Language-Driven Development of Videogames: The <e-Game> Experience*

Pablo Moreno-Ger[1], Iván Martínez-Ortiz[2], José Luis Sierra[1], and Baltasar Fernández-Manjón[1]

[1] Fac. Informática. Universidad Complutense. 28040, Madrid. Spain
{pablom, jlsierra, balta}@fdi.ucm.es
[2] Centro de Estudios Superiores Felipe II. 28300, Aranjuez. Spain
imartinez@cesfelipesegundo.com

Abstract. In this paper we describe a language-driven approach to the development of videogames. In our approach the development process starts with the design of a suitable domain-specific language for building games, along with an abstract syntax for the language and its operational semantics. Next an engine supporting the language is built. Finally games are built using the customized language and they are executed using the engine. This approach is exemplified with the <e-Game> project, which delivers the design of a language and the construction of an engine for the documental development of graphical adventure videogames with educational purposes.

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

1 Introduction

There are fields like edutainment [12], casual gaming [17] or propaganda [26], where rapid development of simple games is required. This development is usually carried out using languages and/or tools specifically tailored to the videogame domain. These languages and tools range from very powerful, but also knowledge-demanding, special-purpose programming languages (e.g. *DIV* [8] and *DarkBasic* [10]) to easy-to-use, but also more limited, authoring systems to be used by non programmers (e.g. *Game Maker* [21]), to scripting solutions based on either general-purpose scripting languages (e.g. *LUA* [11] or *TCL* [20]) or on languages specially tailored to specific videogames.

This search for a good balance between expressive power and simplicity of use and maintenance has also been faced with Language-driven approaches [4, 16]. The specific focus of these approaches is to promote the explicit design and implementation of domain-specific languages [25] as a key feature of the development process. For

* The Projects TIN2004-08367-C02-02 and TIN2005-08788-C04-01 and the Regional Government of Madrid (4155/2005) have partially supported this work. Thanks to the CNICE for the game design documents and graphics assets provided.

R. Harper, M. Rauterberg, M. Combetto (Eds.): ICEC 2006, LNCS 4161, pp. 153 – 164, 2006.

each application domain a suitable language is designed and an interpreter or compiler for such a language is constructed. Being domain-specific, the language can be easily understood and used by the experts in the domain (usually non programmers) in order to *specify* applications in such a domain. Therefore the approach inherits the usability of the production processes based on authoring tools. On the other hand, the approach also provides great flexibility: As long as the design of the languages is an intrinsic activity of the approach, existing languages can be modified and extended and new ones can be created to meet the changing needs of the user community. The main shortcomings of the approach are the costs associated with initial language design and implementation.

The costs associated with language-driven approaches can be amortized in domains where a whole application family (i.e. a set of variants of a single application model) must be produced and maintained [7]. Application families are very frequent in the domain of videogames, where it is possible to distinguish many genres of games, each of them including many common traits that can be abstracted in the design of a domain-specific language or tool. A good example of uniformity is the genre of graphic adventure videogames [13]. In fact, the <e-Game> project provides language-driven methods, techniques and tools for facilitating the production and maintenance of graphical adventure videogames to be applied in education.

In this paper we describe the language-driven approach to the construction of videogames, and we exemplify the different stages of this approach with the experiences gathered from the <e-Game> project. In section 2 we describe the language-driven development approach itself. In section 3 we illustrate this approach with <e-Game>. Finally, section 4 gives the conclusions and lines of future work.

2 A Language-Driven Approach to the Development of Videogames

The rationale behind any process aimed at the language-driven construction of videogames is to provide the advantages of using an authoring approach for each particular family of games, while preserving the flexibility provided by using a full featured programming approach. Indeed, the approach must be conceived as providing an authoring tool (the domain-specific language itself) adapted to the specific needs of each domain, and even of each situation. Besides, a feasible approach must keep the costs associated with the design, implementation and maintenance of domain-specific languages within reasonable limits.

For this purpose, an incremental approach can be adopted, extending and modifying both the language and its implementation in order to adapt them to different situations and families of games. The approach that we propose in this paper encourages such an incremental strategy.

Following our previous experiences with the document-oriented approach to the production and maintenance of content-intensive applications [23, 24], we propose the three views shown in Fig 1 for the characterization of this approach. Next subsections give the details.

