ELSEVIER SCIENCE

Science of Computer Programming

www.elsevier.com/locate/scico

A documental approach to adventure game development

Pablo Moreno-Ger[a,*], José Luis Sierra[a], Iván Martínez-Ortiz[b], Baltasar Fernández-Manjón[a]

[a]*Dpto. Ingeniería del Software e Inteligencia Artificial. Universidad Complutense de Madrid, Spain*

[b]*Centro de Estudios Superiores Felipe II, Aranjuez, Madrid, Spain.*

Elsevier use only: Received date here; revised date here; accepted date here

Abstract In this paper, we propose a documental approach to the development of graphical adventure videogames. This approach is oriented to the production and maintenance of adventure videogames using the game's storyboard as the key development element. The videogame storyboard is marked up with a suitable domain-specific descriptive markup language, from which the different art assets needed are referred, and then the final executable videogame itself is automatically produced by processing the marked storyboard with a suitable processor for such a language. This document-oriented approach opens new authoring possibilities in videogame development and allows a rational collaboration between the different communities that participate in the development process: game writers, artists and programmers. We have implemented the approach in the context of the <e-Game> project, by defining a suitable markup language for the storyboards (the <e-Game> language) and by building a suitable processor for this language (the <e-Game> engine).

Keywords: videogames; adventure games; development process; document-oriented approach; storyboard markup language; game engine

1. Introduction

The development of videogames, which started out as small individual projects initiated by groups of friends, has evolved rapidly in the last few years into a massive entertainment industry. Nowadays, videogames are huge software projects developed by heterogeneous teams, usually grouping more than one hundred people. Therefore, it is no longer possible to have a single programmer coding a game, and advanced software engineering techniques have become a necessity [34].

* Corresponding author. Tel.: +34-91-394-7623;
E-mail adresses: pablom@fdi.ucm.es (P. Moreno-Ger), jlsierra@fdi.ucm.es (J.L. Sierra), imartinez@cesfelipesegundo.com (I. Martínez-Ortiz), balta@fdi.ucm.es (B. Fernández-Manjón).

This paper addresses the relationships between *game writers*, *artists* and *programmers* in the development of *graphical adventure videogames* [15]. The skills of the different participants in the development of this kind of games are usually orthogonal, but their collaborative work is essential for the successful development of the final product. The main role of game writers is to create the *game storyboard*, where the story, development and all the different elements of the game are described. Artists in turn are engaged in producing the artworks that are integrated in the final game (e.g. graphical designs, musical compositions, etc.). Finally, programmers are in charge of implementing and customizing the software infrastructure of the game.

We propose a *documental approach* to rule the collaboration between the aforementioned communities. In this approach, the final program implementing the videogame is not the centre of the development process because the focus is on producing and maintaining the game storyboard. For this purpose, the structure of this document is characterized and formalized in terms of an easy-to-use and easy-to-understand descriptive markup language that can be used by game writers to make the structure of their storyboards explicit. This approach follows the concepts described in [6, 11, 12], where the writer of a document of a given type (a book, a technical manual, etc.) can use a descriptive markup language tailored to such a document type. In the context of this project, additional markup is added to refer the assets provided by the artists and the executable games can be automatically generated from the marked up storyboards by using a suitable processor for the descriptive markup language. An entire family of videogames (in this case, classic graphical adventure games) shares this processor. The processor is provided and maintained by the programmers. In this approach, game writers, who are the true experts in the conception of adventure videogames, rule the writing and development process.

We have implemented the documental approach in the <e-Game> project. This project proposes a development process model centred on the creation of a complete script or storyboard that is then marked with a domain-specific XML-based markup language (the <e-Game> *language*). The resulting marked document, along with the art assets to be integrated in the final game, can be interpreted by a highly modular and extensible processor (the <e-Game> *engine*) in order to automatically produce the executable videogame. Since writers are the main stakeholders in the process, <e-Game> focuses on facilitating their task as much as possible. A high level of technological skill should not be a requirement, which makes is necessary to design the process in such a way that authors will be able to develop all their potential without having to struggle with programming concepts. It should also be noticed that the main contribution and innovation of the work presented in this paper is not the use of specific standards or technologies (e.g. XML), but the use of the basic principles behind descriptive markup in the production and maintenance of graphical adventure videogames. These principles promote the separation between document structure and processing, and lead to a rational separation of roles in the development process.

The structure of this paper is as follows. In section 2 we briefly survey some related work in the field of domain-specific languages and tools for the development of videogames. In section 3 we describe the document-oriented process model proposed by <e-Game>. The <e-Game> language is presented in section 4. The <e-Game> engine is described in section 5. Section 6 presents a qualitative evaluation of <e-Game>'s usability. Finally, section 7 provides some conclusions and lines of future work.

2. Domain-specific Languages and Tools for Creating Videogames

The notion of allowing authors (as opposed to programmers) to create videogames is not new. Videogames capture the imagination of all ages and genres [9] and the concept of allowing authors to develop their own videogames without the distraction of complex low-level programming is quite popular. Many initiatives allow for this kind of rapid development, all with different approaches and varied levels of sophistication. These initiatives can be broadly categorized in three different approaches: *authoring environments* for non-programmers, *scripting solutions*,and *special-purpose programming languages*. The following subsections briefly outline each one of these alternatives.

2.1. Special-purpose Programming Languages

There are programming languages specifically designed for videogame development that require authors to have good programming skills. Perhaps the most popular option nowadays is the *DarkBasic*[1] project [13]. Dark Basic and its spin-offs (DarkBasic Pro or the DarkMatter SDK with C++ support) are probably the main reference when speaking about specific videogame programming. A previous initiative in the field, the DIV Games Studio[2], had a reasonable success in the late 90's, although it was never properly supported by its publishers. However, several open-source initiatives emerged from within the DIV community and projects like *Fenix*[3], *eDIV*[4] or *cDIV*[5] are still valid alternatives to DarkBasic.

A common characteristic of all these languages is that they include functionalities specially suited for videogame development. These functionalities usually act as simplified high-level wrappings of complex low-level constructions typically found in videogames (graphics management, game synchronization, sound management, collision detection, etc.). However, they are full-featured programming languages and, therefore, their use is too complex for the average user without an extensive programming background.

2.2. Scripting languages

Many modern videogames separate low-level logic from data and high-level definitions of the behavior of the game (AI management, game design, etc.). The core of such games is an *engine* that may have been specifically built, reused from previous development or provided by a third party. This engine is written by expert programmers, but it is not a game in itself. The game is actually a set of scripts that are interpreted and executed by the engine [31].

The advantage of scripting approaches is that these languages are usually simpler than fully featured programming languages, which facilitates game programming and maintenance. There are many scripting languages with different approaches and levels of complexity. Sometimes scripting languages are designed for a specific game engine, although general purpose languages such as LUA [14] are used in many developments.

Some commercial games disclose their scripting languages to promote the appearance of *modding communities*, formed by non-affiliated players that create customizations and

[1] http://www.thegamecreators.com/
[2] http://www.divsite.net/
[3] http://fenix.divsite.net
[4] http://ediv.divsite.net
[5] http://cdiv.sourceforge.net/

modifications of the game. Sometimes this approach is so flexible that communities have developed completely new games[6].

This approach has a clear advantage: If the scripting language is targeted to a specific game (or family of games), it is possible to simplify the language by abstracting different elements of the game. This is useful in graphical adventure games that include many widely accepted commonalities. Thus, there are several engines and scripting languages specifically tailored to suit the needs of the genre. Some common examples are initiatives like the *MAD Adventure Game Engine*[7] (which uses LUA as scripting language), *SLUDGE*[8] (Scripting Language for Unhindered Development of a Gaming Environment) or the *WinterMute Engine*[9] (featuring a very powerful Object-Oriented scripting language). A pioneer of this approach was the *SCUMM* system [33], employed in most commercial LucasArts [43] games but not available for the general public.

The genre of graphical adventure games is very synergetic with the Interactive Fiction (IF) community (text-based adventures, in which the computer is used as a medium for the delivery of richly interactive stories). The IF scene is dominated by scripting initiatives like *TADS*[10] (originally developed in 1987 that nowadays uses an object-oriented language with Pascal-like syntax), *INFORM*[11] (developed in 1993 and having a customizable parser which by default supports a C-like language) and *HUGO*[12] (developed in 1995 with a syntax reminiscent of Basic). Although all these initiatives were developed with IF and text-based adventures in mind, nowadays they all support graphics and sounds to different extents, thus making them suitable for the development of graphical adventure games.

Even though scripting languages are usually easier to learn and to use than general-purpose programming languages, they still demand that authors have good programming skills. Therefore, they are not accessible for average authors but only for those with certain skills in computer science or highly motivated enthusiasts.

2.3. Authoring Approaches

On the *no-programming* side of the spectrum, there are several authoring proposals, which allow authors to create videogames without resorting to programming. These initiatives usually trade expressive power for simplicity, although some include an extension mechanism with sophisticated programmatic resources for more advanced users.

One of the most popular initiatives is the *Game Maker*[13], which has been used as a rapid development tool in a number of academic research projects [1, 29, 32]. There are also similar commercial projects like *The FPS Creator* (for quick creation of typical First-Person-Shooters with just a few mouse clicks) or the more sophisticated *The 3D Game Maker* (also GUI-driven, although much more powerful), both produced by the same company that distributes DarkBasic[14]. Finally, the academic-oriented *ToonTalk* [17] takes a radical approach where the

---

[6] The engine of the science fiction game Half-Life is the base for the more realism-oriented Counter Strike, arguably the most popular multi-player First-Person-Shooter

[7] http://mad-project.sourceforge.net/

[8] http://www.hungrysoftware.com/tools/sludge/

[9] http://www.dead-code.org/index2.php/en

[10] http://www.tads.org/

[11] http://www.inform-fiction.org/

[12] http://www.generalcoffee.com/hugo.html

[13] http://www.gamemaker.nl

[14] http://www.thegamecreators.com/

development environment is a game itself, designed to be usable by children but without compromising expressive power. It must be noted that ToonTalk can be used for the development of any kind of program and not only videogames.

Regarding the narrower domain of graphical adventure games, there are also several initiatives that promote an authoring approach. For example, the *Adventure Game Studio*[15] (a drag and drop tool, although it does support some scripting optionally), the *Adventure Maker* project[16] (which can target the games produced to the PlayStation Portable console), or the *3D Adventure Studio*[17] (which provides a very simple GUI for the creation of 3D adventure games, but is still at a very early stage of development).

