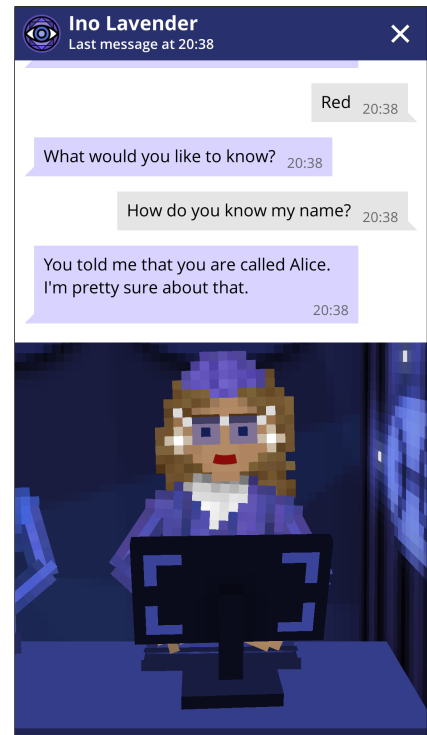
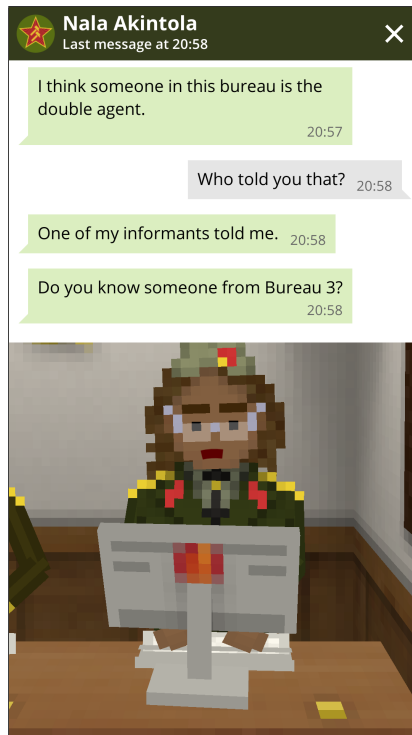
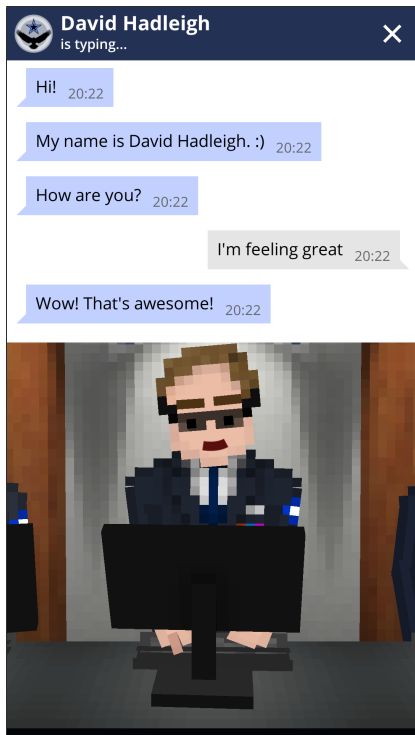


Emergent Narrative through Reasoning Agents in Location-Based Multiplayer Games



Henry Raymond
08-928-061

Master's Thesis
14 December 2017

Supervisors:
Dr. Fabio Zünd
Dr. Stéphane Magnenat
Prof. Dr. Robert Sumner

Prof. Dr. Renato Pajarola

Game Technology Center
Department of Computer Science
ETH Zurich

Visualization and Multimedia Lab
Department of Informatics
University of Zurich

Abstract

Two essential elements of a modern video game are its story and how the story is told. But the number of possible sequences of events that the game's designers must create increases with the player's ability to affect the story. Designers therefore tend to restrict the player's freedom in unrealistic ways, thus making the player's experience less enjoyable.

In this thesis, we present an approach that addresses this problem by treating each non-player character in a game as a reasoning agent. Using a planning approach adopted from the field of artificial intelligence, we let the agents determine their own actions and on this basis let emergent narrative create the story.

The implementation of this thesis comprises two parts: A library called NPCEngine that controls agents in the game world and a mobile game intended as a technology demonstrator.

Acknowledgments

First of all, I would like to express my sincere gratitude to my two supervisors, Dr. Fabio Zünd and Dr. Stéphane Magnenat of the Game Technology Center of ETH Zurich for the support, expertise and time they dedicated to me.

I would like to thank Professor Robert Sumner of ETH Zurich for giving me the opportunity to write this thesis at the Game Technology Center and Professor Renato Pajarola from the Visualization and Multimedia Lab of the University of Zurich for supervising this thesis.

I would also like to thank Julian Scheidegger, Julia Chatain, and Benjamin Bürgisser for their technical help during the development of the mobile game and my father for proofreading this thesis.

Last but not least, I would like to thank the whole team and my fellow students at the Game Technology Center, my mother, and many close friends whose great ideas, creative suggestions and ceaseless interest supported me and my project.

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Procedural Storytelling	2
1.2 Outline	3
2 Related Work	4
2.1 Procedural Storytelling	4
2.2 Emergent Narrative	5
3 NPCengine	6
3.1 Design	6
3.1.1 Problem Definition	6
3.1.2 Domain	7
3.1.3 Storyworld	8
3.1.4 NPCs	8
3.1.5 Storyworld and Knowledge Representation	9
3.1.6 Actions	11
3.1.7 Planning	13
3.1.8 Monte Carlo Tree Search	16
3.1.9 Library Architecture	19
3.2 Implementation	21
3.2.1 Extended Architecture	21
3.2.2 Planning State	23
3.2.3 Domain Definition	26
3.2.4 Storyworld Initialization	35
3.2.5 Token Generator	36
3.3 Results	36
3.3.1 Expressiveness	38
3.3.2 Action Selection	38
3.3.3 Multiplayer Application	38
3.3.4 Technical Practicability	39
3.3.5 Performance	39
3.3.6 Discussion	40
4 Mobile Game	41
4.1 Design	41
4.2 Results	41
5 Conclusion	49
5.1 Future Work	50
5.2 Outlook	50

1 Introduction

Since the dawn of electronic entertainment, the creators of video games have tried to distinguish their products through one essential part of their design: the story. Admittedly, many early games relied heavily on the player's imagination to merge an implied story with abstract gameplay. In modern video games, however, the narrative is often a central part of the player's experience. In many cases, the quality of a game's story will determine its success or failure. There is probably no video game that does not at least hint at a story.

But why does a video game need a story? On closer inspection, most video games consist primarily of relatively abstract game mechanics – mechanics that would feel tedious or even pointless without an accompanying narrative. A game's story gives the player an emotional incentive to explore these mechanics and motivation to use them. In games where the player controls a single character, the story also enhances the player's immersion by tapping into the player's willingness to pretend and relate to the in-game character.

At this point, we must differentiate between *story* and *storytelling*. A story refers to a sequence of more-or-less connected events. Storytelling¹, on the other hand, is a method to convey a story in a way that an audience enjoys. One could argue that storytelling is often more important than the story itself.

The concept of *suspension of disbelief* is essential to storytelling. It denotes our ability to emotionally relate to characters and stories that are obviously fictitious. While telling a story, the storyteller's goal is to maintain suspension of disbelief. If an audience ceases to suspend disbelief, it will usually also lose interest in the story.

Another essential part of a story is its setting – the *storyworld*. The storyworld comprises the locations and surroundings in which the story takes place. But more importantly, the storyworld contains the characters whose actions and interactions tell the story. The storyworld thus provides the storyteller with a set of building blocks that allow him or her to formulate a story. A more complex storyworld allows for richer stories.

The importance of narrative in game design cannot be understated: The story and how it is told are defining features of any game. For instance, the mechanics of many platformer games are arguably very similar – but their setting, narrative, and atmosphere greatly differ. Many game genres such as puzzle, strategy or shooter games give the player a story whose only purpose is to justify their abstract game mechanics. In some games, such as Naughty Dog's critically acclaimed *The Last of Us*, the story is an essential part of the player's experience. However, closer inspection shows that the movie-like cut-scenes that tell the story have little to do with the gameplay. Instead, the narrative emotionally captivates the player for an otherwise fairly generic game.

In contrast, some video games implement a story that fundamentally determines the game mechanics. *Role-playing games* (RPGs) are the most obvious example. In an RPG, the player steps into the shoes of one of the story's characters and experiences the story from the character's perspective. In contrast to the linear narration typical in other games, in an RPG the player experiences the story by interacting with other characters and the surroundings. This genre includes many of the most successful games ever

¹Throughout this thesis – as in most related scientific literature – we will use the terms “storytelling” and “narrative” interchangeably. However, we will favor one term over the other when one is more common in a specific context.

published, such as Rockstar Games' *Grand Theft Auto* franchise, the Square Enix's *Final Fantasy* series, Blizzard's *World of Warcraft*, or Bethesda Game Studios' *Skryim*.

This non-linear way of telling the story entails a new type of narrative design: Within the game, the main sources of narrative are so-called *non-player characters* (NPCs). The game controls these characters and the player can interact with them, usually through pre-scripted dialog sequences. Also essential to most RPGs are so-called *quests*, small missions that NPCs assign to the player. These quests have two goals: First, they direct the player to the next part of the story. And second, these quests convey parts of the story themselves, as they often relate to an overarching storyline.

In terms of narrative, RPGs can be divided into two categories: RPGs with linear storytelling and so-called *open-world* RPGs. In linear storytelling, the sequence of events is the same or very similar in each playthrough. This approach allows the game's designers to invest all their effort in a specific storyline and thereby author a rich story with complex characters. However, the player will have little influence on how events unfold. In other words, his or her *player agency* is very limited.

In contrast, the degree of player agency is very high in open-world RPGs. Here the player is free – at least to a certain degree – to explore the game's storyworld. But this freedom can make telling the story tricky, as designers and story authors must foresee many possible sequences of events. Also, the game must somehow keep the player from doing anything that could stop the story, such as killing an essential character. Deciding between linear storytelling and an open-world approach means dealing with the tradeoff between story richness and player agency. Trying to increase both story complexity and player agency causes the scale of the required content to grow exponentially.

In any case, no matter how much effort its creators put into content complexity, a traditional RPG must limit the player's agency to some degree. A common approach is to block off parts of the storyworld through physical barriers, for instance by having implausibly athletic guards defend an entrance. A less obvious strategy relies on the deterministic behavior of NPCs. Although the player is free to choose when and how he or she interacts with an NPC, that interaction can only yield a limited number of outcomes. In many cases any interaction will lead to just one course of events. Another common problem resulting from NPCs' deterministic behavior arises when their reactions are inconsistent with the storyworld. Typical examples are omniscience and obliviousness. An omniscient NPC may congratulate the player for an action that only the player witnessed. Or an oblivious peasant NPC may attack the player who is famous for slaying all dragons in the land. Faced with NPCs who behave in these ways, some players may cease to suspend disbelief.

Each generation of RPG designers have sought to give the player both a more elaborate story and more agency. This has inevitably led to a massive increase in content – and higher development costs. For instance, according to an analyst, developing *Grand Theft Auto V* (2013) cost over USD 137 million [Sin13]. The focus of much interest is therefore systems that allow for more player agency and a richer story while avoiding exponential growth of content complexity.

1.1 Procedural Storytelling

The goal of *procedural storytelling* (sometimes also called *interactive narrative*) is to find ways to mitigate the tradeoff between story complexity and player agency. Simply put, procedural storytelling is a catch-all term for any system that adapts a story to the player's actions.

One approach in procedural storytelling is to build a system that controls the entire storyworld so as to guarantee that the story takes place as the author intended. Specifically, it detects and tries to resolve conflicts between the current state of the storyworld and the *authorial intent*. For instance, if the character

who is supposed to save the princess unexpectedly dies, the system might have another character take his place. Such systems thus often act as a *deus ex machina*.

Systems that follow this approach to procedural storytelling are often called *drama managers* or sometimes *experience managers*. Such systems have been the subject of much research since the early 1990s. Noteworthy is the work of Mark Riedl and Michael Young [YR03, RSDA08, RTB11], and that of Mubbasir Kapadia [KFZ⁺15, KFS⁺16]. These approaches formulate the task of procedural storytelling as a planning problem. However, the size of state-of-the-art drama manager systems is limited, as the planning complexity rises polynomially with the number of variables in the storyworld. They are therefore limited to offline use.

A different approach to procedural storytelling draws its ideas from robotics. Instead of observing the storyworld as a whole and trying to deal with one complex planning problem, this approach treats each character in the storyworld as an agent. Each agent gets a goal and a set of actions consistent with the intended story. Like a drama manager, the agent achieves its goal by solving a planning problem – but here, each agent plans individually on the basis of the subset of the storyworld observable by the agent. This approach dramatically reduces the computational complexity of the planning task. As events unfold, however, the story may diverge from the authorial intent, as the system has no global control over the storyworld.

Tarn and Zach Adams’ game *Dwarf Fortress* uses this agent-based approach to procedural storytelling to good effect. This base-building game – infamous for its complexity – plays in a procedurally generated world populated by simulated agents. Given that the player’s ability to influence the course of events within the storyworld is practically unrestricted, he or she experiences a unique story. Recent research in the field of agent-based procedural storytelling by James Ryan [RSMWF15, RMWF16] has introduced a new aspect to agent simulation: mimicking a character’s reasoning based on incomplete or false information. This approach involves simulating an NPC’s knowledge about the storyworld.

1.2 Outline

In this thesis, we will present a procedural storytelling system that strongly mirrors Ryan work. In our approach, we treat NPCs as individual agents who plan and act based on a goal and their knowledge of the storyworld. The practical implementation of this thesis has involved two elements: First, we created a library called NPCEngine that lets users define a storyworld and control NPCs who can act within this world. During development, we focused on creating a customizable system that lets users author complex storyworlds and NPC behavior. Second, we developed an espionage-themed mobile game intended as a technology demonstrator for the NPCEngine library.

In Chapter 2, we present an overview of previous research in the field of procedural storytelling, focusing on agent-based approaches and the concept of emergent narrative. Then in Chapter 3 and Chapter 4, we present the two practical parts of this thesis: the NPCEngine library and the mobile game. For both parts, we first analyze the stated problem, then present the chosen design, elaborate on the implementation and finally discuss the results. Lastly, in Chapter 5, we reflect on our work and provide an outlook on possible future research.

2 Related Work

This chapter briefly surveys research related to the topic of this thesis. In Section 2.1 we introduce different approaches to procedural storytelling, focusing on agent-based approaches. Section 2.2 presents a brief introduction to the concept of emergent narrative.

2.1 Procedural Storytelling

Related literature provides a variety of definitions and terms for procedural storytelling. We define it as follows: The concept of creating systems that let users influence the course of a predefined story.

As such, procedural storytelling differs from approaches that adapt a story’s narration to the user’s behavior. Such approaches do not alter the story, only how it is told. Procedural storytelling also excludes story generation, a field that deals with creating stories from scratch. Procedural storytelling differs as well from conversational artificial intelligence, which tries to emulate natural human interaction without any intended narrative.

A common approach to procedural storytelling is the drama manager. Such a system controls the storyworld as a whole and tries to enforce a certain authorial intent. Laurel [Lau86] first introduced this idea by describing an *interactive drama* using a system that could alter a story to fit a customer’s wishes. A key paper on procedural storytelling is Bates’ [Bat92] exploration of possibly using artificial intelligence for drama. He argued that considering the needs, emotions and goals of the individual actors in a story is central to an interactive storytelling system. Kelso et al. [KWB93] continued Bates’ exploration by designing three experiments to determine requirements for a procedural storytelling system.

The most significant body of work in the field of procedural storytelling is the research of Riedl and Young. Though their approaches use drama managers and do not involve agents, they cover many other aspects of procedural storytelling relevant to agent-based approaches. For instance, they extensively research the use of planners and especially partial-order planners for storytelling [YR03, RSY03, RSDA08, Rie09, RTB11]. They also explore the requirements for storytelling systems that interface with a game [Y⁺01].

More recent research into drama managers include work by Shoulson et al. [SGKB13] and Kapadia et al. [KFZ⁺15, KFS⁺16] who use this approach for story authoring systems.

Some research has also addressed defining player behavior. Mateas and Stern [MS02, MS04] introduced *A Behavioral Language* commonly used for behavior definition. ABL is based on a previous language introduced by Loyall [Loy97]. The *hierarchical task networks* (HTNs) introduced by Cavazza et al. [CCM02a, CCM02b] constitute another approach to behavior definition.

Aylett [Ayl99a] implemented the first agent-based storytelling system that creates interactive stories for small children based on the popular Teletubbies television series. She used the insights she gained from this project to define the concept of emergent narrative that we discuss in Section 2.2. Her body of work in the field of agent-based storytelling includes planning [ADP06, ALD⁺06] and how to use generative grammars for text synthesis [ALP04].

Other research into story-related planning includes Kartal et al.'s [KKG14] use of Monte Carlo Tree Search for story generation.

