 Social Media

As social media is a crucial element in cyber bullying situations, we have implemented two social media in our simulator: *Half-Life* (a kind of Facebook[1]) and *Twitter*[2]. Half-Life also has a *Messenger* aspect. Their layouts are discussed in section 4.4.5. Their functionality is discussed below.

[1]`https://www.facebook.com/`, accessed 13/04/2015.
[2]`https://www.twitter.com/`, accessed 13/04/2015.

Half-Life

We have created the Half-Life social media in our simulator (DR1). The possible verbs that trigger its use are: “posts on/to”, “likes (on)”, “replies(-with) on”, and “comments on”. Before posting, commenting or liking on someone else’s Half-Life page, both involved persons have to be friends (using the “(be)friends on” verb) (DR2).

Half-Life Messenger

We have also implemented chatting functionality via the Half-Life Messenger. The verbs that trigger the Messenger are: “chats to” and “says to”. Chatting consists of two parts: a conversation and a message. Only one conversation can exist between two persons. One conversation can have multiple messages. The conversation with the newest message will always appear on top in the left bar of the overlay, this also contains (a part of) the last message.

Twitter

We have recreated a simplistic version of Twitter in our simulator (DR3, DR4). The approval of Twitter was granted to use their layout for the purpose of this simulator. The Twitter overlay is shown when a game move contains one of the following verbs: “tweets”, “retweets”, and “follows on”. A person can only post a message (a Tweet) on his own Twitter page. Retweeting a Tweet copies the original Tweet to both the Twitter pages (it is indicated if a Tweet is retweeted).

4.3.7 Messaging

As a lot of cyber bullying occurs through email and text messages, we have implemented these two methods of messaging in our simulator. Their layouts are discussed in section 4.4.6.

Text

Text messaging is possible in the simulator. In order to depict text messages we use a phone overlay (DR5, DR6). The possible verbs that trigger the phone overlay are: “sends to”, “texts to”, and “receives from”. A person can have one conversation with every person, each of these conversations may contain text messages that have been sent between each other.

Email

Email messages contain two overlays. The first one is used when composing an email and is triggered by the verbs: "sends to" or "(e)mails to" (DR8). The result is a compose email overlay.

The second one is triggered when the verb is "receives from", this results in showing the email inbox of the subject of the game move (DR7, DR8). As in text messages, persons also have conversations with each other where emails are stored in.

4.3.8 Verb Support

In what follows, we explain all the verbs the simulator supports (DR9). We will provide an explanation of all the supported verbs by using it in an example modeled in ATTAC-L, while also giving its JSON representation. Appendix C gives a recap of all the possible verbs.

walks-to & goes-to

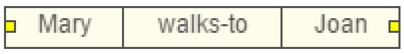


Figure 4.4: Game Move with "walks-to"

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Mary",
                    "_next": [
                        {
                            "word_": "walks-to",
                            "_next": [
                                {
                                    "word_": "Joan"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Mary",
                "_predicate": {
                    "_verb": "walks-to",
                    "_direct": "Joan"
                }
            }
        }
    }
}
```

Listing 4.1: JSON Representation of Figure 4.4

Any character (NPCs or player) can walk to each other by using the *"walks-to"* or *"goes-to"* verb. Figure 4.4 shows an example of a game move using the "walks-to" verb and Listing 4.1 gives the JSON representation of this game move. The subject will always walk towards the direct. If we use the example above, Mary will walk towards Joan. The game move is completed when Mary is in front of and facing Joan.

There is also a possibility to walk to an area in the scene. This requires the direct to be a predefined location, more information on the locations that are defined in the simulator can be found in section 4.3.3.

says to

The verb *"says to"* can be used in two different contexts: talking to someone in person and talking (chatting) on Half-Life with another person. We will explain the first case, talking to someone in person. The second case gives the same result as the "chats to" verb and will be discussed there.

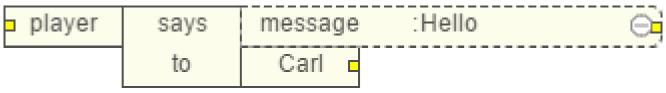


Figure 4.5: Game Move with "says to"

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Hello"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Carl"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Hello"
                    },
                    "_indirect": "Carl"
                }
            }
        }
    }
}
```

Listing 4.2: JSON Representation of Figure 4.5

All characters can talk to each other, but not against themselves. When a game move contains the verb “says to”, the subject and indirect are expected to be characters (NPC or player). If this is not the case, an error will be given and the scenario will be stopped (R5). If however, subject and direct are both characters, the subject will walk towards the indirect (just as in “walks-to”) and the conversation will be shown (R5). See section 4.3.5 for more information about the conversation overlay. If we use the example as shown in Figure 4.5, with its JSON representation shown in Listing 4.2, player will walk to Carl and say “Hello” to him. Note that the direct “a message” can be omitted here, this is the only game move construction where this is possible, because there is no ambiguity here.

chats to

As stated above, *“chats to”* and *“says to”* when used in a game move with the construction as shown in example Figure 4.6, yield the same result.

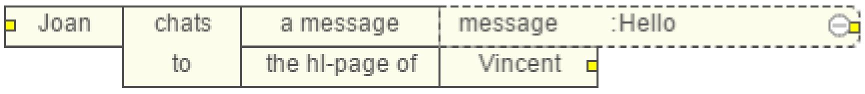


Figure 4.6: Game Move with “chats to”

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Joan",
                    "_next": [
                        {
                            "word_": "chats",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Hello"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "the hl-page of",
                                    "_next": [
                                        {
                                            "word_": "Vincent"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Joan",
                "_predicate": {
                    "_verb": "chats to",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Hello"
                        }
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "hl-page",
                        "_of": "Vincent"
                    }
                }
            }
        }
    }
}
```

Listing 4.3: JSON Representation of Figure 4.6

Every character (NPC or player) has its personal Half-Life page (hl-page). Half-Life and Half-Life Messenger are overlays regarding social media, more information about social media can be found in section 4.3.6. In the “chats to” we only deal with the *Messenger* aspect of Half-Life. In the game move, we have to specify to whose Half-Life Messenger page we are chatting to. As can be seen in the example Figure 4.6, the one sending “Hello” is Joan and the recipient is Vincent. The JSON representation of this example is depicted in Listing 4.3.

sends to (& texts to & mails to)

The *“sends to”* verb can be used for sending two different kinds of messages: text and email. We will start by explaining sending a text message and continue with an email message. We can already note that every character (NPCs and player) has its personal phone (DR5) where text messages are shown and its personal email inbox (DR7). More information about how the messaging system overlays look like can be found in section 4.3.7.

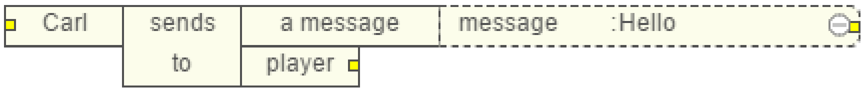


Figure 4.7: Game Move with “sends to” (text)

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "sends",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Hello"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "player"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                    "_subject": "Carl",
                    "_predicate": {
                        "_verb": "sends to",
                        "_direct": {
                            "_det": "a",
                            "_noun": "message",
                            "_attr": {
                                "message": "Hello"
                            }
                        },
                        "_indirect": "player"
                    }
                }
            }
        }
    }
}
```

Listing 4.4: JSON Representation of Figure 4.7

Any character can send a text message to any other character. In order to send a text message the game move has to fulfill three requirements: the verb has to be “sends to” or “texts to”, the direct object has to be either “a message” or “a text”, and the value brick has to have one entry with the property name “message” and a value filled in for the content of the message being sent. An example of a game move containing a “sends to” verb is seen in Figure 4.7, its JSON representation is seen in Listing 4.4.

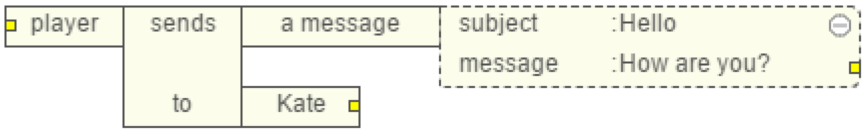


Figure 4.8: Game Move with “sends to” (email)

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "sends",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "subject": "Hello",
                                        "message": "How are you?"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Kate"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "sends to",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": [
                            {
                                "subject": "Hello"
                            },
                            {
                                "message": "How are you?"
                            }
                        ]
                    },
                    "_indirect": "Kate"
                }
            }
        }
    }
}
```

Listing 4.5: JSON Representation of Figure 4.8

Just like with text messages, email messages can be send to any character in the simulator. A game move that sends an email message also has three requirements to fulfill: the verb has to be “sends to” or “(e)mails to”, the direct object has to be either “a message” or “a mail/an email”, and the value brick has to have two entries with their values filled in and their property names being “subject” and “message”. Figure 4.8 shows an example of a game move where the player sends an email to Kate. The subject of this email is “Hello” and the content of the email is “How are you?”. The JSON code of this game move is shown in Listing 4.5.

gives to

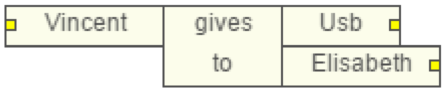


Figure 4.9: Game Move with “gives to”

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Vincent",
                    "_next": [
                        {
                            "word_": "gives",
                            "_next": [
                                {
                                    "word_": "Usb"
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Vincent",
                "_predicate": {
                    "_verb": "gives to",
                    "_direct": "Usb",
                    "_indirect": "Elisabeth"
                }
            }
        }
    }
}
```

Listing 4.6: JSON Representation of Figure 4.9

Using the *"gives to"* verb, any character can give an item to any other character (R13). We have three items in the simulator: a shovel, an USB, and a CD. When the game move is executed, the subject will walk towards the indirect and hand over the item. When the subject is the player, we perform a check to see if the item being given from player to another character is indeed in the player's inventory. Note that we do not make this check if the giver is an NPC. More information about the inventory and items can be found in section 4.3.2.

If we take the example as given in Figure 4.9, with its JSON code given in Listing 4.6, Vincent will walk to Elisabeth and give the USB-stick as soon as he is facing her. The feedback message provided will be: "Vincent gave usb to Elisabeth" (R5).

picks-up & grabs

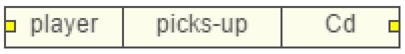


Figure 4.10: Game Move with “picks-up”

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "picks-up",
                            "_next": [
                                {
                                    "word_": "Cd"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "picks-up",
                    "_direct": "Cd"
                }
            }
        }
    }
}
```

Listing 4.7: JSON Representation of Figure 4.10

A game move with the *“picks-up”* or *“grabs”* verb implies that the direct object contains the name of a known item in the simulator. When the subject is player, the item that is picked up is added to the inventory (R12). In the example given in Figure 4.10 with its JSON code in Listing 4.7, the player will walk to where the CD is in the scene and pick it up. At this point the item is removed from the scene and added to the player’s inventory. A feedback message containing “picked up CD” is shown to complete the game move (R5). More information about the inventory and items can be found in section 4.3.2.

receives from

The *“receives from”* verb is used for the receipt of two different kinds of messages: text and email. We start the explanation of receiving a text message and continue with the receipt of an email message. As said in the “sends to” verb, every character (NPCs and player) has its personal phone where text messages are shown and its personal email inbox. More information about how the messaging system overlays look like can be found in section 4.3.7.

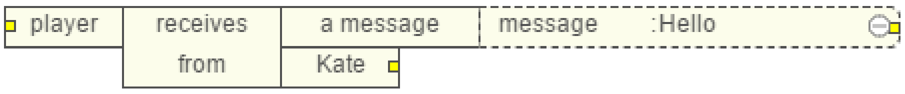


Figure 4.11: Game Move with “receives from” (text)

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "receives",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Hello"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "from",
                            "_next": [
                                {
                                    "word_": "Kate"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "receives from",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Hello"
```

```
                                    }
                                },
                                "_indirect": "Kate"
                            }
                        }
                    }
                }
}
```

Listing 4.8: JSON Representation of Figure 4.11

Any character can receive a text message from any other character. The modeling of the “receives from” verb in ATTAC-L has the same three requirements as the “sends to”: the verb has to be “receives from”, the direct object has to be either “a message” or “a text”, and the value brick has to have one entry with the property name “message” and a value filled in for the content of the message being sent. The difference with “sends to” is that the result of a “receives from” game move regarding a text message shows the perspective from the indirect’s phone. An example of a game move containing a “receives from” verb is given in Figure 4.11, its JSON representation is given in Listing 4.8.

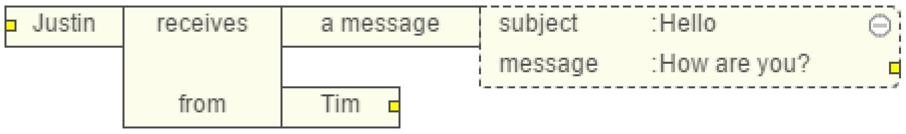


Figure 4.12: Game Move with “receives from” (email)

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Justin",
                    "_next": [
                        {
                            "word_": "receives",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "subject": "Hello",
                                        "message": "How are you?"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "from",
                            "_next": [
                                {
                                    "word_": "Tim"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Justin",
                "_predicate": {
                    "_verb": "receives from",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": [
                            {
                                "subject": "Hello"
                            },
                            {
                                "message": "How are you?"
                            }
                        ]
                    },
                    "_indirect": "Tim"
                }
            }
        }
    }
}
```

Listing 4.9: JSON Representation of Figure 4.12

Just like with text messages, email messages can be received from any character in the simulator. A game move that receives an email message also has three requirements to fulfill: the verb has to be “receives from”, the direct object has to be either “a message” or “a mail/an email”, and the value brick has to have two entries with their values filled in and their property names being “subject” and “message”. The difference with “sends to” is that the result of a “receives from” game move regarding an email message shows the email inbox from the subject’s perspective, instead of the compose mail window. Figure 4.12 shows an example of a game move where Justin receives an email from Tim. The subject of this email is “Hello” and the content of the email is “How are you?”. The JSON code of this game move is shown in Listing 4.9.

befriends on & friends on

In order to be able to post and comment on Half-Life, the involved characters need to be friends on Half-Life. Befriending happens with the *“befriends on”* or *“friends on”* verb. More information about Half-Life can be found in section 4.3.6.

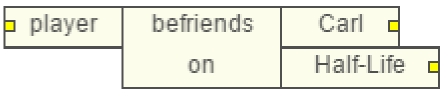


Figure 4.13: Game Move with “befriends on”

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "befriends",
                            "_next": [
                                {
                                    "word_": "Carl"
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "Half-Life"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "befriends on",
                    "_direct": "Carl",
                    "_indirect": "Half-Life"
                }
            }
        }
    }
}
```

Listing 4.10: JSON Representation of Figure 4.13

Figure 4.13 shows an example of a game move befriending player and Carl on Half-Life. Its JSON code is given in Listing 4.10. Note that the indirect has to be “Half-Life”. The result of befriending is that, apart from seeing a new name in the friends list, friends can post, comment and like on each other’s Half-Life page. Posting, commenting and liking is explained next.

posts on & posts to

A message can be posted to either Half-Life or Twitter[3] using *"posts on"* or *"posts to"*. Here we will first discuss the "posts to" for posting a message on Half-Life. Posting a message on Twitter is explained right after.

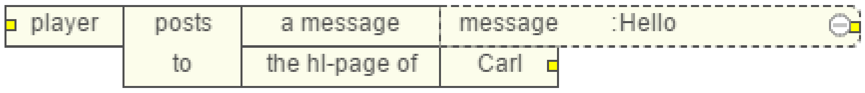


Figure 4.14: Game Move with "posts to" (Half-Life)

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "posts",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Hello"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "the hl-page of",
                                    "_next": [
                                        {
                                            "word_": "Carl"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "posts to",
                    "_direct": {
                        "_det": "a",
```

[3]https://twitter.com/, accessed 15/04/2015.

```
                        "_noun": "message",
                        "_attr": {
                            "message": "Hello"
                        }
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "hl-page",
                        "_of": "Carl"
                    }
                }
            }
        }
    }
}
```

Listing 4.11: JSON Representation of Figure 4.14

Any character (NPC or player) can post on their own Half-Life page (hl-page) or on the Half-Life page of a character which has been befriended in a previous game move (DR2). This can be done by using the "posts on" or "posts to" verb. When posting on your own Half-Life page, the subject and indirect have to be the same. Figure 4.14 shows an example game move of player posting a message ("Hello") on the Half-Life page of Carl (we assume that player has befriended Carl in a previous game move). Listing 4.11 shows the JSON code for this example.

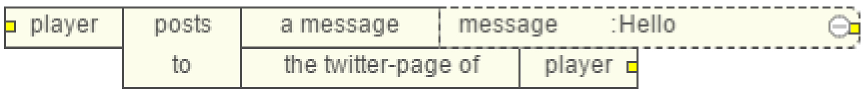


Figure 4.15: Game Move with "posts to" (Twitter)

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "posts",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Hello"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "the twitter-page of",
                                    "_next": [
                                        {
                                            "word_": "player"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "posts to",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Hello"
                        }
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "twitter-page",
                        "_of": "player"
                    }
                }
            }
        }
    }
}
```

Listing 4.12: JSON Representation of Figure 4.15

Posting on Twitter is called “Tweeting”, it can only happen on a character’s own Twitter page. Subject and indirect have to be the same character name in order to post a message (a “Tweet”) on Twitter. If this is not the case, an error message is given saying that you can only post on your own Twitter page (R5). Figure 4.15 shows an example of player posting a message (“Hello”) on his own Twitter page. Listing 4.12 is the JSON code of this example. Because the subject and indirect have to be the same character name, the “posts on” verb is not the most desired one when posting to Twitter. The “tweets” verb is a more efficient and cleaner way of posting a Tweet, it is explained later.

replies on & replies-with to & comments on

When someone posts a message on Half-Life, it is possible to comment on this post. This can be done by using the verb *“replies on”* or *“replies-with* *to"* or *"comments on"*. This implies commenting on the newest post on the indirect's Half-Life page.

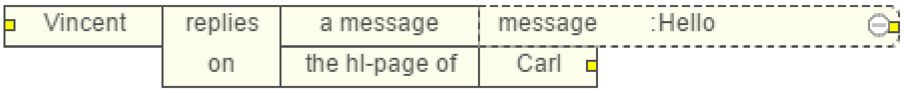


Figure 4.16: Game Move with "replies on"

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Vincent",
                    "_next": [
                        {
                            "word_": "replies",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Hello"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "the hl-page of",
                                    "_next": [
                                        {
                                            "word_": "Carl"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Vincent",
                "_predicate": {
                    "_verb": "replies on",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Hello"
                        }
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "hl-page",
                        "_of": "Carl"
                    }
                }
            }
        }
    }
}
```

Listing 4.13: JSON Representation of Figure 4.16

Figure 4.16 shows an example where Vincent comments ("Hello") on a post on the Half-Life page of Carl. We assume that Vincent and Carl have befriended themselves on Half-Life already, and either Carl or someone else posted a message on Carl's Half-Life page previously. Listing 4.13 shows the JSON code of this example.

likes & likes on

A post on Half-Life can be liked by anyone, including the creator of the post. There are two ways to like a post, they have a different syntax but yield the same result. We will explain both ways to like a post, starting with the *"likes"* verb, and following with the *"likes on"* verb.

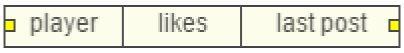


Figure 4.17: Game Move with "likes"

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "likes",
                            "_next": [
                                {
                                    "word_": "last post"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "likes",
                    "_direct": {
                        "_det": "last",
                        "_noun": "post"
                    }
                }
            }
        }
    }
}
```

Listing 4.14: JSON Representation of Figure 4.17

Figure 4.17 implies that the subject (player) likes the last post on his Half-Life page. This post can be one which he posted himself, or it can be a post someone else posted on his page. Its JSON representation is given in Listing 4.14. Note that the direct "last post" can be changed by "last message", the result is the same.

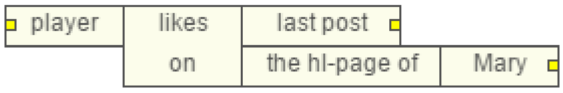


Figure 4.18: Game Move with "likes on"

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "likes",
                            "_next": [
                                {
                                    "word_": "last post"
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "the hl-page of",
                                    "_next": [
                                        {
                                            "word_": "Mary"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "likes on",
                    "_direct": {
                        "_det": "last",
                        "_noun": "post"
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "hl-page",
                        "_of": "Mary"
                    }
                }
            }
        }
    }
}
```

Listing 4.15: JSON Representation of Figure 4.18

While “likes” implies liking a post on one’s own Half-Life page, with “likes on” it is possible to specify the last post to like on anyone else’s Half-Life page. Figure 4.18 shows an example where player likes the last post on the Half-Life page of Mary. The JSON code for this example is given in Listing 4.15. As is the case with “likes”, the direct “last post” can be changed by “last message” without changing the result. The indirect specifies on whose Half-Life page to like the last post. Note that for liking someone else’s post, it is also required to be friends on Half-Life.

tweets

As has been explained for the “posts on” verb, Tweeting can only be done on one’s own Twitter page. However, when using the *“tweets”* verb, this problem is not applicable anymore.

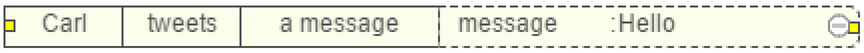


Figure 4.19: Game Move with “tweets”

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "tweets",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Hello"
                                    }
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "tweets",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Hello"
                        }
                    }
                }
            }
        }
    }
}
```

Listing 4.16: JSON Representation of Figure 4.19

As can be seen in Figure 4.19, when using “tweets”, there is no need to specify an indirect. All Tweets using this verb will be posted on the subject’s Twitter page. Listing 4.16 shows the JSON code for this example.

retweets

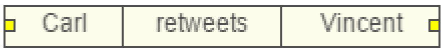


Figure 4.20: Game Move with “retweets”

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "retweets",
                            "_next": [
                                {
                                    "word_": "Vincent"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "retweets",
                    "_direct": "Vincent"
                }
            }
        }
    }
}
```

Listing 4.17: JSON Representation of Figure 4.20

A post (Tweet) on Twitter can be re-tweeted by anyone by using the verb *"retweets"*. This implies increasing the re-tweet counter on the corresponding Tweet and the original Tweet will be copied to the subject's (the one who is re-tweeting) Twitter page, where it states that it is a re-tweeted Tweet. The example given in Figure 4.20 will result in Vincent's original Tweet to be placed on Carl's Twitter page. Note that we assume here that Vincent has posted a Tweet at least once before on Twitter. Listing 4.17 gives the JSON representation of this example.

follows on

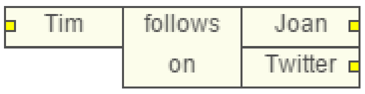


Figure 4.21: Game Move with "follows on"

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "Tim",
                    "_next": [
                        {
                            "word_": "follows",
                            "_next": [
                                {
                                    "word_": "Joan"
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "Twitter"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Tim",
                "_predicate": {
                    "_verb": "follows on",
                    "_direct": "Joan",
                    "_indirect": "Twitter"
                }
            }
        }
    }
}
```

Listing 4.18: JSON Representation of Figure 4.21

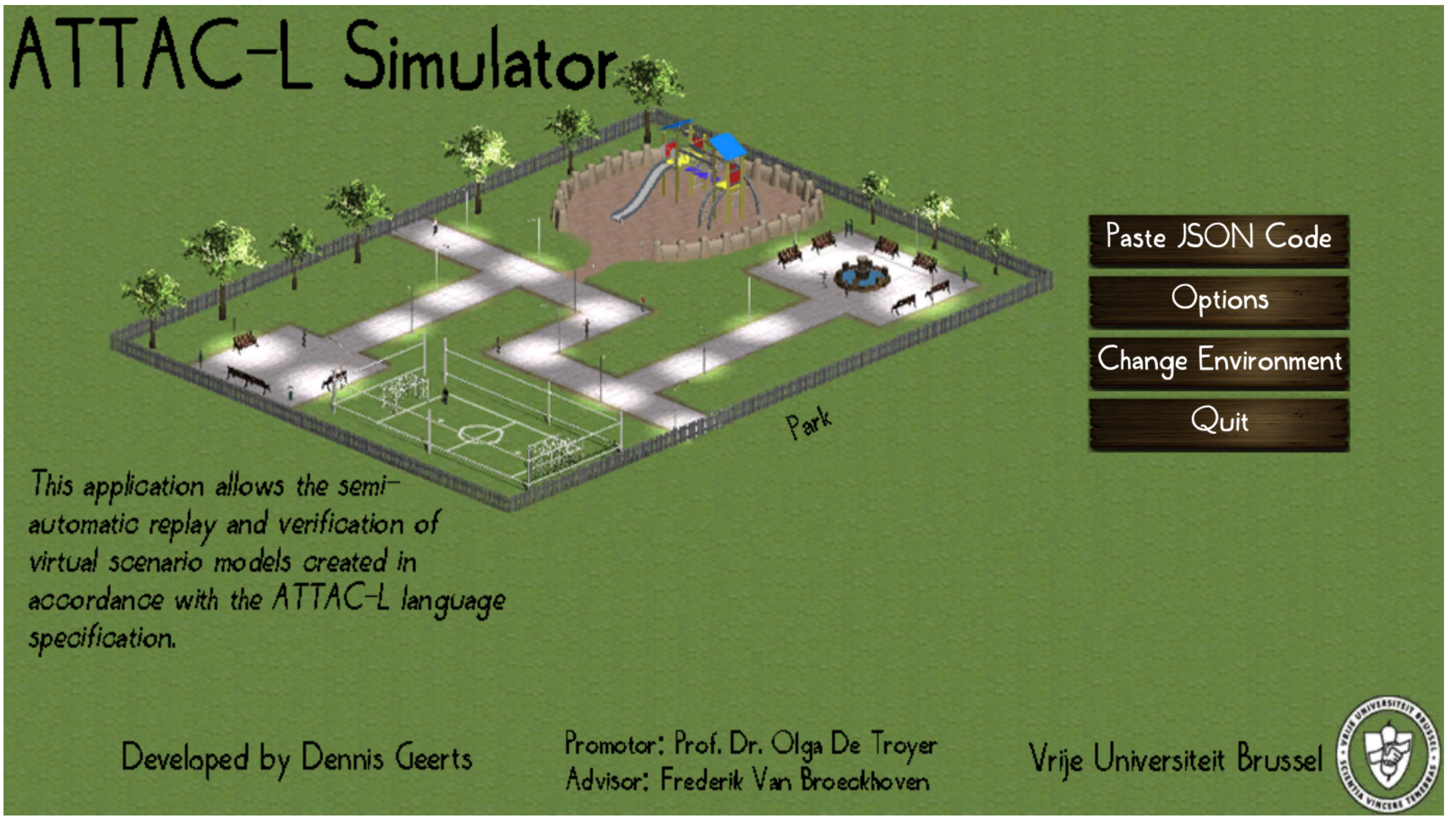
It is also possible to follow someone on Twitter by using the *"follows on"* verb. Figure 4.21, with its JSON representation given in Listing 4.18, gives an example on how to follow someone on Twitter. In this example, Tim will follow Joan; this means that all of Joan's Tweets will be visible on Tim's Twitter page as well. The "following" counter is increased on Tim's Twitter page, while the "follower" counter is increased on Joan's Twitter page.