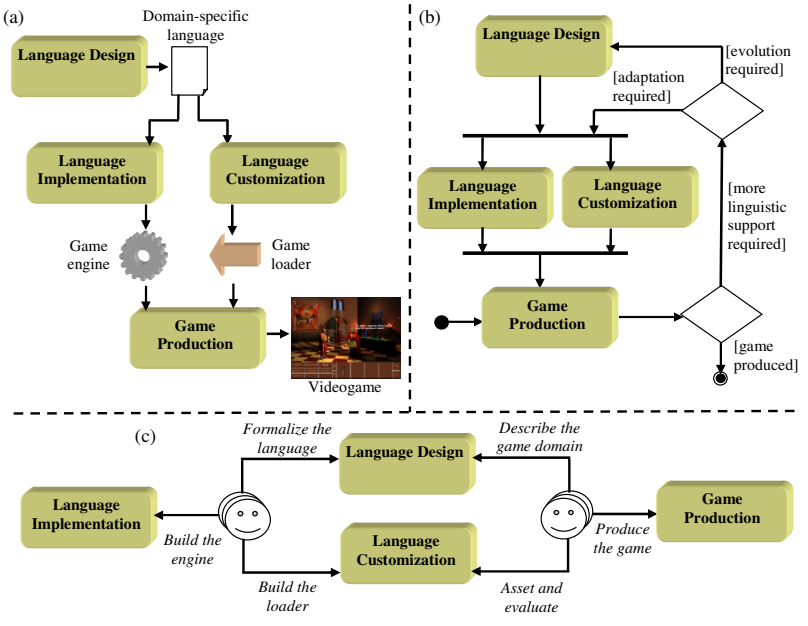

Fig. 1. Language-driven approach to the construction of videogames: (a) products and activities view; (b) sequencing view; (c) participants and roles view

2.1 Products and Activities

The most characteristic activity in the language-driven approach is the *language design* activity. In this activity a suitable *domain-specific language* is designed for the production of the videogames. For this purpose, the activity addresses two different aspects of the language:

- The language's *abstract syntax*. This syntax can be thought of as the information model representing the data of each sentence in the language that is relevant to the subsequent processing (i.e. translation and/or interpretation) of such a sentence [9]. The focus on abstract syntax when designing a domain-specific language simplifies the other design steps and the implementation activity, since the emphasis is put on processing structured data instead of on raw strings. It also keeps the language and the engine independent from the format finally used during authoring, and therefore facilitates reusing the language in different scenarios by defining suitable concrete (textual or even visual [14]) syntaxes. Finally, for reasonably narrow genres (e.g. the domain of graphic adventures addressed) the resulting languages are usually simple and declarative, which induces simpler abstract syntaxes.
- The language's *operational semantics*. This is a formal and implementation-independent characterization of how games described in the language are executed. Therefore this description sets up the basis for building the game engine. The formalization of this semantics is very useful in anticipating the more obscure aspects of the language's dynamic behaviour without being obfuscated by technological and/or implementation details, and also in allowing rapid prototyping.

The *Language implementation* activity deals in turn with the construction of a *game engine*. This engine will be based on the description of the language produced during language design, and especially on the operational semantics.

Another important activity is *language customization*. In this activity a suitable *concrete syntax* is chosen for the language, and this syntax is actually implemented in terms of a *game loader*. This loader performs a translation of games expressed in a specific syntax into a representation compliant with the abstract syntax, thus allowing their execution using the engine. The activity is very valuable for increasing the usability of the language for different communities. Furthermore, the loader can be turned into a full-featured editing and authoring environment for the domain-specific language provided.

Finally, once the game engine and loader are made available, these artefacts can be used for producing specific *videogames* during the *game production* activity.

2.2 Sequencing of the Activities

We promote an incremental process model for the language-driven construction of videogames. In this model languages are not conceived as pre-established, static and unmovable entities, but are dynamic objects that evolve according to the authoring needs that arise during the development of more and more videogames. This evolutive nature alleviates the costs associated with language design and implementation, since these activities can be interleaved with the production of videogames. In addition, it contributes to more usable and affordable languages, since the complexity of the languages is also in accordance with the expressive needs manifested during their application. It avoids the premature inclusion of useless or unnecessarily sophisticated constructs in the languages.