The key related research for this thesis is the work of Ryan et al. [RSMWF15, RMWF16]. In their approach Ryan et al. treat actors in the storyworld as agents that can form knowledge. With this approach they were able to simulate storyworlds of several hundred agents. However, their approach is not implemented in an interactive application such as a game.

Kybartas and Bidarra [KB16], and Riedl and Bulitko [RB12] each published comprehensive surveys on research in the field of procedural storytelling.

2.2 Emergent Narrative

A central insight of Bates' exploration of procedural storytelling [Bat92] is that a story can only be appreciated if the audience is willing to emotionally connect to the characters. In a following work Bates et al. [B⁺94] introduced the idea of *believable agents* as antithesis to the rational agents that the field of artificial intelligence typically studies. The concept of believable agents was explored at depth by Loyall [Loy97].

Aylett [Ayl99b] coined the term *emergent narrative* to describe stories that are not explicitly created by an author but that instead emerge in the heads of the players by experiencing the storyworld's events. Aylett points out that emergent narrative is central to many game genres, such as strategy games. She also states that emergent narrative is also present outside the world of video games, such as in team sports.

Aylett suggests that designers should consider emergent narrative a central element to any approach to interactive storytelling. A risk with relying on emergent narrative, however, is that it relies on the players' willingness to experience events as a story. Aylett therefore proposes several approaches to increase the probability of emergent narrative, such as placing players into social networks and triggering events that are "out of the ordinary".

3 NPCEngine

In this chapter we will present the first practical implementation of the thesis: the storytelling library. This library – called *NPCengine* – lets the user define a storyworld and create NPCs that can remember facts and act upon them.

First, in Section 3.1 we will analyze the problem, find requirements for the system and provide some insight on the theoretical design. Section 3.2 provides an in-depth explanation of various aspects of the implementation. Finally, in Section 3.3 we discuss the results that we obtained with our implementation.

3.1 Design

In this section, we discuss the design of the storytelling library. First, in Section 3.1.1 we look at the problem definition. On this basis, we will examine various parts of the library in greater detail, starting with the definition of the storyworld’s domain in Section 3.1.2. Section 3.1.3 describes the difference between the storyworld domain and the storyworld itself. In Section 3.1.4, we discuss the properties and functionalities that an NPC must possess in order to meet the challenges we have identified.

Two key features of the library are the definition of the storyworld and the representation of NPCs’ knowledge. As these are closely related, we present them together in Section 3.1.5. Based on their knowledge, NPCs must perform actions related to the story. Hence, we first explain in detail the concept and structure of actions in Section 3.1.6. The defining feature of our approach is that NPCs choose actions autonomously based on their knowledge in an effort to attain their goals. This process is commonly called planning. In Section 3.1.7, we discuss how NPCs decide which action to choose. To make truly informed decisions, the NPC must look a few steps ahead and predict the outcome. We use the Monte-Carlo Tree Search algorithm to perform planning that we describe in Section 3.1.8.

We combine all the requirements for the storytelling library to create an abstract architecture for an agent-based storytelling engine in Section 3.1.9.

3.1.1 Problem Definition

The goal of this thesis is to solve specific storytelling problems inherent to some video game genres, especially RPGs: Players want as much player agency as possible while still experiencing a coherent story. However, giving a player more ways to influence the video game’s story forces the game’s designers to anticipate more storylines. Generally, we can state that the complexity of required content grows exponentially with the story’s complexity and player agency. To limit NPC complexity, designers often implement NPCs as state machines who can only react to a player’s actions in a limited number of ways – often just one.

Multiplayer RPGs pose an additional challenge to storytelling, as many players interact with the storyworld at once. Telling one coherent story is nearly impossible, because players start and stop playing at different times. In such games, each player typically experiences a very similar story, told asynchronously. However, this requires the storyworld to stay more or less static so that eager players do not influence the narrative of later players.

The properties we mention above are the basis for three widespread problems in RPGs:

- *Omniscience*: NPCs who know about events that they could not know of in the real world.
- *Obliviousness*: NPCs who are unaware of major events that they could not have missed in the real world.
- *Inconsistency*: Especially in multiplayer RPGs, any player influence on the storyworld is nullified to ensure that future players encounter the same world. For example, NPCs killed by the player will typically reappear moments later.

These problems can greatly impair many players' suspension of disbelief. We therefore tried to tackle these issues with the following problem definition:

Developing a storytelling engine able to synthesize NPC behavior that fits the NPC's observable storyworld.

The work we describe in this thesis is part of a broader project, the Game Science Platform, a software framework for game development in a scientific context. This context gave rise to several requirements for this thesis that were independent of the core subject. For instance, given that the framework functions with the Unity Game Engine, we had to write the library in the C# programming language and offer an application programming interface (API). In multiplayer applications, however, the library could be part of the server-side implementation and should therefore be able to run independently of the Unity Game Engine.

3.1.2 Domain

One core purpose of a storytelling system is to enable the library's users to define their *authorial intent*, in other words, how the story should play out. A fundamental part of the authorial intent is the storyworld. The storyworld consists of elements such as people, monsters, items and locations. More abstractly, we can say that the storyworld consists of a set of objects.

Considering NPCs, we can stipulate that an NPC is nothing more than an object that can also perform actions. For instance, an NPC can talk to a player or another NPC, pick up an item, or walk to a location. Similarly, the player is also just an object in this world.

Each object in the world has a set of properties that are relevant to storytelling, such as color, size or name. Similar objects have the same types of properties, for instance, every `Person` has a *name*. In the spirit of object-oriented programming, we can therefore treat each object as an instance of a model. Furthermore, some actions can only be performed by specific models. For example, only instances of the model `Wizard` can do the action *castSpell*. Just as objects instantiate models, we establish that each of an NPC's interactions with the world is an instance of an abstract action. The system should thus represent the storyworld as a set of abstract models and a set of abstract actions. This approach is common in related literature and is called the *domain* of the storyworld.

Given that we are basing our approach to storyworld domain definition on object-oriented programming, we can use another concept from that field: inheritance. Specifically, a new model should be able to extend an existing one, thus enabling a hierarchical definition of models. For instance, the model `Blacksmith` should inherit certain attributes such as *age* from `Human`, but other properties such as *smithingAbility* should be unique to the model `Blacksmith`.

Figure 3.1 illustrates a possible representation of a storyworld domain. On the left-hand side, we declare four models. Three of them are models for humanoid characters – possibly (but not necessary) NPCs. The models `Author` and `Pilot` extend the model `Human`. On the right-hand side, we list three

possible actions. Each action specifies a model that must underlie an NPC's object if the NPC is to be able to perform that action.

To summarize, the domain definition is central to a storytelling library. The library should therefore enable the user to declare models and actions, and should support inheritance. We discuss the required properties for model and action definitions in Section 3.1.5 and Section 3.1.6.

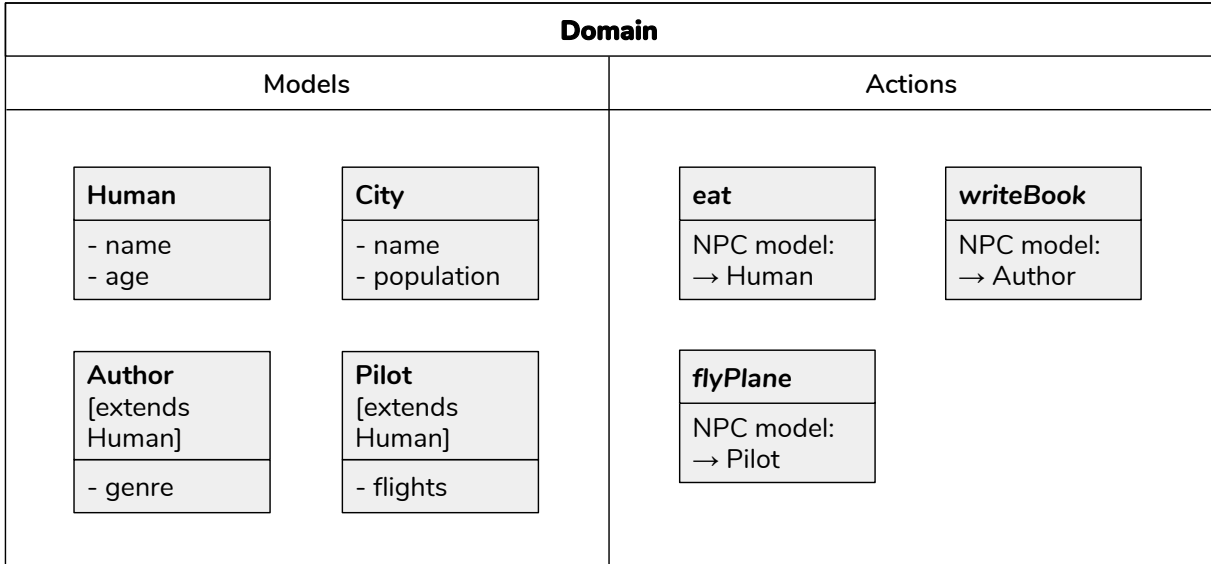


Figure 3.1: Illustration of the representation of models and actions within a domain definition.

3.1.3 Storyworld

The domain is an abstract representation of what *could* exist and what *could* happen. The storyworld, on the other hand, consists of a set of unique elements. Specifically, the storyworld contains objects that are instances of the domain's models. Some of these objects are also NPCs who can perform instances of the domain's actions. In other words, the storyworld as a whole is an instance of the domain.

To allow the user to create a storyworld, the NPC library should therefore offer two functionalities: First, the user should be able to instantiate the domain to create a storyworld. Second, the user should be able to specify which of the instantiated objects are NPCs. We will describe how NPCs work in more depth in the next section.

Figure 3.2 illustrates a possible storyworld instance of the domain defined in Figure 3.1. We've created three objects, each an instance of one of the three models *Author*, *Pilot*, and *City*. Each object will have the properties that the model defines. Given that the models of *Author* and *Pilot* extend the model *Human*, the instantiated objects will also have the properties *name* and *age* of *Human*. We have also defined two of the objects as NPCs. Each NPC can perform the actions assigned to its model. However, given that *Author* and *Pilot* both extend *Human*, the two NPCs will also be able to perform the action *eat* which is assigned to *Human*.

3.1.4 NPCs

By instantiating the domain into a storyworld and defining which objects are NPCs, we have successfully created an initial storyworld. To create a dynamic storyworld, however, we need to let the NPCs and the

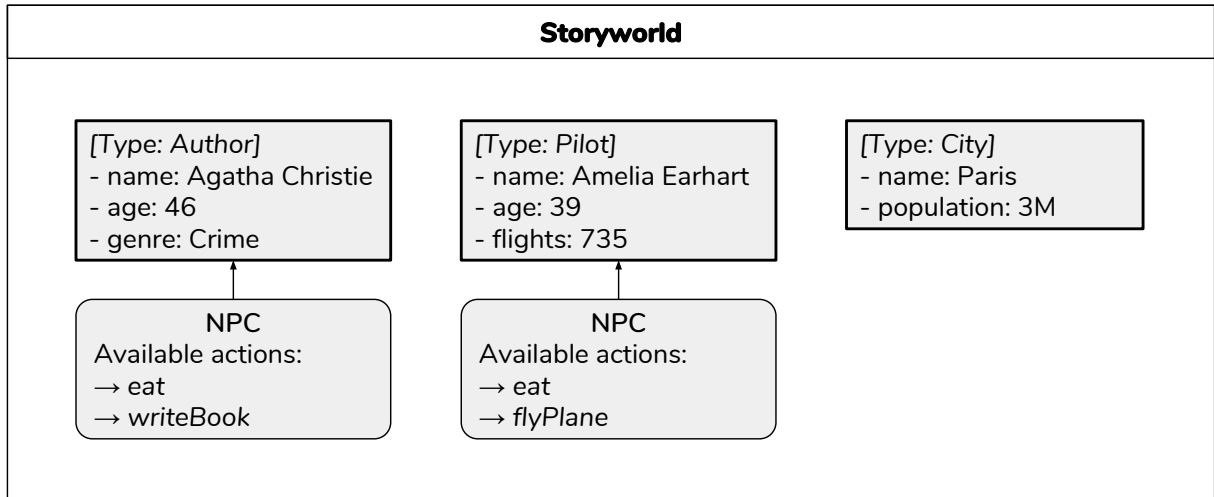


Figure 3.2: Illustration of a storyworld instantiating the domain in Figure 3.1.

players act and interact. Let's focus on NPCs: Given that the NPC's object is an instance of a model, the NPC can perform any action in the domain that its model can access.

An NPC will base its choice of action on the *goal* that the user has assigned to it. A goal is a state of the world that the NPC is trying to reach. A possible goal might be: “become a powerful ruler”. The NPC therefore chooses the action that it believes most likely to lead to its goal. Controlling the NPCs is therefore a major task of the NPC library and consists of two parts: setting goals and triggering actions. Section 3.1.6 and Section 3.1.7 describe these functions.

When considering how NPCs choose a task, we must determine on which facts NPCs base their choices. The NPC should not be able to access all information on the storyworld, as this would render the NPC undesirably omniscient. Instead, we need to let the NPC collect and store information on the storyworld. This means that we must make the NPC able to perceive its surroundings. Technically speaking, the system must let the user give an NPC data on the storyworld. However, perception is not the only way an NPC acquires information. As in the real world, much information might be provided by other characters. Hence, NPCs must be able to use structured data in their communication. This approach lets the NPC receive, but also forward information about the storyworld.

To use the information it has gathered, an NPC must store it in a structured way. Equally importantly, however, the NPC must be able to determine what it *doesn't* know. This allows the NPC to determine what as-yet-unknown information might be relevant for a specific decision – and try to gather that information, either through perception or from another character. Section 3.1.5 explains the representation of NPC knowledge in more detail.

3.1.5 Storyworld and Knowledge Representation

As Section 3.1.2 describes, the domain consists of a set of models and actions. In this section we focus on the required representation of models and instantiated objects. We will discuss actions in the next section.

Imagine that we want to create the storyworld of *The Simpsons'* Springfield. When designing the required domain, our goal is to define a set of `Models` that allow for any town containing humorous animated characters. Also, each `Model` should have a certain number of `ModelProperties` – these are the properties that each instance of the `Model` will have. Within our domain, the first obvious

Model would be *Human*. Among many others, our *Human* will have the *ModelProperties* *name* and *hairColor*. To summarize, the domain contains a set of *Models* that in turn contain sets of *ModelProperties*. The top row of Figure 3.3 shows this relationship.

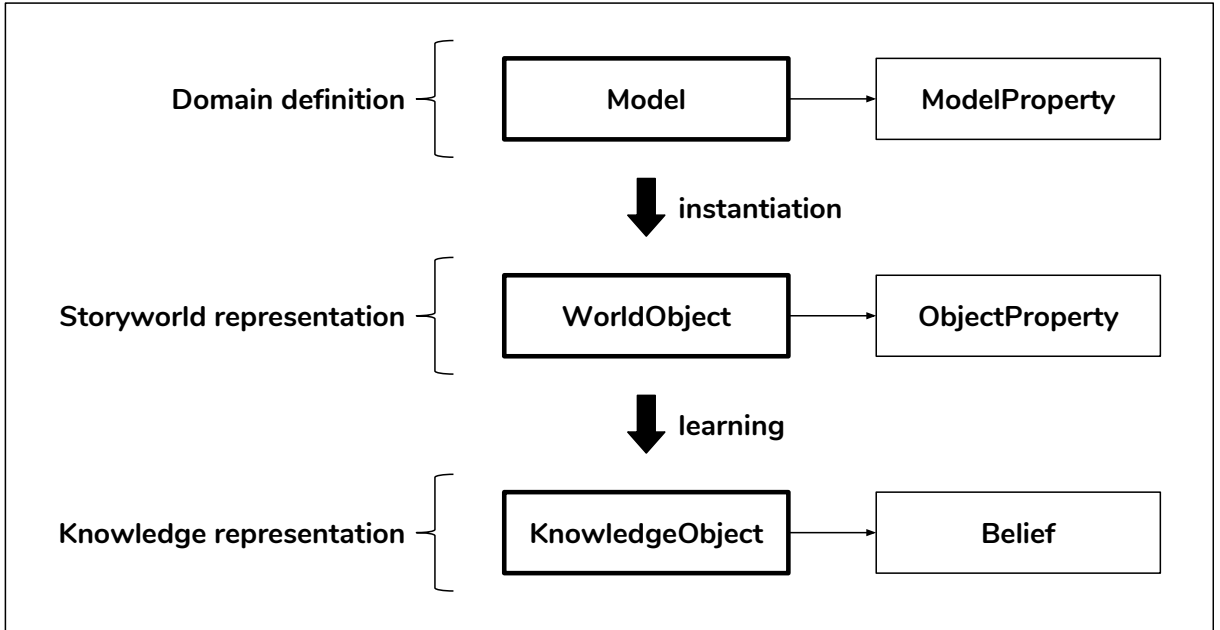


Figure 3.3: Similarities in the representation of the domain, storyworld and knowledge. *WorldObjects* are instances of *Models*. Similarly, *KnowledgeObjects* represent what an NPC knows about a *WorldObject*.