4.4 User Interface

We start by explaining the main menu (R3) and how an ATTAC-L story in JSON code can be pasted to start the simulation (R1). Afterwards we show a screenshot of an ongoing simulation and explain what it shows. We then explain and show the interface for the Inventory. Next, we explain the user interface elements used for conversations and the social media, i.e., conversation overlay and the social media overlays. We conclude this section by explaining how messages are displayed, i.e., the messaging overlays.

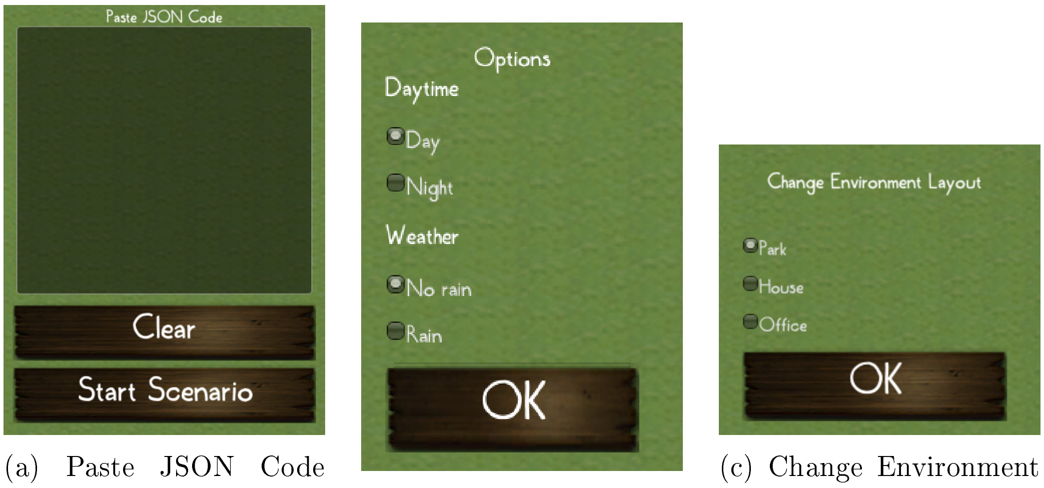
4.4.1 Main Menu

Upon launching the simulator, the main-screen is presented to the user (as illustrated in Figure 4.22). The button "Paste JSON Code" opens a dialog in which ATTAC-L JSON code can be entered. This code represents the textual form of an ATTAC-L storyline model that has to be simulated (Figure 4.23a). After clicking the “Options”-button, the Options dialog is shown (Figure 4.23b). With this dialog, several environment-related properties can be set for the simulation, such as a day- or night-environment or turn on or off rainy weather (R6). Similar for the “Change Environment”-button, which opens the Change Environment-dialog (Figure 4.23c). This dialog controls whether the simulation should be run in a park, office or house environment (R7). We opt for these three environments because most of the scenarios developed are able to fit into one of these environments. The main menu can always be accessed while a simulation is running by pressing the *“ESC”* key (R3).


ATTAC-L Simulator
Paste JSON Code
Options
Change Environment
Quit
Park
Developed by Dennis Geerts
Promotor: Prof. Dr. Olga De Troyer
Advisor: Frederik Van Broeckhoven
Vrije Universiteit Brussel

This application allows the semi-automatic replay and verification of virtual scenario models created in accordance with the ATTAC-L language specification.

Figure 4.22: Main Menu



(a) Paste JSON Code Menu

(b) Options Menu

(c) Change Environment Menu

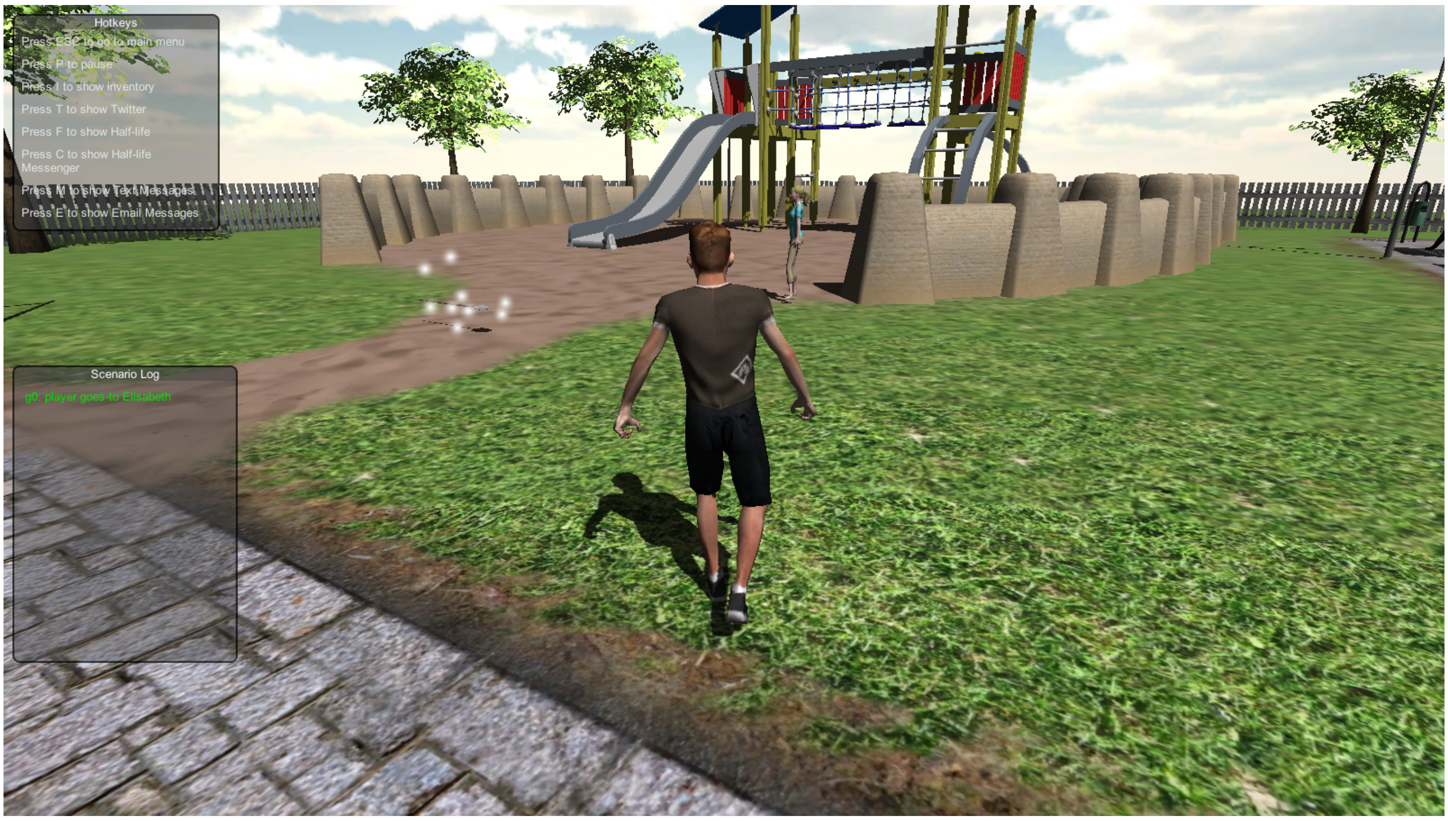
Figure 4.23: Main Menu Interfaces

4.4.2 Simulation

Figure 4.24 shows a screenshot during the execution of a simulation. This example depicts a "goes-to" action, where the player goes to an NPC called Elisabeth. The corresponding ATTAC-L model is shown in Figure 4.25, its JSON is given in Listing 4.19.

As can be seen at the left side of the screen, there are two panels that present information to the user. The top one reminds the user of the *Hotkeys* (i.e., the different keyboard keys providing quick access to a particular overlay in the simulator) available. The bottom panel is the *Scenario Log* (see section 4.3.4).

During a simulation, the user can click *"P"* once to pause the simulation, this will cause the text "Paused" to appear on the screen in front of the simulation. Clicking it twice will resume it again (R2).


Hotkeys
Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages
Scenario Log

g0: player goes-to Elisabeth

Figure 4.24: Simulation Performing the "goes-to" Action (Park Scene)

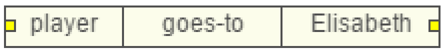


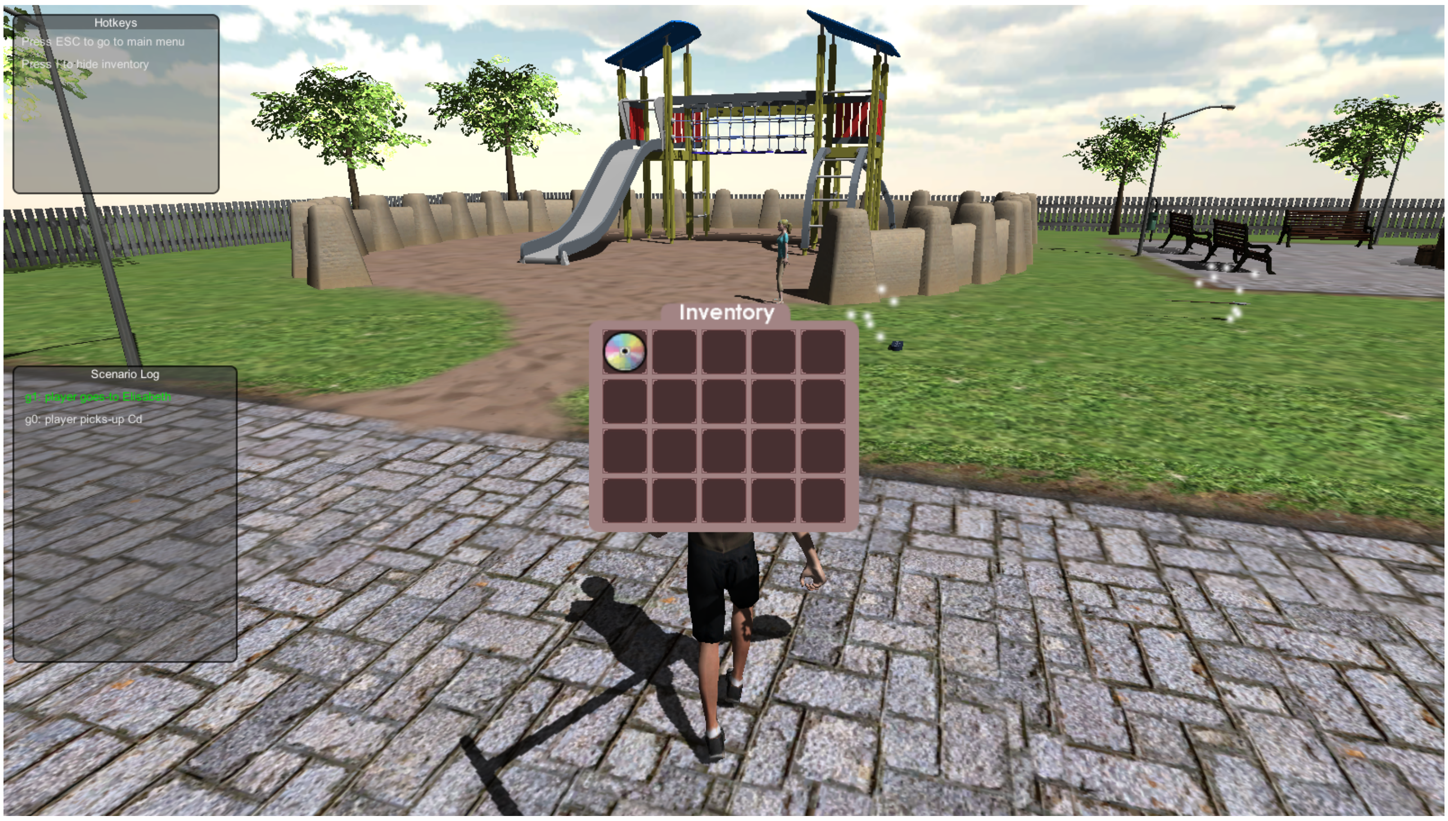
Figure 4.25: ATTAC-L Representation of the Game Move in Figure 4.24