Sequencing of the activities in our approach is in accordance with this pragmatic posture. As depicted in Fig 1b, new development iterations start when new games must be developed or existing ones modified or extended. These production stages can be interrupted in order to refine the game engine and/or the concrete syntax and the associated game loader, in order to introduce a completely new concrete syntax together with the associated loader, or in order to resolve a lack of expressiveness in the current language. This latter situation supposes undertaking a new language design activity, followed by the corresponding implementation and customization of the newly defined features.

2.3 Participants and Their Roles

Our approach explicitly involves two different communities of participants in the development process: *game experts* and *developers*. Game experts group the experts in the different aspects involved in the development of a videogame not directly related to programming (e.g. storyboard writing, graphical design, musical composition, design of the game-play, etc.). Developers in turn represent experts in the different aspects related to programming and software development. This separation of roles, which is taken from our previous experiences with the aforementioned document-oriented approach, is mandatory for a successful application of the approach, although the separation itself is not a novelty in the videogame industry.

In the Language-driven approach, the main responsibility of domain experts is during game production, because they will be the main stakeholders who are in charge of producing the videogames using the language and supporting tools available (the engine and the loader). In turn, the main responsibility for developers is during the language implementation activity, where they build the game engine. The other activities (language design and customization) require active collaboration between experts and developers. During language design, experts must collaborate in order to let developers formulate the right language. For this purpose, developers can use well-known domain analysis methods [2] tailored to the particular domain of videogames. In addition, narrowing the game genre and using rapid prototyping based on the language's operational semantics can help in harmonizing this collaboration. Finally, during language customization experts help developers to define suitable syntaxes. Once the syntax is stable, the construction of game loaders is a routine task that can be tackled by using standard techniques and tools in compiler construction [1] or standard linguistic frameworks like those alluded to in [25].

3 The <e-Game> Project

<e-Game> is a project oriented to the provision of means for the authoring and the deployment of graphic adventure videogames with educational purposes [15, 18]. This project was conducted under the language-driven directives described in the previous section. In this section we detail the consecution of the project in terms of the products yielded by such a language-driven approach.

3.1 The <e-Game> Language: Abstract Syntax and Operational Semantics

During the design of the <e-Game> domain-specific language we chose a characterization of the domain of graphic adventure videogames guided by the instructional uses envisioned [13]. As a result of an initial domain analysis we depicted the main features of the games to be built:

- Adventures occur in a world made of *scenes*. Typical examples of scenes might be a bedroom, a tavern or a street. Scenes have *exits* that lead to other scenes. Besides, some of these scenes are *cutscenes*: fixed scenarios that can be used to include special events in the game flow (e.g. playing a videoclip).
- The player is represented inside the game world by an avatar that plays the leading role in the adventure and navigates the different scenes as commanded by the player.
- The world is populated by *characters*. The player maintains *conversations* with such characters. Conversations follow the model of multiple-choice dialog structures organized as a tree with the player's possible answers as nodes that open new sub-conversations.
- The scenes can contain several *objects*. Good examples of objects might be an axe, a sword, a key or a chest. The player is allowed to perform several types of actions with the objects: he/she can *grab* objects and add them to its personal *object inventory*, *use* objects in his/her inventory with other objects, and *give* inventoried objects to some characters.

Information item	Intended meaning
<scene,*s*>	*s* is a scene.
<cutscene,*cs*>	*cs* is a cutscene.
<start,*s*>	Scene or cutscene *s* is the game's starting point.
<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.
<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.
<object,*s*,o,*c*>	Object *o* is visible in the scene *s* provided that *c* holds.
<grab,*o,c,es*>	If condition *c* holds, object *o* can be grabbed. The effect is the achievement of *es*.
<use-with,o_s, o_t ,*c,es*>	Object *o* can be given to character *ch* when condition *c* holds. Then effects *es* are achieved.
<give-to,*o*, *ch* ,*c,es*>	Object o_s can be combined with object o_t provided that condition *c* holds. Then the 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.