In the IF field there are also examples of the authoring approach that try to eliminate programming to allow authors without a technical background to participate. Since most authors in the scene define themselves as closer to writers of novels than to programmers, it is especially important to have tools that do not require any programming background. In this regard, *ADRIFT*[18] is probably the most extended tool, allowing the author to create the adventures using a number of windows and forms, and with an extended support for graphics and sounds that, again, blurs the barriers between Interactive-Fiction and graphical adventures. A more classical approach like *ALAN*[19] provides a language similar to written English and is considered as a modern alternative to the classic *AGT*[20].

While the author-centred conception maintained by these approaches is in accordance with our documental approach, they are usually oriented to the description of the different features of the videogame as an executable artifact. In addition, all these initiatives assume that the process is focused on the tool that you use. There is no well-defined development process and it is common to start writing the game with the tool itself (although having a full script previously written would be desirable). In this paper we propose a different general approach, focused on producing and maintaining the document with the storyboard describing the game. The proposed development process permits a rational collaboration between the different participants in this process.

3. Document-Oriented Development of Adventure Videogames: The <e-Game> Project

In the last ten years our research group at the Complutense University of Madrid has been applying a *document-oriented* approach to the development of content-intensive applications (e.g. e-learning systems, museum object repositories, knowledge based systems, etc) [10, 35, 36, 40]. These applications share the common feature of integrating large amounts of highly structured contents that are usually authored by domain experts as collections of well-structured documents. With these requirements, the approach focuses on the production and maintenance of the documents with the contents and with other aspects of the application (e.g. some features of the user interface) instead of the application itself. The structure of these documents is then made explicit by marking them up with suitable domain-specific descriptive markup languages, which are specific for each particular application domain. Finally, applications are automatically produced by processing the documents with suitable processors.

[15] http://www.adventuregamestudio.co.uk/
[16] http://www.adventuremaker.com/
[17] http://3das.noeska.com/
[18] http://www.adrift.org.uk/
[19] http://www.alanif.se/
[20] http://www.markwelch.com/agt.htm

Therefore, the approach greatly improves the collaboration between domain experts, who are mainly responsible for producing and maintaining the documents, and developers, who are responsible for formalizing the markup languages and building their processors.

The main goal of the <e-Game> project is to apply the documental approach to the development of graphical adventure videogames [21, 22, 24, 25]. The idea is to allow game writers without a strong technical background to produce and maintain an entire game as a document using a language that is easy-to-understand, which is then fed to a compiler/interpreter that produces a fully functional game. In addition, the approach also defines a model of collaboration with other stakeholders: artists and developers. The resulting process model is outlined in Fig 1, where the development activities, the participants in these activities, and the main products produced during the <e-Game> development of a videogame are outlined. In support of this process model, two different components are introduced: the <e-Game> language, which is used by writers to mark their storyboards up, and the <e-Game> engine, which is used for executing the games from their describing storyboards marked up with the <e-Game> language. While these two components are detailed in the next sections, here we will give a high-level description of the development process model introduced by <e-Game>.

In order to illustrate the development process model, as well as other aspects of the <e-Game> project, a very simple educational graphical adventure game has been taken as a common case study. This game is designed to be used as a support tool in a course on workplace safety regulations. The game describes the story of *José*, a young unemployed worker who gets a job at a construction site. During his first trial week, the worker must perform the different tasks ordered by the *foreman*, paying special attention to workplace safety regulations. Whenever the player omits a safety regulation, he either is injured or is rebuked by the foreman.

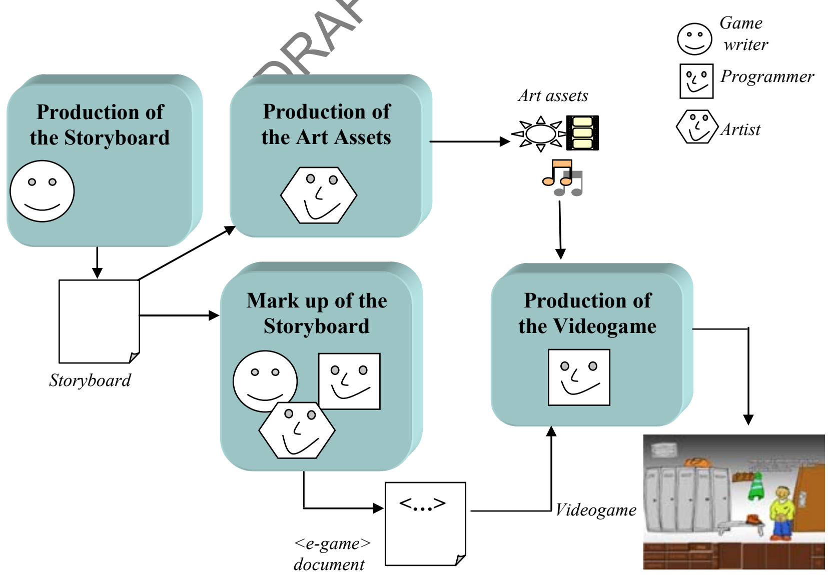
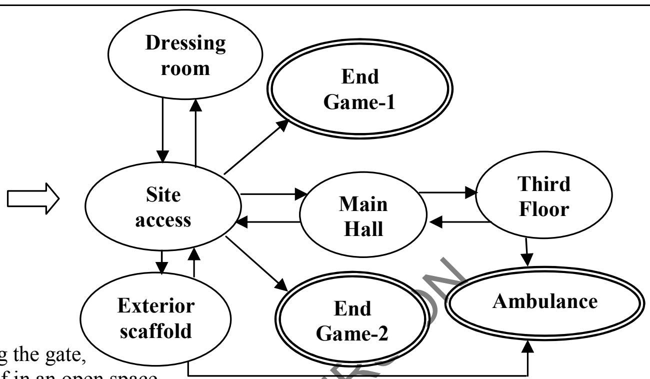


Fig 1. The <e-Game> development process

The following subsections detail the different aspects of the process model from the perspective of its activities.

3.1. Production of the storyboard

The development process model proposed by <e-Game> starts with the independent elaboration of a storyboard of the adventure game written in plain natural language (e.g. English). The game writer carries out this activity. <e-Game> imposes some guidelines restricting the style of the resulting document in order to facilitate its subsequent markup. Summarizing:



Site access
<u>Description</u>
Just after crossing the gate,
José finds himself in an open space.
There is a small pre-fabricated barn apart from the building in construction. The building has four floors and one if its walls is surrounded by a scaffold.
<u>Objects:</u>
Crowbar: There is a crowbar lying on the floor.
<u>Exits:</u>
Barn door: The barn has a door that leads to the "Dressing room"
Building gate: Leads to the "Main Hall".
Exterior gate: Only available after completing all the tasks indicated by the foreman. It leads to either ending cutscene (End Game 1 or End Game 2) according to whether the user has returned all the tools to the toolbox.
<u>Characters:</u>
Foreman: the foreman is standing in front of the building.
<u>Story:</u> (….)

Fig 2. An excerpt of the sample storyboard displaying the map and the description of a scene. End game cutscenes are identified by a double circle.

- The writer should begin by documenting the different *scenes* and *cutscenes* that form the adventure game. *Scenes* are the basic modelling units in <e-Game> and correspond to the different locations that can be visited when playing it. Typical examples might be a dressing room, a construction site or a street. *Cutscenes* in turn can be considered as special kinds of scenes that can be used to include special events in the game flow (e.g. to play a video with some sort of explanation about the game). The writer should also indicate the contents of the scenes (*objects* and *characters*) and indicate the connections between these scenes. The storyboard from our sample also includes a conceptual map of the scenes that clarifies the connections, as depicted in Fig 2. It must be noted that the different objects and characters that populate the scenes are not defined directly. Instead, they will be detailed later. This has the advantage of being more modular and naturally leads to a systematic writing methodology.

Toolbox
Description: It is a standard toolbox
Detailed Description: Safety regulations demand that work tools are always stored properly. I should put here any tools I find around.
Actions: The toolbox can not be grabbed. Different tools may be used with the toolbox and be stored into it.
(…)

Fig 3. Storyboard definitions of some objects.

- After defining all the scenes and cutscenes, the author is advised to give details about the objects populating the scenes. This definition should resemble that given in Fig 3.
- Following the <e-Game> guidelines, the next step is to describe the player and the rest of the characters as in Fig 4. Again, rather than writing out the entire contents of the different conversations, only a brief explanation is given instead. The actual content of the conversation will be detailed later.

José
José is an eager young man trying to make some money. He is not an unexperienced worker, although he is not familiar with the safety regulations in the Spanish workplace.
Description: Hey! It's me!
Detailed Description: I'm not as fit as I used to be. I hope this job will help me to lose some weight.

The foreman
The foreman is well into his forties. He always smiles and treats his employees fairly, although he usually gets angry at the sight of any breach of safety regulations due to a past bad experience that he never talks about.
Description: A friendly man with occasional bursts of anger.
Detailed Description: He is the foreman in this construction site. I think I'd better not make him angry, for my work depends on his reports.
Conversations:

- "Greeting": The first time he speaks, he greets José and welcomes him. He advises José on clothing regulations and suggests he go to the dressing room.
- "Undressed": If after the first conversation José speaks with the foreman, he is directed to the dressing room.
- "First task": The foreman asks José to go to the third floor and collect some sacks of sand that must be brought down to the hall.
- "Complete first task": If José is in the middle of the first task, the foreman insists that he must complete the action. (…)

Fig 4. Storyboard definitions of the player (José) and some characters (The Foreman)

- The last step is to describe the conversations supported by the characters. Conversations are perhaps the most delicate element in a graphical adventure. Most of the game content and clues are obtained by interacting with characters (although the use of visual clues is also common). A long, enticing conversation with varied interaction options and different possible outcomes is one of the aspects that require more dedication from the author. In addition, one of the main checkpoints for the quality of such an adventure is the quality of its dialogues. <e-Game> works with the model of multiple-choice dialog structures organized as a tree with the possible answers as nodes that open to new sub-conversations.