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g0": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "goes-to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "goes-to",
                    "_direct": "Elisabeth"
                }
            }
        }
    }
}
```

Listing 4.19: JSON Representation of Figure 4.25

4.4.3 Inventory

The player is able to access the inventory by pressing *"I"* while the simulation is running. This will pause the simulation and display the inventory of the user in the middle of the screen as illustrated in Figure 4.26. We used the ATTAC-L scenario given in Figure 4.27, so that we have picked up an item (CD) first. Items that are inside the inventory have an icon and a tooltip when hovering over them. It is also possible to rearrange items by dragging and dropping them inside the inventory.



Press ESC to go to main menu
Press I to hide inventory

g1: player goes-to Elisabeth
g0: player picks-up Cd

Figure 4.26: Inventory After Picking Up CD

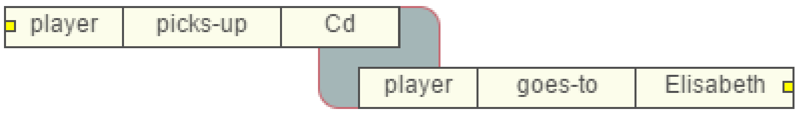
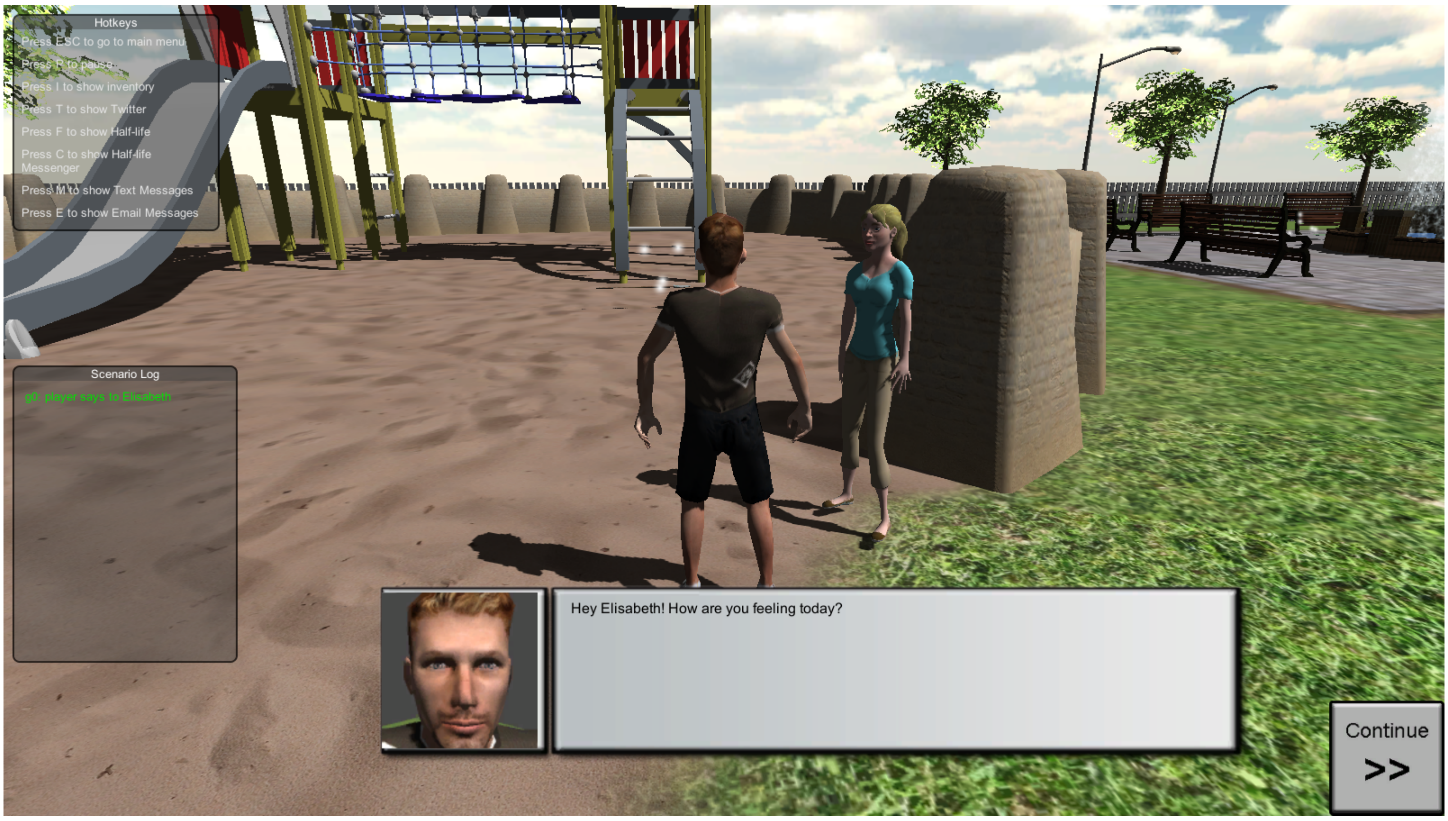


Figure 4.27: ATTAC-L Representation of the Story in Figure 4.26

4.4.4 Conversation Overlay

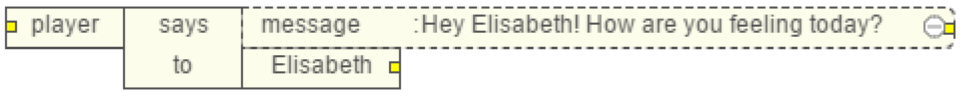
The conversation overlay is shown when the game move contains the verb “says to”. Figure 4.29 is an example game move containing this verb. The resulting overlay is shown in Figure 4.28. The avatar shown on the left of the message is the character that is currently talking.


Hotkeys
Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages
Scenario Log
Continue >>

g0: player says to Elisabeth

Hey Elisabeth! How are you feeling today?

Figure 4.28: Conversation Using the “says to” Action



:Hey Elisabeth! How are you feeling today?

Figure 4.29: ATTAC-L Representation of the Game Move in Figure 4.28

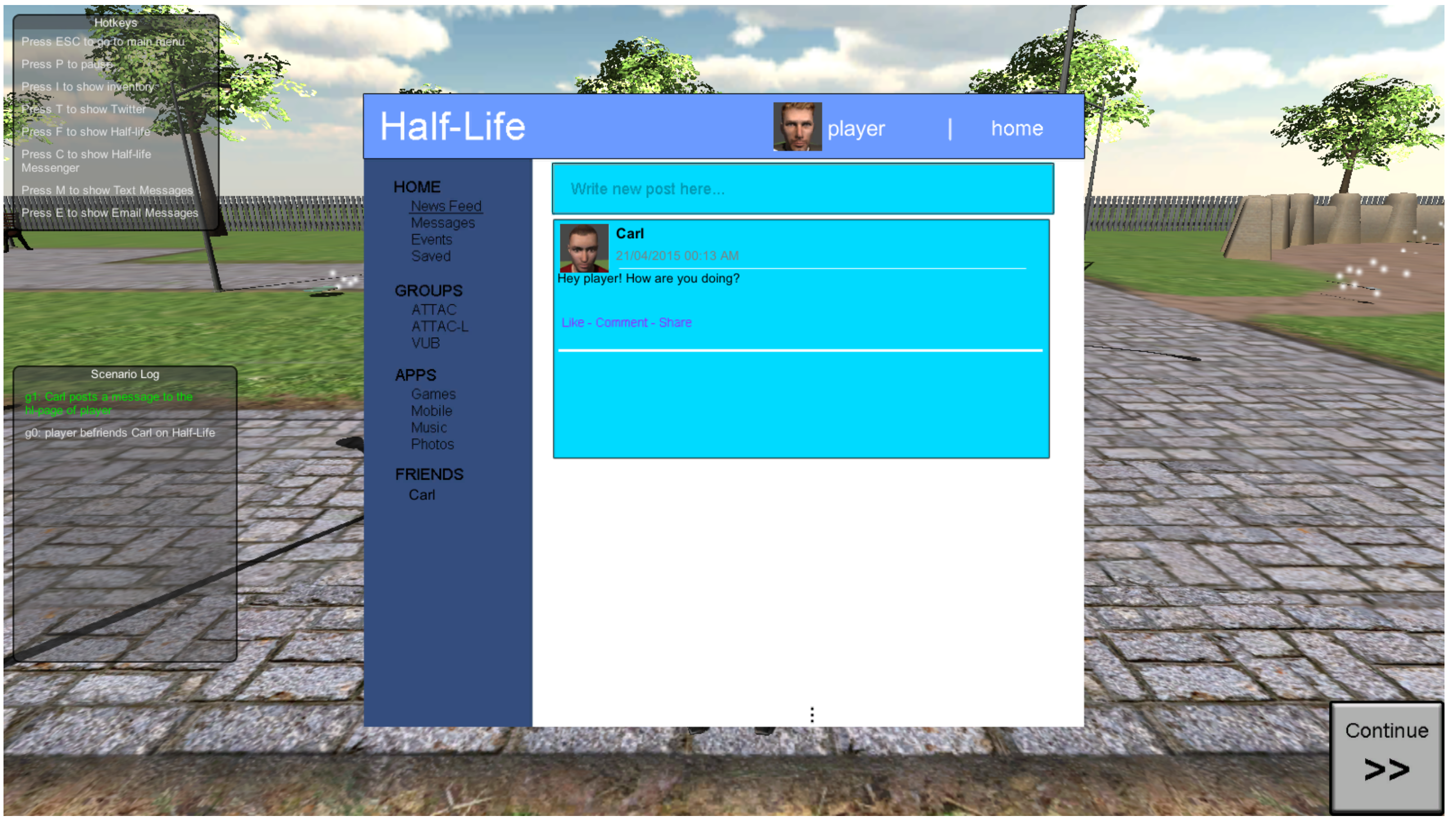
4.4.5 Social Media Overlays

In this section we discuss the visuals used for Half-Life, Half-Life Messenger, and Twitter.

Half-Life

The ATTAC-L story given in Figure 4.31 starts by befriending player and Carl, followed by Carl posting a message to the Half-Life page (hl-page) of player. The resulting Half-Life overlay is shown in Figure 4.30. Note that all the befriended persons are listed (by their name) in the bottom left of the

Half-Life overlay screen. The player's Half-Life page can be brought up by pressing the *"F"* key.



Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages

g1: Carl posts a message to the hl-page of player
g0: player befriends Carl on Half-Life

Carl
21/04/2015 00:13 AM
Hey player! How are you doing?

Figure 4.30: Half-Life Overlay - Performing a "posts to"

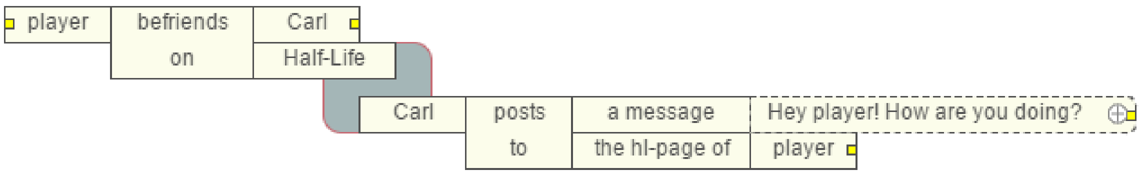
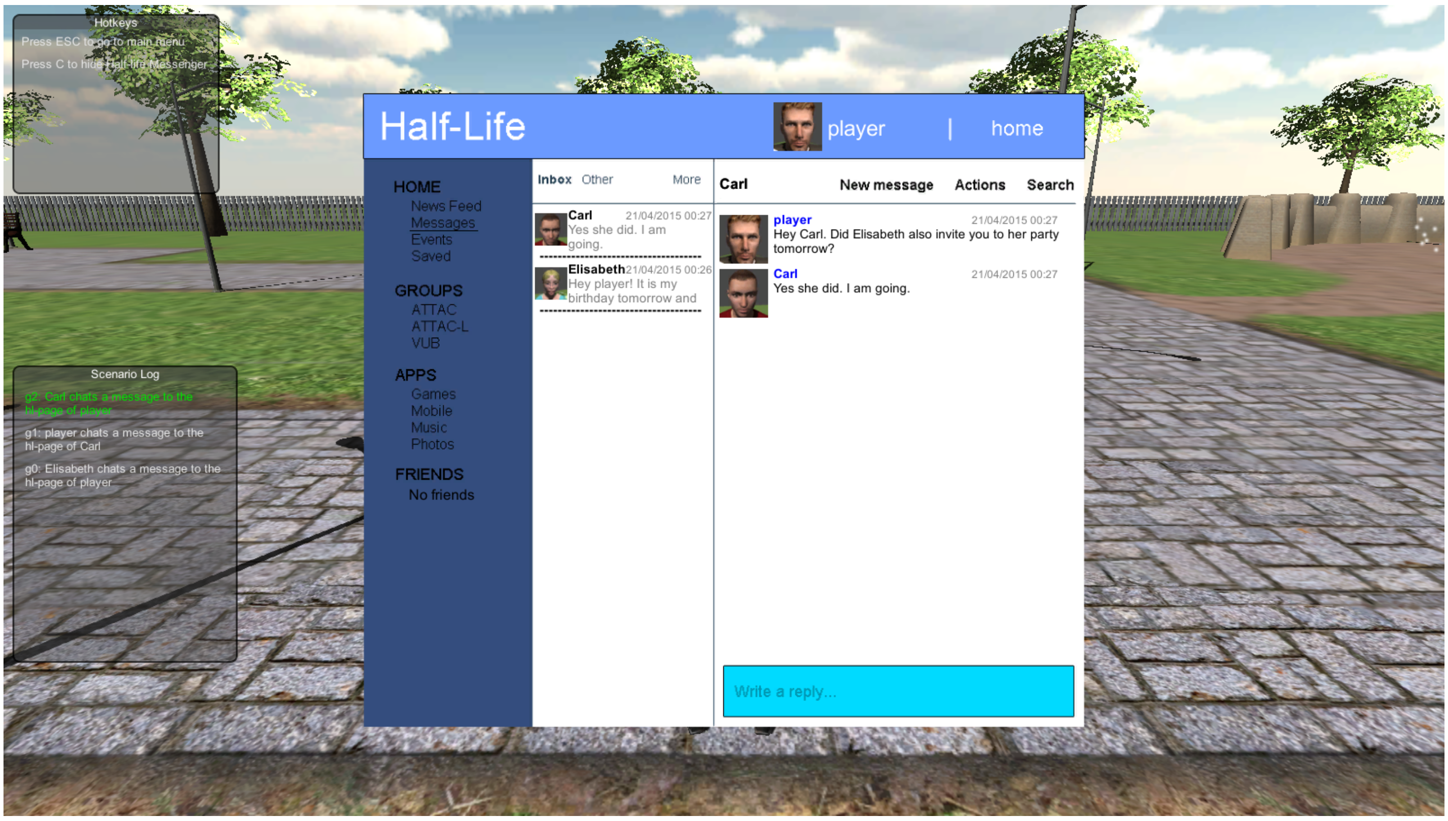


Figure 4.31: ATTAC-L Representation of the Story with a Half-Life Post

Half-Life Messenger

Figure 4.33 shows the ATTAC-L story with two conversations and three messages. Figure 4.32 shows the resulting Messenger overlay for this story. The player's Half-Life Messenger can always be seen by pressing the *"C"* key.



Hotkeys

Press ESC to go to main menu

Press C to hide Half-life Messenger

Scenario Log

g2: Carl chats a message to the hl-page of player

g1: player chats a message to the hl-page of Carl

g0: Elisabeth chats a message to the hl-page of player

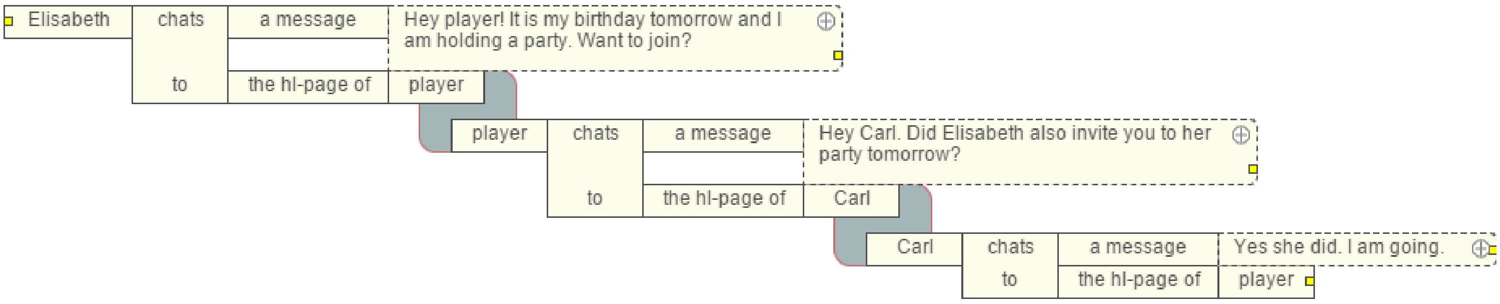
Carl 21/04/2015 00:27 Yes she did. I am going.

Elisabeth 21/04/2015 00:26 Hey player! It is my birthday tomorrow and

player 21/04/2015 00:27 Hey Carl. Did Elisabeth also invite you to her party tomorrow?

Carl 21/04/2015 00:27 Yes she did. I am going.

Figure 4.32: Half-Life Messenger Overlay



Elisabeth chats a message to the hl-page of player: Hey player! It is my birthday tomorrow and I am holding a party. Want to join?

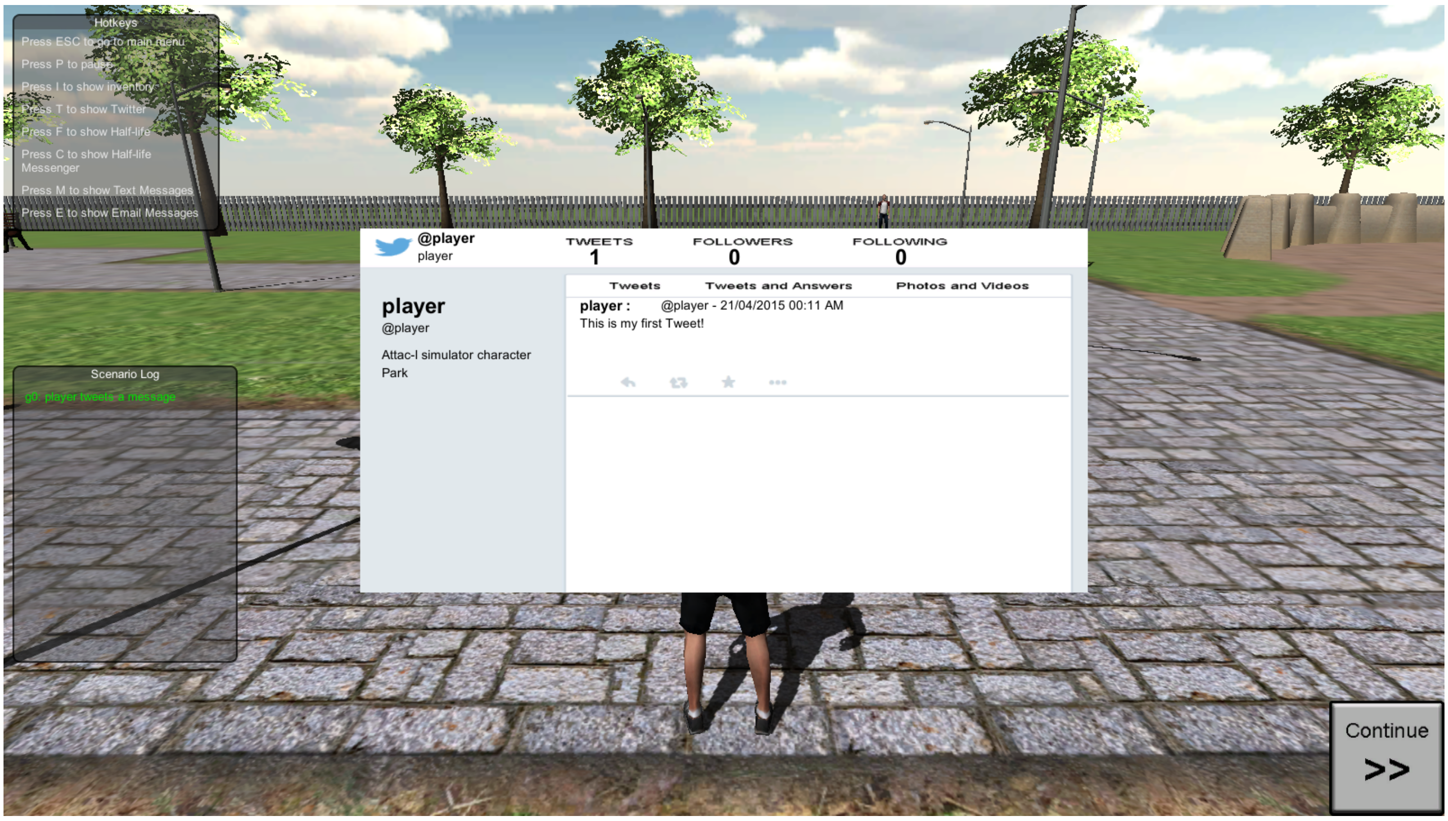
player chats a message to the hl-page of Carl: Hey Carl. Did Elisabeth also invite you to her party tomorrow?

Carl chats a message to the hl-page of player: Yes she did. I am going.

Figure 4.33: ATTAC-L Representation of the Story with Half-Life Messenger

Twitter

Figure 4.35 shows an ATTAC-L game move where player tweets the message "This is my first Tweet!". The corresponding Twitter overlay is shown in Figure 4.34. The player's Twitter page can always be brought up by pressing the *"T"* key.


Hotkeys
Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages
Scenario Log
@player
player
TWEETS
FOLLOWERS
FOLLOWING
Tweets
Tweets and Answers
Photos and Videos
Continue
>>

g0: player tweets a message

TWEETS 1 FOLLOWERS 0 FOLLOWING 0

player
@player
Attac-l simulator character
Park

player : @player - 21/04/2015 00:11 AM
This is my first Tweet!

Figure 4.34: Twitter Overlay - Performing a Tweet

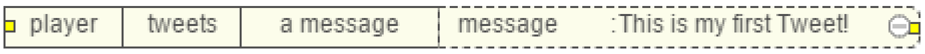


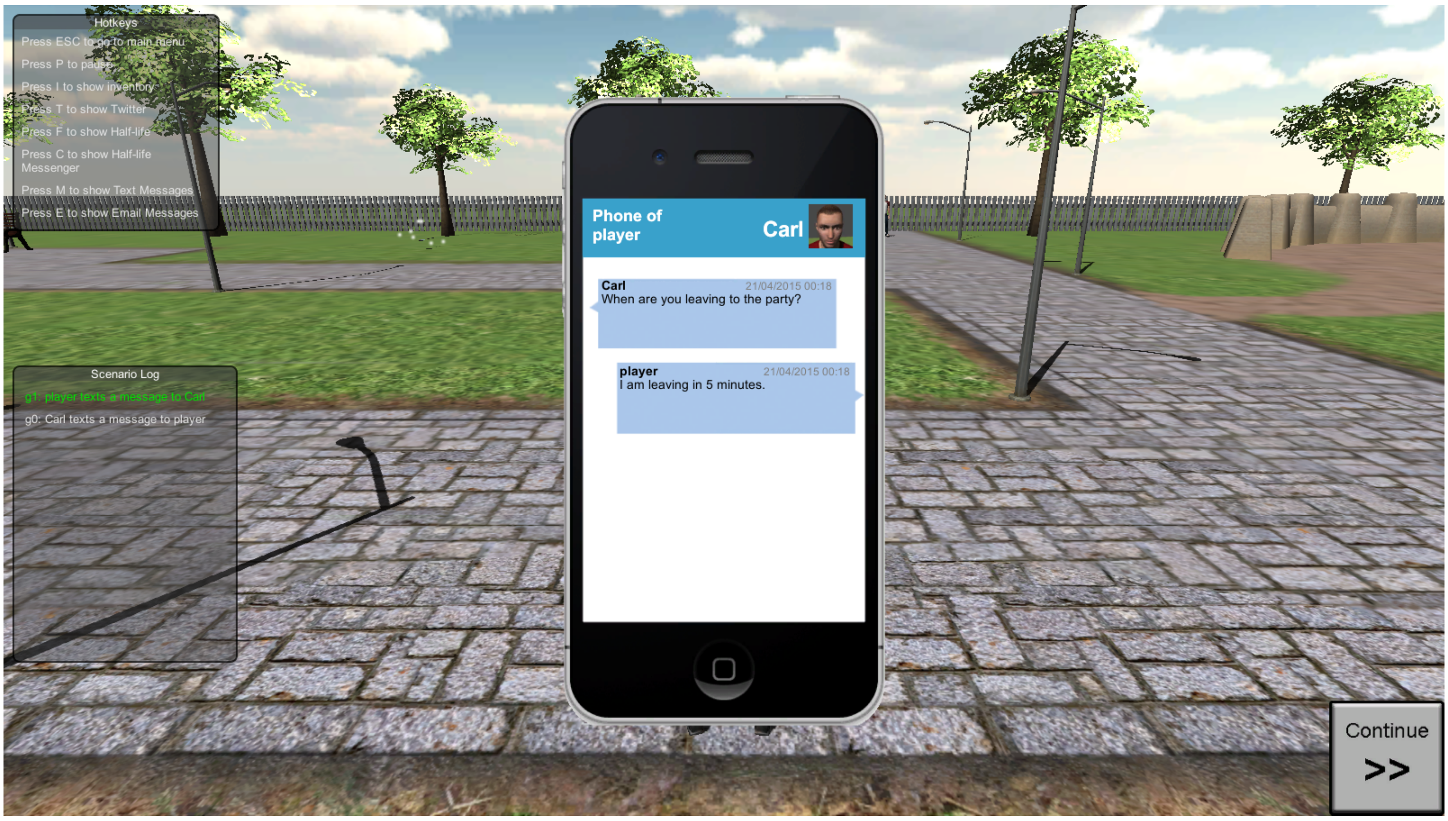
Figure 4.35: ATTAC-L Representation of the Story with a Tweet

4.4.6 Messaging Overlays

In this section we discuss the visuals used for the text messages (phone) and email messages (composing and inbox).

Text

The ATTAC-L story as given in Figure 4.37 starts with Carl sending a text message to player. In the next game move, player sends a text message to Carl. Figure 4.36 shows the result of the given example after the second game move. The top of the phone overlay shows the owner of the phone by means of a picture of the person and his name. The player's phone can be brought up by pressing the *"M"* key.


Hotkeys
Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages
Phone of player
Continue >>

Scenario Log
g1: player texts a message to Carl
g0: Carl texts a message to player

Carl
When are you leaving to the party? 21/04/2015 00:18

player
I am leaving in 5 minutes. 21/04/2015 00:18

Figure 4.36: Text Messages Overlay

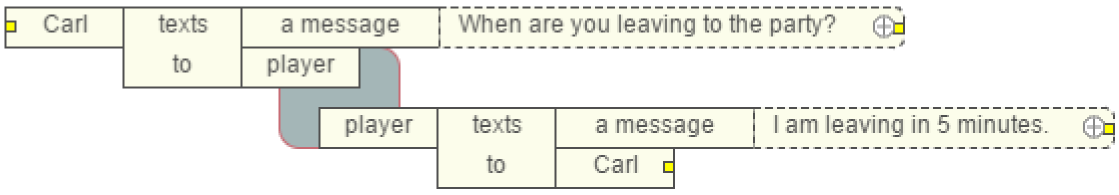
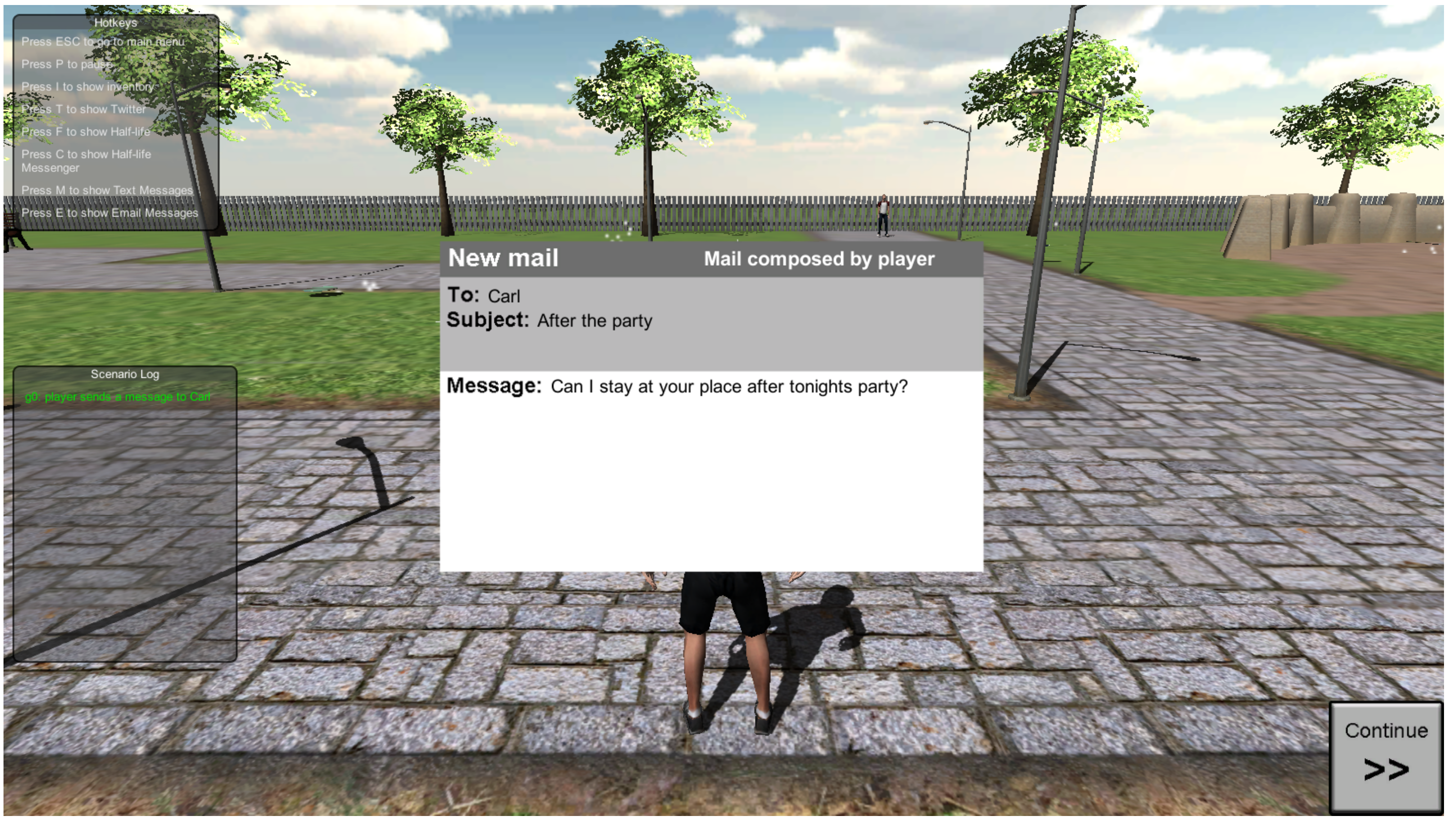

Carl
texts to
a message
player
When are you leaving to the party?
player
texts to
a message
Carl
I am leaving in 5 minutes.

Figure 4.37: ATTAC-L Representation of the Story with Text Messages

Email

The ATTAC-L story given in Figure 4.39 shows the player sending an email to Carl. Note that we know it is an email since the value brick has two entries (subject and message). Figure 4.38 shows the resulting composed email in an email overlay.


Hotkeys
Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages
Scenario Log
Continue
>>

g0: player sends a message to Carl

New mail Mail composed by player

To: Carl
Subject: After the party

Message: Can I stay at your place after tonights party?

Figure 4.38: Compose Email Overlay

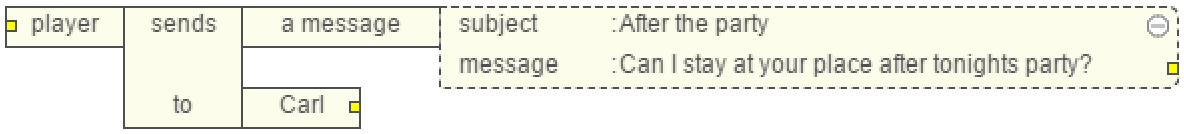
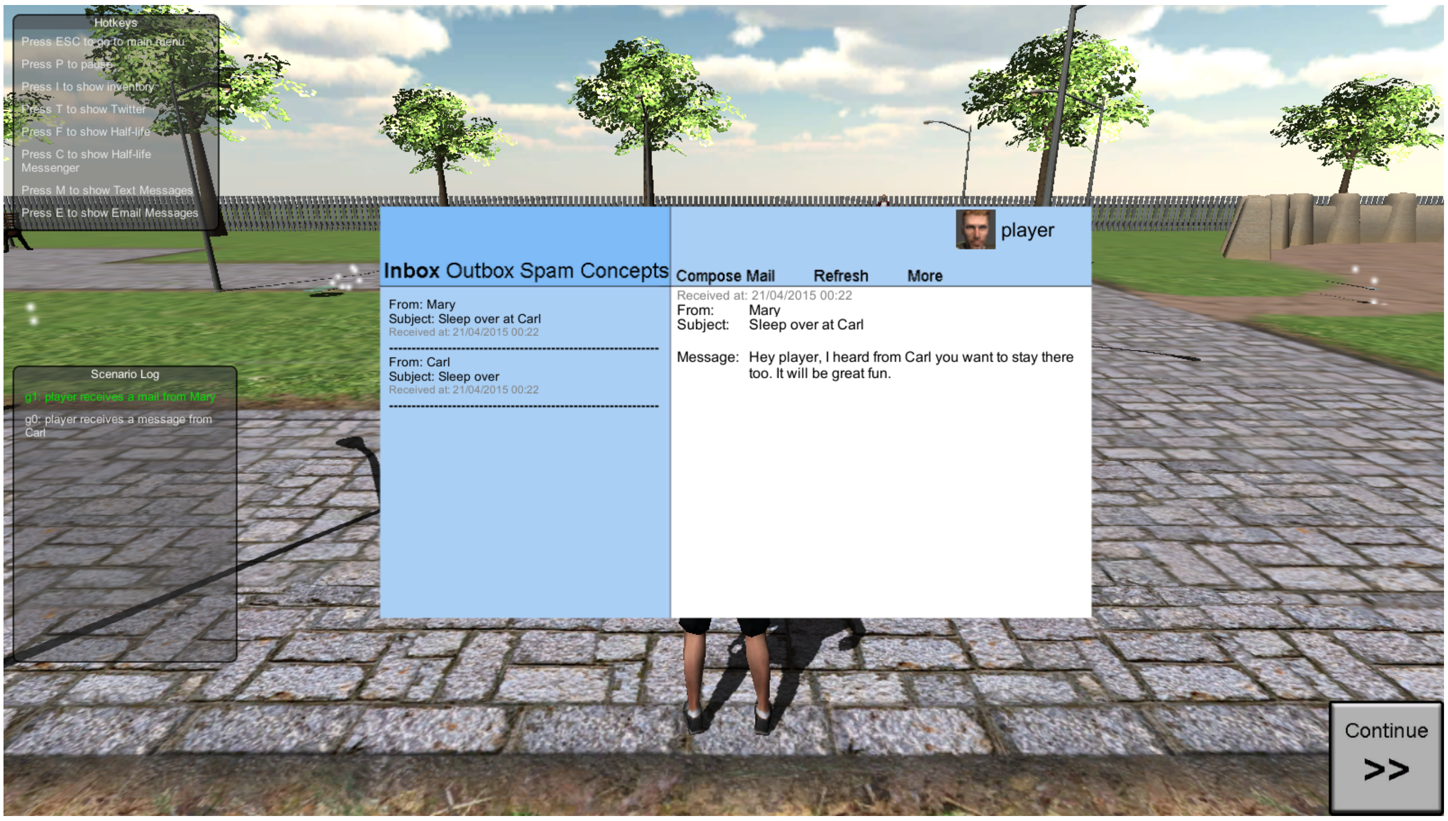


Figure 4.39: ATTAC-L Representation of the Story with Compose Email

The ATTAC-L story given in Figure 4.41 starts with player receiving a mail from Carl, followed by player receiving a mail from Mary. Figure 4.40 shows the resulting email inbox in the overlay. The left hand side of this overlay shows the latest received mails. Note that the newest mail is the one on top, and it is shown in greater detail on the right hand side of the overlay. The owner of the inbox is shown by means of his/her picture at the top right. The player's email inbox can be brought up by pressing the *"E"* key.



Hotkeys
Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages

Scenario Log
g1: player receives a mail from Mary
g0: player receives a message from Carl

Inbox Outbox Spam Concepts

From: Mary
Subject: Sleep over at Carl
Received at: 21/04/2015 00:22

From: Carl
Subject: Sleep over
Received at: 21/04/2015 00:22

player
Compose Mail Refresh More

Received at: 21/04/2015 00:22
From: Mary
Subject: Sleep over at Carl

Message: Hey player, I heard from Carl you want to stay there too. It will be great fun.

Figure 4.40: Email Inbox Overlay

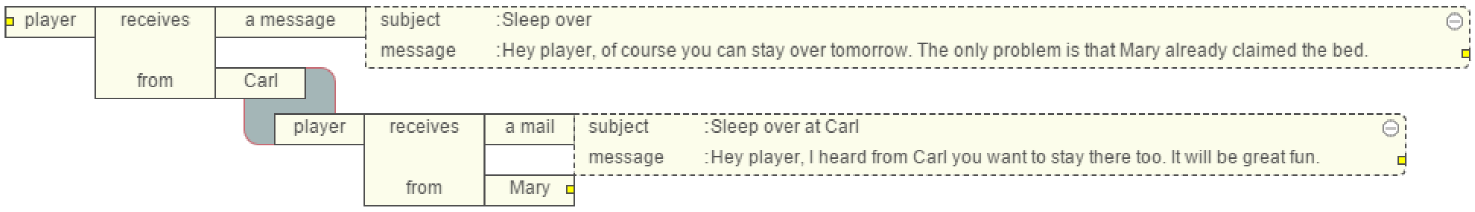

player
receives
a message
subject :Sleep over
message :Hey player, of course you can stay over tomorrow. The only problem is that Mary already claimed the bed.
from
Carl
player
receives
a mail
subject :Sleep over at Carl
message :Hey player, I heard from Carl you want to stay there too. It will be great fun.
from
Mary

Figure 4.41: ATTAC-L Representation of the Story with Email Inbox

5 Implementation

In this chapter, the implementation of the simulator using the Unity game engine is described in more detail. We start by defining the most important concepts in Unity that will be used in our simulator. We then give the system architecture. Following this, we provide the design using UML and Sequence diagrams. Finally we conclude this chapter by explaining the role of the configuration file.

5.1 Concepts in Unity

As we decided to implement our Simulator with Unity (see section 3.2.1), we start by elaborating on some of the key concepts of Unity used for the implementation. We explain these concepts and indicate how they have been used for the implementatin of our simulator.

5.1.1 Scene

A virtual world (i.e., an environment or level), is called a *Scene*. In our simulator, we have three scenes, called the *Park* (Figure 5.1), the *Office* (Figure 5.2), and the *House* (Figure 5.3). The scene is where the characters are placed and the activities will take place.

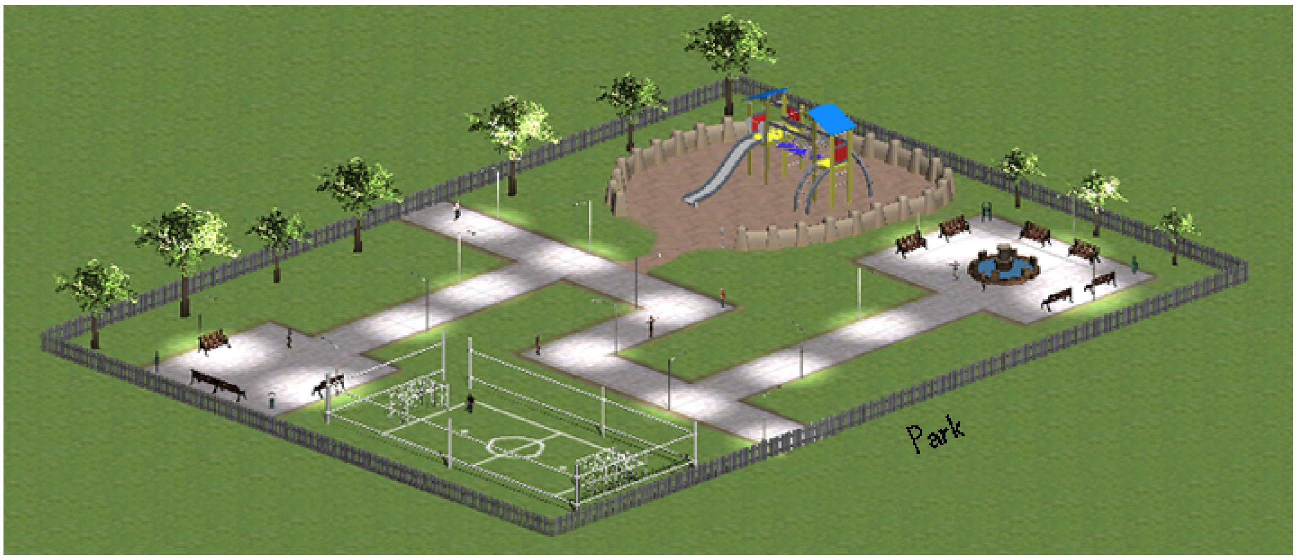


Figure 5.1: Isometric View of the Park Scene

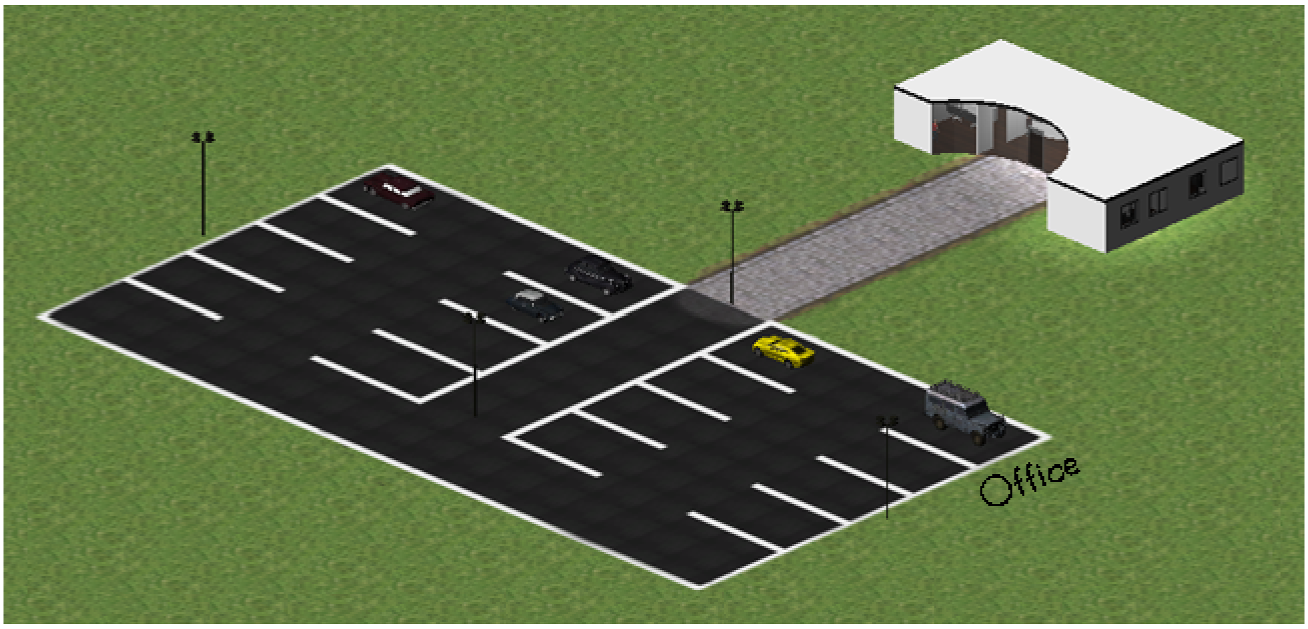


Figure 5.2: Isometric View of the Office Scene

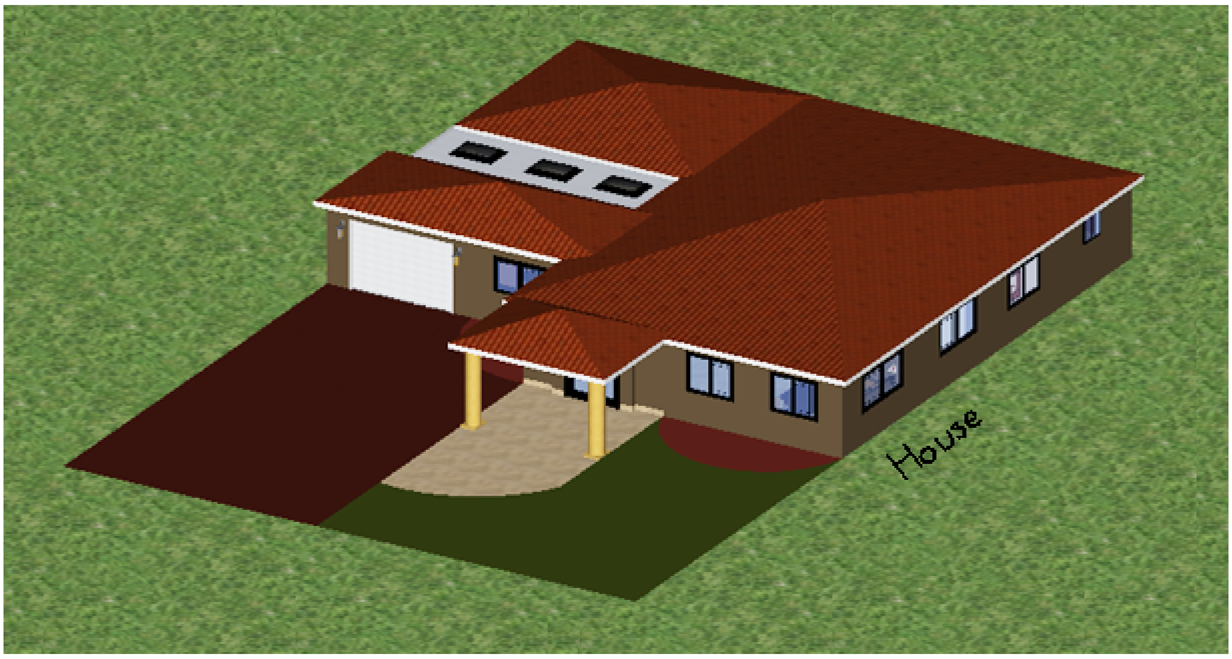


Figure 5.3: Isometric View of the House Scene

5.1.2 GameObject