Now that we defined the domain we can instantiate it to create our version of Springfield. For every character in the town, we will create a *WorldObject* that instantiates the *Model Human*. For each of the *ModelProperties* that we defined for this *Model*, the *WorldObject* will receive a corresponding *ObjectProperty*. In our case, each *Human* will have the *ObjectProperties* *name* and *hairColor*, and corresponding values. For instance, one *WorldObject* will contain the two *ObjectProperties* *name=Marge* and *hairColor=blue*. We can state that, like the domain, the storyworld consists of a set of *WorldObjects* that each have a set of *WorldProperties*, as the middle row of Figure 3.3 shows.

As the previous section described, NPCs must be able to gather knowledge of their surroundings. Given that an NPC cannot know everything, its knowledge is a subset of the storyworld. One can thus argue that an NPC's knowledge is an instance of the storyworld. In keeping with the hierarchical structure that we use for the storyworld, the NPC will create a *KnowledgeObject* for each *WorldObject* that it knows. In our example, if the NPC has seen or heard of Marge, it will create a corresponding *KnowledgeObject*. It will also create a *Belief* for each *ObjectProperty* that the NPC knows the value of. If the NPC passes Marge on the street and does not know her name, it will have a *Belief* *hairColor=blue*. Or if it knows Marge by name but has never seen her in person the NPC might only have the *Belief* *name=Marge*. However, someone might provide the NPC with incorrect information on Marge, which might lead to a *Belief* *hairColor=red*. In another scenario, the NPC might have heard of Marge and have met her in real life but not realized that both were the same person, leading the NPC to create two *KnowledgeObjects* about a single *WorldObject*.

As the bottom row of Figure 3.3 shows, we can once again retain that an NPC's knowledge consists of a set of *KnowledgeObjects*, each containing a set of *Beliefs*. The similar hierarchical structure between the domain, storyworld, and knowledge representations both simplifies our understanding of the matter and lets us use common data representations and algorithms during implementation.

An interesting situation arises when two sources provide the NPC with conflicting information on the same `KnowledgeObject`. The NPC must decide which of the provided values is correct. To do so, the NPC must evaluate and compare certain properties of both `Beliefs`, such as trustworthiness of their sources or whether they are up-to-date. Indeed, some `ObjectProperties`, such as `hairColor`, might change their value over time. Also, unrelated events – such as discovering that the source is a liar – may lead the NPC to reject a previously favored `Belief`. The NPC must therefore store all the `Beliefs` it has ever held about an `ObjectProperty`. Keeping these competing `Beliefs` allows the NPC to decide in an ad-hoc fashion which `Belief` it deems to be correct. Ryan et al. [RSMWF15] proposed this concept of rival *candidate beliefs*. Based on their research, we can establish a set of `Belief` attributes:

- *Subject*: This is the `KnowledgeObject` about which the NPC has created the `Belief`.
- *Subject Property*: This is the `ModelProperty` that the `Belief` pertains to.
- *Value*: The value of the subject property for this specific `Belief`. As we already said, this value might contradict other `Beliefs` about the same subject property.
- *Source*: The NPC or human actor who provided the NPC with the information. If an NPC bases its `Belief` on its own observation, the NPC itself is the source.
- *Strength*: This value states how true the NPC deems a `Belief` to have been at the time of its observation. We can base this value on the trustworthiness of `Belief`'s source and possibly other factors, such as the source's own certainty on the value or its "rememberability" (i.e., how easily an actor can remember the property). The strength of a `Belief` should decay over time, to simulate forgetfulness in an NPC. The importance of the `Belief`'s subject property or a forgetfulness constant of the NPC can influence the forgetfulness factor.
- *Accuracy*: This value states how true the NPC deems that a `Belief` currently is. Differentiating between strength and accuracy lets the NPC form `Beliefs` on properties that are expected to change over time. The accuracy is therefore not only based on the `Belief` strength, but also on the time that has passed since the NPC last knew the `Belief` was up-to-date. The factor for this decay should reflect the probability that the subject property will change. For instance, the accuracy of a property `mood` should decay quickly, while the accuracy of `firstName` should not change over time.

3.1.6 Actions

In the context of the NPC's choosing which action to perform, we treat each action as an atomic and instantaneous transition from one state of the storyworld to the next. This is a common approach in related research. The NPC library therefore does not simulate the NPCs continuously. Physical interactions, text output and animations are tasks of the user application. The application calls the library only when an NPC must decide what to do next. Such an occasion might regularly arise as part of the application's update call or be triggered by interaction with another character. The library's task is then to determine – on the basis of the NPC's knowledge of the storyworld – which action to take. We split these actions into two distinct groups:

- *Reflective actions*: These are actions that influence only the acting NPC and other non-acting objects. For instance, the NPC puts on a hat. Such actions are self-contained as only one NPC is involved. However, there is no external interaction to provoke reflective actions. The system must therefore let the user trigger the NPCs performance of reflective actions regularly – typically by means of a periodic update call.

- *Interactive actions*: These actions involve the NPC and an acting counterpart. This counterpart can be a player or another NPC – from the NPC’s perspective both are just objects in the world. For example, asking another character for information is an interactive action. The differentiating characteristic of such actions is that the counterpart’s behavior potentially influences the NPC’s actions. Interactive actions cover most behavior that a player witnesses from an NPC. Unlike reflective actions, interactive actions can be provoked by a counterpart, often the player. The system should therefore allow the user to pass the result of an interactive action to an NPC and let the NPC react to the input. The messages used for communication must hence have a simple interface. This is especially true in a client-server environment, where the server computes the NPC’s behavior, while the counterpart might be a player performing actions on a client application.

How players perform actions is not the responsibility of the library. However, player actions that involve NPCs must use the same message interface as the NPCs’ interactive actions. Using a common interface allows the NPC to react appropriately. Conversely, NPCs use the same format to forward the results of their actions involving the player. We believe that this form of communication allows for more realistic interactions, since an NPC does not know whether its counterpart is a human player or another NPC.

While interactive actions require two actors, there is no restrictions for reflective actions. The NPC can perform reflective actions either alone or while interacting with another character. For instance, an NPC might perform the action `writeNotes` when alone or while the player provides the NPC with information. The set of actions an NPC can choose among is hence only restricted in non-interactive settings.

To determine which action the NPC should choose, and how to perform it, we need to know certain action-specific attributes. For instance, the definition of each action must state under which conditions an NPC can perform it as well as its effect on the storyworld. However, the expected effect might not correspond to the actual effect. For instance, asking another character for information does not guarantee that the other character will actually provide it.

On the basis of the inheritance features that Section 3.1.2 describes, an action definition in the domain should include the following attributes:

- *Actor model*: This declares which NPCs can perform this action. Specifically, this refers to one domain `Model`. Any NPC whose `WorldObject` instantiates that `Model` or inherits from it will consider this action while deciding on which action to perform.
- *Counterpart model*: This attribute is only necessary for interactive actions. It determines whether an NPC can interact with a counterpart through this action. For example, an NPC should only consider `askToGrantWish` when the counterpart instantiates the `Fairy` model.
- *Precondition*: This attribute is a predicate. If the NPC evaluates this to be true based on its current knowledge of the storyworld, then the NPC can consider this action. For instance, the precondition for `eat` might be `isHungry AND hasFood`.
- *Postcondition*: This attribute specifies what outcome the NPC *expects* by performing the action. We formulate it as a function that the NPC executes within the representation of its current knowledge of the storyworld. When the NPC is planning, we use this attribute to determine which action to pick next. The result of the postcondition function is this is a prediction of the most likely effect of the action based on the information available to the NPC.
- *Effect*: Like the postcondition, this attribute is a function. However, unlike the postcondition, the system calculates the effect only once the NPC has *decided* to do this specific action. Unlike the postcondition, the effect function thus interacts with the storyworld and other NPCs. For instance,

the effect of an interactive action might require input from a counterpart. Ideally, the result of the effect function is identical to that of the postcondition. However, depending on the situation, they may diverge significantly.

Table 3.1 shows an example action definition that allows an NPC to ask its counterpart for its name. We define both the actor and counterpart’s models as `Human`. The definition implies that all humans have a name and can be asked for it. On the other hand, it also means that any non-human NPC either has no name or cannot be asked for it. The precondition states that the NPC should only consider performing the action if it doesn’t know the counterpart’s name. When deciding which action to choose, the NPC executes the postcondition and determines the probable outcome – in this case, that the NPC will know the counterpart’s name if the counterpart doesn’t lie. After considering all possible actions, the NPC might consider the one in Table 3.1 the most promising. The NPC will then execute the action’s effect function, which involves requesting and processing the counterpart’s input.

Action askForName	
actorModel	Human
counterpartModel	Human
precondition	<code>actor.knows(counterpart.name) == false</code>
postcondition	<pre> if actor.knows(counterpart.isLiar) : actor.knows(counterpart.name) = false else : actor.knows(counterpart.name) = true </pre>
effect	<pre> actor.ask(counterpart, "What is your name?") answer = actor.getAnswer(counterpart) if isName(answer) : actor.learn(counterpart.name, answer) </pre>

Table 3.1: Example in pseudocode of the definition within the domain of an action that lets an NPC ask another character for its name.

While defining the domain’s actions, we must consider that in some situations, none of the defined actions are appropriate. Such circumstances emerge when the precondition functions of all actions return a value of false. Given that the user defines the preconditions, we cannot prevent such situations. Our library should therefore provide each NPC with a fallback action that it can perform if it finds itself in such a deadlock.

3.1.7 Planning

As the previous section described, we treat actions performed by the NPCs as atomic and instantaneous. But time may pass *between* actions – during which conditions within the storyworld may change. Similarly, the NPC might inadvertently influence the storyworld by performing an action. We must therefore expect unpredictable changes to the storyworld between each action. NPCs should therefore not follow preplanned sequences of actions. Instead, when the user application requests the NPC to perform an action, the NPC should determine the one that is the most appropriate at that time.

The task of finding the best action on the basis of the information available is one of the core fields of artificial intelligence and is commonly called *planning*. A generalized planning problem requires three inputs:

- *Initial state*: A set of properties that define the current world state. This is commonly defined as a logical proposition.
- *Actions*: The set of actions that we can choose from. Each action has a precondition, i.e. the condition under which an NPC is allowed to perform it, and a postcondition, i.e. the action's effect on the world state. Applying the postcondition to a world state yields a new world state.
- *Target state*: This is the world state that we want to reach. Like the initial state, it is often defined as a logical proposition.

For our NPC's planning problem, the initial state is the NPC's knowledge as described in Section 3.1.5, as it represents the world state as the NPC observes it. However, since the NPC's knowledge may contain erroneous information, the planning process might yield an action that is suboptimal or even unsuited to reach the intended goal. This is fully intentional behavior, as it emulates our human fallibility when provided with incorrect information.

Next, the actions required for our planning problem are those actions in the domain that this NPC's model can perform. As we stated in Section 3.1.6, each action has a pre- and postcondition. However, unlike the generalized planning problem above, we cannot be certain of the actual results of an action. We must therefore differentiate between the expected result, i.e. the postcondition, and the actual result, i.e. the effect function.

Given that we are trying to mimic human decision-making, defining a fixed target state is difficult. An NPC might for instance strive to be happily married, rich or powerful – or any combination thereof. Some goals might be unreachable, leaving us with an unsolvable planning problem.

Another issue arises when we consider how many successive actions the NPC might have to perform in order to reach the goal. We humans do not plan every step of our lives at birth. Instead, we plan a few steps ahead, trying to predict which decision will lead to the most favorable outcome. Similarly, realism requires that our NPCs not plan too far ahead. Moreover, planning the entire sequence of actions might be prohibitively expensive. As the space of possible storyworlds is potentially infinite, this could lead to an endless planning task.

To resolve these issues, we don't aim for a specific target state. Instead, we pick the actions that are the most promising for the NPC. We therefore replace the third input of our planning problem, the target state, with an *evaluation function*. This function returns the quality of a potential storyworld state in terms of the NPC's goal. The evaluation function thus implicitly defines the NPC's goal, as evaluating the target state would yield the highest possible score. Using an evaluation function also lets us express more precisely the vague goals that are typical of human nature.

Listing 3.1 illustrates an example evaluation function that takes into account several different criteria for a potential storyworld. Having access to a variety of control flow commands such as conditional expressions or loops lets the user define an evaluation function that is much more expressive than a fixed goal state.

As with the target state, we can use functions instead of immutable expressions in other parts of the planning process for a more versatile planning environment. Appropriate applications of such functions are actions' pre- and postconditions. We are unaware of any related research that has let pre- and postcondition contain functions. Past approaches limit the precondition's definition to a conjunctive predicate of literals. Similarly, the approaches define postconditions as propositions devoid of conditional effects. These limitations allow the use of propositional logic to solve the planning problem. These approaches

```

1 evaluate(state) :
2
3     goodness = 0.0
4
5     if state.myself.hasFood > state.myself.hasHunger :
6         goodness += 1.0
7
8     goodness += state.relationship.Happiness
9
10    if state.relationship.isMarried :
11        goodness += 0.5
12
13    foreach neighbor in state.neighbors :
14        if neighbor.gold < state.myself.gold
15            goodness += 0.2
16
17    if state.kingdom.king == state.myself :
18        goodness += 0.1 * state.kingdom.numberOfWorkers
19
20    return goodness

```

Listing 3.1: Example of an evaluation function that an NPC can use to determine how good it deems a potential state to be.

can also benefit from efficient planning algorithms that rely on the grouping of literals. In contrast, for the library we present here, we decided to favor expressiveness over efficiency. We therefore explicitly expect the pre- and postconditions to be executable functions, as Table 3.1 shows.

To understand how our NPC performs planning, let’s consider a simplified version of the process, where we only look one step ahead. Figure 3.4 summarizes this approach. To determine which action to choose, we need to complete the following steps:

1. We need to determine which actions the NPC can perform. In our architecture, each NPC is a `WorldObject` within the storyworld representation (see Section 3.1.3). Given that each action states which `Model` can perform it, we can assemble the list of performable actions by picking those actions that the `Model` of the NPC’s `WorldObject` can perform.
2. We further reduce the list of actions by executing each action’s precondition with the initial state as input. The initial state is the current world state from the NPC’s perspective, i.e. the NPC’s knowledge. Each precondition is a function that returns a boolean value stating whether the NPC can perform the related action under the given circumstances or not. To obtain the list of actions that the NPC deems appropriate, we discard any functions whose precondition function returns a value of false.
3. We initialize each remaining action as a so-called *action instance*. The action instance contains two elements: a copy of the current state and a version of the domain action in which placeholders such as actor and counterpart have been populated.
4. Once instantiated, the state contained in the action instance is independent of all other states. We can therefore now execute each of the actions’ postcondition functions, which will only alter the instance’s state. We are then left with a set of hypothetical states, each differing from the initial state by the expected effect of one of the actions.
5. We can now execute the evaluation function on each of the states. The function will return the “goodness” of the state in terms of the NPC’s goal. We then have the NPC choose its next action by picking the action whose state was evaluated to be the best.

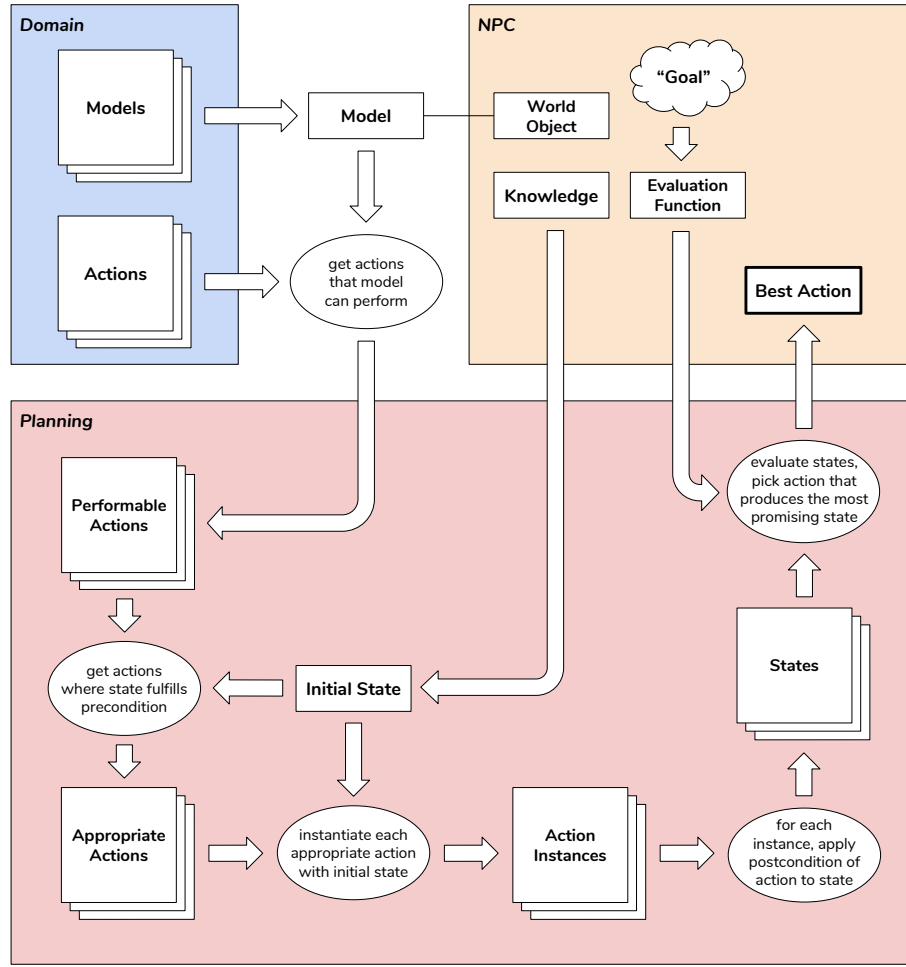


Figure 3.4: Simplified visualization of the library's planning process.