We recommend using a format to describe the conversations that reflects the notion of a tree like the one suggested in Fig 5.

```
F: Well José, have you measured the scaffold?
  ->J: No sir, not yet
       F: And what are you waiting for?
       J: At once, sir
  ->J: Yes sir, it's ready
       F: And...
          -> J: It's rather high, sir
                F: That is not a reliable measurement!
                F: I need precise data (go-back)
          ->J: It is 2.5 meters high, sir
                F: Hmmmm....
                F: That is over the 2 meter limit
                F: It must have handrails then and the plank must be wider than 60 centimeters
                F: Does it comply to those specifications?
                  ->J: Yes, it does
                         F: Ah, that's perfect
                         F: In that case, I need you to climb to the top of the scaffold
                         F: and help David with the window frames
                         J: Yes, sir (end of conversation)
                  ->J: No, it doesn't
                         F: I can't believe they installed a bad scaffold!
                         F: Please, check it out again or I will have to call them
                         F: and have a new scaffold installed
                         F: That would delay the entire schedule! (end of conversation)
```

Fig 5. Conversation between José and the foreman

After completing the conversations, the writer has a document that describes the complete game, defining scenes, navigation, objects, characters and all the possible interactions. This document is the most valuable asset produced during the development process, since it contains the essence and is the key to the success of the final videogame. Therefore, the documental approach develops the idea that the entire development process be focused on this document. In the end, the videogame itself will be automatically produced by processing this document, although this requires the author to define its structure explicitly.

3.2. Mark up of the storyboard

The goal of this activity is to mark up the storyboard using a suitable domain-specific markup language that indicates its structure and the semantics of each portion of the document. In addition, the language proposed in <e-Game> also includes markup to describe presentational and operational aspects needed for the execution of the actual game. The <e-Game> language itself will be detailed in section 4.

The main workload of this activity is also for the game writers. Nevertheless, the markup must also formalize aspects that are not so straightforward for the writers and may require the assistance of the other stakeholders.

First, as will be detailed in section 4, the language requires some occasional deep thinking while formalizing abstract ideas about the state of the game. In the example shown in Fig 6, the statement *"Only available after completing all the tasks indicated by the foreman"* found in the description of the exit *Exterior Gate* in the definition of *Site Access* is establishing a condition for traversing the exit that must be adequately formalized. For this purpose, the programmers may also be involved in the mark up process, in order to advise writers about the description of these *operational aspects* of the <e-Game> document.

```
<exit x="50" y="100" width="20" height="20">
   <documentation>
      Exterior gate: Only available after completing all the tasks
      indicated by the foreman. It leads to either ending cutscene
      (End Game 1 or End Game 2) according to whether the user has
      returned all the tools to the toolbox.
    </documentation>
    <next-scene idTarget="EndGame1">
      <condition>
          <active flag="EndThirdTask"/>
          <active flag="StoredHammer"/>
      </condition>
     </next-scene>
     <next-scene idTarget="EndGame2">
      <condition>
          <active flag="EndThirdTask"/>
          <inactive flag="StoredHammer"/>
       </condition>
     </next-scene>
</exit>
```

Presentational aspects

Operational aspects

Fig 6. Example of markup in <e-Game>. The markup makes the rhetorical structure of the storyboard explicit, and also includes other operational and presentational aspects.

As shown in Fig 6, there are some presentational characteristics that must also be made explicit in the markup. In the example, it is necessary to indicate the coordinates and dimensions of the rectangle that delimitates the exit (i.e. the zone that triggers the change of scene when clicked with the mouse). This process requires access to the art assets, and therefore, the collaboration of the artists.

Nevertheless, our experience is that these situations can be tackled without excessive effort in typical adventure games. The process is not overly complex, and most writers are comfortable enough so as to need barely any assistance, as detailed in section 6.

3.3. Production of the art assets

In this activity, the artists, following the descriptions of the storyboard, produce the different artwork that will finally be integrated into the game. The resulting files can be referred from the markup added to the storyboard, therefore yielding a complete description of the final videogame.

In Fig 7 some art assets for the proposed case study are shown. It is important to note that these assets can be changed by others if required, while most of the effort applied in the production of the storyboard and the markup process is preserved (presentational markup as object coordinates may need some adjustments). This possibility is a direct consequence of the documental approach and the descriptive markup spirit, which promotes an explicit separation between contents, structure and the subsequent processing of the marked documents.

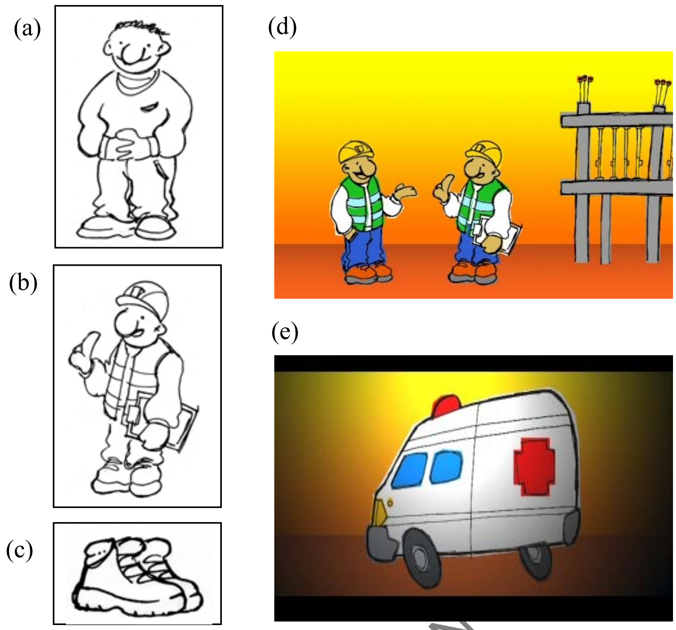


Fig 7. Some simple artworks from the case study: (a) draft of the player's avatar, (b) the Foreman, (c) sprite for an object (boots), (d) a composed scene (Site Acess), (e) a frame of a cutscene (ambulance).

3.4. Production of the videogame

Once the <e-Game> document with the marked storyboard and the art assets are available, the videogame can be produced by processing all this material with the <e-Game> engine (Fig 8).

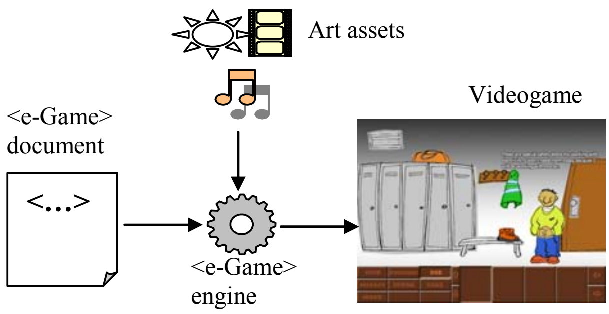


Fig 8. The videogame is automatically generated by processing the <e-Game> document and the art assets with the <e-Game> engine.

The <e-Game> engine, which will be described in section 5, is highly modular and configurable. This allows programmers to extend it with the appropriate components in order to accommodate the particular needs of a videogame. For instance, the introduction of more sophisticated art assets and presentation requirements in a game can be readily accommodated in a systematic way by customizing the <e-Game> engine with new presentation capabilities.

In addition, the newly added components can be reused both in the production of future versions of the game and of other similar games.

4. The <e-Game> Language

The <e-Game> language allows game writers to describe graphical adventure game as a document containing the game's storyboard, and to mark this document up using easy-to-use and easy-to-understand descriptive markup. Therefore, the game writer does not specify how the characters move or how the lighting works, but what the actual content of the game is (i.e. scenarios, items, conversations, etc.).

The <e-Game> language is an XML *application* (i.e. a markup language defined using XML). XML is the *de facto* standard markup metalanguage recommended by the World Wide Web Consortium [3]. Human readability for the defined markup languages is one of the key XML design features, which makes XML especially well suited for our needs. In addition, XML languages can also readily include mechanisms for identification and reference of document elements, which will be very useful in the context of <e-Game> for modelling scenes, characters and conversations separately and then linking them by using references. Finally, XML also allows the description of the markup languages using declarative grammatical formalisms [20, 28] such as DTDs (*Document Type Definitions*) [42] and XML Schema [8]. It is important to note that XML is used here with an emphasis on its original use as a document markup metalanguage, instead of its use as a storage format definition language.

Although we have formulated <e-Game> in terms of an XML Schema in order to facilitate language extensibility and evolution, for the sake of simplicity in this paper we will describe the structure of the language using an XML DTD, since this formalism is less verbose and more easily understood by most people.

In the following subsections, the different aspects of the language are detailed from a technical point of view. The examples relate to the aforementioned educational game about workplace safety regulations.

4.1. Top-level <e-Game> document

As described in the previous section, an <e-Game> document is structured as a sequence of *scenes* and *cutscenes*, *objects*, the description of the *player*, *characters* and *conversations*, which are marked up as an `eGame` element, as indicated in Fig 9.

```
<!ELEMENT eGame (title?, story?, (scene | cutscene)+, object*,
                 player, character*, conversation*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT story ANY>
```

Fig 9. <e-Game> DTD fragment defining the top-level structure for an <e-Game> document

As indicated in Fig 9, an <e-Game> document also includes the game's title (it must be marked up as a `title` element), and a summary of the game's story (marked up as a `story` element). This kind of human-readable documentation will usually be ignored by the engine described in section 5, although it will be very valuable for designers and developers during the production and maintenance stages, since it is an integral part of the game's storyboard.

4.2. Flags and Conditions

An unstructured plain game, where every door is always open, every character always says the same things and every exit leads to the same place is too limiting. An absolute lack of order could lead to incoherency and could make it difficult to perceive any progress in the game. A graphical adventure requires a sense of narrative coherence in the succession of events. We can achieve this by introducing a notion of *state*. Every action performed in the game should be able to affect future actions. Some objects may be hidden until something happens (e.g. the object appears only if the player has performed a given action), some exits may be locked (e.g. you can't use the elevator unless you have been instructed to do so and received the key), and a character may offer a different conversation (e.g. the foreman does not always give the same commands). The <e-Game> language allows a declarative representation of the game's state which is mainly based on boolean propositional variables called *flags*. These variables can be used to describe the conditions that the player must have previously achieved in order to be allowed to carry out an action in a game, such as to see an object or activate an exit. When a flag holds, it is said to be *active*. Otherwise, it is said to be *inactive*.

This can be conceptually modelled by allowing each interaction (with an object or character) to activate a condition. Then, we can add preconditions to anything we want in the form of a list of conditions that indicate which actions must have been performed. The state in any given point of the game is determined by set of actions that have already been performed.