When creating a game (or simulation) in Unity, every object is considered a *GameObject*. Examples of GameObjects are: the player character, NPC characters, lights, etc. All GameObjects present in the current scene are organized in a hierarchical manner. Dragging one GameObject under another creates the concept of "Parenting". Every child will inherit the movement and rotation of its parent. A GameObject is the basis for every logical component of a game. They are containers for one or more *Components*, which take care of behavior, positioning, visuals, physics, etc. Every GameObject always contains a "Transform" component which cannot be removed. This contains the position and orientation of the object. Multiple components can be added to a single GameObject. Some examples of components used in our simulator are: Animation, Box Collider, Nav Mesh Agent, Mesh Renderer, Mesh Collider, GUIText, and Script. We explain them further.

- An *Animation* component is used to manage the animations of a 3D model, e.g., the walk and idle animations of a character.

- A *Box Collider* creates a box around the GameObject and triggers an event if something collides with it.

- A *Mesh Collider* has the same purpose, but for a more complex shape. A mesh is a 3D figure built from flat polygons (mostly triangles), the *Mesh Renderer* simply renders it onto the scene.
- *Nav Mesh* and *Nav Mesh Agent* components are used in conjunction to deal with automatic path finding in a scene (see section 5.1.4).
- *GUIText* is used to place GUI-elements on the screen.
- *Script* allows attaching scripts to GameObjects. The scripts can be written in C#, JavaScript or Boo. The scripts for our simulator are written in C#.

5.1.3 Prefabs

A *Prefab* is a GameObject which has predefined components and settings. A prefab can be seen as a template out of which new GameObjects (i.e., instances) can be created. Changes made to the prefab are also reflected to all of its instances. However, changing components and settings for an individual GameObject based on a prefab remains possible.

In our simulator there are several prefabs used:

Notification
: Whenever feedback is given (e.g., "player picked up shovel" or "player befriended Carl on Half-Life") the notification prefab is used to create the GameObject that displays the message. It contains a "GUIText" in which a message can be placed and has a script attached that scrolls the text up and fades it out slowly.

Rain
: Rain can be turned on or off in the main menu. When turned on, the rain prefab is used to create the GameObject that makes it rain. For performance reasons we chose to only make it rain in a radius around the player. Since the camera stays attached to the player, it gives the illusion it is raining in the entire scene. Every rain drop is a GameObject; it has to be duplicated a lot to have the appearance of rain, hence the need of a prefab.

Conversation Overlay
: The way of showing a conversation between two characters is always the same, independent of the characters involved or the scene. The conversation prefab consists solely of a "Script" component. This script deals

with drawing the overlay on the screen and fitting the correct conversation text on top of it. More information about the Conversation Overlay can be found in section 4.3.5.

Items

Several items can be present in the scene for the player to be picked up. For now, we have three items defined in our simulator: a shovel, an USB(-stick), and a CD. Each of these items has their own prefab. These prefabs contain the following components: “Mesh Renderer” (its shape), “Box Collider” (to check whether a character is on top of the item), “Particle System” (for highlighting purposes), and a “Script”. The latter handles the logic of picking up the item: first it checks if a character is on top, continuing with displaying the pick-up animation of the character, adding the item to the inventory (if the character is the player); removing the item from the scene; and displaying a notification message stating that the item is picked up. More information about items can be found in section 4.3.2.

5.1.4 Nav Mesh

Nav Mesh (Navigational Mesh) is Unity’s built-in path finding system. Since in the simulation characters move automatically, it allows us to provide intelligent and accurate movement for them. Traditional path finding in a 3D world around 3D objects is slow because of the complexity of the objects. This is why Unity uses a Nav Mesh; it is a simple 3D mesh coming from the geometry of more sophisticated elements in a scene. Because of the “simplicity” it is easier to navigate and find paths in the Nav Mesh. Figure 5.4 shows the Nav Mesh for our Park scene, note that only the area selected in blue is walkable.

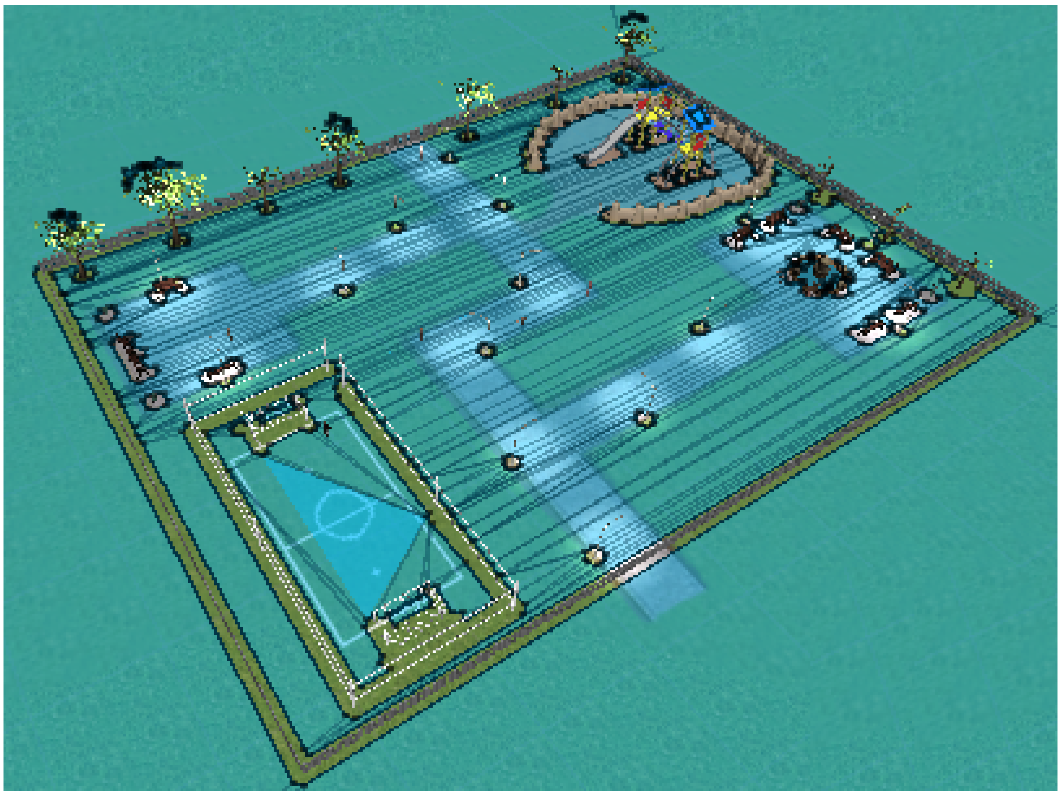


Figure 5.4: Nav Mesh of Park

5.2 System Architecture

Figure 5.5 presents the high-level design of our simulator. In the next sub sections, we describe the major components of the system and how they interact with one another.

Note that a game engine consists of layers with components. A general system architecture of a game engine has been given in section 3.2.1, Figure 3.5. Many of these components are also present in Unity, however we decided not to depict them if they are of no or little importance in our simulator.

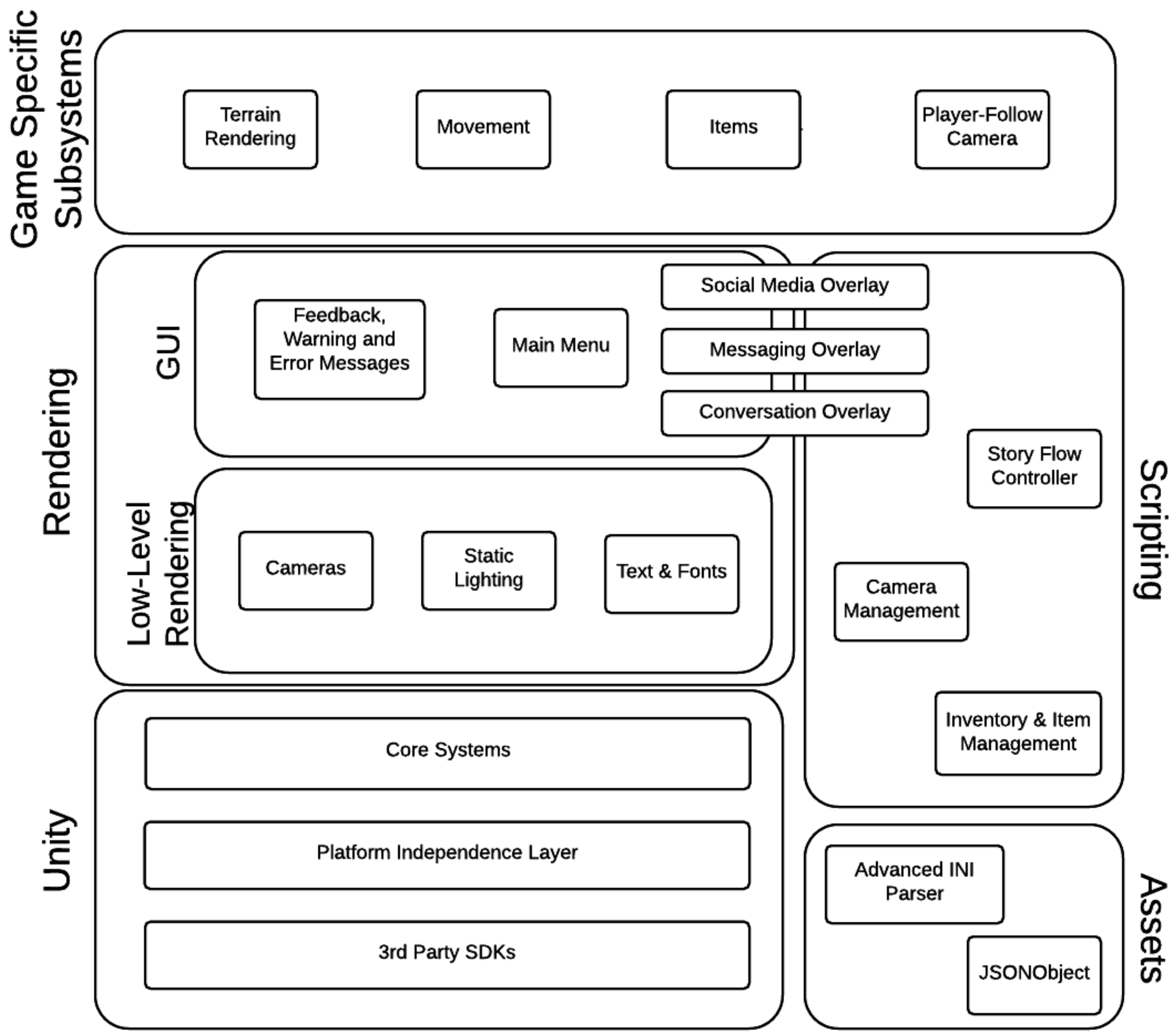


Figure 5.5: System Architecture

5.2.1 Unity

The Unity component consists of several technologies layered on top of each other. These are systems that are being taken care of by Unity itself. The most important ones are 3rd Party SDKs (e.g., DirectX, OpenGL, PhysX);

Platform Independence Layer (e.g., Platform Detection, File System Access); and Core Systems (e.g., Math Library, Memory Allocation).

5.2.2 Assets

The Assets component consists of third party libraries. We used two assets. The first one is called JSONObject and it is used for the parsing of the JSON input. The second one is called Advanced INI Parser and we use it to read values from our configuration file.

5.2.3 Rendering

The Rendering component consists of a Low-Level Rendering and GUI component. We will discuss both of them here.

Low-Level Rendering

The Low-Level Rendering component is part of the Rendering system of a game engine. The main focus in this component is that the geometric primitives are rendered as quickly and richly as possible. The viewpoint of the player is not taken into account here. In our simulator, the corresponding subcomponents are Text & Fonts, Cameras, and Static Lighting. As said, these subcomponents are rendered as quickly as possible, they will receive more attention in later components when they are needed.

GUI

Our GUI component deals with overlaying 2D graphics on the 3D scene for the purpose of user interaction. Examples of such GUI components in our simulator are: Feedback, Warning and Error Messages, Main Menu, Social Media Overlay, Messaging Overlay, and Conversation Overlay.

5.2.4 Scripting

The Scripting component contains all our scripts written in C#. Scripts are used to program the logic of the game. Some areas for which they are used include: Inventory & Item Management; Camera Management (making sure the camera keeps its viewpoint behind the player); Story Flow Controller (to interpret the ATTAC-L story); and Social Media, Messaging, and Conversation Overlay (to call the corresponding GUI overlays at the appropriate time).

5.2.5 Game Specific Subsystems

This is the top-level component. It deals with features of the game. Examples are: Terrain Rendering, Movement (making the characters move), Items, Player-Follow Camera (to make sure if the player moves, the camera moves with him).

5.3 Design

In this section we give the UML (Unified Modeling Language) class and sequence diagram for the simulator. We provide explanations below each figure.

Figure 5.6 shows a compact version of the UML class diagram of our simulator. This diagram only shows the classes, without methods and variables. For the full diagram, refer to Figure D.1 in Appendix D. In the following sections, a brief explanation of the different classes is given.

EscapeGUI

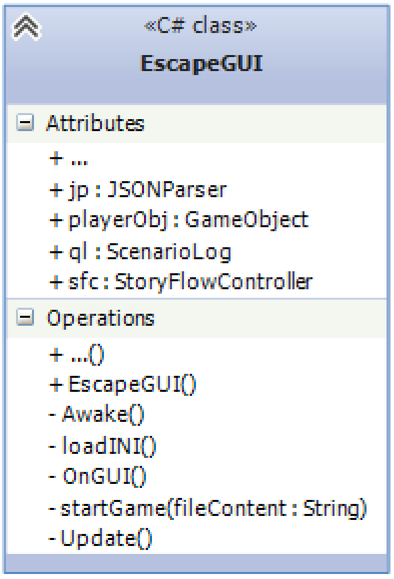


Figure 5.7: EscapeGUI Class

When launching the simulator or pressing “ESC” during a simulation, the main menu appears. This menu is defined in the EscapeGUI class, given in

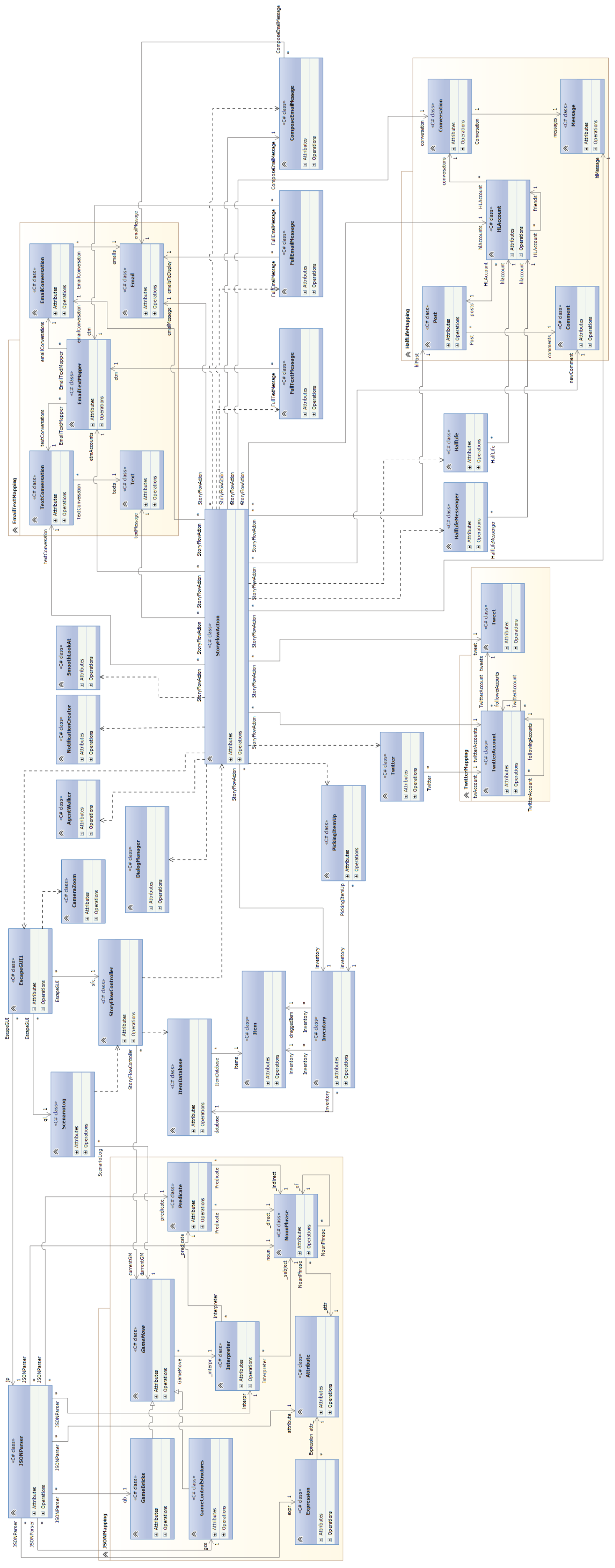


Figure 5.6: UML Class Diagram (Compact)

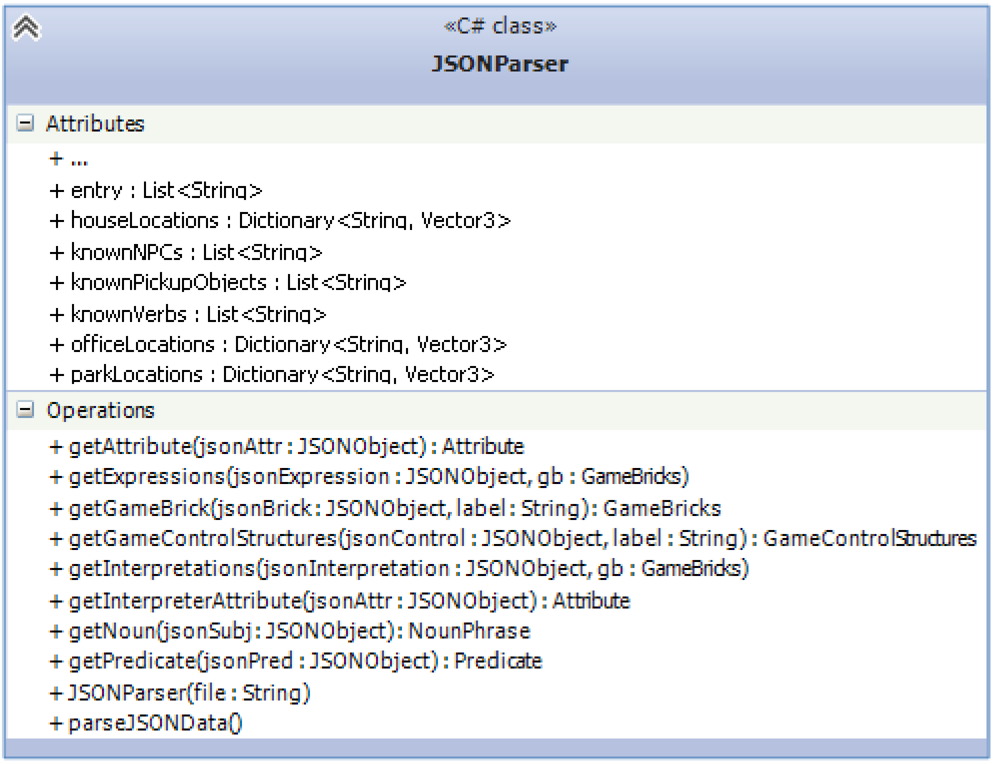
Figure 5.7, it is directly attached to our (empty) "Main Menu" GameObject via a Script component. This means that this class will be executed as soon as the simulator is started, and only stops executing when the simulator is turned off. Because of this property, it takes care of the following:

- Everything related to the main menu (options and environment changing).
- Hotkey buttons behavior.
- Reading the configuration file and setting the variables accordingly.
- If a story is started, it checks whether the JSON code is valid and starts the simulation or gives appropriate feedback.

As soon as the JSON is pasted in the input box and the Start Scenario button is pressed, we launch the *startGame* method. Before the simulation is started, we have to check the ATTAC-L JSON code for validity. We do this by calling the *parseJSONData* method from the JSONParser class, it is explained below. If the JSON is valid the StoryFlowController and QuestLog classes are initialised and the simulation will be started. The data structure used to store the JSON is explained below in JSONMapping.

The *loadINI* method is used to read the configuration file.

JSONParser