3.1.8 Monte Carlo Tree Search

In the planning process that Section 3.1.7's enumeration describes, we are only looking one step ahead. We therefore expect that at least one of the available actions will have a more favorable predicted outcome than the others. However, considering that goals may be abstract and high-level, we will likely have to plan several steps ahead before seeing any change in the evaluation of the hypothetical states.

This complicates our planning algorithm: Although our goal is still only to determine the next action that an NPC should perform, the impact of that action may only be detectable after the NPC has performed several consecutive actions. The planning process of our library should therefore not only be able to determine the effect of one action but also forecast the effects of several consecutive actions.

We must therefore introduce a data structure that can represent many combinations of future steps. To achieve this, we use a tree structure in which each level's branching corresponds to the available actions at that point. Each node in the tree represents an action instance, i.e. an action and a resulting state. Therefore, each path from the root to a leaf constitutes one possible series of actions. Figure 3.5 shows such a tree for an NPC prince trying to marry a princess. Let's assume *dragonAlive=true* is part of the initial state. The top-level node labeled *slayDragon* consists of the action *slayDragon* and a state that is a copy of the initial state except that *dragonAlive=false*.

Planning several steps ahead also has some implications for how we define actions. To limit the number of potential actions, an NPC only considers those actions that it itself can perform. This means

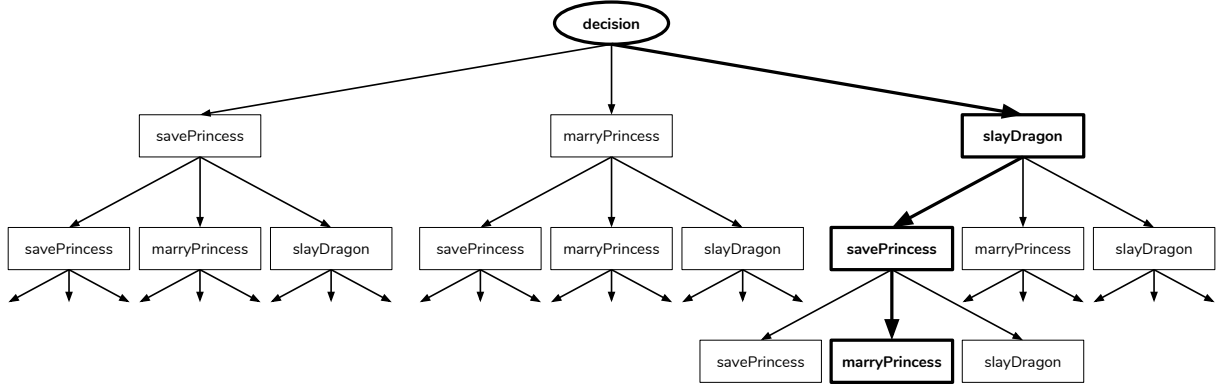


Figure 3.5: Tree structure of possible action sequences in a world where a dragon is holding a princess prisoner. The goal of the NPC prince is reach a state where he is married to the princess. If the NPC plans only one step ahead, it cannot determine which action to choose first.

that it ignores potential intermediate actions by other actors. We must therefore define the postcondition function of each action so as to take account of all the probable actions that others may perform before the NPC performs its next action. For instance, the action `askForTime` should not have the postcondition

```
counterpart.hasBeenAskedForTime = true
```

but instead

```
if counterpart.ownsWatch : actor.knowsTime = true
```

From a computational perspective, planning several steps ahead is unfortunately very expensive, as the number of possible action combinations grows exponentially with the number of steps we are looking ahead. Also, since we are trying to emulate human behavior, we can expect the number of possible actions for each step to be relatively large – implying a high branching factor. We therefore need a suitable tree search algorithm to efficiently find the most promising series of actions.

Related research uses a variety of planning strategies involving various tree search algorithms. A focus on efficiency and accuracy usually guide the choice of a specific approach. In contrast, we are less concerned with the planning accuracy of our library. We hope that players will interpret incorrect behavior as human-like fallibility. We are nevertheless still interested in an efficient tree search algorithm, as we expect our tree to be too large for naive search strategies.

Our library therefore uses the *Monte-Carlo Tree Search* (MCTS) algorithm. Ertel et al. [ESS89] first introduced this approach in 1989 and Google’s AlphaGo [SHM⁺16] artificial Go player has popularized it in recent years. The core idea behind MCTS is that we iteratively test different paths from the root to a leaf node. In each iteration, we try to learn from previous iterations and favor more promising branches of the tree. MCTS is thus a heuristic approach to tree search. It can guarantee the correct result only in infinite time. The deviation between the MCTS result and the perfect result converges logarithmically towards zero with the number of iterations.

Given its iterative nature, MCTS can be interrupted at any point during the search process. When interrupted, it will yield the best result that it could find up to that point. This lets us define a fixed computational budget for the tree search and nevertheless be sure that we will have a result after the budget is depleted. Given that we strive to achieve plausibility and not infallibility in our library’s NPCs, we can accept a non-perfect result from the tree search.

3 NPCengine

An MCTS search starts out with a tree consisting only of the root node. In our case, this is the initial state. During each iteration, the MCTS search perform the following four steps:

1. *Selection*: We traverse the tree following one path from root to leaf. At each node we pick the child node according to some predetermined reasoning. Let's pick a random child for now; we will explain our actual approach below. We continue picking child nodes until we've reached a leaf, i.e. a node without children. In the first iteration, this node will be the root.
2. *Expansion*: We create a set of child nodes for the selected leaf. In our case, we take the resulting state of the selected node and create a child action instance for every possible action. Ideally we would create all children. However, if we expect the branching factor to be too high, we limit their number. We then select one of the children at random.
3. *Simulation*: We evaluate the newly selected node. For our planning purposes, we first apply the action's postcondition function to the state of the action instance and then evaluate the resulting state by using our evaluation function.
4. *Back-propagation*: We take the value returned by the evaluation function and pass it up the tree until we reach the root node. At each node we update the information necessary for the next selection step.

The MCTS's method for choosing a node's child during the selection step determines the efficiency of the search algorithm. In *pure MCTS*, we pick a child at random, as stated above. This means that over time, we will have searched each branch of the tree equally.

However, for a more efficient search, we should instead favor the more promising branches of the tree. One possible solution would be to pick the child node whose branch's states evaluate on average to the highest value. This approach could, however, yield a situation where we start erroneously favoring a branch due to a some random, non-representative, overly-positive results. To avoid getting stuck in such a local maximum we must sometimes select less-promising branches.

Kocsis and Szepesvári [KS06] propose a child node selection method called *Upper Confidence Bound 1 applied to trees* (UCT) that does just that. During the back-propagation step of the MCTS search we calculate the UCT value for each visited node. The UCT value consists of the mean of the evaluation function's results for all states in the node's branch. But it also contains the upper confidence bound of that branch. We can calculate the UCT value by using the following formula:

$$\bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}$$

where

$\bar{x}_i$ is the mean result of the evaluation function for that branch, scaled to a value between 0 and 1

n is the total number of iterations and

n_i is the number of iterations that have passed through this branch.

Each time the MCTS search picks a child node during a selection step, it will choose the child with the highest UCT value.

This gives an MCTS search that uses UCT an interesting behavior: If the MTCS search hasn't selected a node often, the node's upper confidence bound will be the main determinant of its UCT value. Conversely, the more the search algorithm selects the node, the lower the node's upper confidence bound

will be. Hence if the search hasn't selected the children of a node often, the child that the search has selected the least will have the highest upper confidence bound and therefore the highest UCT value. This means that in a freshly-expanded node, all children will have the same chance of being selected.

However, once the MCTS search has selected a node frequently, the upper confidence bound will be much lower, meaning that the mean of the evaluation becomes the main determinant of the UCT value. In other words, the more the search has selected the children of a node, the more the algorithm will favor the most promising nodes.

Using UCT with MCTS has some drawbacks. For instance, it does not consider the risk of a branch. A branch with many equally-promising paths thus may get the same attention as a branch yielding both phenomenal and terrible evaluation results.

Using UCT and MCTS also has implications for our planning process. For instance, given that the MCTS explores freshly-created child nodes equally and that we can interrupt the search at any iteration, we must ensure that children are picked randomly, as otherwise the MCTS might favor the first branches over later ones, leading to a bias in our planning. We must also let the user fix a computational budget for the MCTS search. If the user expects high branching factors, he or she should be able to limit the number of children per node by setting a maximum number. On the other hand, if the branching factor is low, the MCTS may perform an unrealistically deep search in the tree. We should therefore also let the user limit the depth of the MCTS search.

3.1.9 Library Architecture

The requirements and insights from the preceding sections let us design a rough architecture for the library. The class diagram in Figure 3.6 presents an overview of the required elements. We divided the architecture of the library into five areas:

- *Domain definition:* The user can use these classes to define the abstract models and abstract actions that the storyworld can contain (see Section 3.1.2). A `Model` object will contain one or more `ModelProperty` instances, each defining a property that `WorldObjects` of a specific model can have. Each `Action` defines which `Model` can perform it.
- *Storyworld:* The storyworld is an instance of the domain (see Section 3.1.3). Its centerpiece is a `World` entity which consists of several `WorldObjects` and `Actors`. Each `WorldObject` represents an entity in the storyworld that instantiates a `Model` from the domain. A set of `ObjectProperty` objects describe each entity by specifying values for each of the `Model`'s `ModelProperty` objects.
- *Actor management:* This area is responsible for the acting and interacting parts the storyworld. The user can define certain `WorldObjects` to be `Actors`. In the context of the library, we call both agents and players "actors", given that they "act" within the storyworld. Therefore, both the `Agent` and `Player` classes inherit from the abstract class `Actor`. Interaction within the storyworld occurs between `Actors`. NPCs thus cannot differentiate between human players and other NPCs.
- *Knowledge representation:* Each `Agent` stores its knowledge of the storyworld in a `Knowledge` object. We represent the `WorldObjects` known to the NPC as a `KnowledgeObject` within the `Knowledge` object (see Section 3.1.5). We store the `ModelProperties` that the NPC knows the values to in `Beliefs`. Importantly, the NPCs' knowledge is independent of the storyworld representation. `KnowledgeObject` and `Beliefs` do not refer to specific objects in the storyworld. This allows the NPC to confuse people or objects, or build knowledge on entities that don't actually exist.

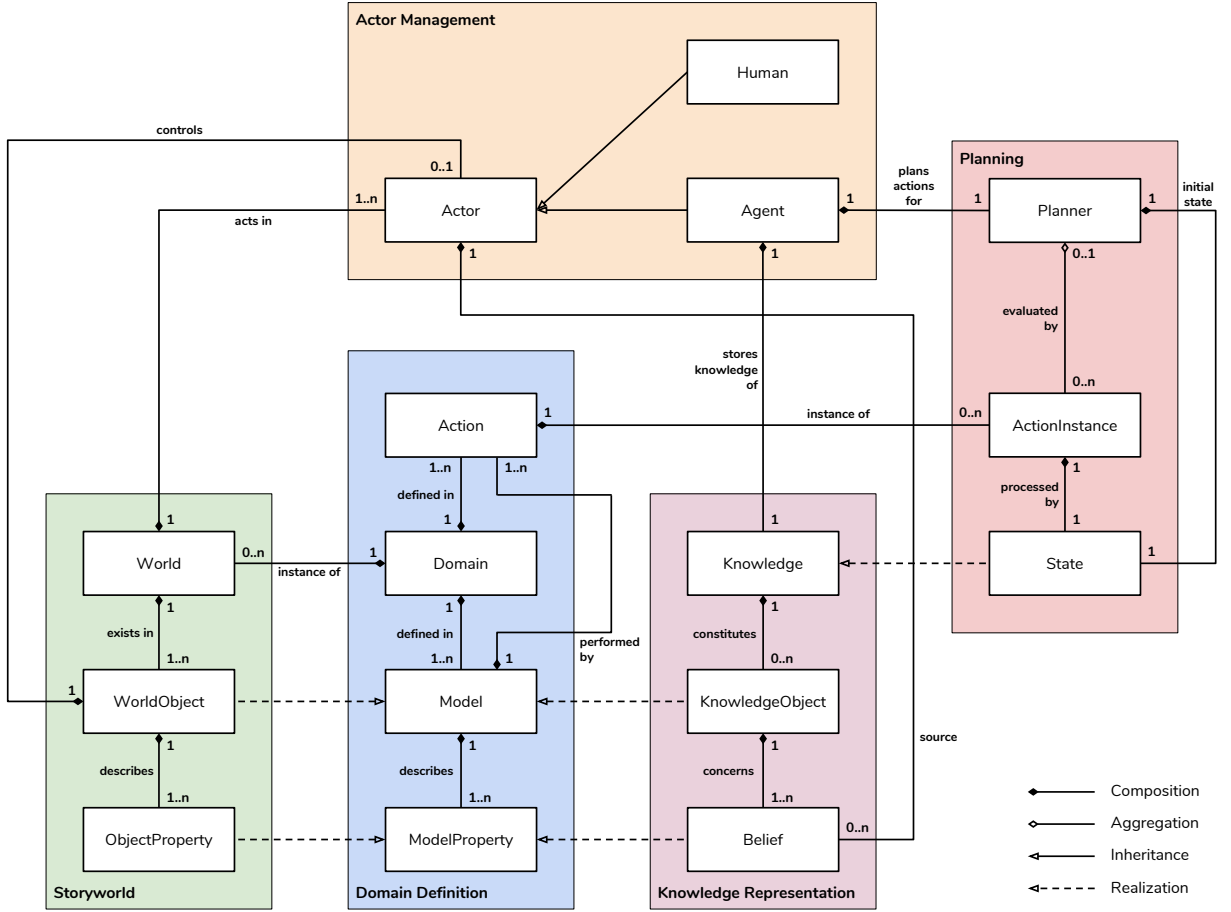


Figure 3.6: Simplified class diagram for the architecture of the procedural storytelling library. We group the classes into five areas: (1) the domain definition, which establishes the available models and actions; (2) the storyworld, which stores the current state of the world; (3) actor management, which allows the user to create and control NPCs and players; (4) the knowledge representation, which tracks each NPC’s knowledge of the storyworld; and (5) planning, which we task with finding actions for the NPCs.

- *Planning*: An NPC uses this area to plan its actions. Each Agent has a Planner object that controls the planning process. It stores the current state of the NPC’s Knowledge as a State object. The State object contains the same information as the Knowledge object, but optimizes the access, duplication and modification of the data during the planning process. While planning, the Planner will create ActionInstances, each having a State, which represents the hypothetical state of the world if the NPC were to pick that Action (see Section 3.1.7).

3.2 Implementation

This section describes how we implemented the NPCengine library. We start with an overview of the library's architecture in Section 3.2.1. The subsequent sections explain various areas of the library in more detail; some discuss the implementation of the design that Section 3.1 describes, while others introduce additional functionalities that we included in the library.

Concerning the core implementation, we look at the following subjects: Section 3.2.2 explains the basic structure of the state we use for planning, given that the attributes of a state determine how the user must formulate actions. In Section 3.2.3, we describe how users can define a storyworld domain by using the library. On the basis of this domain, a user can instantiate a storyworld, as we elaborate in Section 3.2.4.

In the remaining section, we discuss a feature that extends the core NPC system: To ensure that a client application can react to all possible tokens, we implemented a token generator that we present in Section 3.2.5.

3.2.1 Extended Architecture

The library architecture that we presented in Section 3.1.9 only covers the core functionality of our NPC system. The implementation of our library comprises a larger set of modules, many of which serve as an interface between the system's core and the library's user.

Figure 3.7 shows the architecture we implemented. Given that the library is intended for multiplayer applications, we designed the library in line with the server-client pattern. However, the user can employ these client-side and server-side sections of the library together in a single-player application. We also created a module that a user can employ during development and that is thus independent of the client-side or server-side sections of the library.

We use two technologies throughout the library: JavaScript Object Notation (*JSON*) and the *ECMAScript* scripting language. We've chosen these two technologies because of their common syntax and their frequent use in modern applications.

The domain definition, serialized communication messages, and our internal database all use the JSON notation to store data. For serialization and deserialization we use Newtonsoft's *Json.NET* library, as it is the most widely used JSON serializer available as a .NET library.