(a)

```
<!ELEMENT condition (%basic-condition; | either)+>
<!ELEMENT active EMPTY>
<!ATTLIST active flag NMTOKEN #REQUIRED>
<!ELEMENT inactive EMPTY>
<!ATTLIST inactive flag NMTOKEN #REQUIRED>
<!ELEMENT either (%basic-condition;)+>
<!ENTITY % basic-condition "(active|inactive)">
```

(b)

```
<condition>
   <active flag="FirstTaskInitiated"/>
   <either>
       <inactive flag="UsedSandSack1Container"/>
       <inactive flag="UsedSandSack2Container"/>
   </either>
</condition>
```

Fig 10. (a) DTD fragment for conditions; (b) an example of condition

In Fig 10a the markup used to describe conditions is formalized. Notice that conditions are expressed as *conjunctive normal forms* on the flags. Indeed, atomic conditions are introduced either with an `active` element (requiring a flag to be active) or with an `inactive` one (requiring it to be inactive). The flags themselves are indicated using `flag` attributes. Clauses (disjunctions of atoms) are expressed using `either` elements. In turn, conjunctions are marked up as `condition` elements. As stated in section 6, we have realized that, in the context of graphical adventure games, the use of conjunctive normal forms is more natural for writers than the use of disjunctive ones (i.e. the presence of several groups of conditions interpreted as different alternatives).

Fig 10b shows a meaningful condition in our case study. It is important to note that in spite of the formal grounding of the condition system, the resulting markup is not far from how an author would write conditions in a storyboard: “The worker can only use the lift if he has been instructed to go to the third floor (FirstTaskInitiated) but hasn’t yet collected both sacks of sand (one of the sacks can not have been dumped in the container yet)”.

4.3. Effects

<e-Game> also allows a declarative representation of the effects caused by the different actions performed during the game. These effects are described using the markup formalized in Fig 11a. In Fig 11b the use of this markup is exemplified in the case study. Lists of effects are marked up as `effects` elements. Individual effects can be in turn of the following types:

- Activation of a flag. The most transcendent effect of an action is activating a flag (and thus modifying the state of the game). This is noted as an `activate` element and the activated condition is indicated with a `flag` attribute. Notice that in <e-Game> it is not possible to *deactivate* flags, since it includes a monotonic notion of logical truth: once a proposition becomes true, it will stay true forever. Intuitively, a flag represents an achievement, which cannot be “unachieved”.

(a)
```
<!ELEMENT effects ((activate|consume-object|speak-player|
                    speak-char)*,trigger-cutscene?)>
<!ELEMENT activate EMPTY>
<!ATTLIST activate flag NMTOKEN #REQUIRED>
<!ELEMENT consume-object EMPTY>
<!ELEMENT speak-player (#PCDATA)>
<!ELEMENT speak-char (#PCDATA)>
<!ELEMENT trigger-cutscene EMPTY>
<!ATTLIST trigger-cutscene idTarget IDREF #REQUIRED>
```

-0-

(b)
```
<effects>
     <speak-player>Aaaahhhhh!!!</speak-player>
     <activate flag="PlayerDamaged"/>
     <trigger-cutscene idTarget="Ambulance"/>
</effects>
```

Fig 11. (a) Markup for effects; (b) example of effects.

- Consumption of an object. Like most of the usual adventure games, games in <e-Game> maintain an *inventory* of objects, which can be used in different ways. Some of these uses (e.g. combining the object with another one in a specific way) can cause the consumption of the used objects. This change is expressed using a `consume-object` element. The object actually consumed will be determined by the context of the effect.
- Lines spoken by the characters. Some actions can cause the characters populating a scene to say something. Lines said by the player’s avatar are marked up as `speak-player` elements, while phrases said by characters are marked up as `speak-char` elements. In this last case, the character that speaks the line is determined by the context.
- Triggering a cutscene. Some actions can cause the visualization of a cutscene illustrating their consequences. This effect is described using a `trigger-cutscene` element. The cutscene is referred to with an `idTarget` attribute. If there is a sequence of effects, this type of effect must be the last one to appear, since when entering a cutscene, the action will continue from that cutscene.

4.4. Resources

The presentation of the different elements (scenes, cutscenes, objects, player and characters) involved in a game requires the location of external assets provided by the artists (e.g. background bitmap, environmental music, hardness maps, etc). Therefore, for each element to be presented it is possible to associate a set of *resources* containing references to all the artworks required. The markup used in <e-Game> to refer to these assets is depicted in Fig 12a. Each asset is referred to using an `asset` element. The `type` attribute allows the identification of the asset's type (e.g. a *jpg* image, an *mp3* file, etc.), while `uri` is used to locate the actual asset. Also, notice that a set of resources can be made conditional on an appropriate condition. This allows authors to tailor the presentation of the elements to different circumstances (e.g. a character that changes her clothes or a chest that appears open or closed depending on the state of the game).

(a)

```
<!ELEMENT resources (condition?, asset+)>
<!ATTLIST resources id ID #IMPLIED>
<!ELEMENT asset EMPTY>
<!ATTLIST asset type CDATA #REQUIRED uri CDATA #REQUIRED>
```

-0-

(b)

```
<resources>
    <asset type="image/jpeg" uri="images/background1.jpg"/>
    <asset type="audio/mpeg" uri="sounds/working1.mp3"/>
</resources>
```

Fig 12. (a) Markup for the resources; (b) sample resources for a scene of the game.

In Fig 12b we illustrate the use of resources with a (very simplified) example where a background image and an ambient sound for a scene are referred to.

(a)

```
<!ELEMENT scene (documentation?, resources*, exits, objects?,
                 characters?)>
<!ATTLIST scene id ID #REQUIRED start (yes|no) "no">
<!ELEMENT documentation ANY>
<!ELEMENT exits (exit+)>
<!ELEMENT objects (object-ref+)>
<!ELEMENT characters (character-ref+)>
```

-0-

(b)

```
<scene id="SiteAccess">
 <documentation>
Just after crossing the gate, José finds himself in an open space.
There is a small pre-fabricated barn apart from the building in
construction. ...
...
 </documentation>
 <resources>  ...   </resources>
 <exits>      ...   </exits>
 <objects>    ...   </objects>
 <characters> ...   </characters>
</scene>
```

Fig 13. (a) Markup for the scenes; (b) simplified example of using the (a) markup for describing a scene.

4.5. Scenes

Scenes in <e-Game> are marked up with `scene` elements, using the structure formalized in Fig 13a. According to this definition, each scene can contain the fragment of the storyboard that describes its structure, context and other aspects concerning it (`documentation` element). The scene also contains the list of alternative resources for its visualization, the sequence of exits leading to other scenes or cutscenes (`exits` element), the objects (`objects` element) and the characters (`characters` element) that may appear in the scene. Notice that each element of type `scene` has associated an `id` attribute that identifies the element within the document. This feature, which is also shared by the other top-level elements, allows other elements to refer to it. In addition, it is possible to indicate that the scene is the starting point of the game by using the attribute `start`. These markup conventions are exemplified in Fig 13b, where the (simplified) top-level markup for a scene in the case-study is depicted.

```
<!ENTITY % position   "x NMTOKEN #REQUIRED y NMTOKEN #REQUIRED">
<!ENTITY % rectangle "%position; width NMTOKEN #REQUIRED
                             height NMTOKEN #REQUIRED">
<!ELEMENT exit (documentation?, next-scene+)>
<!ATTLIST exit %rectangle;>
<!ELEMENT next-scene (condition?, effects?)>
<!ATTLIST next-scene idTarget IDREF #REQUIRED
                     xTarget NMTOKEN #IMPLIED
                     yTarget NMTOKEN #IMPLIED>
```

Fig 14. Markup for the exits.

(a)
```
<!ELEMENT object-ref (documentation?,condition?)>
<!ATTLIST object-ref idTarget IDREF #REQUIRED %position;>

<!ELEMENT character-ref (documentation?,condition?)>
<!ATTLIST character-ref idTarget IDREF #REQUIRED %position;>
```

(b)
```
<object-ref idTarget="Hole" x="45" y="50">
 <documentation>Hole:There is a big hole in the middle of the room.</documentation>
  <condition>
     <inactive flag="UsedFencesHole"/>
  </condition>
</object-ref>
...
<character-ref idTarget="Foreman" x="45" y="200">
  <documentation>The foreman is standing in front of the building.</documentation>
</character-ref>
```

Fig 15. (a) Markup for the references to the objects and the characters in a scene;
(b) example of use of the markup in (a).

In Fig 14 the markup for the description of each exit is formalized. Each exit is marked up as an element of type `exit`. The exit is located in the scene with a bounding rectangle defined by the *x* and *y* coordinates in its upper-left corner, its height and its width (this information is encoded into the `exit` elements with attributes). In <e-Game> it is possible to make the place where an exit leads conditional according to a condition formulated in the game's state. Therefore, and as indicated in Fig 14, an `exit` element also contains a sequence of possible follow up scenes, each one marked up as a `next-scene` element. These elements can contain the conditions that must hold, and the effects achieved by traversing the exits under such conditions (in this context, such effects only contemplate the activation of flags). In turn, the target of the exit, which must be a scene or a cutscene, is referred to using `idTarget` attributes, and for scenes, the starting position for the player's avatar in the target scenes is referred with xTarget and yTarget attributes. The use of this markup is illustrated in Fig 6 with an exit leading to different scenes depending on the holding condition.

Finally, the markup for the objects and the characters that can appear in a scene are characterized in Fig 15a. Again, it is important to take into account that such objects and characters are not described at this point. Instead, they are described as top-level elements in the <e-Game> document and they are referred to in the scenes using object-ref and character-ref elements. The references are indicated using idTarget attributes.

In addition, it is important to note that the visibility of an object or character in a scene can depend on a condition of the game's state (element condition), as stated in Fig 15a. Indeed, objects and characters will be visible *only* when their associated conditions hold. Finally, the position of objects and characters in the scene are given using x,y attributes and their role in the scene can be documented using a documentation element. The use of this markup is exemplified in Fig 15b.

4.6. Cutscenes

Cutscenes are marked up as cutscene elements, as formalized in Fig 16a. The structure of these elements is very simple. Indeed, the description of the cutscene can contain the fragment of the storyboard describing it, a list of alternative sets of assets used to visualize it, and a list with the next scenes that can be visited once the cutscene has been played. As with scenes, it is possible to mark a cutscene up to identify it as the game's starting point (start attribute). In addition, when the cutscene does not have a practicable next scene in the current state of the game, the game finishes. Fig 16b depicts the markup for a simple cutscene.