```
«C# class»
JSONParser

Attributes
+ ...
+ entry : List<String>
+ houseLocations : Dictionary<String, Vector3>
+ knownNPCs : List<String>
+ knownPickupObjects : List<String>
+ knownVerbs : List<String>
+ officeLocations : Dictionary<String, Vector3>
+ parkLocations : Dictionary<String, Vector3>

Operations
+ getAttribute(jsonAttr : JSONObject) : Attribute
+ getExpressions(jsonExpression : JSONObject, gb : GameBricks)
+ getGameBrick(jsonBrick : JSONObject, label : String) : GameBricks
+ getGameControlStructures(jsonControl : JSONObject, label : String) : GameControlStructures
+ getInterpretations(jsonInterpretation : JSONObject, gb : GameBricks)
+ getInterpreterAttribute(jsonAttr : JSONObject) : Attribute
+ getNoun(jsonSubj : JSONObject) : NounPhrase
+ getPredicate(jsonPred : JSONObject) : Predicate
+ JSONParser(file : String)
+ parseJSONData()
```

Figure 5.8: JSONParser Class

Figure 5.8 shows the JSONParser class. In order to parse the JSON, we use the JSONObject class[1], it is freely available on the Unity Asset Store. Everything is stored in the data structure as explained below in JSONMapping. We start by checking if the JSON format is valid and continue if it is. In order to check if the contents of the JSON are eligible with our simulator, we keep a list of: accepted verbs, known NPCs (names), known items, and known locations (name and position) per scene.

[1] `https://www.assetstore.unity3d.com/en/\#!/content/710`, accessed 05/05/2015.

JSONMapping

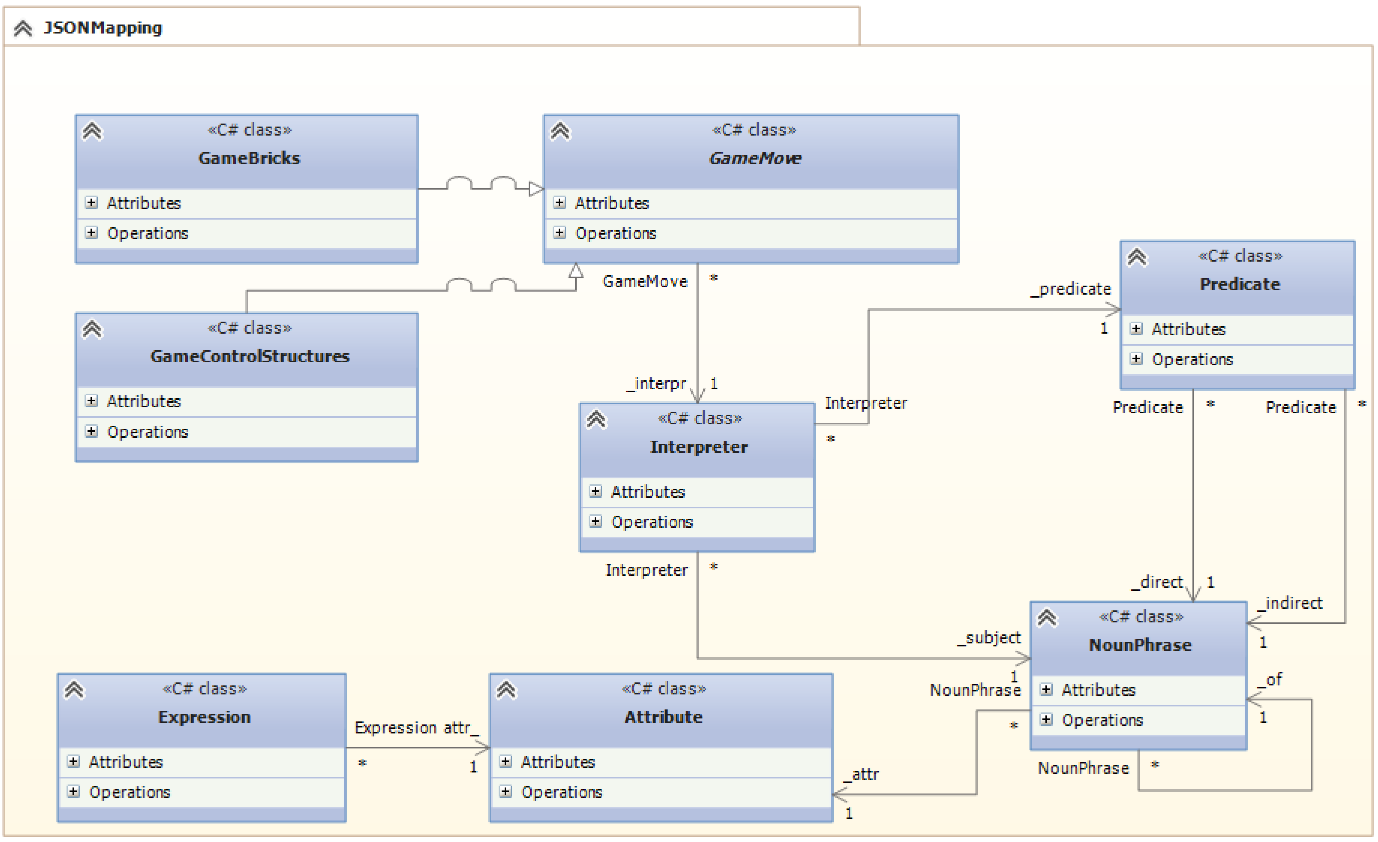


Figure 5.9: JSONMapping Namespace

JSONMapping is a namespace which contains the following classes used to store the JSON code (see Figure 5.9):

- *GameMove (abstract class)*
  A GameMove contains all the elements that make up a game move. It is either a construction of bricks (GameBricks) or a control structure (GameControlStructures). The other classes mentioned in JSONMapping are all mappings of elements which appear in every GameBricks game move. We will explain the data structure as implemented in C#.

  – *GameBricks*
    GameBricks is derived from GameMove and uses a string "_next" (the flowlabel) and an "_expr" element.

  - *GameControlStructures*
    GameControlStructures is derived from GameMove and uses a string "_type" (cho, oid, or con) and a list of strings "_paths" (every entry is a flowlabel).

- *Expression*
  Expression is the mapping of the "_expr" element, which contains "word_" and "attr_".

- *Attribute*
  Attribute is the mapping of the possible "_attr" element, this occurs when there is a value brick present. It contains a dictionary of string values.

- *Interpreter*
  Interpreter is the mapping of the "_interpr" element, it always contains a "_subject" and "_predicate" element, and optionally a "_passive" element. The _subject element is defined as a NounPhrase, while the _predicate is defined as a Predicate.

- *NounPhrase*
  NounPhrase is the mapping of the "_det", "_noun", "_nom", "_of" and "_attr" fields.

- *Predicate*
  Predicate contains a "_direct" and "_indirect" as a NounPhrase and "_verb" as a string.

ScenarioLog

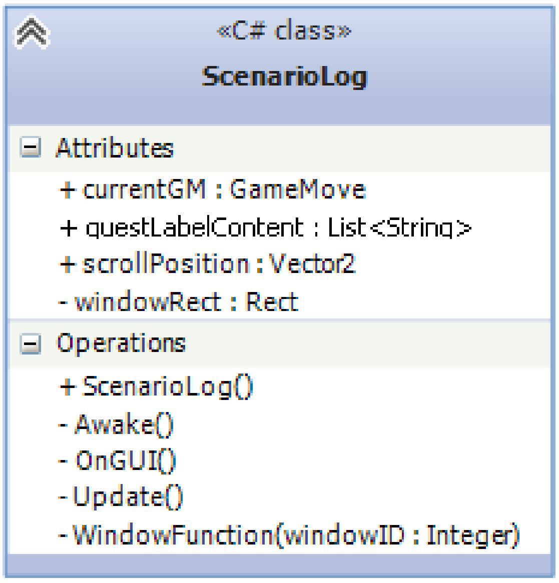


Figure 5.10: ScenarioLog Class

The ScenarioLog class, given in Figure 5.10, deals with drawing the Scenario Log panel on the left hand side of the screen. It requires an instance of a GameMove so that it can retrieve the flowlabel and content of the current game move being executed, and store it in a list for displaying in the panel.

StoryFlowController

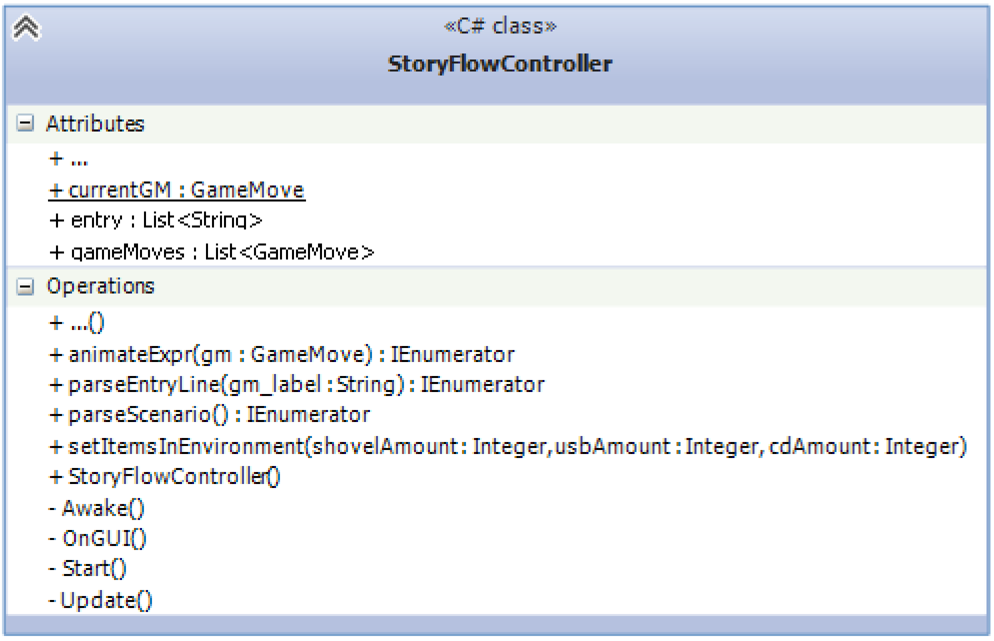

«C# class»
StoryFlowController
Attributes
+ ...
+ currentGM : GameMove
+ entry : List<String>
+ gameMoves : List<GameMove>
Operations
+ ...()
+ animateExpr(gm : GameMove) : IEnumerator
+ parseEntryLine(gm_label : String) : IEnumerator
+ parseScenario() : IEnumerator
+ setItemsInEnvironment(shovelAmount : Integer, usbAmount : Integer, cdAmount : Integer)
+ StoryFlowController()
- Awake()
- OnGUI()
- Start()
- Update()

Figure 5.11: StoryFlowController Class

After the JSON code is entered in the designated input box in the main menu, it will be validated and the simulation can be started. Upon clicking the start button, the EscapeGUI class will attach the StoryFlowController class (see Figure 5.11) to the player's GameObject. Before the execution of the first game move, the items are placed in the scene using *setItemsInEnvironment*. The amount is decided by the values in the configuration file. As soon as all the items are placed at random locations in the scene, we start the simulation's core loop *parseScenario*. This method contains a for-loop which goes over the list of entries (most of the time this only contains one entry), and starts the *parseEntryLine* method for each entry. This method will check if the current game move is a brick or a control structure. If it is a brick, we perform the simulation of the game move by calling the *animateExpr* method. This method consists of a switch-statement, based on the verb we call the appropriate simulation method in StoryFlowAction, described below. After we simulated the first game move brick, we check if it has a "_next" element and call *parseEntryLine* again with the next game move. If the game move is a control structure, we perform the appropriate action. For choice, we place the contents in a list and show the choice buttons on the screen. The button chosen represents the game move to be executed in the *parseEntryLine* method. For order independence or concurrency we execute the *parseEntryLine* method one by one, by its contents. Note that with order independence we shuffle the contents list first, so the order is random.

StoryFlowAction

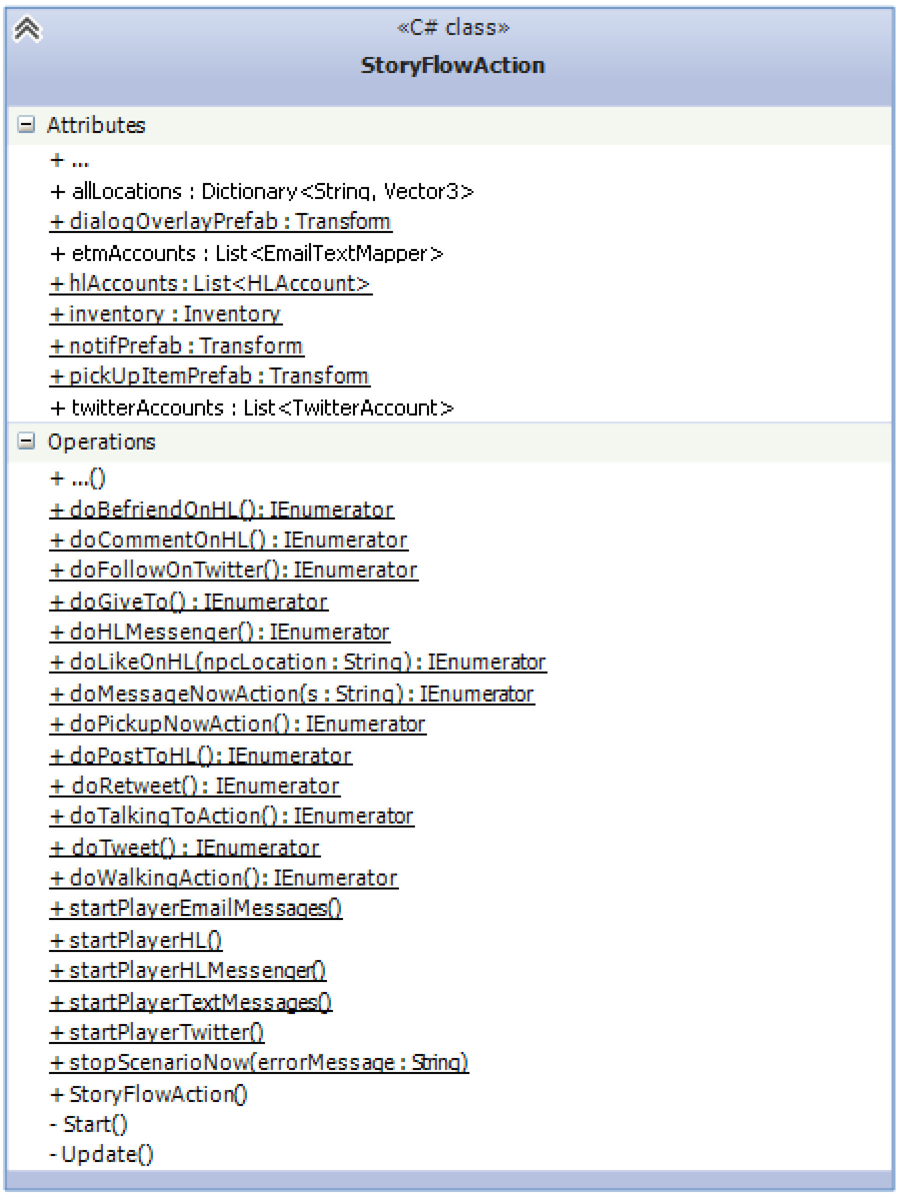

«C# class»
StoryFlowAction
Attributes
+ ...
+ allLocations : Dictionary<String, Vector3>
+ dialogOverlayPrefab : Transform
+ etmAccounts : List<EmailTextMapper>
+ hlAccounts : List<HLAccount>
+ inventory : Inventory
+ notifPrefab : Transform
+ pickUpItemPrefab : Transform
+ twitterAccounts : List<TwitterAccount>
Operations
+ ...()
+ doBefriendOnHL() : IEnumerator
+ doCommentOnHL() : IEnumerator
+ doFollowOnTwitter() : IEnumerator
+ doGiveTo() : IEnumerator
+ doHLMessenger() : IEnumerator
+ doLikeOnHL(npcLocation : String) : IEnumerator
+ doMessageNowAction(s : String) : IEnumerator
+ doPickupNowAction() : IEnumerator
+ doPostToHL() : IEnumerator
+ doRetweet() : IEnumerator
+ doTalkingToAction() : IEnumerator
+ doTweet() : IEnumerator
+ doWalkingAction() : IEnumerator
+ startPlayerEmailMessages()
+ startPlayerHL()
+ startPlayerHLMessenger()
+ startPlayerTextMessages()
+ startPlayerTwitter()
+ stopScenarioNow(errorMessage : String)
+ StoryFlowAction()
- Start()
- Update()

Figure 5.12: StoryFlowAction Class

The action to be simulated is based on the game move's verb, these actions are selected in a switch-statement in the StoryFlowController class. E.g., the verb "walks-to" is linked to the action "doWalkingAction". All these action methods are defined in the StoryFlowAction class (Figure 5.12). The methods defined in this class that start with “do” deal with moving the player or NPC, showing overlays, picking up items, etc. It basically performs what is expressed in the game move. The methods with the prefix “start” get called from the EscapeGUI class when the corresponding hotkey is pressed, these methods show the overlays. The method *stopScenarioNow* is called when an error occurs in the current game move, it will give detailed information about the error before halting the simulation and returning to the main menu.

Item, ItemDatabase & Inventory

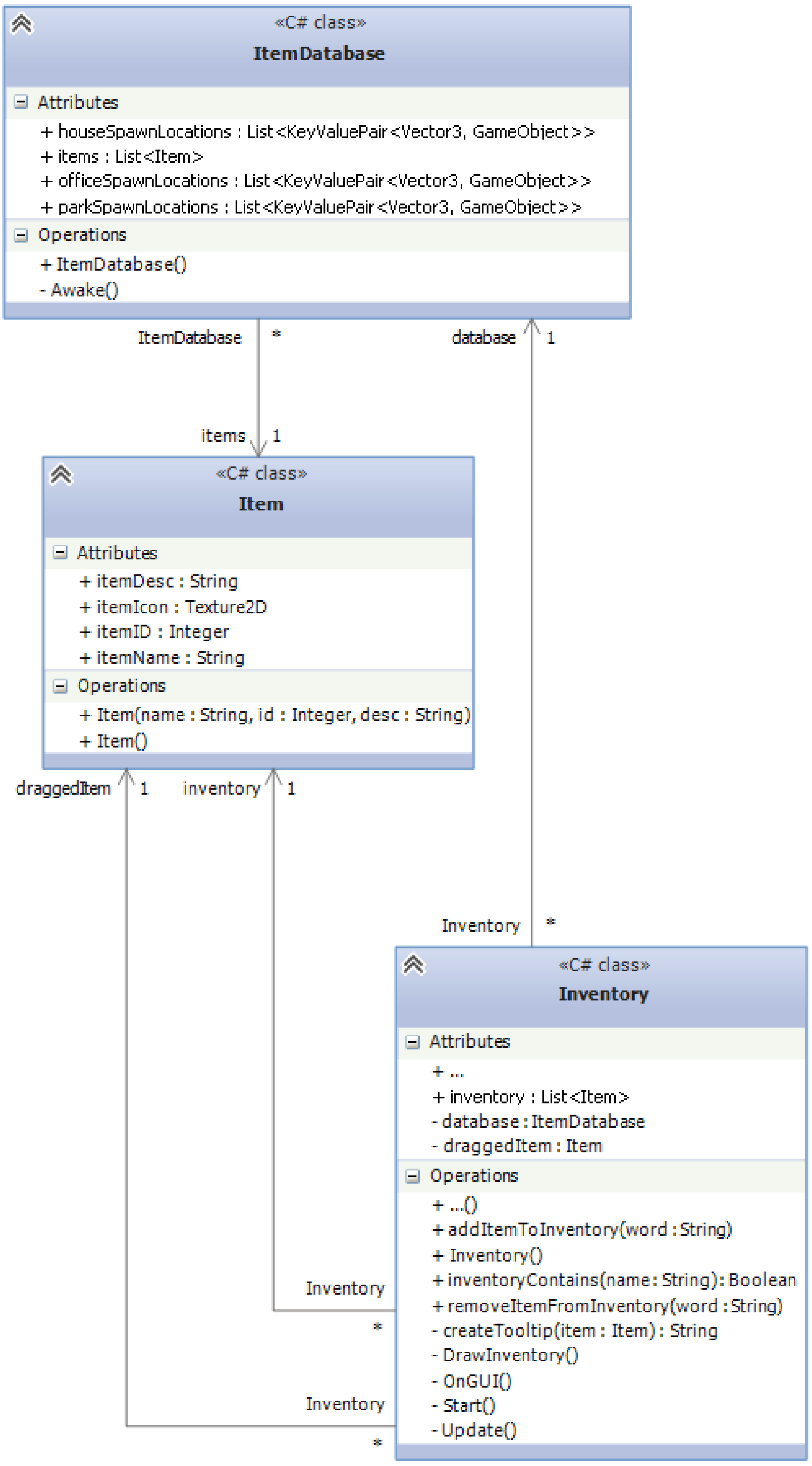


Figure 5.13: Item, ItemDatabase and Inventory Class

Figure 5.13 shows the classes Item, ItemDatabase, and Inventory, and how they interact with each other.

The Item class defines the data structure of an item, i.e. name, id, description, and an icon (2D texture).

The ItemDatabase contains a list where the items that are known in the simulator can be defined. In our simulator this list contains three items: a shovel, an USB, and a CD. Note that for the purpose of our simulator, a list is sufficient. When a large collection of items is necessary, other implementations are required (e.g., a true database). This class also contains three other lists (*parkSpawnLocations*, *officeSpawnLocations*, and *houseSpawnLocations*), each of these lists' entries contain a KeyValuePair of its location in the 3D scene (defined as Vector3) and a reference to its GameObject. These locations are all the possible places where an item can be located in a scene. The reference is required to be able to delete the item's GameObject when it is picked up, initially this is set to null. It is updated to match the item's prefab later.

The Inventory class contains methods for drawing the inventory on the screen (called when the "*I*" hotkey is pressed), controlling the drag & drop behavior, adding and removing items, etc. The Inventory class is attached to a GameObject with a Script component, since there is only one inventory available in the simulator (the player's). Note that instead of creating a new instance of Inventory every time (e.g. in StoryFlowAction or PickingItemUp), the instance is retrieved from the GameObject.

PickingItemUp

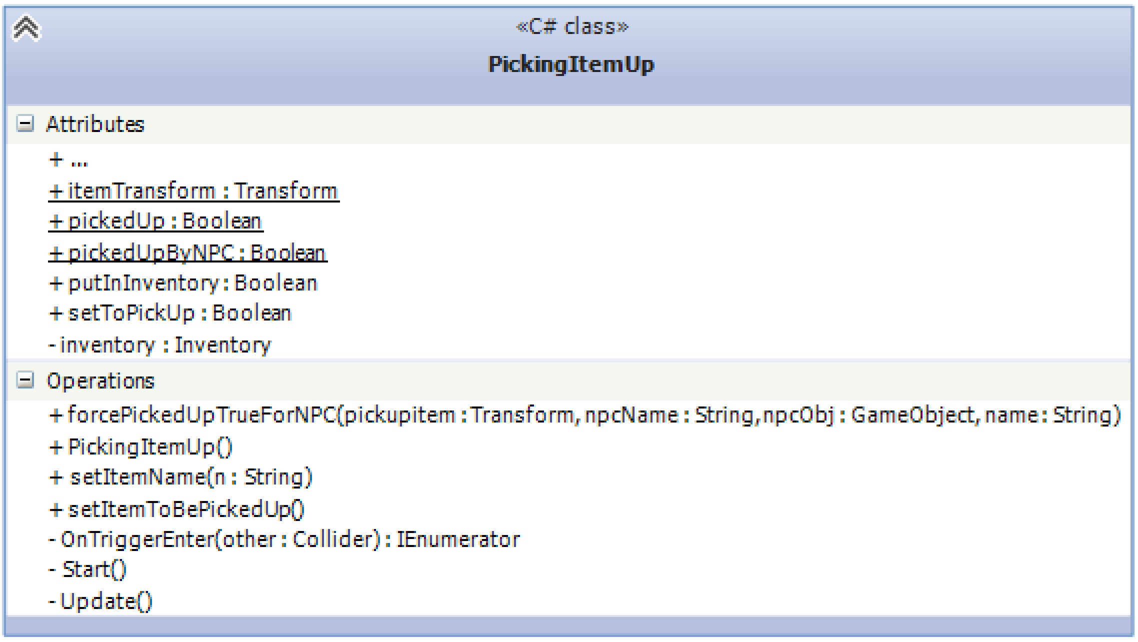

«C# class»
PickingItemUp
Attributes
+ ...
+ itemTransform : Transform
+ pickedUp : Boolean
+ pickedUpByNPC : Boolean
+ putInInventory : Boolean
+ setToPickUp : Boolean
- inventory : Inventory
Operations
+ forcePickedUpTrueForNPC(pickupitem : Transform, npcName : String, npcObj : GameObject, name : String)
+ PickingItemUp()
+ setItemName(n : String)
+ setItemToBePickedUp()
- OnTriggerEnter(other : Collider) : IEnumerator
- Start()
- Update()

Figure 5.14: PickingItemUp Class

The PickingItemUp class, given in Figure 5.14, checks if the item where the player or NPC is on top of is the item to be picked up. If it is, the item is removed from the scene, added to the inventory (if it is player), and a notification is shown containing information about what item is picked up by who. The check for the correct item is necessary to avoid accidentally picking up an item when walking over one, while going towards another.