When initializing the library, the user defines all functions in ECMAScript 5. This includes the pre- and postcondition, the effect functions of actions, and the NPCs' goals. As we examine below in Section 3.2.2, actions can also contain variables. The user defines these variables in ECMAScript as well. We use all these functions during our server-side planning process. For efficient execution, we use the *Jurassic* ECMAScript runtime environment. We chose the Jurassic library over other available .NET libraries for efficiency: unlike other ECMAScript libraries, Jurassic is not an interpreter. Instead, Jurassic compiles ECMAScript code into .NET bytecode. This means that we can use the ECMAScript functions during planning without the overhead of interpreting the ECMAScript code at each iteration.

We also use ECMAScript for text synthesis in the client-side section of the library. Given that the client side only has access to the data that the NPCengine forwards by means of the communication messages, we deemed including the Jurassic library unnecessary. Instead, we implemented a limited ECMAScript interpreter called SimpleJS, which can interpret ECMAScript expressions but which supports neither functions nor control flow commands.

3 NPCengine

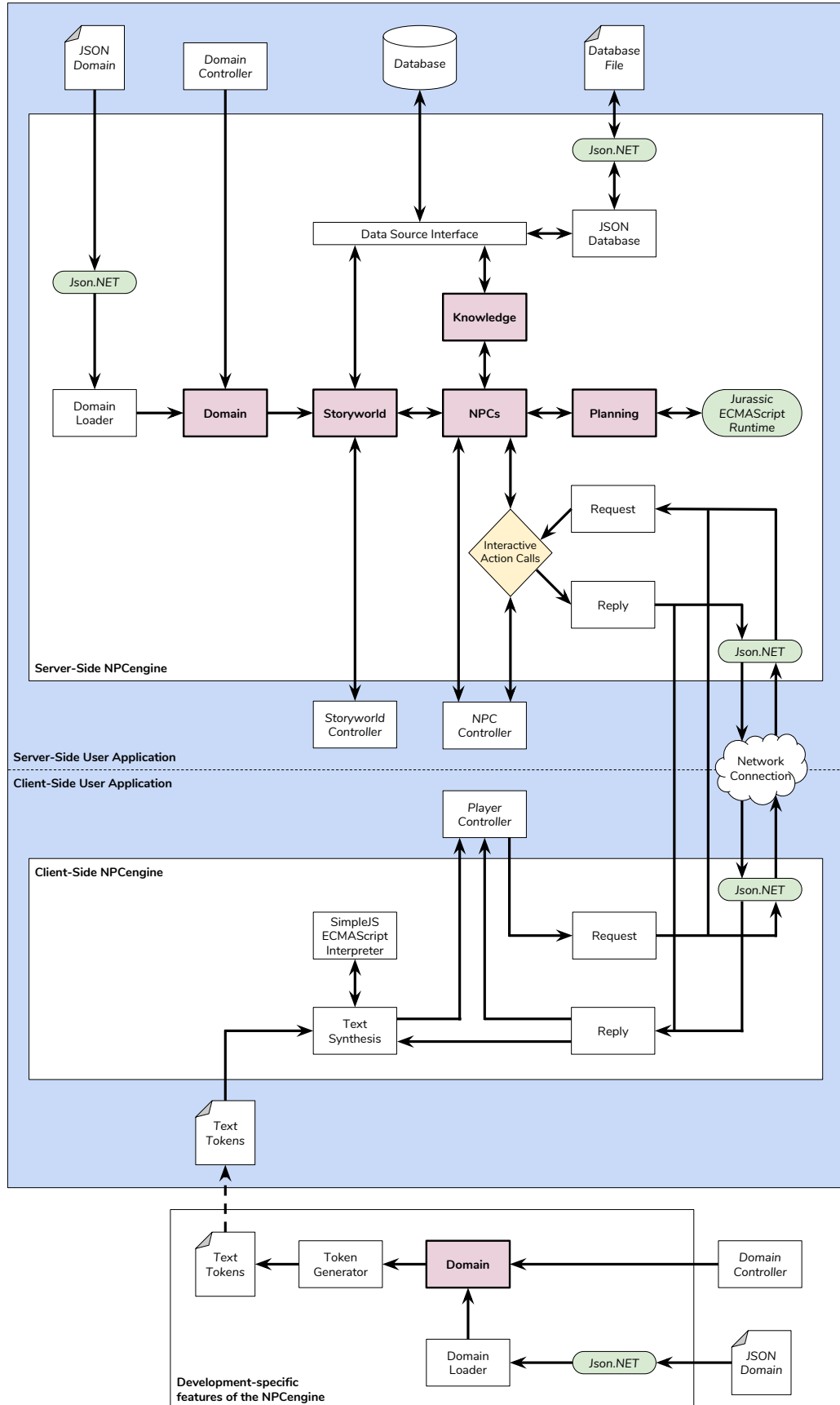


Figure 3.7: Extended architecture of the NPCengine. We divide the library into three application fields: The server side, the client side, and the section the user employs for development.

The API of the NPCEngine's server-side section lets the user access four elements of the system that the user must manage:

- *Domain definition*: The user must initialize the system with a domain. He or she can do so by loading JSON files containing the domain definition (see Section 3.2.3). We've created a domain loader module that is responsible for parsing, assembling and validating the domain from JSON files. The user can also define and alter the domain through API calls. However, the NPCEngine locks the domain once the user has instantiated a storyworld.
- *Database*: The user must provide the library with a database to store the loaded domain, the initialized storyworld, and the NPCs' knowledge. The library contains a data source C# interface, which the user must implement and link to an external database. We've designed the interface around a relational database scheme. We also provide the user with a JSON database, a ready-made implementation of the data source interface that uses a JSON file as storage. The inefficiency of this JSON database implementation is irrelevant to the efficiency of the library as whole, as each NPC call accesses the database only once.
- *Storyworld*: The user can instantiate the domain into a storyworld. Once created, all objects of the storyworld are freely accessible through the API, allowing the user to influence the world as he or she wishes. For instance, the user can freely extend the storyworld at any time.
- *NPCs and players*: The user can turn objects of the storyworld into NPCs and players. The user can set each NPC's goal and have the user's application add or remove beliefs from the NPCs' knowledge. To have an NPC perform an action, the user has two options: Either the user triggers a reflective call for one NPC, in which case the NPC will only consider reflective actions, or the user can start an interaction between two actors, each being either an NPC or an actor. The user triggers these interactive calls with a *request* and the call returns a *reply*. Communication between actors consists solely of request and reply messages. In a multiplayer environment, where these messages are sent over a network, the user can serialize both the request and the reply messages as JSON strings.

The NPCEngine can function without its client-side section. A user can easily write a client application that interprets and sends JSON versions of the request and reply messages. However, the client side of the NPCEngine offers modules that let the user's application interpret replies coming from the server. The user can find this helpful as he or she decides how to process the data contents of the reply message coming from the server. One important module within the NPCEngine's client side is the text synthesis module, which lets the user synthesize localized dialog.

The final section of the NPCEngine is the token generator module. The user can utilize the token generator to determine all possible token variations contained in NPCs' message replies (see Section 3.2.5). It works very simply: The user provides a domain, either through a JSON definition or through the API, as we described above for the server-side domain definition. In a brute-force manner, the token generator will then apply all actions to all applicable model combinations, returning a list of all possible tokens. The user can employ this list to determine what kind of tokens the client application might encounter. The text synthesis module we mentioned above also uses the list.

3.2.2 Planning State

The purpose of the NPCEngine is to emulate human behavior. But the realism of an implementation depends on the variety and expressiveness of the NPCs' behavior. These properties in turn depend on the quality of the user-authored actions. Our goal is therefore to provide the user with a versatile and easy-to-use way of defining actions.

Two properties determine the expressiveness of an action's definition: How precisely we let the user determine the situation in which an NPC can perform the action and how precisely the user can describe the action itself. These qualities are the action's pre- and postcondition as well as the effect functions of an action. As we mentioned above, the user writes these three kinds of functions in ECMAScript. Using this programming language lets the user employ any algorithm that ECMAScript supports. For instance, a user could implement their own planning algorithm within a single action. However, the powerful technical properties of these functions are only effective if they have access to useful data. We therefore offer an API within the ECMAScript runtime that the user can access from within the action functions.

When using this API, we must always keep in mind the scope within which we are executing our functions. The NPCengine executes the effect function within the scope of the current knowledge state. However, as we apply the pre- and postcondition functions during the planning process, we in fact execute them within a copy of the current knowledge state – a copy representing a hypothetical future state. Our API can therefore also provide data from a hypothetical knowledge state, and not just from the NPC's current knowledge.

To provide data on an NPC's knowledge, our API returns a `KnowledgeObject`. Each `KnowledgeObject` is uniquely defined by its model and ID number. A `KnowledgeObject` also provides a set of helpful methods. Here are three examples:

- **unknownProperties()**
Returns an array of the properties that the NPC does not know about this `KnowledgeObject`.
- **learn(propertyName, value, source)**
Lets the NPC learn the value of a property of this `KnowledgeObject`. The `source` parameter is the `KnowledgeObject` of the NPC or player who provided the information.
- **getSource(propertyName)**
Returns the `KnowledgeObject` (of an NPC or a player) that is the source of the given property's value within this `KnowledgeObject`.

We plan to implement another method, `getCompetingBeliefs(propertyName)`, that would return beliefs with lower accuracy for a given property, but have yet to implement it.

Our API consists of three parts: A set of global objects describing the current call, a set of functions that lets the user access the NPC's knowledge, and possibly a set of user-defined global variables. First, let's look at the objects that provide information on the current call. These are the objects we define globally within the ECMAScript runtime:

- me** This is the `KnowledgeObject` that defines the NPC's knowledge about itself. The user must define whether the NPC has perfect knowledge about itself or not. By intentionally giving the NPC no knowledge about itself, we could emulate amnesia.
- you** (*only for interactive calls*)
This `KnowledgeObject` defines the NPC's knowledge about its current counterpart. The library automatically creates a new `KnowledgeObject` if the NPC is interacting with this counterpart for the first time.
- us** (*only for interactive calls*)
This is a `Relationship` object in which the NPC stores information on its relationship with the counterpart. Two NPCs will not share a `Relationship` object, but instead will each have a `Relationship` object for the other. Once again, the library automatically creates a new `Relationship` object if this is the first time that the NPC has met its counterpart. The object contains the following properties:

- `respect`
- `familiarity`
- `sympathy`
- `lastMeeting`

The first three properties are values between 0 and 1 that the user can employ to reflect an NPC's emotions towards its counterpart. We chose these three dimensions because we felt that they are typically independent of each other and are broader than “trust” or “love”. If a user needs more dimensions to represent a relationship, they can add more properties in a custom relationship object. The `lastMeeting` contains the timestamp of the most recent interactive action that the NPC performed with the counterpart.

conv (only for interactive calls)

This is a `Conversation` object in which the NPC stores information on the current interaction. The user can define a timeout value in seconds; if neither actor performs a call within that timeframe, their conversation will expire. The library will then automatically create a new `Conversation` object the next time the NPC interacts with the counterpart. The `Conversation` object contains the following properties:

- `state`
- `subject`
- `subjectProperty`
- `lastInteraction`

The `state` property is a string that we let the user define freely. The user can therefore create a state machine for an interaction by defining states such as *greeting* and *smallTalk*. The `subject` property is a pointer to a `KnowledgeObject` and the `subjectProperty` points to a `ModelAttribute`. This lets the user store the subject of the current conversation precisely. Finally, `lastInteraction` stores the timestamp of the most recent interactive call that the NPC got from the counterpart.

global As of this writing, this object contained only one property: `time`, which is the current timestamp. However, extending this object to contain more information on the storyworld would be easy.

States for interactive calls therefore have five global variables, whereas those of reflective calls have two. We kept variable names intentionally short to improve the readability of user-written functions. When we start a new planning routine, we first define the values of these global objects within the initial state. During the planning process, any pre- or postcondition functions can access and change any of these values. For instance, while planning we might consider the action *changeSubject* with the postcondition `conv.state = "talkAboutWeather"`. Executing this postcondition will affect the `conv` global object of the state.

The second part of our API comprises the functions that let the user access the NPC's knowledge. Our API offers ten functions for this purpose; here are the four most important ones:

- **`allObjects()`**
Returns all `KnowledgeObjects` that the NPC possesses.
- **`allModels()`**
Returns all `Model` objects about which the NPC can form an `KnowledgeObject`.
- **`getObject(reference)`**
Returns the `KnowledgeObjects` with the given model and ID.

- **getModel (name)**

Returns the `Model` object with the given name.

A `Model` object provides another set of methods. Here are the two most important ones:

- **getObject (id)**

Returns the `KnowledgeObjects` with the given ID.

- **createObject ()**

Returns a new `KnowledgeObjects` implementing that `Model`.

The third part of the API comprises user-defined variables. Within the domain, the user can define such variables in an action's definition. The values of these variables are therefore specific to one action instance. The action functions can access them as global variables within the ECMAScript scope. We discuss the definition of such variables in more detail in the following section.

3.2.3 Domain Definition

To define the domain of a storyworld, the user of the NPCEngine can define sets of models and actions. Our API lets the user create the domain by using method calls. But we also provide a domain loading module that can load a domain definition from JSON files. In this section, we focus on domain definition using JSON.

Given that the domain consists of lists of models and actions, we encourage the user to provide their domain in the form of multiple JSON files. This lets the user organize their domain in a tidy way.

Model definitions

First, let's look at model definitions in JSON. We loosely base our JSON notation for models on the notation for schemas published by the Open API Initiative. Listing 3.2 shows JSON code that defines three models: `Person`, `Villager` and `Ring`. The models each contain a set of properties, each of which has a name and a type. The NPCEngine supports the basic data types `boolean`, `integer`, `float`, `string` and `time` (a 64-bit integer). It also supports `scale`, a floating-point number limited to values between 0 and 1. For properties that can only assume a limited number of values, the NPCEngine also offers the `enum` type. This type requires that the property have an `enumerators` attribute, listing the available values. Finally, the NPCEngine lets properties have the type `reference`, which contains a model and an ID. Such a property is thus a pointer to either a `WorldObject` or a `KnowledgeObject`, depending on whether the property is part of the storyworld or of an NPC's knowledge.

A model can also extend an existing model. In Listing 3.2, the model `Villager` extends the model `Person`. An object instantiating `Villager` thus has the properties of `Villager` and of `Person`. A `Villager` NPC can also perform all the actions that a `Person` can.

Action definitions

The second purpose of domain definition is to allow the user to define actions. Related research has often used *A Behavior Language* (ABL) for this purpose. Mateas and Stern [MS02, MS04] introduced ABL, whose syntax is Java-like. For the NPCEngine, we ported this syntax to JSON and enhanced it with attributes necessary for our library. Listing 3.3 shows a portion of an example JSON file that defines two actions.

```

1  "models" : {
2
3      "Person" : {
4          "properties" : {
5              "gender" : {
6                  "type" : "enum",
7                  "enumerators" : [ "m", "f" ]
8              },
9              "name" : {
10                 "type" : "string"
11             },
12             "age" : {
13                 "type" : "int"
14             },
15             "ring" : {
16                 "type" : "reference"
17             }
18         },
19     },
20
21     "Villager" : {
22         "extends" : ["Person"],
23         "properties" : {
24             "job" : {
25                 "type" : "string"
26             }
27         }
28     },
29
30     "Ring" : {
31         "properties" : {
32             "size" : {
33                 "type" : "float"
34             }
35         }
36     }
37 }
38 }

```

Listing 3.2: Example JSON file that defines three models: Person, Villager and Ring.

Each action definition within a JSON domain file can have the following attributes:

me	This is the name of one of the domain's models. Any NPC whose <code>WorldObject</code> implements this model (or an extension of it) can perform this action.
you	If this attribute is present, the action is an interactive one; if the attribute is omitted, the action is reflective. To perform this action, the NPC's counterpart <code>WorldObject</code> must implement this model (or an extension of it).
precondition	Written as ECMAScript code. The NPC executes this code when determining whether it can perform this action in a given state. The resulting value must be either true or false.
postcondition	Written as ECMAScript code. The NPC executes this code when determining the expected effects of an action in a given state.

```

1 {
2   "actions" : [
3
4     {
5       "me" : "Person",
6       "precondition" : "me.tiredness > 0.9 && getObject(me.location).isIndoors",
7       "postcondition" : "me.tiredness = 0.0;",
8       "steps" : [
9         { "output" : "[imTired]" },
10        { "call" : "[goToSleep]" }
11      ]
12    }
13  ],
14
15  "actionGroups" : [
16
17    {
18      "me" : "Villager",
19      "precondition" : "us.familiarity > 0.5 && conv.state == 'smallTalk'",
20      "actions" : [
21
22        {
23          "you" : "Person",
24          "precondition" : "!you.isKnown('name')",
25          "postcondition" : "you.flagAsKnown('name');",
26          "steps" : [
27            { "output" : "[whatIsYourName?]" },
28            {
29              "input" : "string",
30              "reaction" : [
31                { "effect" : "you.learn('name', @input, you)" }
32              ]
33            }
34          ]
35        }
36      ],

```

Listing 3.3: First part of an example JSON file. This section defines two actions, one of which is part of an action group.

steps This attribute lists a set of commands that the NPC will process sequentially when performing the action.