Fig. 2. Main types of information items that constitute the abstract syntax for the <e-Game> language

- Games include a declarative notion of *state*, which is mainly based on boolean propositional variables called *flags*. Flags 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, activate an exit or initiate a conversation with a character. When a flag holds, it is said to be *active*. Otherwise, it is said to be *inactive*. Conditions will be expressed as *conjunctive normal forms* on the flags (i.e. as sequences of alternatives made of elementary activation and deactivation conditions).
- Finally, the different actions undertaken by the player can produce several *effects*. These effects can be of different types: (i) activation of a flag; (ii) consumption of an object in the inventory as a consequence of combining it with another or giving it to a character; (iii) speech uttered by a character when he/she receives an object; and (iv) triggering a cutscene. Notice that in <e-Game> it is not possible to *deactivate* flags. Intuitively, a flag represents an achievement, which cannot be unachieved.

The abstract syntax for an <e-Game> language able to express these features can be easily characterized as set of tuples, which will be called <e-Game> *information items*. In Fig 2 the main types of information items comprised by this abstract syntax are summarized (for the sake of simplicity we omit information items used for presentational purposes, such as for instance the coordinates and the dimensions of an exit, or the external assets required to render a character). 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. In turn, effects are represented as a list of tuples representing the individual effects (e.g. <activate,*f*> for activating the flag *f*). Lists themselves are 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 are represented as lists of tuples with the basic conversation steps. Option lists in the conversation are represented as lists of pairs of the form <*o*,*conv*>, where *o* is the text of the option and *conv* is the sub-conversation that follows. In Fig 3 the representation of a fragment of a conversation using this abstract syntax is depicted. This conversation concerns a simple educational game about safety regulations at work.

```
<conversation, Foreman,
                <<speak-char, "Well José, did you measure the
                              scaffold">,
                 <<response,
                  <<"No sir, not yet",
                    <<speak-char, "And what are you waiting for, boy?">,
                    <<speak-player, "At once, sir">,
                    <end-conversation,<>>>>>,
                  <<"Yes sir, it's ready",
                    <<speak-char, "And...">,
                    <<response,
                     ...  >,
                 {{SecondTaskInitiated, UsedMeasureTapeScaffold},∅}>
```

Fig. 3. A fragment of conversation represented in the abstract syntax of the <e-Game> language

This abstract syntax, although reasonably readable, is not intended to be used by authors directly. It sacrifices clarity for a representation that allows a reasonably straightforward definition of its formal operational semantics. The operational semantics for the <e-Game> language is specified using the style of *structural operational semantics*, a common method of specifying the operational semantics of artificial computer languages [19, 22].

In Fig 4 we show some of the semantic rules for the <e-Game> language that formalize the behaviour associated with conversations. In such rules, expressions of the form $\vdash \Phi$ are used to introduce both generic set-theoretical statements that must hold and <e-Game> specific predicates that are readily translated into such kind of statements (such *translation* is addressed by other semantic rules). In turn $s_o \rightarrow s_1$ is used for denoting a basic transition between execution states. Such states are represented as 5-tuples with the form $\langle\theta,G,\sigma,in,out\rangle$ where: (i) θ is a set of attribute-values pairs which represent the *control state* of the execution; (ii) G is the set of information items that represents the game; (iii) σ is the *game state*, which contains a pair for the active flags and another one for the objects in the inventory; (iv) *in* is an *input stream* that contains the commands representing the player's interactions; and (v) *out* is an *output stream* where the commands for producing the game's presentation are written.

The notation of both the abstract syntax and the operational semantics is oriented to allowing a relatively effortless implementation of the <e-Game> engine and facilitating prototyping processes [5]. As we will see later, the abstract syntax can be customized to a more usable language with a lower cognitive overload for authors.

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

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

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

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

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

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

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

Fig. 4. <e-Game>'s semantics rules associated with conversations

3.2 The <e-Game> Engine

The architecture of the <e-Game> engine is depicted in Fig. 5. The design of this architecture was driven by the operational semantics of the <e-Game> language. According to this architecture, the engine is comprised of the following elements:

- 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: *control rules*, which roughly correspond to the semantic rules of the <e-Game> operational semantics, and *GUI components*, which implement interaction and presentation services to support the final presentation layer of the videogame.
- The *game controller*, which encodes the operational behaviour of the engine. This controller can be customized with an appropriate set of control rules, and it includes a *control driver* that implements the selection and application strategies for such rules.
- The *user interface*, which takes care of the basic interactions with the player and of the presentation of the game. This interface is customized with a suitable collection of GUI components, and its behaviour is controlled by a pre-established *GUI shell*.
- *Input* and *output streams* that connect the controller and the user interface.
- A *game generator*. This artifact processes the game's information items, selects the appropriate game components, and registers them in the core and user interface of the engine. This component is in turn architected with an extensible set of *configuration rules*.