All of our items are prefabs. When our items are about to be put in the scene, we retrieve the item's prefab and attach the PickingItemUp class to it. We use the *setItemName* method to indicate what item the script is being attached to. As it has been said above, the lists containing our items is represented by entries with a KeyValuePair consisting of a Vector3 and a GameObject. The latter refers to the item's prefab, containing the PickingItemUp Script component.

CameraZoom

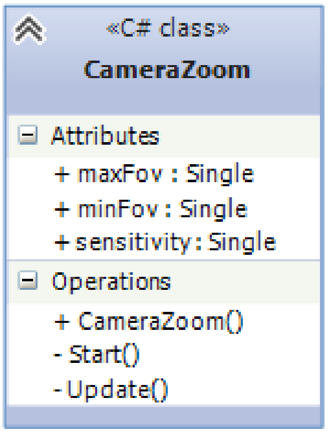


Figure 5.15: CameraZoom Class

This class changes the field of view of the camera by zooming in or out with the mouse scroll wheel. CameraZoom (see Figure 5.15) is attached to the camera's GameObject. The EscapeGUI class takes care of disabling the zooming functionality when the menu is open, and enabling it again when closed. It does this by simply enabling or disabling the CameraZoom Script component of the GameObject.

NotificationCreator

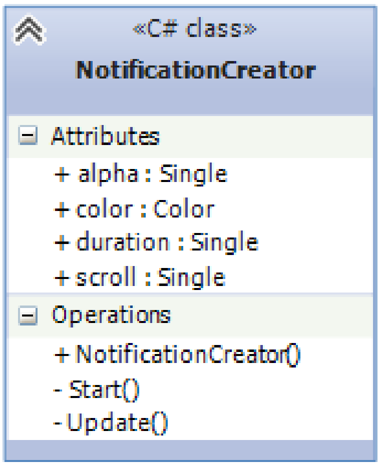


Figure 5.16: NotificationCreator Class

The NotificationCreator class (Figure 5.16) is put in a prefab as a Script component, together with a GUIText. The class deals with showing an information message that slowly fades out over time.

SmoothLookAt

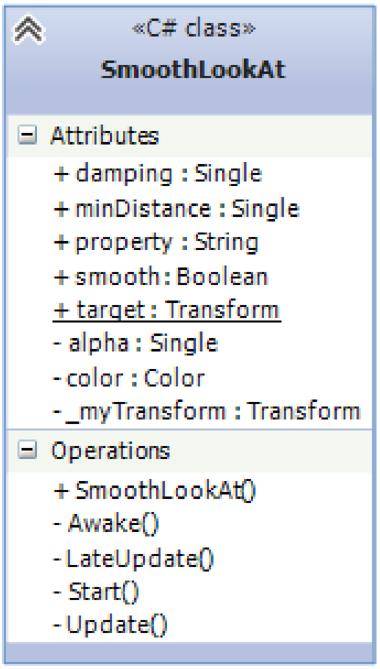


Figure 5.17: SmoothLookAt Class

Whenever an action occurs where movement happens and the player is not involved in the game move, the camera needs to look to where the action is happening. The SmoothLookAt class (Figure 5.17) takes a target (NPC's GameObject) where the camera locks on to, this will mostly be the subject of the current game move that is being simulated. Note that the camera stays attached to the player, but its view point is changed to be aimed towards the target. As the target moves, the camera has to smoothly rotate to follow the target.

AgentWalker

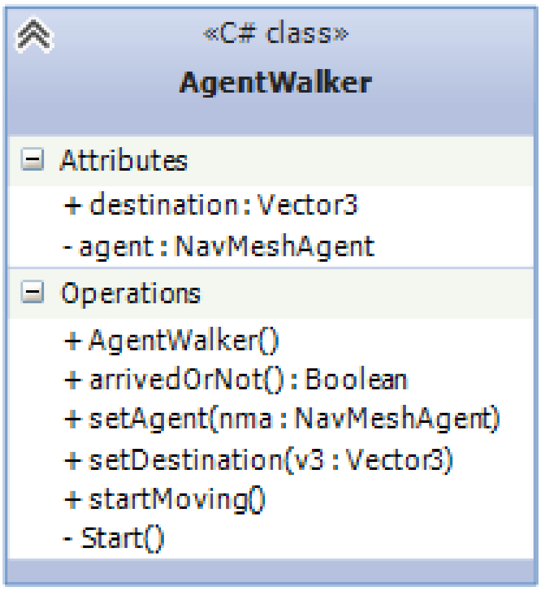


Figure 5.18: AgentWalker Class

The AgentWalker class, given in Figure 5.18, is used to control the NavMeshAgents. It contains methods to set the agent, its destination, and check whether he arrived at the destination already. It is used in StoryFlowAction extensively to make a character start walking towards a set destination, and continuing with the next game move only when he arrived.

DialogManager

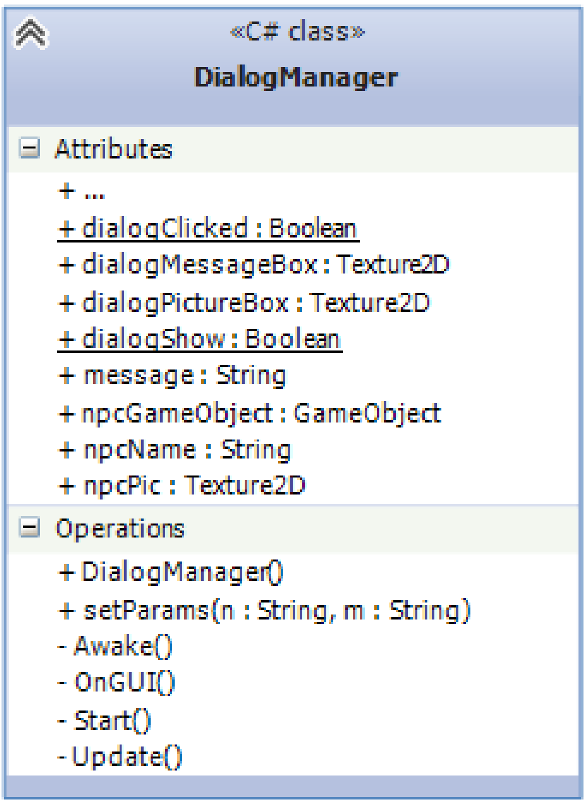


Figure 5.19: DialogManager Class

The DialogManager class (Figure 5.19) is put in a prefab as a Script component. It deals with drawing the conversation overlay and the text inside of it. This prefab is called every time two characters (NPCs or player) talk to each other.

EmailTextMapping

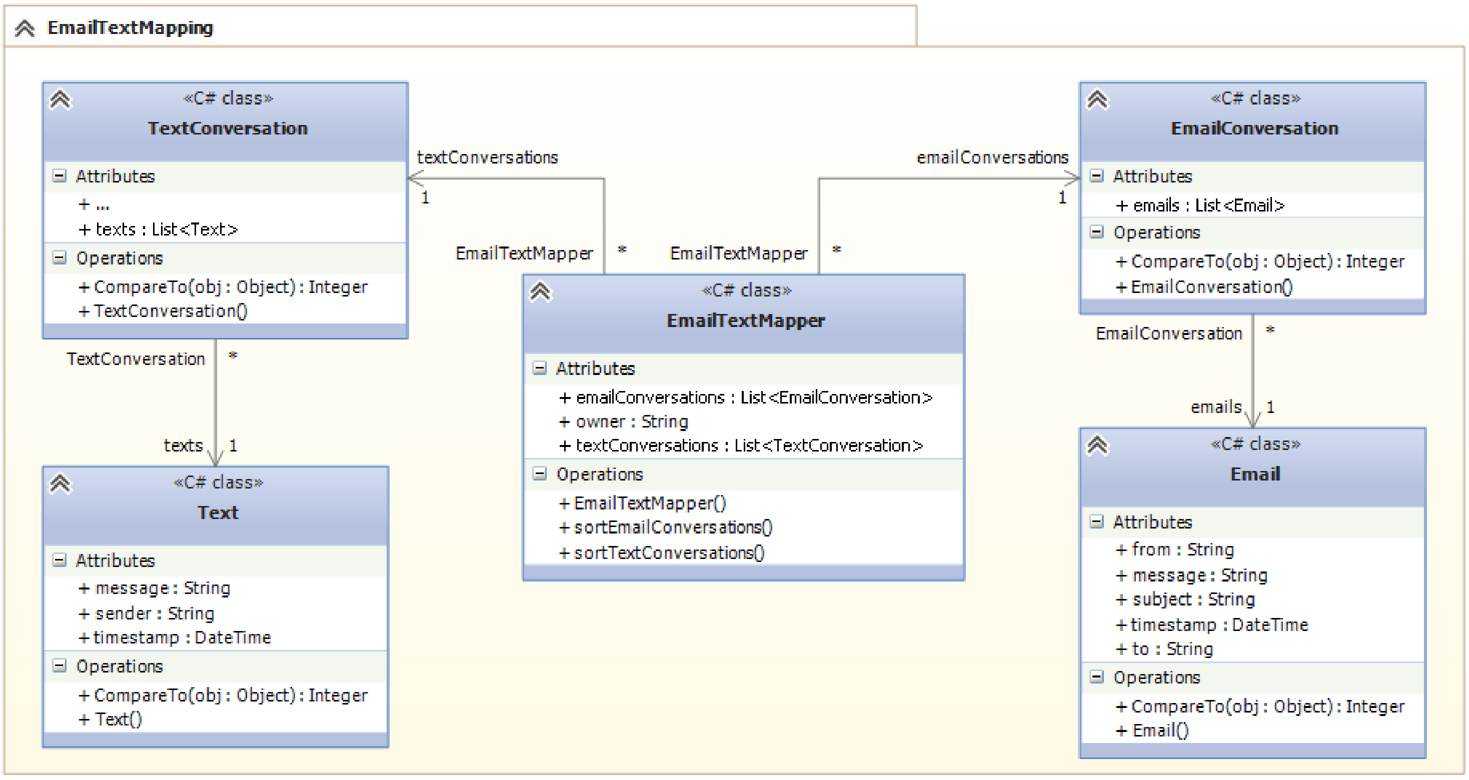


Figure 5.20: EmailTextMapping Namespace

EmailTextMapping is a namespace which contains the following classes to store the email or text message information in (see Figure 5.20):

- EmailTextMapper
- Text
- TextConversation
- Email
- EmailConversation

EmailTextMapper represents the object of the person's phone and inbox. It contains an owner name and two lists with respectively textConversations and emailConversations.

Texting involves two characters, a sender and a receiver. Any character can text with any other character. When two characters text to each other, they are having a conversation. A conversation can have one or more texts. A text message contains a sender, message, and timestamp.

The same story applies for emails and email conversations. Except that an email contains a sender (from), receiver (to), subject, message, and timestamp.

FullTextMessage, FullEmailMessage & ComposeEmailMessage

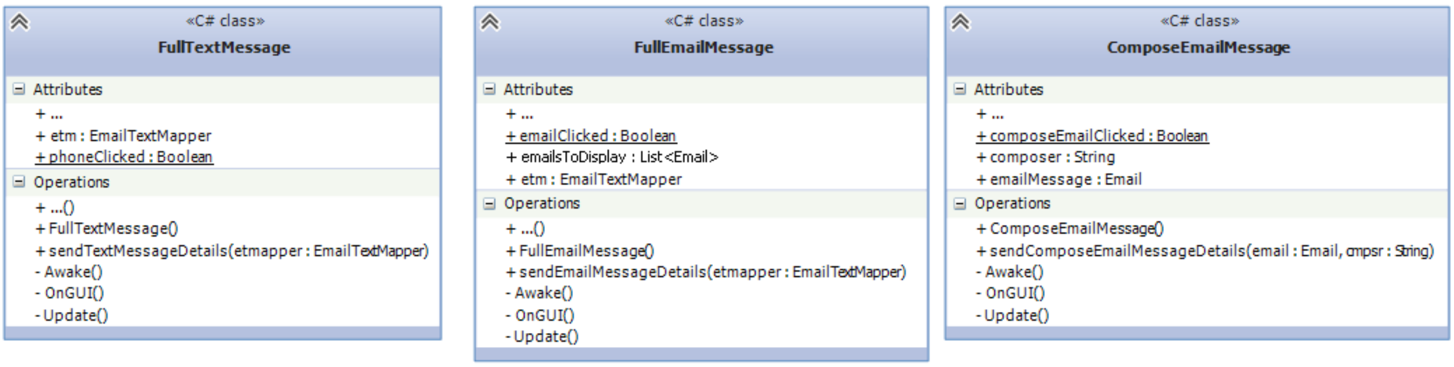


Figure 5.21: FullTextMessage, FullEmailMessage and ComposeEmailMessage Class

FullTextMessage, FullEmailMessage, and ComposeEmailMessage (see Figure 5.21) serve the same purpose, showing their corresponding overlays with the correct information inside. These classes get called from the involved methods inside the StoryFlowAction class.

HalfLifeMapping, HalfLife & HalfLifeMessenger

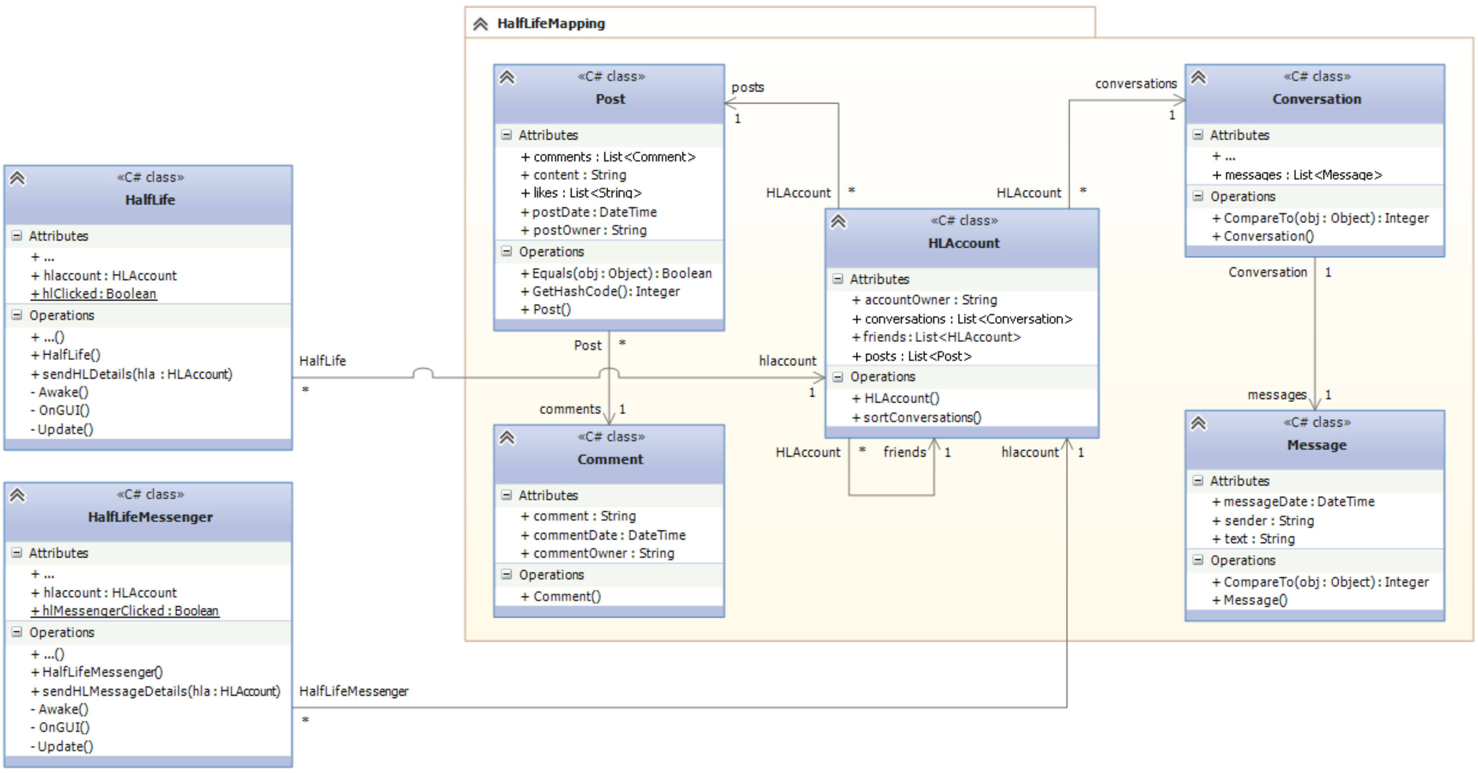


Figure 5.22: HalfLifeMapping Namespace, HalfLife and HalfLifeMessenger Class

Figure 5.22 shows the HalfLifeMapping namespace and the HalfLife and HalfLifeMessenger classes.

HalfLifeMapping is a namespace which contains the following classes to store the Half-Life information in:

- HLAccount
- Conversation
- Message
- Post
- Comment

Half-Life is divided in two parts, regular Half-Life and Half-Life Messenger. Every character has its personal Half-Life (Messenger), so both have the HLAccount in common. This contains the account owner's name, a list of friend names, list of conversations, and a list of posts.

When talking about Half-Life Messenger, it is similar as explained in EmailTextMapping.

Half-Life on the other hand, consists of one or more posts which can have comments on them. A post contains the name of who posted it, the content, a list of names who liked it, a list of comments, and a timestamp. A comment contains the name of who commented, the comment itself, and the timestamp.

The HalfLife and HalfLifeMessenger classes deal with showing the correct overlay and the information inside of it. These classes get called from the involved methods inside the StoryFlowAction class.

TwitterMapping & Twitter

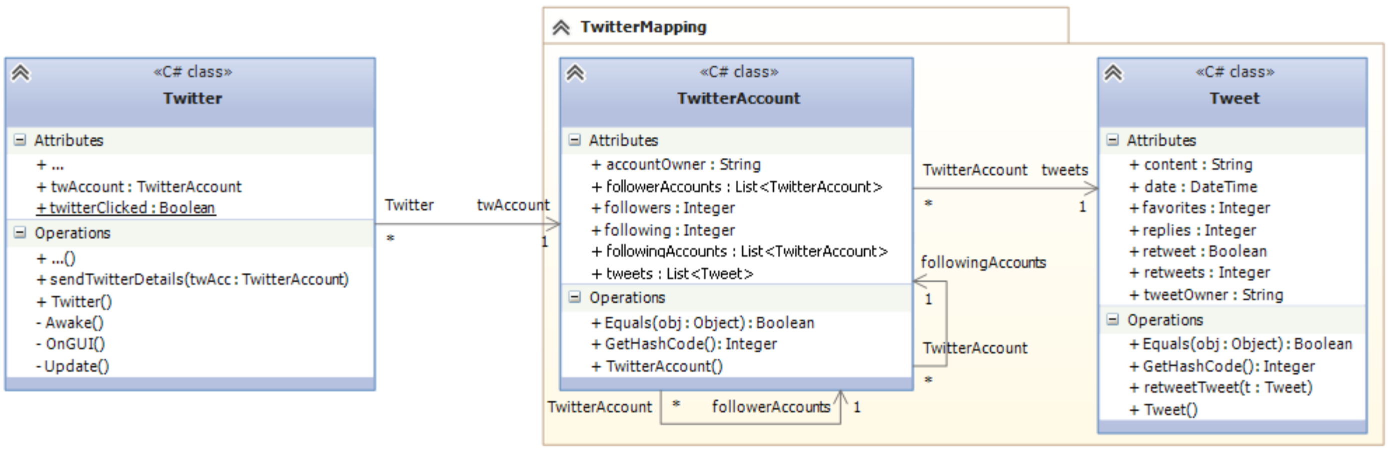


Figure 5.23: Twitter Namespace and Twitter Class

Figure 5.23 shows the TwitterMapping namespace and the Twitter class.

TwitterMapping is a namespace which contains the following classes to store the Twitter information in:

- TwitterAccount
- Tweet

Every character has its personal Twitter account. The information necessary is kept inside TwitterAccount and consists of: the name of the owner, a list of Tweets, the amount of followers and following, a list of follower accounts and following accounts.

A person can only Tweet on his own Twitter page and retweet or favorite someone else's Tweet. The Tweet class consists of: name of the Tweet creator; the Tweet's content; the number of replies, retweets, and favorites; a timestamp; and a retweet boolean.

The Twitter class deals with showing the correct overlay and the information inside of it. These classes get called from the involved methods inside the StoryFlowAction class.

5.3.1 Sequence Diagram

Figure 5.24 gives a general high-level sequence diagram of our simulator. We expect the simulator to run already. We will explain the different steps, corresponding with the numbers on the diagram.

1. Show Main Menu

   Since we expect the simulator to be running, the main menu is shown. The GameObject with EscapeGUI attached to it takes care of displaying this menu on the screen.

2. ATTAC-L Modeling Tool

   The simulator user needs to model a scenario in the ATTAC-L Modeling Tool. After this is done, it is exported in JSON format.

3. Paste ATTAC-L JSON Export

   The simulator user pastes the JSON code of the scenario he modeled in the corresponding input box in the main menu. As has been said, EscapeGUI takes care of this.

3.1 Validate JSON

   Before EscapeGUI starts the simulation, it needs to check if the entered JSON is valid. It does this by sending the input (JSON) to the JSONParser. This checks if the JSON is formatted correctly and if there are no unknown verbs.

3.2 JSON Validated

   If the JSON is valid, the JSONParser will put the corresponding error flags (booleans) to false.

3.3 Add QuestLog Script to GameObject

If all the error flags are set to false, this signals to the EscapeGUI to start the simulation. First it adds the QuestLog script to a GameObject so that the quest log becomes active for the rest of the simulation.

3.4 Add StoryFlowController Script to GameObject

After QuestLog has been added to a GameObject (thus activated), the same is done for StoryFlowController. This GameObject controls the flow of the scenario simulation.

3.4.1 parseScenario()

The *parseScenario()* method consists of a loop that runs for every element e (a flowlabel) in the list of entries. If the ATTAC-L Scenario has multiple entries, this loop will run multiple times.

3.4.1.1 parseEntryLine(e)

In every iteration of the parseScenario() loop, *parseEntryLine(e)* is called, where e is the current entry. This method consists of an IF-clause, where we check if the sequence of bricks corresponding to e (the sequence's flowlabel) is a game move or a game control structure. In the first case (3.4.1.1) we immediately perform the simulation for the action by calling *animateExpr()* (3.4.1.1.1), this method contains a SWITCH-statement that maps all supported verbs to actions in the StoryFlowAction class (3.4.1.1.2).

In the latter case and if the control structure is a choice, we show the choice options on the screen (3.4.1.1.4). The option chosen by the user is returned (3.4.1.1.5) and parseEntryLine(chosen) is called with the chosen option. If the control structure is order independence or concurrency, we call parseEntryLine(GCSPaths) for each of its contents.

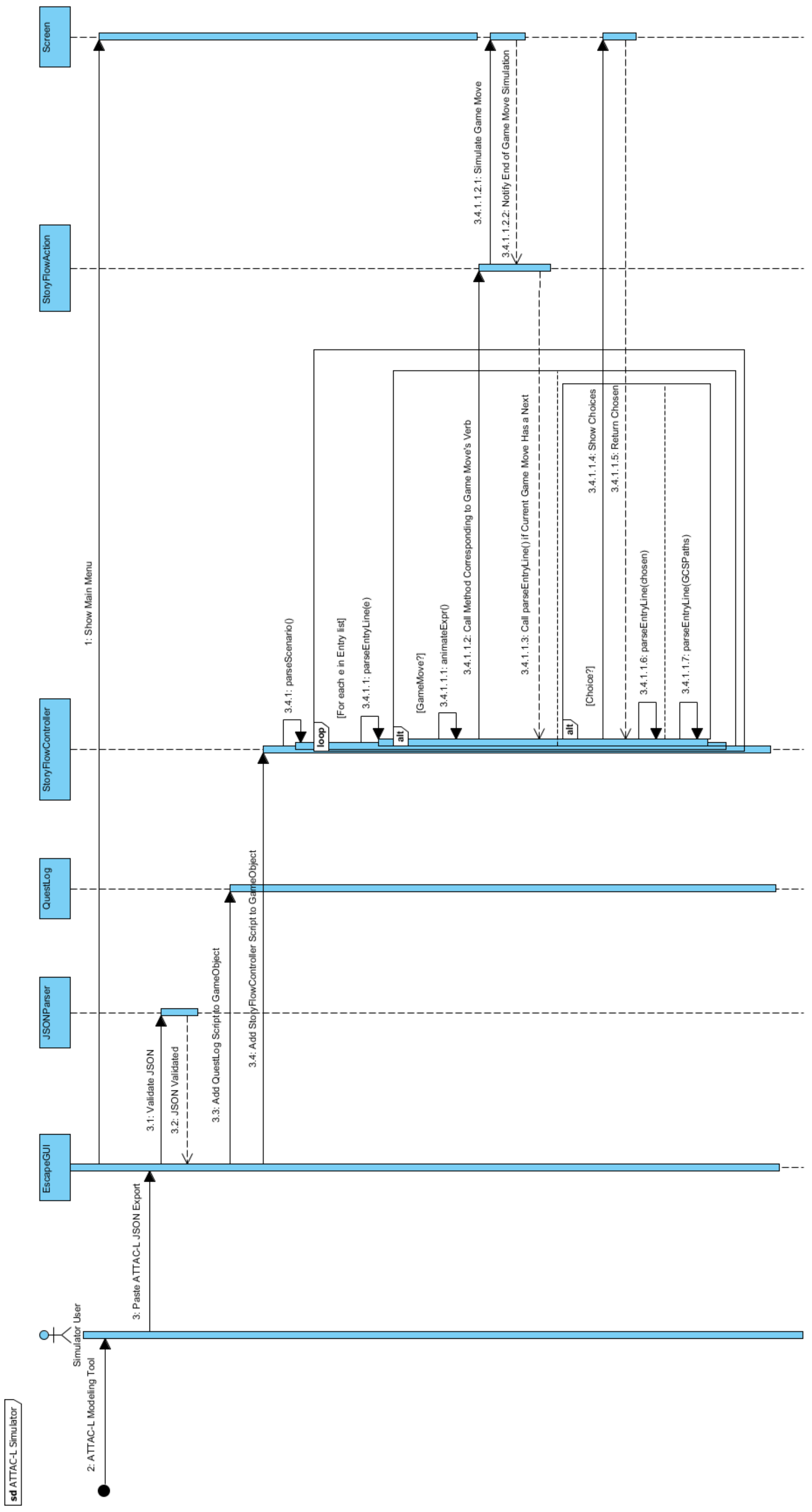

sd ATTAC-L Simulator
Simulator User
EscapeGUI
JSONParser
QuestLog
StoryFlowController
StoryFlowAction
Screen
1: Show Main Menu
2: ATTAC-L Modeling Tool
3: Paste ATTAC-L JSON Export
3.1: Validate JSON
3.2: JSON Validated
3.3: Add QuestLog Script to GameObject
3.4: Add StoryFlowController Script to GameObject
3.4.1: parseScenario()
loop
[For each e in Entry list]
3.4.1.1: parseEntryLine(e)
alt
[GameMove?]
3.4.1.1.1: animateExpr()
3.4.1.1.2: Call Method Corresponding to Game Move's Verb
3.4.1.1.2.1: Simulate Game Move
3.4.1.1.2.2: Notify End of Game Move Simulation
3.4.1.1.3: Call parseEntryLine() if Current Game Move Has a Next
alt
[Choice?]
3.4.1.1.4: Show Choices
3.4.1.1.5: Return Chosen
3.4.1.1.6: parseEntryLine(chosen)
3.4.1.1.7: parseEntryLine(GCSPaths)

Figure 5.24: Sequence Diagram

5.4 Configuration File

Upon executing the simulator for the first time, a configuration file is created in the same folder as the executable file. The configuration file is a plain .txt file, written in .INI standards. The default values of the file can be seen in Listing 5.1.

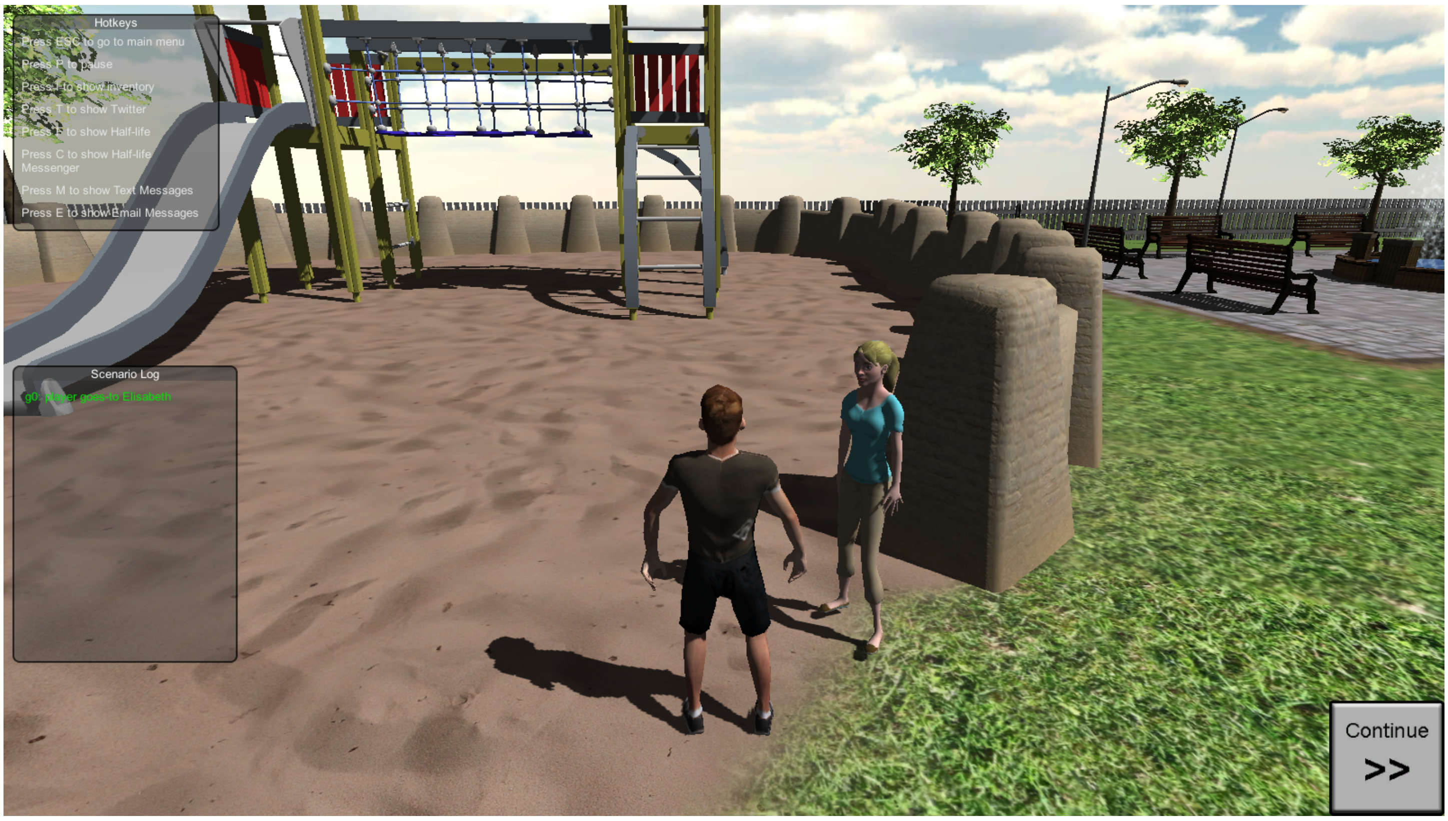
```
[Flow]
manual=false