Steps within Actions

The effect function that we discussed in previous sections only lets the user change the NPC's knowledge. So we need to give the user a way of also altering the storyworld to reflect such effects. We thus use the attribute *steps* instead of a single effect function in order to give the user a more elaborate structure in which to define an action's results.

The NPCEngine does not let the NPC influence the storyworld directly, this is the responsibility of the user application. When the user application calls an NPC to choose an action using the NPCEngine, the NPCEngine must therefore return a detailed description of the action the NPC chose, so that the user application can perform the action as intended. For this purpose, the NPCEngine provides the user with four types of steps:

3 NPCengine

<i>effect</i>	This step represents the effect function that we discussed in previous sections. The user must write this step as an ECMAScript code function. This function is executed within the scope of the NPC's current knowledge and is not returned to the user application. Hence any change to the knowledge will have no effect on the state of the storyworld.
<i>call</i>	This is a <i>token</i> , i.e. a string with which the NPCengine tells the user application what action the NPC chose. The user should employ a <code>call</code> step to trigger some NPC behavior within the user application. This behavior can be independent of the NPC's counterpart, if one is present, so we don't need to pass the token to the counterpart.
<i>output</i>	An <code>output</code> step is similar to a <code>call</code> step and also consists of a token that the NPCengine sends to the user application. However, its purpose is interaction. The <code>output</code> step should lead the NPC to either say or do something to its counterpart within the user application. The user application should provide the NPC's counterpart with the <code>output</code> step's token.
<i>input</i>	In the list of steps withing an <code>steps</code> attribute, only the last step can be an <code>input</code> step. This step type is the counterpart to the <code>output</code> step type. Instead of sending a token to the counterpart, an <code>input</code> step informs the counterpart that the NPC is expecting a value or a token. In other words, the NPC expects the counterpart to say or do something. We discuss how <code>input</code> steps work in more detail below.

Tokens

Tokens are central to our API. A token is a string that refers unambiguously to one statement or behavior of an actor. To differentiate tokens from other code, we require that square brackets surround tokens' names. We use tokens for all communication between NPCs or between an NPC and a player.

The NPCengine is not responsible for any ambiguous interactions between actors in the storyworld. If the user application allows for ambiguous behavior, it must interpret that behavior itself and call the NPCengine by means of an unambiguous token. In particular, the NPCengine does not support `input` or `output` in natural language. If a user application uses natural language, it must employ some approach from the field of conversational artificial intelligence to translate between natural language and the tokens.

Until now, we have been discussing values that an NPC returns to a `call`. We must, however, also make sure that the requests the user application passes to the NPC are not ambiguous. This is the purpose of an `input` step in an action's definition. The `input` step's declaration lets the user specify which values or tokens the user application can send to the NPC in response to an action. By limiting the number of responses, we make sure that the NPC can react appropriately. An `input` step thus specifies which type the user application's response value can have. The `input` step supports the same types as the model properties: `boolean`, `integer`, `float`, `string`, `time`, `scale`, `enum` and `reference`. An `input` step should also have a `reaction` attribute. Like the `steps` attribute of an action, the `reaction` attribute consists of a list of steps. The NPCengine will perform the steps of the `reaction` attribute when the counterpart responds to an `input` request. To prevent the user from nesting `input` steps, the `reaction` attribute cannot contain any additional `input` steps.

Action Groups

As described in Section 3.1.2, we want to support inheritance in both models and actions. However, unlike models, actions do typically not share a full set of attributes with any other model. We therefore introduce *action groups* as a mechanism to group several actions that share some common attributes. Any action that is part of an action group will inherit all the attributes the user has defined for the action group. If both the action group and the action define a pre- or postcondition, the action will execute both definitions sequentially. Individual actions can override the action group's values for the other attributes by redefining them.

Simple Example Actions

Listing 3.3 shows two simple actions: The first one allows an NPC with a `WorldObject` that implements the model `Person` to go to sleep. Given that the action has no `you` attribute, it is a reflection action that the NPC performs without a counterpart. The `precondition` attribute states that the NPC must be very tired and indoors to perform this action. The `postcondition` states that the expected outcome of the action is that the NPC will no longer be tired – though the actual outcome of the action may be different. When performing the action, an NPC processes two steps that the NPCEngine returns to the user application: an `output` step that will make the NPC say that it is tired and a `call` step triggering a behavior by means of the token `[goToSleep]`.

The second example action is part of an action group. The `me` attribute of the action group states that only a `Villager` NPC can perform actions that the action group contains. Also, the action group's precondition states that the NPC can perform these actions only if the NPC is relatively familiar with its counterpart and that both are currently making small talk. In this case, the attributes of this action group will apply to all the actions it contains. The NPCEngine will execute the precondition of the second action after it executed the precondition of the action group.

An NPC can perform the second action to ask its counterpart for its name. because this is an interactive action, it needs the attribute `you`, defined as any actor implementing `Person`. The precondition states that an NPC can only perform the action if it doesn't know the counterparts' name; the postcondition states that the NPC expects to know that name after it performs the action. The action comprises two steps: First, the NPC outputs a question, asking for the counterpart's name. Second, the NPC requests an input of type `string` from the counterpart. This `input` step contains a `reaction` attribute defining the steps that the NPC will process, if the counterpart responds. In this case, there is one `effect` step, causing the NPC to add the counterpart's name to its knowledge.

Script in Tokens

We can observe that the actions that Listing 3.3 presents are not very powerful. For instance, the first action definition covers a very useful task – at least in a world where NPCs have to sleep. But given that the NPCEngine will always return the same two tokens, each NPC will perform the action the same way every time. Though the user application could introduce some variation, we make the definition more powerful by allowing the user to insert ECMAScript in the tokens. Before sending the token to the user application, the NPCEngine will evaluate any part of a token appearing in curly brackets. For instance, if a user wants to make the NPC's behavior for the first action of Listing 3.3 more appropriate to the situation, they could replace the `output` step with the one in Listing 3.4.

Input Step Options

Another shortcoming appears, when we consider that the response of a counterpart to most interactive actions won't be a single value. We therefore offer another type for input steps: `options`. An input step with the type `options` requires an additional attribute, also called `options`. The `options` attribute lists all the ways in which the counterpart can respond to the action. Each option requires a `token` attribute defining the token with which the counterpart must respond and a `reaction` attribute listing the steps that the NPC will perform should the counterpart choose that option.

Listing 3.5 shows a continuation of the JSON file from Listing 3.3 that defines one additional action. An NPC performing this action will ask its counterpart for a gold coin. The second step of the action's definition is a `input` step with type `options`. The counterpart can choose between two tokens: `[no]` and `[yes]`. If the counterpart responds with one of the two tokens, the NPC will perform the steps in the corresponding `reaction` attribute. If the counterpart does not respond or responds with a different token, the NPC will perform neither of the reactions.

User-Defined Variables

A third limitation to the domain's action definitions – as we've described them until now – is very obvious in the second action of Listing 3.3: Using this action only lets an NPC obtain the value of one property for one object. Let's consider a very inquisitive NPC who wants to know everything about the world. The user would have to define an action for every property of every object in the storyworld, which is obviously impractical.

We therefore introduce user-defined variables, which are the most powerful features of the NPCEngine that the user can employ to create expressive action definitions. Within an action definition, the user can insert the attribute `variables`, which defines an array of variables. Each variable definition can have the following attributes:

<i>name</i>	This mandatory attribute defines the name of the variable as a string. The NPCEngine uses this name to declare a global variable within the ECMAScript scope of each action instance of this action. The post-condition, effect and token functions of this action will all have access to this variable. However, the precondition cannot contain any variables, as the NPCEngine assigns values to the variables after evaluating the precondition. By defining an appropriate precondition, the user can thus avoid situations where the NPCEngine cannot assign values to the variables.
<i>definition</i>	This attribute is also mandatory and the user must define it in the form of ECMAScript code. When the NPCEngine executes it, the code must

```
9 { "output" : "[!you ? 'imTired' :  
  (conv.state !== 'discussion' ? 'imGoingToBed' : 'sorryToInterrupt') ]]" },
```

Listing 3.4: Alternative `output` step for the first action in Listing 3.3. If the NPC is alone, it will just state that it is tired. If the NPC is interacting with a counterpart, but not having a discussion, the NPC will mention to the counterpart that it is going to bed. Or else, if it is having a discussion with a counterpart, it will excuse itself.

```

37     {
38         "you" : "Villager",
39         "postcondition" : "me.gold += 1;"
40         "steps" : [
41             { "output" : "[canIHaveMoney?]" },
42             {
43                 "input" : "options",
44                 "options" : [
45                     {
46                         "token" : "[no]",
47                         "reaction" : [
48                             { "output" : "[noProblem]" }
49                         ]
50                     },
51                     {
52                         "token" : "[yes]",
53                         "reaction" : [
54                             { "output" : "[thankYou]" },
55                             { "call" : "[takeMoney]" },
56                             { "effect" : "me.gold += 1;" }
57                         ]
58                     }
59                 ]
60             }
61         ]
62     }

```

Listing 3.5: Continuation of the example JSON file from Listing 3.3. This action lets the NPC ask its counterpart for one gold coin. The NPC expects the counterpart to either accept or decline. The `input` step defines this choice by using the `options` type.

return an ECMAScript array. The elements of this array represent all values that the variable can assume for the given state. If the variable can only assume one value for any given state, the user can wrap the declaration in square brackets to create a ECMAScript array with a single value.

condition

This attribute serves as an additional precondition to the action. However, unlike the action's own precondition, it has access to the `this` variable and all previously defined variables of this action instance.

tokenGeneratorOptions The token generator uses this attribute (see Section 3.2.5).

The central feature of the `definition` function of a variable is that it returns an array of values. This lets us treat each value in the array as a candidate assignment of the variable and use our planning algorithm to determine which value is the most promising in order to reach the NPC's goal. Assigning values to variables involves creating a copy of the action instance for every potential value and assigning one of the values to each copy. In this way, each possible value corresponds to a new action that the NPC can consider while planning.

Although variables let the user create much more expressive actions more easily, they also greatly increase the branching factor in the planning tree. For instance, if we have three applicable actions without variables, our planning process will have a branching factor of three. If we have three actions, however, each with two variables that can each take any of ten values, we get a branching factor of 300. For this reason, the NPCengine lets the user set a constant that defines the maximum number of action

instances that the planning process can create on the basis of a single variable assignment. But if the user sets this threshold one or more magnitudes lower than the actual branching factor, the planning algorithm will explore too small a section of the search space and thus fail to yield a sensible result. A user should therefore always consider the tradeoff between expressiveness and the resulting branching factor when defining actions.

When using variables, we would often like to select objects of different models or a selection of specific properties from a model. However, the methods listed above only allow the user to select either one or all models, objects or properties. Of course, the user can create their own selection process within the `definition` function for a variable. Such a function may be costly, however, as it would involve looping over a potentially large array of objects. When selecting certain models, we could consider using inheritance within the domain. For example, we could select all models that inherit from one model. This is impractical, however, as the inheritance within the domain may not correspond to the grouping we want to achieve when assigning values to variables.

Hence we introduce *variable templates* to simplify the selection of certain models, objects and properties. Within the model definition, we can give models and properties the attribute `variableTemplates`. This attribute consists of a list of names of variable templates. Listing 3.6 shows an abbreviated model definition using the `variableTemplates` attribute.

```

1 "models" : {
2
3   "Apple" : {
4     "variableTemplates" : [ "edible", "throwable" ],
5     "properties" : { ... }
6   },
7
8   "Soup" : {
9     "variableTemplates" : [ "edible" ],
10    "properties" : { ... }
11  },
12
13  "Stone" : {
14    "variableTemplates" : [ "throwable" ],
15    "properties" : { ... }
16  }
17
18 }
```

Listing 3.6: Abbreviated JSON definition of three models: Apple, Soup and Stone. Each model uses the `variableTemplate` attribute to define which of the two variable templates, `edible` and `throwable` the model supports.

These definitions of variable templates let the user define which templates an object must support in order for the NPCengine to assign that object to a variable. To do so, the user can employ the following methods within the ECMAScript scope of a planning state:

- **`allModelsWithTemplate(templateName)`**
Returns all `Model` objects that support the given template and about which the NPC can form knowledge.
- **`allObjectsWithTemplate(templateName)`**
Returns all `KnowledgeObject` instances that the NPC knows of and whose models support the given template.

Similarly each Model object offers the following methods:

- **allPropertiesWithTemplate(templateName)**
Returns all of the model's property names that support the given template.
- **isTemplateApplicable(templateName)**
Returns true if the model supports the given template.

Elaborate Example Action

By combining all the abilities of the NPCengine we have mentioned until now, we can create very powerful actions. For instance, by performing the action that Listing 3.6 shows, an NPC could ask for the value of any property of any object in the storyworld – provided that the NPC already knows the object and that the property supports the variable template `learnable`.

```

1      {
2          "variables": [
3              {
4                  "name" : "it",
5                  "definition" : "getObjects()",
6              },
7              {
8                  "name" : "prop",
9                  "definition" : "it.unknownProperties()",
10                 "condition" : "prop.isTemplateApplicable('learnable')",
11             }
12         ],
13         "postcondition" : "it.flagAsKnown(prop.name, you);",
14         "steps" : [
15             {
16                 "output" : "[letsTalkAbout->{it._type}.{it._id}]"
17             },
18             {
19                 "output" : "[whatIs->{it._type}.{prop.name}?]"
20             },
21             {
22                 "input" : "{prop.type}",
23                 "optionsFromProperty" : "prop",
24                 "optionsTokenFormat" : "[itIs->{it._type}.{prop.name}=@option]",
25                 "reaction" : [
26                     { "effect" : "it.learn(prop.name, @input, you);" }
27                 ]
28             }
29         ]
30     },

```

Listing 3.7: Example of a very expressive action. An NPC performing this action can ask about any learnable property of any object that it knows.

We assume that the action in Listing 3.6 is part of an action group that defines its `me`, `you` and `precondition` attributes – though this action doesn't necessarily need a precondition. Next, we define two variables:

1. We assign all `KnowledgeObjects` that the NPC knows of to the variable `it`. We therefore create as many action instances as the NPC has `KnowledgeObjects`. In each action instance, the NPCengine assigns `it` a different `KnowledgeObject`.

2. We assign to the variable `prop` all the properties that the ECMAScript expression `it.unknownProperties()` returns. Given that in each action instance the variable `it` is a different `KnowledgeObject`, the method `unknownProperties()` will in turn yield a different set of possible values for `prop`. The NPC might know the values of all properties of some `KnowledgeObjects`, in which case the method returns an empty array.

Now we copy each action instance for each potential value of `prop`. If the array of values is empty, the NPCEngine discards the action instance. We now have a number of action instances equal to the total number of unknown properties. Each action instance is unique in terms of the assignment of the variables `it` and `prop`.

Finally, the NPCEngine executes the ECMAScript function that we defined in the `condition` attribute. For each action instance, the NPCEngine evaluates `prop.TemplateApplicable('learnable')`. If the function returns the value `false`, the NPCEngine discards the action instance. This will leave us with a set of action instances, of which each defines the value of `prop` as a property that supports `learnable`.

In the postcondition, we state that the NPC can expect to know the unknown value after performing the action. The NPCEngine will consider this while planning. If the planning process chooses one of the action instances to be the most promising one, the NPC will perform the following steps:

1. The NPC outputs a token informing the counterpart that it wants to talk about a specific `KnowledgeObject`. The NPCEngine executes the parts of the tokens appearing between curly brackets. This might lead the NPC to output `[letsTalkAbout->Person.15]`.
2. The NPC outputs a token asking the counterpart for the value of a specific property, for instance `[whatIs->Person.hairColor?]`.
3. The NPC requests a value or a token from the counterpart. By using ECMAScript in the `input` attribute we can have the NPCEngine dynamically assign the type. If the property for which the action is requesting the value is of type `enum`, we can have the NPCEngine also pass the options from the enumerators specified in the model definition. This might yield option tokens such as `[itIt->Person.hairColor=blond]` and `[itIt->Person.hairColor=black]`. Finally, if the counterpart answers the action's input request, `reaction` attribute will lead the NPCEngine to store the input in the NPC's knowledge, setting the counterpart as the source.

User-Defined Functions

The NPCEngine's domain definition supports another minor feature: user-defined global functions. The user can define a set of functions in the domain's JSON files. These functions will then be available to the user within the ECMAScript scope. This is especially useful if a user regularly employs a set of commands and would like to implement them as a utility function. Listing 3.8 shows such a function definition.

These user-defined global functions may also be useful for actions whose ECMAScript functions are exceptionally long. The user can remove those functions from the action definition and define them as global functions, improving the action definition's readability.