(a)
```
<!ELEMENT cutscene (documentation?,resources*,next-scene*)>
<!ATTLIST cutscene id ID #REQUIRED start (yes|no) "no">
```

(b)
```
<cutscene id="LiftUp">
  <documentation>The lift is moving up</documentation>
  <resources>
      <asset type="video/mpeg" uri="video/liftup.mpg"/>
  </resources>
  <next-scene idTarget="ThirdFloor"/>
</cutscene>
```

Fig 16. (a) Markup for the cutscenes; (b) an example of cutscene.

4.7. Objects

Objects in an <e-Game> document are marked up as object elements, as stated in Fig 17a. Hence that <e-Game> discriminates between two different situations:

- There is a single object distinguished (e.g. a toolbox in the case study), which is identified using an id attribute (Fig 17b).
- There is a collection of objects sharing the same features (e.g. in the case study we can find two sacks of sand, which are identical from a descriptive point of view). In this case the particular objects that share the description are enumerated using instance elements (Fig 17c).

As illustrated in Fig 17b and Fig 17c, the textual description of an object is marked up as a description element. Authors must provide three kinds of descriptions: a name (name element), a brief description (brief element), and a detailed one (detailed element).

```
(a) <!ELEMENT object (documentation?,instance*,
                     resources*,description,actions?)>
    <!ATTLIST object id ID #IMPLIED>
    <!ELEMENT instance EMPTY>
    <!ATTLIST instance id ID #REQUIRED>
    <!ELEMENT description (name,brief,detailed)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT brief  (#PCDATA)>
    <!ELEMENT detailed (#PCDATA)>
                                   -0-
(b) <object id="Toolbox">
     <documentation>...</documentation>
     <resources>...</resources>
     <description>
       <name>Toolbox</name>
       <brief>It is a standard toolbox.</brief>
       <detailed>Safety regulations demand that work tools are always stored
          properly. I should put here any tools I find around.</detailed>
    </description>
        ...
    </object>
                                   -0-
    <object>
(c)   <documentation>....</documentation>
      <instance id="SandSack1"/>
      <instance id="SandSack2"/>
      <resources>...</resources>
      <description>
         <name>Sand Sack</name>
         <brief>It is full of sand.</brief>
          <detailed>The sacks seem very heavy. I may be able to carry more than one
       with great effort, although safety regulations do not recommend it.</detailed>
      </description>
      ...
    </object>
```


Fig 17. (a) Markup for the objects; (b) a single object; (c) a collection of objects sharing common descriptive features.

In addition, objects in <e-Game> also include the set of actions that the player is allowed to carry out with the object. Markup for actions is formalized in Fig 18a. Actions are enclosed in an `actions` element and can be of the following types:

- The player can pick up the object. This action is marked up as a `grab` element (Fig 18b). Objects without a grab action cannot be picked up.
- A player can combine the object with another one. This action is marked up as a `use-with` element, and the object that can be combined with the current one is referred to using an `idTarget` attribute (Fig 18c).
- The player can also give an item to a character. The corresponding action is marked up as a `give-to` element. The character is referred to with an `idTarget` attribute (Fig 18d).

All these actions may include natural language documentation, as well as a condition and effects. In the case of *grab* actions, the effects are constrained to the activation of flags, triggering cutscenes or having the player speak a line of dialoge. In the case of *use-with* and *give-to* actions, they can also include object consumption and lines spoken by the player or the character affected.

```
    <!ELEMENT actions (grab|use-with|give-to)+>
(a) <!ELEMENT grab (documentation?,condition?, effects?)>
    <!ELEMENT use-with (documentation?,condition?, effects?)>
    <!ATTLIST use-with idTarget IDREF #REQUIRED>
    <!ELEMENT give-to (documentation?,condition?, effects?)>
    <!ATTLIST give-to idTarget IDREF #REQUIRED>
                    -0-
    <grab>
      <documentation>The hole must have been secured before grabbing
(b)                  the sacks.</documentation>
      <condition>
          <inactive flag="UsedFencesHole"/>
      </condition>
      <effects>
        <trigger-cutscene idTarget="Ambulance"/>
      </effects>
    </grab>
                    -0-
    <use-with idTarget="Toolbox">
(c)   <documentation>It can be used with the toolbox (Dressing room)</documentation>
      <effects>
         <activate flag="UsedCrowbarToolbox"/>
       </effects>
    </use-with>
                    -0-
    <give-to idTarget="Foreman">
(d)    <documentation>It can be given to the foreman, who asks José to go
          to the dressing room and put it in the toolbox.</documentation>
       <effects>
          <speak-char>Put that into the toolbox which is in the dressing
                  room, please</speak-char>
       </effects>
    </give-to>
```

Fig 18. (a) Markup for the actions involving an object; (b) example of a complex *grab* action that triggers a cutscene; (c) example of a *use with* action; (d) example of a *give to* action.

4.8. Characters and the Player

Characters are, to some extent, similar to objects, as reflected in Fig 19a. It is possible to have either individual characters or collections of common characters. Descriptions of characters are also analogous to those of objects. Nevertheless, while objects support actions, characters support conversations that can be maintained with the player (`conversations` element) as indicated in section 3. Each possible conversation is cross-referred using the `idTarget` attribute of a `conversation-ref` element. In addition, authors are also allowed to specify a condition that must hold before starting a conversation with a character (if not specified, it is assumed to be true). Also notice that Fig 19a includes markup for the player's description (`player` element).

In Fig 19b an example of character is depicted. In this example the reference to a potential conversation is also detailed, while the others are omitted for brevity.

```
(a) <!ELEMENT character (documentation?,instance*,resources*,
                         description,conversations?)>
    <!ELEMENT player (documentation?,resources*, description)>
    <!ATTLIST character id ID #IMPLIED>
    <!ELEMENT conversations (conversation-ref+)>
    <!ELEMENT conversation-ref (documentation?,condition?)>
    <!ATTLIST conversation-ref idTarget IDREF #REQUIRED>
                         -o-
(b) <character id="Foreman">
      <documentation>The foreman is well into his forties. He always smiles and
       treats his employees fairly, although he usually gets angry at the sight of any
       breach of safety regulations due to a past bad experience that he
       never talks about.</documentation>
      <description>
         <name>Foreman</name>
         <brief>A friendly man with occasional bursts of anger.</brief>
         <detailed>He is the foreman at this construction site. I think I'd better
          not make him angry, cause my work depends on his reports.</detailed>
       </description>
       <conversations>
          <conversation-ref idTarget="greeting">
             <documentation> The first time he speaks, he greets José and welcomes
                 him. He advises José on clothing regulations and suggests he
                  go to the dressing room.</documentation>
              <condition>
                 <inactive flag="SpeakGreeting"/>
              </condition>
           </conversation-ref>
            ...
       </conversations>
    </character>
```

Fig 19. (a) Markup for the characters; (b) an example (the foreman).

4.9. Conversations

<e-Game> contemplates tree-like conversations between characters and the player, whose structure is characterized in Fig 20a. Fig 20b depicts the markup for part of the conversation shown in Fig 5. According to this markup:

- A conversation always starts with a *dialogue* between the character and the player's avatar. This dialogue is characterized as a talk initiated by the character (`speak-char` element), and continued by either the character (`speak-char` element) or the player's avatar (`speak-player` element).
- The dialogue finishes when a list of options is offered to the (human) player (`response` element), or when the conversation itself is finished (`end-conversation` element, which also encloses the effects achieved by the conversation).
- Each option is in turn characterized by the option itself (marked with an `speak-player` element), followed by another dialogue that could lead to another response (thus leading to the previously mentioned tree-like structure), end the conversation, or go back to the previous set of responses (`go-back` element).

(a)

```
<!ELEMENT conversation (%dialogue;, %continuation;)>
<!ATTLIST conversation id ID #REQUIRED>
<!ENTITY % dialogue "(speak-char,(speak-char|speak-player)*)">
<!ENTITY % continuation "(response|end-conversation)">
<!ELEMENT response (option)+>
<!ELEMENT option (speak-player,%dialogue;,(%continuation;|go-back))>
<!ELEMENT go-back EMPTY>
<!ELEMENT end-conversation (effects?)>
```

-0-

(b)

```
<conversation id="CompleteSecondTask">
  <speak-char>Well José, did you measure the scaffold?</speak-char>
  <response>
    <option>
      <speak-player>No sir, not yet</speak-player>
      <speak-char>And what are you waiting for, boy?</speak-char>
      <speak-player>At once, sir</speak-player>
      <end-conversation/>
     </option>
     <option>
       <speak-player>Yes sir, it's ready</speak-player>
       <speak-char>And...</speak-char>
       <response>
          <option>
             <speak-player>It's rather tall, sir</speak-player>
             <speak-char>That is not a reasonable measure!</speak-char>
             <speak-char>I need precise data</speak-char>
             <go-back/>
           </option> (...)
```

Fig 20. (a) Markup for the conversations; (b) part of a conversation.

5. The <e-Game> Engine

<e-Game> documents and the art assets can feed an <e-Game> *engine* in order to execute the documented videogames. The core of this engine is based on the addition of suitable *operational semantics* to the <e-Game> language described in the previous section. This operational semantics is detailed in subsection 5.1. In addition, the engine has a highly modular architecture in order to facilitate its adaptation to many different application contexts and to accommodate future evolutions of the <e-Game> language. This architecture is briefly outlined in subsection 5.2.

5.1. An Operational Semantics for the <e-Game> language

Given that <e-Game> is a descriptive markup language for adventure game storyboards, <e-Game> documents can be used for many different purposes (like producing well formatted XHTML documents using an XSL Transformation [4]). Nevertheless, the more relevant use of these documents is to produce running games by automatically processing these documents with the <e-Game> engine. This use relies in turn on the addition of well-defined operational semantics for the <e-Game> language. The formal specification of these semantics will be a very useful guide for developers who build and maintain the <e-Game> engine. Therefore, the primary goal of this specification is to provide a strong, unambiguous and implementation-independent description of the dynamic behaviour of <e-Game>, which is a key aspect for a successful language design and implementation process [23]. Besides, regardless of its formal flavour, the specified semantics are based on the usual interactions and system behaviours found in the genre of graphical adventure games, instead of on sophisticated or abstract mathematical concepts. This makes the language easy to use for authors, who will usually have enough intuitive knowledge about these interactions and behaviours.

$$\frac{\Phi_o ;;...;; \Phi_k}{\Psi_o ;;..;; \Psi_n}$$

Fig 21. Structure of the inference rules. Each Φ_i in the premise and Ψ_j in the conclusion are expressions in a suitable formal language.