[Items]
shovel=2
usb=3
cd=3
```

Listing 5.1: Default Configuration File

We can see from Listing 5.1 that there are four variables that can be set. We will start by explaining the three variables in *Items* and afterwards explain the *Flow* variable.

The number we place behind shovel, USB, or CD indicates the amount of that particular item will be present in the scene. If this number is 0 or negative, that item will not be present. Note that if the combined amount of the three items is higher than the maximum amount possible (see Section 4.3.2 for the maximums), we reduce each item amount by one and check if the total amount is below the maximum. If it is still too high, we repeat the process. We will explain this by using an example. We assume the Park scene is selected; its maximum amount of items is 32. In our configuration file we set "shovel=30", "usb=5", "cd=5". The total amount of these three items is 40, which is 8 above the maximum. This means that the simulator will reduce the amount of all three items by one, and it will do this three times. The total amount is now 31 (27 shovels, 2 USBs, 2 CDs). These amounts will be placed in the simulation when the Park scene is chosen.

The "[Flow]" element contains one variable, a boolean, called "manual". When it is set to false, the simulation will ask for player confirmation after each game move. A "continue" button has to be clicked every time to advance to the next game move. Figure 5.25 shows this button at the bottom right of the screen. If this variable is set to true, the simulation will only be interrupted when player input is required (e.g., in the case of a choice or during a conversation).



Press ESC to go to main menu
Press P to pause
Press I to show inventory
Press T to show Twitter
Press F to show Half-life
Press C to show Half-life Messenger
Press M to show Text Messages
Press E to show Email Messages

g0: player goes-to Elisabeth

Figure 5.25: Continue Button

6 Conclusions

In this final chapter we conclude our thesis by summarizing the research and work done. Next, we discuss the limitations of our work and propose future work. We end this thesis by discussing the limitations in language support.

6.1 Summary

The purpose of this thesis was to create a simulator for the semi-automatic play and verification of virtual scenario models, created with the ATTAC-L language. We started this thesis by introducing the context and the problem. In chapter 2, we discussed related work regarding 5 frameworks for story building and made a distinction based on their main purpose: learning programming or prototyping tools. The frameworks discussed are Scratch, Greenfoot, Alice, e-Adventure, and 80Days.

Chapter 3 gave definitions for games, narrative-based games, and serious games. We also went into further detail on a classification for serious games. In this chapter we also explained the main architecture of game engines. After comparing different game engines, we decided to realize the simulator by means of the Unity game engine. We have also described ATTAC-L and its different components, as well as how they are used together to model a scenario. We concluded this chapter by explaining how an exported ATTAC-L scenario in JSON format looks; this export is the input for our simulator.

In chapter 4 we introduced our ATTAC-L simulator. We started by listing the requirements necessary to be able to simulate a scenario using the ATTAC-L simulator. Afterwards we explained the main menu and depicted the simulation. Afterwards great detail was given to the different verbs that can be used in an ATTAC-L scenario that have an action attached to it. We concluded this chapter by explaining the different features (e.g. different persons, the possible items, etc.) the simulator supports.

In chapter 5 we discussed the implementation of the simulator. We started by explaining the key concepts in Unity that are used. We then depicted our system architecture and explained the different components. Afterwards the design of the system was explained in detail by means of the UML class diagram and sequence diagram for the system. We concluded this chapter with explaining the role of the configuration file.

6.2 Limitations and Future Work

In this section we discuss the limitations in our simulator and indicate how they can be overcome in further work.

6.2.1 Limitation of the Simulation Overlays

The overlays that occur when the simulation is running (e.g., Email, Half-Life, etc.) are not interactive and only show a "limited" amount of information. We discuss how they can be improved.

Email Inbox

When looking at the email inbox overlay (shown by pressing the *"E"* hotkey or with the "receives from" verb in a game move), only the latest (or current) received email is shown in detail on the right hand side. The left hand side contains a list of only the latest 5 emails received. In a next version of the simulator the inbox can be made interactive, where the bar on the left side is scrollable to show every email. Clicking one of these mails would show the detailed version on the right.

Text Messages

When clicking the *"M"* hotkey or a game move containing the verb "sends to", the phone overlay with text messages are shown. Currently the phone overlay only shows the last 4 received and/or sent text messages. When the phone overlay is brought up, it only shows the message conversation with the last person who was communicated with (via text). In a future version, the content of the phone can become scrollable so that more messages can be shown. An inbox could also be created, just as with emails, for the phone's text messages. This inbox would contain the different text conversations with other persons. Clicking on one of these conversations would show all the text messages in this conversation.

Half-Life

The Half-Life overlay (*"F"* hotkey or the "posts to" verb) also has no interactivity attached to it. It only shows the two latest posts on the corresponding profile. As well as showing only the two latest comments (if any) on these posts. The same solution can be brought up as with Email and Text, making the main page (right side) of Half-Life a scrollable area so all the posts can be seen. Another option could be to create a "Show Older" button, which would show the next two posts. For the comments, a "Show All" button can provide the functionality to show all the comments under each other.

Half-Life Messenger

The Half-Life Messenger overlay (*"C"* hotkey or the "chats to" verb) only shows as many messages that can fit on the right side (7 at a time). This can be solved by using a scrollable area or a "Show Older" button, as explained above. The middle bar, contains all the conversations the Half-Life account owner has with different characters. Currently this supports enough space to show the conversation with every character available. However, if more characters are added to the simulator, this available space will no longer be sufficient. Making this into a scrollable area is also advised in this case.

Twitter

The Twitter overlay (*"T"* hotkey or the "tweets" verb) can only show the three latest Tweets (or retweets) at the same time. The Tweet area can become scrollable again to be able to see all the Tweets. Another solution could be the "Show Older" button as explained above.

6.3 Limitations in Language Support

6.3.1 Verbs

Appendix C shows a list of all the verbs that can be used in our simulator. New functionality (verbs) can always be added to the simulator. This can be done by adding the new verb in the known verbs list in the JSONParser class. Then, the verb must be added in the switch statement in the StoryFlowController, where this launches the associated method (also to be implemented) in StoryFlowAction.

6.3.2 Characters

There are 9 characters in our simulator (8 NPCs and the player's character). In order to add a new character to the simulator, the first thing that is required is a 3D character model with animations (or compatible with the other animations). A new GameObject has to be created for this NPC, with an Animation, NavMeshAgent, and Mesh Renderer component attached to it. The last thing we have to do, is add the name of the character to the list of known NPCs in the JSONParser class.

6.3.3 Animations

The only animations that the player's character and NPC have in common are walking and idle. Only the player's character has a pick-up animation attached. This animation is not compatible with some of the other NPC models. A new pick-up animation can either be created or an already existing one (which is compatible with all character models) can be used. Every character (NPC and player's character) has an Animation component attached to its GameObject; the different animations get added in this component and can be launched using code.

6.3.4 Items

Currently we only have three types of items in our scene that can be picked up: a shovel, an USB, and a CD. In order to create a new pick-up item in our simulator, we need to have a 3D model of the item. We should then create a GameObject with the PickingItemUp script, Mesh Renderer, and Box Collider component attached to it. Then it needs to be transformed into a prefab. A new item has to be instantiated in the ItemDatabase class by creating an instance of the Item class. We also have to add the name of the item to the known items list in the JSONParser class. There also has to be a sprite in the Resources folder of Unity available, with the same name as the item. This sprite is the icon shown if the item is in the inventory.

Bibliography

Adobe Flash [Computer software]. (2015). Retrieved from https://get.adobe.com/nl/flashplayer/ on April 13, 2015.

Alice [Computer software]. (2015). Retrieved from http://www.alice.org/ on March 28, 2015.

ARMA 2 [Computer software]. (2009). Retrieved from http://www.arma2.com/ on March 31, 2015.

BlueJ [Computer software]. (2015). Retrieved from http://bluej.org/ on March 27, 2015.

Caltagirone, S., Keys, M., Schlief, B., & Willshire, M. J. (2002). Architecture for a massively multiplayer online role playing game engine. *Journal of Computing Sciences in Colleges*, *18*(2), 105–116.

Corman, S. R. (2013). The difference between story and narrative. Retrieved from http://csc.asu.edu/2013/03/21/the-difference-between-story-and-narrative on April 3, 2015.

Corti, K. (2006). Games-based learning; a serious business application. *Informe de PixelLearning*, *34*(6), 1–20.

Counter Strike: Global Offensive [Computer software]. (2015). Retrieved from http://blog.counter-strike.net/ on March 30, 2015.

CryEngine [Computer software]. (2015). Retrieved from http://www.cryengine.com/ on March 22, 2015.

Day-Z [Computer software]. (2015). Retrieved from https://dayzmod.com/ on March 31, 2015.

Dickey, M. D. (2006). Game design narrative for learning: Appropriating adventure game design narrative devices and techniques for the design of interactive learning environments. *Educational Technology Research and Development*, *54*(3), 245–263.

Djaouti, D., Alvarez, J., & Jessel, J.-P. (2011). Classifying serious games: The g/p/s model. *Handbook of research on improving learning and motivation through educational games: Multidisciplinary approaches*, 118–136.

e-Adventure [Computer software]. (2012). Retrieved from `http://e-adventure.e-ucm.es/` on March 28, 2015.

Eberly, D. H. (2010). *Game physics.* Taylor and Francis.

Fifa [Computer software]. (2014). Retrieved from `https://www.easports.com/nl/fifa` on March 30, 2015.

Friendly ATTAC. (2012). *Adaptive Technological Tools Against Cyberbullying.* Retrieved from `http://www.friendlyattac.be/en/` on February 21, 2015.

Furtado, A. W., & Santos, A. L. (2006). Using domain-specific modeling towards computer games development industrialization. In *The 6th oopsla workshop on domain-specific modeling (dsm06).*

Furtado, A. W. B., & de Medeiros Santos, A. (2006). Sharpludus: improving game development experience through software factories and domain-specific languages. *Universidade Federal de Pernambuco (UFPE) Mestrado em Ciência da Computação centro de Informática (CIN).*

Gobel, S., Salvatore, L., & Konrad, R. (2008). Storytec: A digital storytelling platform for the authoring and experiencing of interactive and non-linear stories. In *Automated solutions for cross media content and multi-channel distribution, 2008. axmedis'08. international conference on* (pp. 103–110).

Greenfoot [Computer software]. (2015). Retrieved from `http://www.greenfoot.org/` on March 27, 2015.

Gregory, J. (2009a). Game engine architecture. In (Second ed., chap. 1). CRC Press.

Gregory, J. (2009b). Game engine architecture. In (Second ed., chap. 10;11;12;13). CRC Press.

Juul, J. (2010). The game, the player, the world: Looking for a heart of gameness. *PLURAIS-Revista Multidisciplinar da UNEB*, *1*(2).

Kickmeier-Rust, M., Mattheiss, E., & Albert, D. (2011). Experiences with an approach to an unobtrusive assessment of motivational states in immersive, narrative learning environments. In *Proceedings of the 7th european conference on management leadership and governance: Ecgbl 2011* (p. 315).

Kickmeier-Rust, M. D., Göbel, S., & Albert, D. (2008). 80days: Melding adaptive educational technology and adaptive and interactive storytelling in digital educational games. In *Proceedings of the first international workshop on story-telling and educational games (stegŠ08).*

Klabbers, J. H. (2003). The gaming landscape: a taxonomy for classifying games and simulations. In *Digra conf.*

Kolo, C., & Baur, T. (2004). Living a virtual life: Social dynamics of online gaming. *Game studies*, *4*(1), 1–31.

League of Legends [Computer software]. (2015). Retrieved from `http://euw.leagueoflegends.com/` on March 30, 2015.

Lindley, C. A. (2005). Story and narrative structures in computer games. *Bushoff, Brunhild. ed.*

Luoma, J., Kelly, S., & Tolvanen, J.-P. (2004). Defining domain-specific modeling languages: Collected experiences. In *4 th workshop on domain-specific modeling.*

Marchiori, E. J., Del Blanco, Á., Torrente, J., Martinez-Ortiz, I., & Fernández-Manjón, B. (2011). A visual language for the creation of narrative educational games. *Journal of Visual Languages & Computing*, *22*(6), 443–452.

Mass Effect [Computer software]. (2012). Retrieved from `http://masseffect.bioware.com/` on April 4, 2015.

Michael, D. R., & Chen, S. L. (2005). *Serious games: Games that educate, train, and inform.* Muska & Lipman/Premier-Trade.

MIT Media Lab. (2015). *Scratch - Imagine, Program, Share.* Retrieved from `https://scratch.mit.edu/` on March 27, 2015.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, *24*(3), 45–77.

Petridis, P., Dunwell, I., de Freitas, S., & Panzoli, D. (2010). An engine selection framework for high fidelity serious games. In *The 2nd international conference on games and virtual worlds for-serious-applications (vsgames 2010), braga, portugal.*

Pulse!! [Computer software]. (2007). Retrieved from `http://serious.gameclassification.com/EN/games/1017-Pulse/index.html` on March 30, 2015.

Sherlock Holmes: Crimes & Punishments [Computer software]. (2014). Retrieved from `http://www.sherlockholmes-thegame.com/` on March 30, 2015.

Shute, V. J., Ventura, M., Bauer, M., & Zapata-Rivera, D. (2009). Melding the power of serious games and embedded assessment to monitor and foster learning. In *Serious games: Mechanisms and effects* (Vol. 2, pp. 295–321). Routledge/LEA Philadelphia, PA.

Starcraft II [Computer software]. (2014). Retrieved from `http://us.battle.net/sc2/en/` on March 30, 2015.

Surgeon Simulator [Computer software]. (2013). Retrieved from `http://www.surgeonsim.com/` on March 31, 2015.

Susi, T., Johannesson, M., & Backlund, P. (2007). Serious games: An overview.

The Sims [Computer software]. (2014). Retrieved from `http://www.thesims.com/` on March 30, 2015.

Torrente, J., Del Blanco, Á., Marchiori, E. J., Moreno-Ger, P., & Fernández-Manjón, B. (2010). < e-adventure>: Introducing educational games in the learning process. In *Education engineering (educon), 2010 ieee* (pp. 1121–1126).

Unreal Engine [Computer software]. (2015). Retrieved from `https://www.unrealengine.com/` on March 22, 2015.

Van Broeckhoven, F., & De Troyer, O. (2013). Attac-l: A modeling language for educational virtual scenarios in the context of preventing cyber bullying. In *Serious games and applications for health (segah), 2013 ieee 2nd international conference on* (pp. 1–8).

World of Warcraft [Computer software]. (2015). Retrieved from `http://us.battle.net/wow/en/` on March 30, 2015.

A

Appendix A

Listing A.1 is the JSON representation of the ATTAC-L story shown in Figure 3.23 in section 3.3.4.