3.2.4 Storyworld Initialization

The API of the NPCEngine to initialize the storyworld consists of the following elements: Loading the domain and creating a world (see Listing 3.9), defining world objects and NPCs (see Listing 3.10) and finally creating players (see Listing 3.11)

```

1 {
2   "functions" : {
3
4     "distance" : [
5       "function(posA, posB) {",
6         "    var deltaX = posA.x - posB.x;",
7         "    var deltaY = posA.y - posB.y;",
8         "    return Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));",
9       "}"
10    ]
11  }
12 }
13 }

```

Listing 3.8: User-defined global utility function to determine the distance between two coordinates.

```

1 // Load domain from JSON
2 DomainLoader domainLoader = new DomainLoader();
3 domainLoader.AddJSON(GetFile("Domain/Models.json"));
4 domainLoader.AddJSON(GetFile("Domain/Actions.json"));
5 domainLoader.AddJSON(GetFile("Domain/Functions.json"));
6 Domain domain = domainLoader.CreateDomain();
7
8 // Create a new data source (which is a JSON Database)
9 IDataSource dataSource = new JsonDatabase("Database/jsonDatabase.json");
10
11 // Create a new NPCEngine world
12 WorldConfig worldConfig = new WorldConfig(
13     180000, // timeout of conversations
14     180000 // timeout of reactions
15 );
16 World world = new World(domain, dataSource, worldConfig);

```

Listing 3.9: Example code to create a World with the NPCEngine.

3.2.5 Token Generator

Due to the expressiveness of the actions' definitions, it may be hard for a user of the NPCEngine to determine all possible tokens that the NPCEngine might return. We therefore implemented the token generator module. Similar to the storyworld, it requires a domain definition. It then applies all possible value combinations to generate a JSON list of all tokens that the NPCEngine can produce.

Given that the user can employ the token generator during development of the application, the token generator won't have access to a real state. And given that the user can use variables within tokens, the token generator won't be able to determine which values the variables can assume. The user can therefore employ the `tokenGeneratorOptions` attribute of the variable definition to pass an array of values to the token generator that represent all possible values that the NPCEngine may encounter for that variable.

3.3 Results

The goal of the first part of our implementation was to design and create a library able to synthesize NPC behavior that fits the NPC's observable storyworld. We also wanted the library's implementation to sup-

3 NPCEngine

```
1 // Create a world object that implements model "Villager"
2 WorldObject aliceWO = world.CreateWorldObject("Villager");
3 aliceWO["gender"] = "f";
4 aliceWO["name"] = "Alice";
5 aliceWO["gold"] = 7;
6
7 // Create an NPC with previously-created world object
8 AgentConfig agentConfig = new AgentConfig(
9     10, // max number of sequential actions per call
10     AgentConfig.PlanningStrategyConfig.MCTS, // planning algorithm
11     100, // max action instances per variable
12     100, // max planning iterations
13     5, // max planning depth
14     "conv.state = 'greeting';"
15 );
16 Agent alice = world.InitAgent(aliceWO, agentConfig);
17
18 // Allow the agent to have knowledge about up to 50 world objects of type "Person"
19 alice.AllowMemoryOf("Person", 50);
20
21 // Set the agent's goal
22 agent.SetGoal(@"
23 (function() {
24     return me.gold;
25 })()", 1.0);
```

Listing 3.10: Example code to create a world object and an NPC.

```
1 // Create new world object with model "Hero"
2 WorldObject bobWO = this.world.CreateWorldObject("Hero");
3 bobWO["name"] = "Bob";
4
5 // Initialize human with the previously created world object
6 Human bob = world.InitHuman(bobWO);
7
8 // Make agent gain knowledge about the human's name
9 bob.GainKnowledgeOf(bobWO, hero, new List<string>() { "name" });
```

Listing 3.11: Example code to create a world object and a player.

port multiplayer games. We identified three properties of current video games that can impair suspension of disbelief in some players: omniscient NPCs, oblivious NPCs and an inconsistent storyworld.

We have identified a set of properties that let us assess the qualities of our implementation. The realism of an NPC's behavior depends on two properties: First, the NPC's behavior must be comparable to human behavior. We discuss this in Section 3.3.1. Second, the behavior must be appropriate to the situation. In other words, an NPC's reasoning about a particular action should be sound. We examine this in Section 3.3.2. Section 3.3.3 presents the NPCEngine's multiplayer capability.

One core requirement for any software library is its practicality. We discuss the NPCEngine's technical properties in Section 3.3.4 and its performance in Section 3.3.5.

We conclude this chapter with a discussion in Section 3.3.6.

3.3.1 Expressiveness

The degree to which an NPC's behavior appears human strongly depends on how the game visually and auditorily represents it. Given that the user application is responsible for animation and dialog, the NPCengine has an only indirect influence on these aspects of the behavior's realism.

To appear human, however, an NPC's behavior must also be varied and expressive. Because the NPCengine determines which action an NPC performs, it strongly influences these properties. Defining a varied set of actions is the responsibility of the library's user, but the NPCengine can support the user by letting him or her define highly expressive actions easily.

As we implemented the NPCengine's definition scheme for actions, we therefore focused on usability and expressiveness. First, for the action definitions we've used JSON, a popular format that is easy to read. Also, many external applications support the JSON format. Second, we've based our notation for actions on the existing ABL notation for agent behavior, making it easily accessible to users that already have experience with similar applications. Third, we've augmented the existing ABL standard with a set of features that render the notation significantly more powerful: inheritance, tokens and the user's ability to employ ECMAScript in most attributes.

The NPCengine also lets the user application access all values of the storyworld and of the NPCs' knowledge at any time. A game using the NPCengine can therefore implement a drama manager that continuously alters the storyworld without interfering with the NPCengine.

3.3.2 Action Selection

The suitability of an NPC's behavior is central to its believability. Choosing the most appropriate action for a given situation is therefore the NPCengine's core purpose.

The NPCengine thus treats NPCs as *reasoning* agents, i.e. agents who can independently determine the best next action based on a set of facts. We also implemented them as *planning* agents. This means that an NPC looks several steps ahead and considers the possible effect of a sequence of actions. By implementing the MCTS planning algorithm, we have enabled the NPCengine to efficiently search among a potentially unlimited number of action combinations.

An NPC's decisions are truly believable, however, when it bases them on its incomplete knowledge of the world. We therefore let NPCs build knowledge on the storyworld. Because the NPC bases its planning on its limited knowledge instead of the objective storyworld, its behavior is believably fallible.

3.3.3 Multiplayer Application

To our knowledge, no previous research has combined agents who plan based on their knowledge, with multiplayer interaction. This was thus an important goal for the NPCengine.

The NPCs of the NPCengine can hence differentiate between individual players. Every NPC stores information on its relationship with each player it ever met. NPCs can track several simultaneous interactions, remembering the current subject of each interaction. Our design of the NPCengine treats each call for planning as an instantaneous, atomic event. An NPC can thus have several simultaneous conversations that all can access the same knowledge. If an NPC receives information in one conversation relevant to another, the NPC can immediately use the information in the other conversation. If a user finds the NPC's ability to have multiple simultaneous conversations unrealistic, he or she can introduce a lock that blocks another player from interacting with the NPC if the NPC is already interacting with one.

Two other advantages of the NPCEngine's design for multiplayer applications are the following: First, given that the NPCEngine performs planning on copies of the NPC's knowledge, we can envision a multithreaded version of the NPCEngine that could process several calls simultaneously. Second, to prevent the volume of the NPCs' knowledge from growing exponentially with the size of the storyworld, we let the user limit the number of objects an NPC can know. This also emulates the limits of human memory.

One shortcoming of the NPC engine is that it can only deal with bilateral interactions. By using appropriate models, a user can create tuples that emulate multilateral interactions, but they will not support the expressiveness possible in bilateral interactions.

3.3.4 Technical Practicability

The NPCEngine has several technical requirements. On one hand, the library should be able to run in the C# environment of the Unity game engine, and on the other, the library must also be able to run on an independent server. Throughout development, we continuously tested the NPCEngine both within the Unity game engine and within a separate application. The NPCEngine functions flawlessly in both environments. Any application that can interface .NET bytecode can use the NPCEngine, given that C# can be compiled to .NET bytecode.

The NPCEngine's C# and ECMAScript APIs are straightforward but also powerful. The NPCEngine employs the popular standards JSON and ECMAScript, so developers familiar with these technologies will find the NPCEngine easy to use. These standards also ease code transfer from other sources to the NPCEngine.

Finally, because the NPCEngine uses JSON for most interfaces with the user application, a user can easily implement a library extension to the NPCEngine. A possible extension that we've already implemented is a custom client side for the NPCEngine. Another sensible extension that we envision is a visual editor for domain definition. A user could easily implement both extensions without access to the NPCEngine's source code.

3.3.5 Performance

The practicability of the NPCEngine also depends on its performance in real-world use. The NPCEngine should be able to answer a user application call within a time frame that is acceptable for a video game.

We conducted a performance test within a limited storyworld environment. The average processing time for an interactive call was 176 milliseconds. Table 3.2 lists the results and parameters that we used for the performance test. A more complex storyworld would allow a performance test that better emulates real-world application.

Although not a decisive attribute, the NPCEngine's search precision may be important for some applications. Given that the NPCEngine's planning process uses the MCTS search algorithm, the NPCEngine shares the precision characteristic for MCTS. If the optimal action can be found in a finite number of search iterations, the NPCEngine will inevitably find it if it performs at least the necessary number of iterations. Below that threshold, the error will logarithmically converge towards zero with the number of iterations.

Average processing time	176 milliseconds
Shortest processing time	103 milliseconds
Longest processing time	439 milliseconds
Sample size	50 interactive calls
Branching factor during planning	9
Limit to number of actions performed per call	100
Limit to number of instances per variable	100
MCTS search budget	20 iterations
Limit to search depth	5
Number of NPCs	54
Hardware	Laptop using Intel i7 processor with 2.6 GHz

Table 3.2: Results and conditions for our performance test. Given the limited storyworld, these results don't represent those of a real-world application.

3.3.6 Discussion

The NPCEngine satisfies all requirements of the initial problem definition. But the results a user can attain depend on the user application employing the NPCEngine. The NPCEngine thus offers an easy and versatile way to formulate actions; this encourages the user to create expressive behavior.

By applying the MCTS search algorithm, the NPCEngine can reliably find the most promising action for the NPC. To prevent the computational complexity of the planning task from growing exponentially with the storyworld, MCTS also lets the user define a planning budget. However, we have not yet tested the NPCEngine's performance in a real-world setting. Given the NPCEngine's performance in an artificially small environment, we expect the NPCEngine to have performance issues in real-world settings.

We believe that the NPCEngine offers an approach that can solve all three of the issues that we identified in current video games. The first two are oblivious and omniscient NPC behavior. Such behavior results from an NPC's actions not reflecting what the NPC should know about the storyworld. In contrast, NPCs of the NPCEngine chose their actions based on their knowledge only. Having an oblivious or omniscient NPC in the NPCEngine would require that the user application to intentionally falsify the NPC's knowledge.

The third issue is the inconsistency typical in modern video games, which often result from game designers' intentional decisions to limit player agency. The approach that we've implemented with the NPCEngine no longer requires game designers to limit their player's agency.

4 Mobile Game

In this chapter we present the second practical implementation of the thesis: the mobile game. The goal of the game is to serve as technology demonstrator for the NCPengine.

In Section 4.1 we discuss the design of the mobile game. Section 4.2 presents and discusses the results of our mobile game implementation.

4.1 Design

Our mobile game has several requirements. First it must be able to easily represent the output of the NCPengine. We therefore chose to implement the interaction between the NPCs and the player as a chat. On one hand this allows us to easily represent the NCPengine's tokens, as we can translate each token into a natural language message. Also, by using a communication method that is familiar to most players, the players will be able to interpret emotions into the natural language message, similar to the way they do during real chat conversations. Also, it is easier for a user to design expressive emotionality through text than through animations.

We chose an espionage setting for the game as it is ideally suited for a variety of game mechanics that involve limited knowledge, planning and large amount of interaction between different characters.

The mobile game must also support two additional technologies: Augmented reality and a location-based mechanic.

4.2 Results

The primary goal of our mobile game was to allow us to test and demonstrate the NCPengine. The game achieves its purpose: it lets us show that a user can employ the NCPengine to control NPCs in a game and that each of these NPCs can store knowledge and plan actions based on its knowledge.

Figure 4.1 shows the decoy mini-game that appears each time the player opens our game. Figure 4.2 shows the "radar" map view on which the player can see his or her location and the position of "signals" that the player can use to connect to a room of NPCs. Figures 4.3 and 4.4 demonstrate the augmented reality feature that the player must use to see a room. Finally, Figure 4.5 shows the view that the player uses to chat with the NPCs.

Table 4.1 lists the steps of an interaction that the mobile game produced by using the NCPengine. It demonstrates that the NPC considers actions' preconditions and picks one based on its goal. It also shows that the NCPengine stores beliefs on storyworld objects and assigns them a source and an accuracy.

The secondary goal of the mobile game was to test game mechanics only possible with the NCPengine. Due to time constraints, we have not yet used the NCPengine to experiment with such mechanics. But the mobile game's functionality lets us implement new mechanics in the future. The espionage setting

that we chose for the game is very suitable for a variety of game mechanics involving limited knowledge and planning.

The mobile game also demonstrates that the NPCEngine supports multiplayer applications, since several players using separate devices can interact with an NPC at the same time. It also shows that the NPCEngine's networking features work as intended.

As part of the problem description, our mobile game had to implement two technologies independent of the NPCEngine: augmented reality and location-based mechanics. In the mobile game, the augmented reality feature using Vuforia works flawlessly. The tracking is stable enough to let a player focus on a small portion of a tracking tag. We've also implemented a location-based functionality that displays a visually appealing map based on the MapBox platform and limits the player's interaction options based on his or her location.

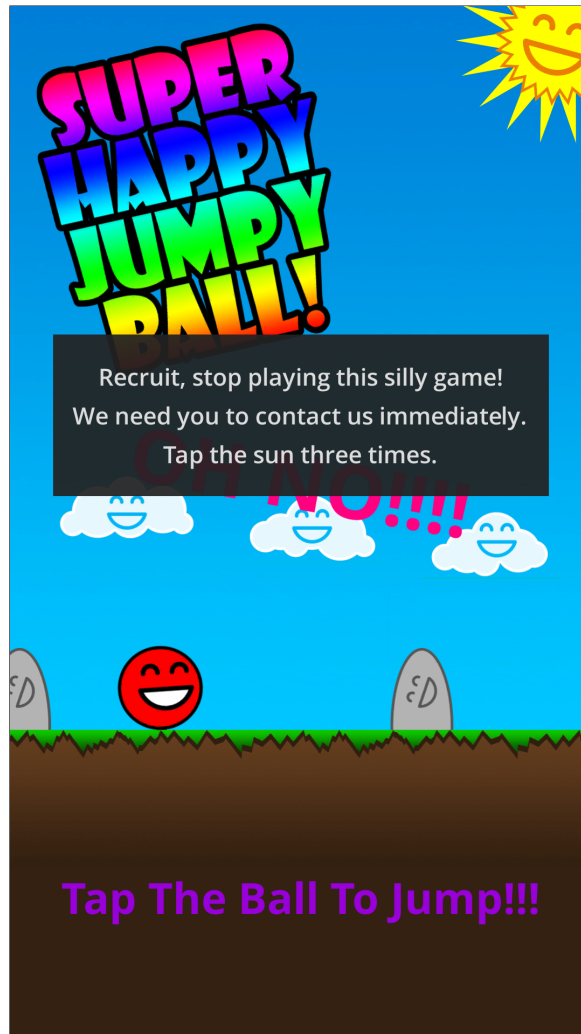


Figure 4.1: Our decoy mini-game called Super Happy Jumpy Ball. Its purpose is to increase the player's immersion by implying that he or she has access to a secret application hidden within the Super Happy Jumpy Ball game. We also use the time the player spends in in the mini-game to initialize the augmented reality feature, load the map, and to let the device's GPS location settle. In the mini-game, the scene continuously scrolls to the left and the player must try to jump over the stones by tapping the red ball. The game has neither a win nor a lose condition. The sprites, colors and animations are intentionally overly cheerful and annoying. Tapping the sun in the top right corner three times makes the mini-game's sprites fall off the screen, revealing the map view shown in Figure 4.2.



Figure 4.2: The mobile game’s map view. The red dot shows the player’s location. The corner protruding from the red dot shows the azimuth towards which the device is pointing, based on the device’s internal compass. The white antenna icons show the position of “signals” that the player can use to connect to a room. Each of the nine rooms has a unique icon visible above the antenna. When the player taps on an antenna icon, a white circle indicates the signal’s range. The name and “strength” of the icon also appear in the bottom bar. The signal’s strength depends solely on the player’s distance from the midpoint of the signal and has no gameplay implication. By tapping the “connect” button, the player can launch the augmented reality view shown in Figure 4.3. The button labelled “Log” in the top right opens a history of previous interactions.



Figure 4.3: Demonstration of the mobile game’s augmented reality feature, which uses the Vuforia framework. We provide the player with an A6-sized cardboard booklet containing three augmented reality markers, one for each of the factions of the game. When the Vuforia framework detects the correct tag, it displays the corresponding room. Tapping on one of the six NPC figures or name tags makes the camera zoom in on that character and display the chat interface shown in Figure 4.5

By providing the player with a physical booklet, we hope to create the feeling that he or she owns a secret holographic telecommunication device. When the game first detects a tag we play a short static noise animation that looks like a holographic display connecting to an analog signal. Our game calculates the angle between the NPCs’ eyes and the camera, allowing for an occasional animation that makes an NPC appear to glance at the player. Each of the three factions in the game have their own room texture – this is the texture of the “Global Freedom Agency”.