The description of the <e-Game>'s operational semantics follows the style of the *structural* approach to the specification of the operational semantics of artificial programming languages [16, 27, 30]. This approach leads to reasonably understandable specifications, which can also be easily prototyped in order to check the adequacy of the subsequent implementation with very little additional effort [5]. According to the approach, the operational semantics of a language is characterized by modelling the behaviour of an abstract machine that executes this language as a *formal calculus* made up of *inference rules* like those depicted in Fig 21. The reading of such rules is the usual one: when all the elements in the premise hold, the elements in the conclusion hold. Empty premises can be omitted, and the resulting rules are used to introduce *axioms* into the calculus. Most of the interesting calculi will usually consist of an infinite number of inference rules. In order to give a finite characterization, a finite number of *rule patterns* can be provided instead by using *syntactic variables*. We will use a *cursive* font to denote syntactic variables in our specification. In addition, this specification will mainly adopt a *small-step* specification style, since we are interested in the rigorous modelling of the basic state transitions of the <e-Game> engine. Nevertheless, we will also give a *big-step* characterization of the overall behaviour of the engine by taking, as usual, the transitive closure of the transition relation in the *small-step* one. In our formalization we will be use the following notations:

$$\frac{\vdash \langle p,v \rangle \in \rho}{\vdash \rho_p = v}$$

$$\vdash \left(\rho_p := v\right) = \left\{\langle p',v' \rangle \mid \langle p',v' \rangle \in \rho \wedge p' \neq p\right\} \cup \left\{\langle p,v \rangle\right\}$$

Fig 22. Consulting and updating the values in set of property-value pairs.

- $\vdash \Phi$ for denoting a set-theoretical formula Φ that must hold. In our formalization, we will freely make use of typical first-order logic and set theoretical constructs and operations, and therefore it will be assumed that the <e-Game> semantics will be built upon an appropriate axiomatization for such constructs (see, for instance, [19]). On the contrary, we only define the set-related notations specifically introduced for the <e-Game> semantics. This is the case of the sets of property – value pairs that will be used for several purposes in the semantics, in order to facilitate the modular evolution of the specification without the need to resort to more sophisticated formalisms with built-in modularity facilities, like [26]. In Fig 22 we provide consulting and updating facilities for managing these sets. With ρ_p we will denote the value of property p in set ρ. With $\rho_p := v$ we will denote the set that results from substituting the value of p in ρ by v.
- $s_o \rightarrow s_1$ for denoting a basic (small-step) state transition.

- $s_o \rightarrow_+ s_1$ for denoting that the state s_1 can be reached from the state s_o with a sequence of one or more basic state transitions (i.e. for denoting a big-step state transition).

In the following sections we describe this <e-Game>'s operational semantics. We will start by designing a suitable formal representation for the computation states of a very high-level and abstract intended view of the <e-Game> engine. Then we will formalize the <e-Game>'s semantics rules.

5.1.1. Computation states

In Fig 23 a very high-level view of the <e-Game> engine is informally outlined. According to this view, the engine is made up of a *core* and a *user interface*, which are connected using two streams: *input* and *output*. The user interface collects the *user's inputs* and puts them in the input stream. The core in turn encodes the operational behaviour of the engine, and therefore is able to process the inputs to write *presentation commands* in the output stream according to the description in the <e-Game> document taken as input. These commands are in turn interpreted by the user interface in order to produce suitable presentations.

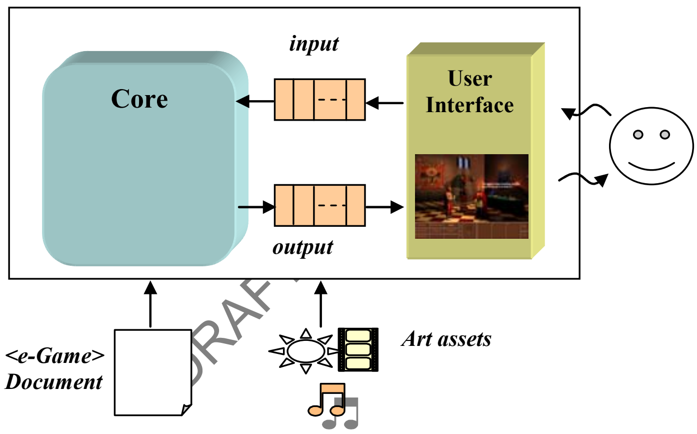


Fig 23. A very high-level view of the <e-Game> engine.

Therefore, the computation states for this engine will include a suitable abstract representation of the e-game document (which will be maintained invariable during the entire execution), the internal game's state maintained by the game's core, the control state of this core, and the state of the input and the output streams. Notice that the user interface will be actually abstracted in terms of the user's inputs and the presentation commands read and written in such streams. Therefore, the computation state will be formally represented by 5-tuples $<\theta,G,\sigma,in,out>$ where:

- *G* is an abstract representation of the game that is being played. This representation is a set of *information items* containing all the data about the <e-Game> document required to execute the game. For the purpose of formalization these items will be represented as ordered tuples, and they will include tuples of (but not necessarily restricted to) the types described in Fig 24. Notice that this representation, which is specially tailored to the formalization described in this section, can be readily generated for each <e-Game> document as a domain-specific *infoset* [7] for such a document. For simplicity, we omit the formalization of the translation process. Also, notice that the representation can include additional information that, like the assets associated with the different elements, could be required for the user interface in order to interpret the presentation commands correctly. In these items, conditions are further represented as ordered pairs <*f*+,*f*->, where *f*+ is the set of flags that must be active, and *f*- the set of those that must be inactive. Rules in Fig 25 introduce shortcuts for accessing these two components. In turn, effects are represented as a list of tuples representing the individual effects. In this representation, the first element identifies the effect's type, and the other elements represent the effect's arguments (e.g. <activate,*f*> for activating the flag *f*). Lists themselves will be represented either by <> (in case of the empty list) or by <*e*,*l*> (in case of a list with head *e* and with rest *l*). Finally, conversations will be represented as lists of tuples representing the basic conversation steps. Option lists in the conversation will be represented as lists of pairs of the form <*m*,*conv*>, where *m* is the message to be said by the player, and *conv* is the conversation that follows.

Information item	**Intended meaning**	**Information item**	**Intended meaning**
<scene,*s*>	*s* is a scene.	<cutscene,*cs*>	*cs* is a cutscene.
<next-scene,*cs*,*ns*,*c*,*es*>	If condition *c* holds, once the cutscene *cs* is finished, it is possible to enter *ns* and get *es* as effects.	<grab,*o*,*c*,*es*>	If condition *c* holds, object *o* can be grabbed. The effect is the achievement of *es*.
<next-scene,*s*,*i*,*ns*,*c*,*es*>	If condition *c* holds, it is possible to go from scene *s* to *ns* by traversing the exit number *i* and to achieve *es* as effects.	<use-with,o_s, o_t ,*c*,*es*>	Object o_s can be combined with object o_t provided that condition *c* holds. Then the effects *es* are achieved.
<object,*s*,o,*c*>	Object *o* is visible in the scene *s* provided that *c* holds.	<give-to,*o*, *ch* ,*c*,*es*>	Object *o* can be given to character *ch* when condition *c* holds. Then effects *es* are achieved.
<character,*s*,*ch*,*c*>	Character *ch* is visible in scene *s* provided that *c* holds.	<conversation,*ch*,*conv*,*c*>	The conversation *conv* can be maintained with character *ch* when condition *c* holds.
<start,*s*>	Scene or cutscene *s* is the starting point.		

Fig 24. Information items in the abstract representation of the game relevant for the operational semantics.

$$\frac{\vdash c = \langle fs, \_ \rangle}{\vdash \mathrm{F}^{+}(c) = fs}$$

$$\frac{\vdash c = \langle \_, fs \rangle}{\vdash \mathrm{F}^{-}(c) = fs}$$

Fig 25. Shortcuts for accessing the components of a condition. With _ we will denote an anonymous unique syntactic variable.

- θ is the control state of the engine's core. This state is used to decide how the execution is to proceed. It is represented as a set of property – value pairs. In Fig 26 we characterize the types of control states to be used in the semantics. Ctrl-enter(*s*) indicates that the player is entering the scene or cutscene s, while ctrl-in(s) indicates that he/she is actually in s. Ctrl-app-effects(es, θ') indicates that effects es must be applied, and, if the application is completed, control state θ' must be used in continuation. Ctrl-use-with (o,es, θ') indicates an object as an additional context for the application of such effects, and ctrl-give-to(o,ch,es, θ') indicates an object and a character as such an additional context. Ctrl-talking(s,ch,$conv$) indicates that the conversation $conv$ is being carried out with character ch in scene s. Ctrl-goto-choosing(θ', os) changes such a state to let the player choose an option, while ctrl-goto-talking(θ', $conv$) changes the choosing state to the talking one and sets the conversation associated with the option chosen. Finally, ctrl-game-over is the state associated with the end of the game. In Fig 27 a set of rules for discriminating between these control states and accessing some of their relevant properties are introduced.

$$\vdash \text{ctrl-enter}(s) = \{\langle \text{ctrl,enter}\rangle, \langle \text{scene},s\rangle\}$$

$$\vdash \text{ctrl-in}(s) = \{\langle \text{ctrl,in}\rangle, \langle \text{scene},s\rangle\}$$

$$\vdash \text{ctrl-app-effects}(es,\theta) = \{\langle \text{ctrl,app-effects}\rangle, \langle \text{effects},es\rangle, \langle \text{next-ctrl},\theta\rangle\}$$

$$\vdash \text{ctrl-use-with}(o,es,\theta) = \text{ctrl-app-effects}(es,\theta) \cup \{\langle \text{obj},o\rangle\}$$

$$\vdash \text{ctrl-give-to}(o,ch,es,\theta) = \text{ctrl-app-effects}(es,\theta) \cup \{\langle \text{obj},o\rangle, \langle \text{char},ch\rangle\}$$

$$\vdash \text{ctrl-talking}(s,ch,conv) = \{\langle \text{ctrl,talking}\rangle, \langle \text{scene},s\rangle, \langle \text{char},ch\rangle, \langle \text{conv},conv\rangle\}$$

$$\vdash \text{ctrl-goto-choosing}(\theta,os) = (\theta_{\text{ctrl}} := \text{choosing})_{\text{options}} := os$$

$$\vdash \text{ctrl-goto-talking}(\theta,conv) = (\theta_{\text{ctrl}} := \text{talking})_{\text{conv}} := conv$$

$$\vdash \text{ctrl-game-over} = \{\langle \text{ctrl,game-over}\rangle\}$$

Fig 26. Characterization of the control states of the engine's core.