```
{
    "_entries": [
        "g0"
    ],
    "_flow": {
        "g16": {
            "_expr": [
                {
                    "word_": "Elisabeth",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Thank you very much!"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "player"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Elisabeth",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Thank you very much!"
                    },
                    "_indirect": "player"
                }
            }
        },
        "g15": {
            "_next": "g16",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Here you go, Carl found your USB!"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Here you go, Carl found your USB!"
                    },
                    "_indirect": "Elisabeth"
                }
            }
        },
        "g14": {
            "_next": "g15",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "gives",
                            "_next": [
                                {
                                    "word_": "Usb"
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "gives to",
                    "_direct": "Usb",
                    "_indirect": "Elisabeth"
                }
            }
        },
        "g13": {
            "_next": "g14",
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "gives",
                            "_next": [
                                {
                                    "word_": "Usb"
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "player"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "gives to",
                    "_direct": "Usb",
                    "_indirect": "player"
                }
            }
        },
        "g12": {
            "_next": "g13",
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "I do not really know her that well.
                                    Here, you can give it to her."
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "player"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "I do not really know her that well. Here, you
                            can give it to her."
                    },
                    "_indirect": "player"
                }
            }
        },
        "g11": {
            "_next": "g12",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Give it back to her, it is hers!"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Carl"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Give it back to her, it is hers!"
                    },
                    "_indirect": "Carl"
                }
            }
        },
        "g20": {
            "_expr": [
                {
                    "word_": "Vincent",
                    "_next": [
                        {
                            "word_": "befriends",
                            "_next": [
                                {
                                    "word_": "Carl"
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "Half-Life"
                                }
                            ]
                        }
```

```json
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Vincent",
                "_predicate": {
                    "_verb": "befriends on",
                    "_direct": "Carl",
                    "_indirect": "Half-Life"
                }
            }
        },
        "g21": {
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "befriends",
                            "_next": [
                                {
                                    "word_": "Mary"
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "Half-Life"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "befriends on",
                    "_direct": "Mary",
                    "_indirect": "Half-Life"
                }
            }
        },
        "g23": {
            "_expr": [
                {
                    "word_": "Vincent",
                    "_next": [
                        {
                            "word_": "comments",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Send it to me please!"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "the hl-page of",
                                    "_next": [
                                        {
                                            "word_": "Carl"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Vincent",
                "_predicate": {
                    "_verb": "comments on",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Send it to me please!"
                        }
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "hl-page",
                        "_of": "Carl"
                    }
                }
            }
        },
        "g24": {
            "_expr": [
                {
                    "word_": "Mary",
                    "_next": [
                        {
                            "word_": "comments",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Can you send it to me please?"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "on",
                            "_next": [
                                {
                                    "word_": "the hl-page of",
                                    "_next": [
                                        {
                                            "word_": "Carl"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Mary",
                "_predicate": {
                    "_verb": "comments on",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Can you send it to me please?"
                        }
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "hl-page",
                        "_of": "Carl"
                    }
                }
            }
        },
        "g22": {
            "_type": "oin",
            "_paths": [
                "g23",
                "g24"
            ]
        },
        "g19": {
            "_next": "g22",
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "posts",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Me and player found Elisabeth’s homework! Who else wants it?"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "the hl-page of",
                                    "_next": [
                                        {
                                            "word_": "Carl"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "posts to",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Me and player found Elisabeth’s homework!
                                Who else wants it?"
                        }
                    },
                    "_indirect": {
                        "_det": "the",
                        "_noun": "hl-page",
                        "_of": "Carl"
                    }
                }
            }
        },
        "g18": {
            "_next": "g19",
            "_type": "oin",
            "_paths": [
                "g20",
                "g21"
            ]
        },
        "g17": {
            "_next": "g18",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Don’t tell her but send me a copy too,
                                     else I will tell on you!"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Carl"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Don’t tell her but send me a copy too, else I
                            will tell on you!"
                    },
                    "_indirect": "Carl"
                }
            }
        },
        "g10": {
            "_type": "cho",
            "_paths": [
                "g11",
                "g17"
            ]
        },
        "g9": {
            "_next": "g10",
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Yes I did. Should I give it back to her or pretend I never found it and copy her homework?"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "player"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Yes I did. Should I give it back to her or pretend I never found it and copy her homework?"
                    },
                    "_indirect": "player"
                }
            }
        },
        "g8": {
            "_next": "g9",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Hey Carl, Elisabeth told me she lost her USB stick containing her homework here today. Did you find it by accident?"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Carl"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Hey Carl, Elisabeth told me she lost her USB stick containing her homework here today. Did you find it by accident?"
                    },
                    "_indirect": "Carl"
                }
            }
        },
        "g7": {
            "_next": "g8",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Ok, I will go and ask him!"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Ok, I will go and ask him!"
                    },
                    "_indirect": "Elisabeth"
                }
            }
        },
        "g25": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "I need to go now I am sorry. Go ask it yourself."
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "I need to go now I am sorry. Go ask it yourself."
                    },
                    "_indirect": "Elisabeth"
                }
            }
        },
        "g6": {
            "_type": "cho",
            "_paths": [
                "g7",
                "g25"
            ]
        },
        "g5": {
            "_next": "g6",
            "_expr": [
                {
                    "word_": "Elisabeth",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "I just saw Carl picking something up! Could you go and ask if he found it?"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "player"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Elisabeth",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "I just saw Carl picking something up! Could
```

```
                            you go and ask if he found it?"
                    },
                    "_indirect": "player"
                }
            }
        },
        "g4": {
            "_next": "g5",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "I cannot find it. Are you sure you lost it here?"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "I cannot find it. Are you sure you lost it here?"
                    },
                    "_indirect": "Elisabeth"
                }
            }
        },
        "g3": {
            "_next": "g4",
            "_expr": [
                {
                    "word_": "Carl",
                    "_next": [
                        {
                            "word_": "picks-up",
                            "_next": [
                                {
                                    "word_": "Usb"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Carl",
                "_predicate": {
                    "_verb": "picks-up",
                    "_direct": "Usb"
                }
            }
        },
        "g2": {
            "_next": "g3",
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "_next": [
                                {
                                    "word_": "a message",
                                    "attr_": {
                                        "message": "Sure, I can help you find it!"
                                    }
                                }
                            ]
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "_det": "a",
                        "_noun": "message",
                        "_attr": {
                            "message": "Sure, I can help you find it!"
                        }
                    },
                    "_indirect": "Elisabeth"
                }
            }
        },
        "g26": {
            "_expr": [
                {
                    "word_": "player",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "This is not my problem. Leave me alone!"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "Elisabeth"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "player",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "This is not my problem. Leave me alone!"
                    },
                    "_indirect": "Elisabeth"
                }
            }
        },
        "g1": {
            "_type": "cho",
            "_paths": [
                "g2",
                "g26"
            ]
        },
        "g0": {
            "_next": "g1",
            "_expr": [
                {
                    "word_": "Elisabeth",
                    "_next": [
                        {
                            "word_": "says",
                            "attr_": {
                                "message": "Hey player, I lost my USB stick containing my homework. Can you help me find it?"
                            }
                        },
                        {
                            "word_": "to",
                            "_next": [
                                {
                                    "word_": "player"
                                }
                            ]
                        }
                    ]
                }
            ],
            "_interpr": {
                "_subject": "Elisabeth",
                "_predicate": {
                    "_verb": "says to",
                    "_direct": {
                        "message": "Hey player, I lost my USB stick containing my
                            homework. Can you help me find it?"
                    },
                    "_indirect": "player"
                }
            }
        }
    }
}
```

Listing A.1: JSON Representation of Figure 3.23

B
Appendix B

Locations

Park

- Playground
- Fountain
- Football-field
- Sitting-area

Office

- Cubicle
- Small-cubicle
- Meeting-room

House

- Lounge
- Entrance
- Dining-room
- Fire-place
- Small-Bedroom
- Double-bedroom
- Small-bathroom
- Living-room
- Kitchen
- Master-bedroom
- Master-bathroom

Appendix C

Below we show all the verbs (actions) that are supported by our simulator.

Simulator Supported Verbs

- "walks-to"
- "goes-to"
- "says to"
- "chats to"
- "sends to"
- "texts to"
- "emails to"
- "mails to"
- "gives to"
- "picks-up"
- "grabs"
- "receives from"
- "befriends on"
- "friends on"
- "posts on"
- "posts to"
- "replies on"
- "replies-with to"
- "comments on"
- "likes"
- "likes on"
- "tweet"
- "tweets"
- "retweets"
- "follows on"

D Appendix D

Figure D.1 shows the complete UML class diagram of our simulator. Note that some classes contain too many variables or methods to depict on this diagram, this is why we opt to only display the important ones. Three dots indicate that there are more methods or variables present than depicted.

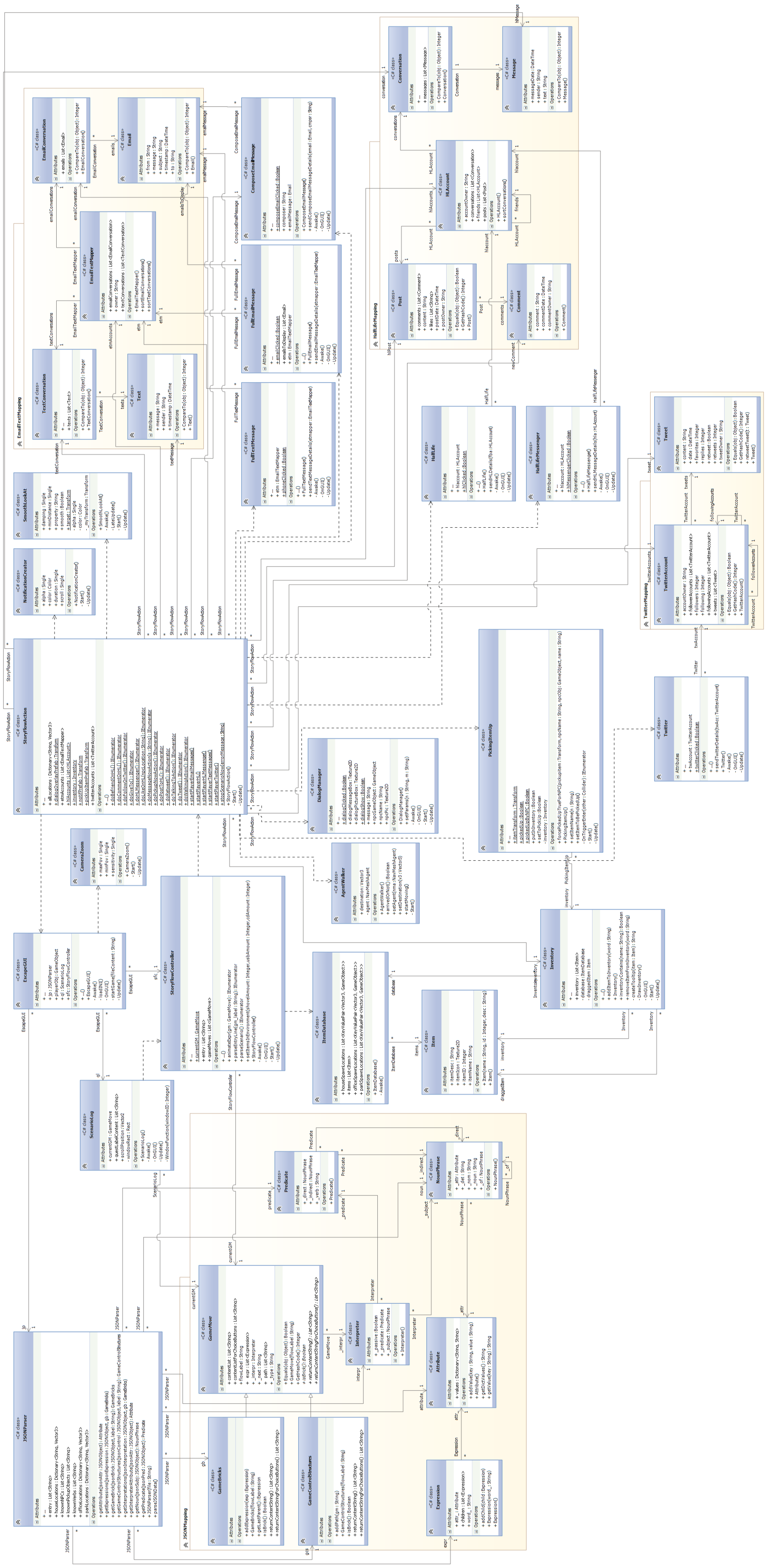


Figure D.1: UML Class Diagram