Figure 4.4: Demonstration of the mobile game's augmented reality feature displaying two different rooms from various angles. The left screenshot shows the room texture of the faction called "The Enlightened Angels of Peace" while the right one shows the texture of the "World Liberation Front".

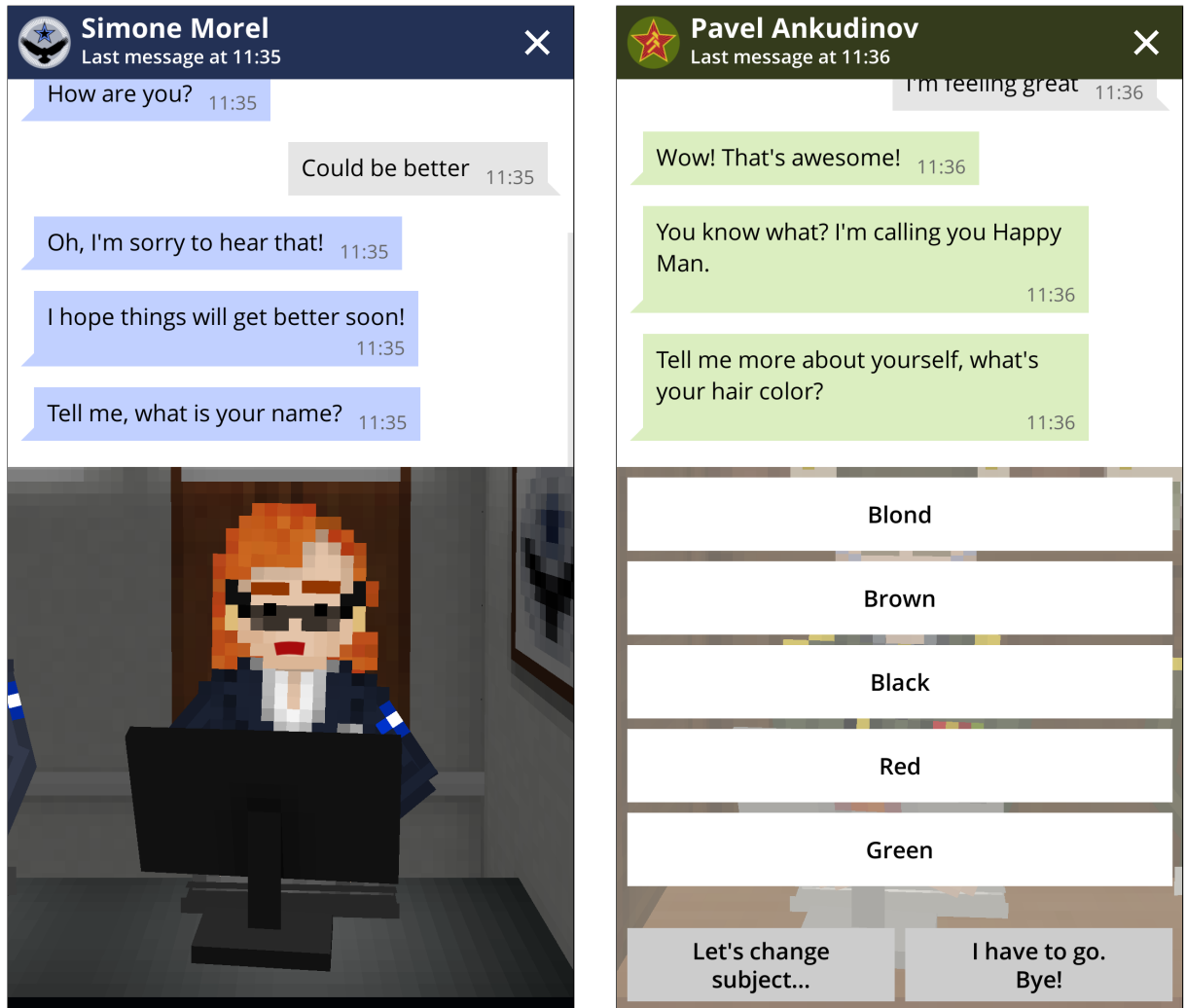


Figure 4.5: The chat view that the player uses to interact with the NPCs. The top half displays the chat’s history. Its design mirrors that of many chat applications for mobile devices. We hope this gives the player a sense of familiarity that encourages him or her to suspend disbelief. Each time the NPC performs an action that includes an *input* step, the available options appear in the bottom half of the screen. In the future, we intend to let the player interrupt a conversation by using the buttons “Let’s change subject...” and “I have to go. Bye!” at the bottom of the screen. However, we have not yet implemented this functionality.

Me	Hi!
Jules Dupont	Hey!
Jules Dupont	My name is Jules Dupont. :)
Jules Dupont	How are you?
Me	Terrible
Jules Dupont	Oh, I'm sorry to hear that!
Jules Dupont	I hope things will get better soon!
Jules Dupont	Tell me, what is your name?
Me	Henry
Jules Dupont	Tell me more about yourself, what's your hair color?
Me	Brown
Jules Dupont	What would you like to know?
Me	How am I feeling?
Jules Dupont	You told me that you are feeling terrible. I'm pretty sure about that.
Jules Dupont	What would you like to know?
Me	How do you know my name?
Jules Dupont	You told me that you are called Henry. I'm pretty sure about that.
Jules Dupont	What would you like to know?
Me	Actually, my mood changed.
Jules Dupont	Well, how are you feeling now?
Me	I'm feeling great.
Jules Dupont	Wow! That's awesome!
Jules Dupont	You know what? I'm calling you Happy Man.
Jules Dupont	What would you like to know?
Me	How do you know my name?
Jules Dupont	I decided that you are called Happy Man. That is how you are called.
Jules Dupont	What would you like to know?

Table 4.1: Chat log of the interaction between a player and an NPC in our mobile game. The agent's evaluation function yields a high value for states in which the NPC knows the player's name. But the NPC cannot ask for its counterpart's name before introducing itself and asking about the player's well-being. As the log shows, the NPC thus first pick those actions before asking for the player's name. The NPC also can pick the action that asks for the player's hair color. Given that this is not part of its goal, it only picks that action once it knows the player's name.

After an initial phase of the conversation, the NPC keeps returning to the question "What would you like to know?", allowing us to test its knowledge. The NPC's answers contain the value and the source of the belief, while the statements "I'm pretty sure about that." and "That is how you are called." indicate the NPC's certainty about a specific belief.

5 Conclusion

The goal of this thesis was to design a system able to synthesize the behavior of non-player characters (NPCs) that fit the NPC's observable storyworld. In doing so, we tried to address several common problems with storytelling in modern video games. On one hand, players expect an ever-increasing level of player agency. However, NPCs do not provide the appropriate expressive behavior, as they are typically implemented as state machines.

On the other hand, game designers deliberately limit player agency. Given the central role of narrative in many games, designers put much effort in authoring compelling stories. To give a player more influence on the storyworld, game designers must create more variations of the story. But faced with the impossibility of designing every possible sequence of events, designers resort to approaches that make their storyworld unrealistic or inconsistent.

In this thesis, we first presented a design for a procedural storytelling library that enables both a rich storyworld and high player agency. The library's central purpose is to control NPCs within a storyworld. We base our approach on the work of Ryan et al. [RSMWF15, RMWF16]. The library treats each NPC as a reasoning agent: First, the library lets the NPC store its knowledge about the storyworld. Second, the library decides which actions the NPC should perform based on its knowledge. To have the library find an action for an NPC, the library's user provides a goal and a set of actions for the NPC. The library chooses the action most likely to let the NPC reach its goal. The library treats this choice as a planning problem. By using Monte Carlo Tree Search (MCTS) during planning, the library can find the best option despite a limited computational budget.

In the first practical part of this thesis, we implemented this design. A user can employ the library, called the NPCEngine and written in C#, within the Unity game engine or in an independent application. The NPCEngine supports expressive action definitions that employ ECMAScript functions and inheritance. The NPCEngine also uses the JSON format throughout its API, allowing users to easily implement extensions to the library. And it provides a set of useful modules for development related to the library.

The NPCEngine addresses all the problems with video games that we identify above. A game using the library can therefore offer high player agency and a compelling story. The NPCEngine supports multiplayer applications. However, we have not yet tested the NPCEngine in a real-world environment. A performance test suggests that the NPCEngine may suffer from performance issues in large game worlds.

The second practical part of this thesis was an espionage-themed mobile game that serves as a technology demonstrator for the NPCEngine. Our goal for the mobile game was to test the NPCEngine and demonstrate its capabilities. The mobile game employs two additional technologies: a location-based mechanic using the MapBox platform, and augmented reality using the Vuforia framework. The mobile game demonstrated that users can employ the NPCEngine to control NPCs in a single- or multiplayer game.

Overall, we were able to implement a system fulfilling the challenges that the problem definition had posed. Though we haven't yet used the NPCEngine in a real-world setting, we are confident that it can produce NPC behavior that reflects the NPC's knowledge more realistically than many modern video games.

5.1 Future Work

Our next step in the development of the NPCEngine is testing it in a real-world setting. On one hand, we need to evaluate whether we can achieve the expected expressiveness. On the other, we need to determine whether the NPCEngine’s performance is sufficient for a large storyworld.

To truly evaluate the realism that our approach allows, future work should include devising new game mechanics that our approach makes possible. Social interactions such as lying and withholding information may be essential elements of such game mechanics. Having NPCs interact between each other may also be an interesting element. Such research would also involve creating various game prototypes to analyze these mechanics.

User studies can also provide insight into whether players prefer the NPC behavior under our approach to traditional NPC behavior. We could, for instance, imagine that players in fact appreciate a certain degree of determinism in NPCs. One could also explore how players put their greater agency to use and devise new forms of interaction.

On a technical level, future research could make the planning algorithm more efficient. For instance, we now copy the entire state of the NPC’s knowledge at each node of the planning tree. An approach that just saves the differences from the parent node may be more efficient from a memory and computational perspective. Other possible improvements include making the MCTS search multithreaded, which may significantly improve its performance.

Using functional programming principles, future work could extend the ECMAScript API to let users define functions more concisely. Also, the NPCEngine needs a more straightforward way of handling competing beliefs. One key element in dealing with competing beliefs is establishing a good weighting system to help compare them.

On a longer timescale, future research should explore the possible transfer of our approach to other fields. We can imagine applications of our approach in simulator-based training, psychology research or story authoring.

5.2 Outlook

Over the last few decades, video games have become one of the most widespread forms of entertainment. As an ever-increasing part of the world’s population regularly plays video games, developers have strived to attract customers with increasingly complex and realistic games. However, a major hurdle to increasing realism is the behavior of in-game characters.

We are convinced that in coming years, much research in the field of video game development will focus on increasing the realism of NPC behavior. The two defining elements of this thesis, planning and knowledge representation, will surely be among the subjects studied most. As a result of this research effort, we expect to see significantly more realistic video game characters in the near future. We hope that this thesis will contribute to that development.

Bibliography

- [ADP06] Ruth Aylett, Joao Dias, and Ana Paiva. An affectively driven planner for synthetic characters. In *Icaps*, pages 2–10, 2006.
- [ALD⁺06] Ruth Aylett, Sandy Louchart, Joao Dias, Ana Paiva, Marco Vala, Sarah Woods, and Lynne Hall. Unscripted narrative for affectively driven characters. *IEEE Computer Graphics and Applications*, 26(3):42–52, 2006.
- [ALP04] R Aylett, S Louchart, and J Pickering. A mechanism for acting and speaking for empathic agents. In *Workshop Empathic Agents, The Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS04)*, New York, 2004.
- [Ayl99a] RS Aylett. Virtual teletubbies: reapplying a robot architecture to virtual agents. In *Proceedings of the third annual conference on Autonomous Agents*, pages 338–339. ACM, 1999.
- [Ayl99b] Ruth Aylett. Narrative in virtual environments-towards emergent narrative. In *Proceedings of the AAAI fall symposium on narrative intelligence*, pages 83–86, 1999.
- [B⁺94] Joseph Bates et al. The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125, 1994.
- [Bat92] Joseph Bates. Virtual reality, art, and entertainment. *Presence: Teleoperators & Virtual Environments*, 1(1):133–138, 1992.
- [CCM02a] Marc Cavazza, Fred Charles, and Steven J Mead. Character-based interactive storytelling. *IEEE Intelligent systems*, 17(4):17–24, 2002.
- [CCM02b] Marc Cavazza, Fred Charles, and Steven J Mead. Planning characters’ behaviour in interactive storytelling. *Computer Animation and Virtual Worlds*, 13(2):121–131, 2002.
- [ESS89] Wolfgang Ertel, Johann Schumann, and Christian B Suttner. Learning heuristics for a theorem prover using back propagation. In *OeGAI*, pages 87–95. Springer, 1989.
- [KB16] Ben Kybartas and Rafael Bidarra. A survey on story generation techniques for authoring computational narratives. *IEEE Transactions on Computational Intelligence and AI in Games*, 2016.
- [KFS⁺16] Mubbasir Kapadia, Seth Frey, Alexander Shoulson, Robert W Sumner, and Markus H Gross. CANVAS: computer-assisted narrative animation synthesis. In *Symposium on Computer Animation*, pages 199–209, 2016.
- [KFZ⁺15] Mubbasir Kapadia, Jessica Falk, Fabio Zünd, Marcel Marti, Robert W Sumner, and Markus Gross. Computer-assisted authoring of interactive narratives. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, pages 85–92. ACM, 2015.
- [KKG14] Bilal Kartal, John Koenig, and Stephen J Guy. User-driven narrative variation in large story domains using monte carlo tree search. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 69–76. International Foundation for Autonomous Agents and Multiagent Systems, 2014.

Bibliography

- [KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, volume 6, pages 282–293. Springer, 2006.
- [KWB93] Margaret Thomas Kelso, Peter Weyhrauch, and Joseph Bates. Dramatic presence. *PRES-ENCE: Teleoperators & Virtual Environments*, 2(1):1–15, 1993.
- [Lau86] Brenda Kay Laurel. *Toward the design of a computer-based interactive fantasy system*. PhD thesis, The Ohio State University, 1986.
- [Loy97] Aaron B Loyall. Believable agents: Building interactive personalities. Technical report, Carnegie-Mellon Univ Pittsburgh PA Dept of Computer Science, 1997.
- [MS02] Michael Mateas and Andrew Stern. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4):39–47, 2002.
- [MS04] Michael Mateas and Andrew Stern. A behavior language: Joint action and behavioral idioms. In *Life-Like Characters*, pages 135–161. Springer, 2004.
- [RB12] Mark Owen Riedl and Vadim Bulitko. Interactive narrative: An intelligent systems approach. *Ai Magazine*, 34(1):67, 2012.
- [Rie09] Mark O Riedl. Incorporating authorial intent into generative narrative systems. In *AAAI Spring Symposium: Intelligent Narrative Technologies II*, pages 91–94, 2009.
- [RMWF16] James Ryan, Michael Mateas, and Noah Wardrip-Fruin. A simple method for evolving large character social networks. *Proc. Social Believability in Games*, 2016.
- [RSDA08] Mark O Riedl, Andrew Stern, Don Dini, and Jason Alderman. Dynamic experience management in virtual worlds for entertainment, education, and training. *International Transactions on Systems Science and Applications, Special Issue on Agent Based Systems for Human Learning*, 4(2):23–42, 2008.
- [RSMWF15] James Owen Ryan, Adam Summerville, Michael Mateas, and Noah Wardrip-Fruin. Toward characters who observe, tell, misremember, and lie. *Proc. Experimental AI in Games*, 2, 2015.
- [RSY03] Mark Riedl, Cesare J Saretto, and R Michael Young. Managing interaction between users and agents in a multi-agent storytelling environment. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 741–748. ACM, 2003.
- [RTB11] Mark Riedl, David Thue, and Vadim Bulitko. Game AI as storytelling. *Artificial Intelligence for Computer Games*, 125, 2011.
- [SGKB13] Alexander Shoulson, Max L Gilbert, Mubbasir Kapadia, and Norman I Badler. An event-centric planning approach for dynamic real-time narrative. In *Proceedings of Motion on Games*, pages 121–130. ACM, 2013.
- [SHM⁺16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- [Sin13] Brendan Sinclair. GTA V dev costs over \$137 million, says analyst. <http://www.gamesindustry.biz/articles/2013-02-01-gta-v-dev-costs-over-USD137-million-says-analyst>, February 2013. Accessed: 2017-12-06.

Bibliography

- [Y⁺01] R Michael Young et al. An overview of the mimesis architecture: Integrating intelligent narrative control into an existing gaming environment. In *The Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, pages 78–81, 2001.
- [YR03] R Michael Young and Mark Riedl. Towards an architecture for intelligent control of narrative in interactive virtual worlds. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 310–312. ACM, 2003.