$$\frac{\vdash \theta_{\text{ctrl}} = \text{enter} \;;;\; \vdash \theta_{\text{scene}} = s}{\vdash \text{is-enter}(\theta,s)}$$

$$\frac{\vdash \theta_{\text{ctrl}} = \text{in} \;;;\; \vdash \theta_{\text{scene}} = s}{\vdash \text{is-in}(\theta,s)}$$

$$\frac{\vdash \theta_{\text{ctrl}} = \text{app-effects} \;;;\; \vdash \theta_{\text{effects}} = es}{\vdash \text{is-app-effects}(\theta,es)}$$

$$\frac{\vdash \theta_{\text{ctrl}} = \text{talking} \;;;\; \vdash \theta_{\text{conv}} = conv}{\vdash \text{is-talking}(\theta,conv)}$$

$$\frac{\vdash \theta_{\text{ctrl}} = \text{choosing} \;;;\; \vdash \theta_{\text{options}} = os}{\vdash \text{is-choosing}(\theta,os)}$$

Fig 27. Shortcuts for discriminating between control states and accessing some of their properties.

- σ represents the game state. As with the control state, it will be represented as a set of property – value pairs. In the semantics presented in this paper, this state includes two properties: the set of active flags (*flags*) and the set of objects in the inventory (*inv*). The game's initial state is characterized in Fig 28. Rules in Fig 29 introduce a notation for testing when a condition *c* holds in a game state σ' –holds(c,σ'); when an object or a character *e* is in a scene *s* of the game *G* provided the game state σ' –is-in-scene(G,σ',s,e); when an object *o* is visible in such conditions, either by being in the scene or being in the inventory –is-visible(G,σ',s,o); and when a cutscene *cs* triggers the end of the game *G* given the state σ' –is-game-over(cs, σ', G).

$$\vdash \text{gs-init} = \{\langle \text{flags},\varnothing\rangle,\langle \text{inv},\varnothing\rangle\}$$

Fig 28. The game's initial state.

$$\frac{\vdash \mathrm{F}^{+}(c) \subseteq \sigma_{\text{flags}} \;;;\; \vdash \mathrm{F}^{-}(c) \cap \sigma_{\text{flags}} = \varnothing}{\vdash \text{holds}(c,\sigma)}$$

$$\frac{\vdash \langle \text{object}, s, e, c\rangle \in G \vee \langle \text{character}, s, e, c\rangle \in G \;;;\; \vdash \text{holds}(c,\sigma)}{\vdash \text{is-in-scene}(G,\sigma,s,\mathrm{e})}$$

$$\frac{\vdash \text{is-in-scene}(G,\sigma,s,o) \vee o \in \sigma_{\text{inv}}}{\vdash \text{is-visible}(G,\sigma,s,o)}$$

$$\frac{\vdash \langle \text{cutscene}, cs\rangle \in G \;;;\; \vdash \nexists \mathrm{ns},\mathrm{c},\mathrm{es}(\langle \text{next-scene}, cs, \mathrm{ns}, \mathrm{c}, \mathrm{es}\rangle \in G \wedge \text{holds}(\mathrm{c},\sigma))}{\vdash \text{is-game-over}(cs,\sigma,G)}$$

Fig 29. Some rules for dealing with the game's state.

User's input	Intended meaning	User's input	Intended meaning
<go,*e*>	Go to the exit *e* in the current scene.	<give-to,*o*,*ch*>	Give object *o* stored in the inventory to character *ch*.
<inspect,*e*,*l*>	Inspect element *e* (object or character) in the current scene with a detail level *l* (*brief* or *detailed*).	<talk-to,*ch*>	Initiate a conversation with character *ch*.
<grab,*o*>	Grab object *o* in the current scene.	<select,*o*>	Continue the current conversation from option *o*.
<use-with,o_s,o_t>	Combine object o_s in the inventory with object o_t, in the current scene.		

Fig 30. Encoding of the user's inputs.

- *in* is the input stream. This stream will be represented as a list. In this case this list will contain the user's actions, which will be represented as tuples of the types shown in Fig 30.
- *out* is the output stream. It will contain the presentation's commands represented as tuples, whose intended meaning will be clear from the context of use (e.g. <do-inspect,*o*,*l*> will be the presentation command associated with the inspection of object *o* with a level of detail *l*). The stream itself will be represented using <> for the empty stream, and pairs of the form <*s*,*e*> for the result of appending element *e* to stream *s*.

5.1.2. Semantic rules

In Fig 31 the rules characterizing the state transitions caused by the application of effects are shown:

- Rule *end-apply-effects* deals with an empty list of effects. In this case, the control state indicated as a continuation is established as a new control state.
- The other rules address the application of the different types of effects. The *activate-flag* rule adds the activated flag to the set of active flags in the game's state. The *speak-player1* and *speak-char1* rules write suitable presentation commands in the output to force the player's avatar or the character to speak. The *consume-obj* rule extracts the current object from the inventory (this object will be the value of the *obj* property). Finally, the *trigger-cs* rule causes the indicated cutscene to be entered by setting a suitable control state. Notice that all the rules except this last one imply the application of the rest of the effects. On the contrary, *trigger-cs* interrupts the application of the list of effects (regardless of the fact that the corresponding effect should be the last on such a list), discards the continuation state, and forces a control state leading to the triggering of the cutscene.

Rules in Fig 32 formalize the walk through the scenes and the cutscenes:

- Rule *entering* establishes what happens when a scene or cutscene is entered: a presentation command for playing it is written in the output.
- The logic for abandoning a cutscene is mirrored in rules *leaving-cs* and *game-over*. In *leaving-cs* a suitable next scene is discovered, whose condition is active and therefore the referred destination can be entered. In turn *game-over* deals with the case where such a next exit is not found. In this case the game finishes.
- The logic for abandoning scenes is in turn reflected in *leaving-s*, which is almost identical to *leaving-cs* with the exception of the user being the one who chooses the exit where he/she wants to go.

$$\frac{\vdash \text{is-app-effects}(\theta,\langle\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{next-ctrl}}, G, \sigma, in, out\rangle}\ \textbf{end-apply-effects}$$

$$\frac{\vdash \text{is-app-effects}(\theta,\ \langle\langle \text{activate-flag}, f\rangle, es\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{effects}} := es, G, \sigma_{\text{flags}} := \sigma_{\text{flags}} \cup \{f\}, in, \langle out, \langle \text{do-activate-flag}, f\rangle\rangle\rangle}\ \textbf{activate-flag}$$

$$\frac{\vdash \text{is-app-effects}(\theta, \langle\langle \text{speak-player}, m\rangle, es\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{effects}} := es, G, \sigma, in, \langle out, \langle \text{do-speak-player}, m\rangle\rangle\rangle}\ \textbf{speak-player1}$$

$$\frac{\vdash \text{is-app-effects}(\theta,\ \langle\langle \text{speak-char}, m\rangle, es\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{effects}} := es, G, \sigma, in, \langle out, \langle \text{do-speak-char}, \theta_{\text{char}}, m\rangle\rangle\rangle}\ \textbf{speak-char1}$$

$$\frac{\vdash \text{is-app-effects}(\theta, \langle \text{consume-object}, es\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{effects}} := es, G, \sigma_{inv} := \sigma_{\text{inv}} - \{\theta_{\text{obj}}\}, in, \langle out, \langle \text{do-consume-object}, \theta_{\text{obj}}\rangle\rangle\rangle}\ \textbf{consume-obj}$$

$$\frac{\vdash \text{is-app-effects}(\theta, \langle\langle \text{trigger-cs}, cs\rangle, \_\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \text{ctrl-enter}(cs), G, \sigma, in, \langle out, \langle \text{do-trigger-cs}, cs\rangle\rangle\rangle}\ \textbf{trigger-cs}$$

Fig 31. Application of the effects.

$$\frac{\vdash \text{is-enter}(\theta,s)}{\langle \theta,G,\sigma,in,out\rangle \rightarrow \langle \text{ctrl-in}(s),G,\sigma,in,\langle out,\langle \text{do-enter},s\rangle\rangle\rangle}\ \textbf{entering}$$

$$\frac{\vdash \text{is-in}(\theta,cs)\ ;;\ \vdash \langle \text{cutscene},cs\rangle \in G\ ;;\ \vdash \langle \text{next-scene},cs,ns,c,es\rangle \in G\ ;;\ \vdash \text{holds}(c,\sigma)}{\langle \theta,G,\sigma,in,out\rangle \rightarrow \langle \text{ctrl-app-effects}(es,\text{ctrl-enter}(ns)),G,\sigma,in,\langle out,\text{do-leaving}\rangle\rangle}\ \textbf{leaving-cs}$$

$$\frac{\vdash \text{is-in}(\theta,cs)\ ;;\ \vdash \text{is-game-over}(cs,\sigma,G)}{\langle \theta,G,\sigma,in,out\rangle \rightarrow \langle \text{ctrl-game-over},G,\sigma,in,\langle out,\text{do-finish}\rangle\rangle}\ \textbf{game-over}$$

$$\frac{\vdash \text{is-in}(\theta,s)\ ;;\ \vdash \langle \text{scene},s\rangle \in G\ ;;\ \vdash \langle \text{next-scene},s,e,ns,c,es\rangle \in G\ ;;\ \vdash \text{holds}(c,\sigma)}{\langle \theta,G,\sigma,\langle\langle \text{go},e\rangle,in\rangle,out\rangle \rightarrow \langle \text{ctrl-app-effects}(es,\text{ctrl-enter}(ns)),G,\sigma,in,\langle out,\langle \text{do-leaving},e\rangle\rangle\rangle}\ \textbf{leaving-s}$$

Fig 32. Entering and leaving scenes and cutscenes, and finishing the game.

The inspection of objects and characters is formalized in Fig 33 with the rule *inspect*. Notice that only the visible elements (including the objects in the inventory) can be inspected.

$$\frac{\vdash \text{is-in}(\theta,s)\ ;;\ \vdash \text{is-visible}(G,\sigma,s,e)}{\langle \theta,G,\sigma,\langle\langle \text{inspect},e,l\rangle,in\rangle,out\rangle \rightarrow \langle \theta,G,\sigma,in,\langle out,\langle \text{do-inspect},e,l\rangle\rangle\rangle}\ \textbf{inspect}$$

Fig 33. Inspecting objects and characters.

The different actions applicable to objects (grabbing them, combining them with other objects, and donating them to other characters) are formalized in Fig 34:

- Grabbing an object is supported by the *grabbing* rule. For an object to be grabbed it must be in the scene, grabbing it must be permitted, and the user must *want* to grab it. The object is added to the inventory, and the corresponding effects are applied.

$$\frac{\vdash \text{is-in}(\theta,s)\ ;;\ \vdash \text{is-in-scene}(G,\sigma,s,o)\ ;;\ \vdash \langle \text{grab},o,c,es\rangle \in G\ ;;\ \vdash \text{holds}(c,\sigma)}{\langle \theta,G,\sigma,\langle\langle \text{grab},o\rangle,in\rangle,out\rangle \rightarrow \langle \text{ctrl-app-effects}(es,\theta),G,\sigma_{\text{inv}} := \sigma_{\text{inv}} \cup \{o\},in,\langle out,\langle \text{do-grab},o\rangle\rangle\rangle}\ \textbf{grabbing}$$

$$\frac{\vdash \text{is-in}(\theta,s)\ ;;\ \vdash \langle \text{use-with},o_s,o_t,c,es\rangle \in G\ ;;\ \vdash o_s \in \sigma_{inv}\ ;;\ \vdash \text{is-in-scene}(G,\sigma,s,o_t)\ ;;\ \text{holds}(c,\sigma)}{\langle \theta,G,\sigma,\langle\langle \text{use-with},o_s,o_t\rangle,in\rangle,out\rangle \rightarrow \langle \text{ctrl-use-with}(o_s,es,\theta),G,\sigma,in,\langle out,\langle \text{do-comb},o_s,o_t\rangle\rangle\rangle}\ \textbf{combining}$$

$$\frac{\vdash \text{is-in}(\theta,s)\ ;;\ \vdash \langle \text{give-to},o,ch,c,es\rangle \in G\ ;;\ \vdash o \in \sigma_{inv}\ ;;\ \vdash \text{is-in-scene}(G,\sigma,s,ch)\ ;;\ \text{holds}(c,\sigma)}{\langle \theta,G,\sigma,\langle\langle \text{give-to},o,ch\rangle,in\rangle,out\rangle \rightarrow \langle \text{ctrl-give-to}(o,ch,es,\theta),G,\sigma,in,\langle out,\langle \text{do-give-to},o,ch\rangle\rangle\rangle}\ \textbf{giving}$$

Fig 34. Using the objects.

- The combination of objects is addressed in the *combining* rule. The source object must be in the inventory, the target object must be in the scene, the combination must be allowed by the game description, it must be feasible in the current state, and the user must trigger such a combination. The result is the application of the effects induced by the action.
- Finally, the donation of an object to a character is contemplated by the *giving* rule. The object donated must be in the inventory, the character must be in the scene, the donation must be contemplated in the game description, and it also must be viable in the current state and desired by the user. As a result the donation is carried out by executing the corresponding effects.

The characterization of the operational semantics of the conversations with the characters is addressed by the rules in Fig 35:

- The *init-conv* rule characterizes the beginning of a conversation: the character is in the current scene, is able to maintain a conversation in the current game state, and the user initiates such a conversation. Then the control state is properly established to proceed with the conversation.
- Dialogue that proceeds automatically between the characters and the player's avatar is managed by the *speak-player2* and the *speak-char2* rules. These rules reflect the writing of the corresponding presentation commands in the output.
- Interaction with the user in order to let him/her choose an option is addressed by the *choosing*1 and *choosing*2 rules. The first rule deals with the presentation of a list of options during the course of the conversation. Notice that the option list itself is stored in the control state in order to backtrack to it if needed. The second rule deals with the actual selection of an option and the conversation continues by the branch attached to that option.
- Backtracking to the nearest option list is actually addressed by the *going-back* rule.
- Finally, the *ending-conv* rule deals with the end of the conversation and applies the corresponding effects.

$$\frac{\vdash \text{is-in}(\theta,s)\ ;;\ \vdash \text{is-in-scene}(G,\sigma,s,ch)\ ;;\ \vdash \langle \text{conversation}, ch, conv, c\rangle \in G\ ;;\ \vdash \text{holds}(c,\sigma)}{\langle \theta, G, \sigma, \langle\langle \text{talk-to}, ch\rangle, in\rangle, out\rangle \rightarrow \langle \text{ctrl-talking}(s, ch, conv), G, \sigma, in, \langle out, \langle \text{do-talk-to}, ch\rangle\rangle\rangle}\ \textbf{init-conv}$$

$$\frac{\vdash \text{is-talking}(\theta, \langle\langle \text{speak-player}, m\rangle, conv\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{conv}} := conv, G, \sigma, in, \langle out, \langle \text{do-speak-player}, m\rangle\rangle\rangle}\ \textbf{speak-player2}$$

$$\frac{\vdash \text{is-talking}(\theta, \langle\langle \text{speak-char}, m\rangle, conv\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{conv}} := conv, G, \sigma, in, \langle out, \langle \text{do-speak-char}, \theta_{\text{char}}, m\rangle\rangle\rangle}\ \textbf{speak-char2}$$

$$\frac{\vdash \text{is-talking}(\theta, \langle\langle \text{options}, os\rangle, \langle\rangle\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \text{ctrl-goto-choosing}(\theta, os), G, \sigma, in, \langle out, \langle \text{do-choosing}, os\rangle\rangle\rangle}\ \textbf{choosing1}$$

$$\frac{\vdash \text{is-choosing}(\theta, os)\ ;;\ \vdash \langle o, conv\rangle \in os}{\langle \theta, G, \sigma, \langle\langle \text{select}, o\rangle, in\rangle, out\rangle \rightarrow \langle \text{ctrl-goto-talking}(\theta, conv), G, \sigma, in, \langle out, \langle \text{do-choosen}, o\rangle\rangle\rangle}\ \textbf{choosing2}$$

$$\frac{\vdash \text{is-talking}(\theta, \langle \text{go-back}, \langle\rangle\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \theta_{\text{ctrl}} := \text{choosing}, G, \sigma, in, \langle out, \langle \text{do-going-back}, \theta_{\text{options}}\rangle\rangle\rangle}\ \textbf{going-back}$$

$$\frac{\vdash \text{is-talking}(\theta, \langle\langle \text{end-conversation}, es\rangle, \langle\rangle\rangle)}{\langle \theta, G, \sigma, in, out\rangle \rightarrow \langle \text{ctrl-app-effects}(es, \text{ctrl-in}(\theta_{\text{scene}})), G, \sigma, in, \langle out, \text{do-end-conv}\rangle\rangle}\ \textbf{ending-conv}$$

Fig 35. Talking.

Finally,the big-step rules in Fig 36 deal with the behavior of the engine itself. Rule *playing* takes the transitive closure of the state transition relation. In turn, rule *play* models the complete behavior, starting at an initial state and ending at a final one.

As a final remark, notice that a game can exhibit several sources of non-determinism, as revealed by this formal specification of the semantics (e.g. an exit can be practicable in several ways, a character is able to maintain several conversations, combining two objects or giving one to a character can have several associated effects, etc.). Practical implementation will deal with a non-deterministic situation by randomly choosing one of the possible outcomes. These situations could be considered as a flaw in a fixed storyboard. However, they should not be considered a design flaw in the <e-Game> language. Although in the current implementation the engine issues a warning whenever these situations are detected, authors may want to innovate and explore this kind of non-deterministic behavior.

$$\frac{s_o \rightarrow s_1 \;;;\; s_1 \rightarrow_+ s_2}{s_o \rightarrow_+ s_2}\ \textbf{playing}$$

$$\frac{\vdash \langle \text{start}, s\rangle \in G \;;;\; \langle \text{ctrl-enter}(s), G, \text{gs-init}, in, \langle\rangle\rangle \rightarrow_+ \langle \theta, G, \sigma, \langle\rangle, out\rangle \;;;\; \vdash \theta_{\text{ctrl}} = \text{game-over}}{\langle G, in\rangle \rightarrow_+ \langle \theta, G, \sigma, \langle\rangle, out\rangle}\ \textbf{play}$$

Fig 36. Playing the game.

5.2. Implementation details

The architecture of the <e-Game> engine is depicted in Fig 37. This architecture is a refinement of the high-level view of Fig 23 and its design has been driven by the operational semantics described above. According to this architecture, the engine includes the following elements:

- A *tree builder*. This artefact is based on a standard DOM parser [2], and it builds a tree representation of the <e-Game> document.
- A *component repository*. This repository contains a set of *game components*, which can be adequately selected and assembled to create the final videogame. There are two kinds of game components: *core rules*, which roughly correspond to the small-step rules of the <e-Game> operational semantics, and *GUI components*, which implement interaction and presentation services for supporting the final presentation layer of the videogame. These components follow a common component model, which is an evolution of the model described in [37].
- The *core* and *user interface* were presented in the previous subsection. The engine core can be customized with an appropriate set of core rules, and it includes a *core driver* that implements the selection and application strategies for such rules. The behaviour of this driver roughly corresponds with that modelled by the big-step rules of the semantics, enriched with randomized conflict resolution. In turn, the user interface is customized with a suitable collection of GUI components, and its behaviour is controlled by a pre-established *GUI shell*.
- A *game generator*. This artifact processes the document, selects the appropriate game components, and registers them in the core and user interface of the engine. The game generator is architected according to the model for the incremental construction of processors for domain-specific markup languages described in [39]. Therefore, it contains a general-purpose *tree processor* and a set of *operationalizers* that are used to assign a set of semantic attributes and an evaluator to each node of the document tree. In turn, the evaluators will be responsible for computing these values for the semantic attributes associated with their nodes.