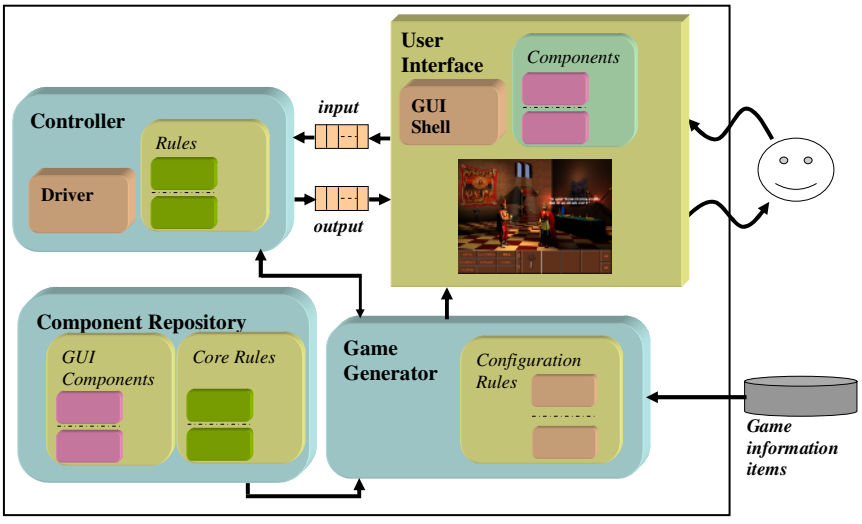

Fig. 5. Architecture of the <e-Game> engine

This architecture is modular enough to accommodate evolutions in the <e-Game> language, as we realised at earlier design stages of this language.

3.3 The <e-Game> Language as a Descriptive Markup Language for Game Storyboards

We have customized the <e-Game> abstract language to yield a XML-based descriptive markup language [3, 6] that can be used to directly mark up the game storyboards. This customization facilitates the production and maintenance of the graphic adventure videogames for game writers. Indeed, being descriptive and mirroring the structure of the storyboards, the language is easily understandable to game writers, which can use it to make the structure of their storyboards explicit. Besides, the markup can also be used to refer to the art assets required to display the different game components. These assets can be provided by another community of experts (the artists).

As a consequence of markup-based customization, a collaboration model for the development of graphical adventure videogames arises, which is based on our previous work on the document-oriented approach. The development process is ruled by game writers, although it also involves the other participants (including artists and developers) in a rational way. According to this model, game writers prepare the storyboards in plain English and then they mark them up with the <e-Game> markup language. In this process they can be advised by developers regarding the most abstract aspects of the language (e.g. specification of conditions), and also by artists regarding the aspects related to the art assets (e.g. coordinates and other presentational information). In Fig. 6a we depict the concrete syntax for describing conversations (we use XML DTDs notations for the sake of brevity, although the syntax is actually

formalized using an XML Schema, in order to facilitate its future maintenance and evolution). In Fig. 6b we show a fragment of storyboard for the safety regulations game that corresponds to the conversation already represented in Fig. 3. In Fig. 6c we show the result of marking up this fragment.

The game loader for this concrete syntax has also been provided with a modular architecture to facilitate its extension and evolution. This architecture is based on the model for the incremental construction of processors for domain-specific markup languages described in [24].

(a)
```
<!ELEMENT conversation (%dialogue;, %continuation;)>
<!ATTLIST conversation id ID #REQUIRED>
<!ENTITY % dialogue "(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?)>
```

(b)
```
F: Well José, have you
     measured the scaffold?
->J: No sir, not yet
     F: And what are you
              waiting for,
                      boy?
     J: At once, sir
->J: Yes sir, it's ready
     F: And...
```

(c)
```
<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> (…)
```

Fig. 6. (a) Formalization of the markup for conversations; (b) a fragment of conversation in a storyboard; (c) markup of the fragment in (b)

3.4 Production of Videogames with <e-Game>

Until now we have applied <e-Game> in the development of some educational videogames about safety regulations and work risks prevention (Fig. 7a), and also in several experiences carried out in collaboration with the CNICE, the Spanish National Center of Information and Educative Communication, the biggest repository of educational games in Spain (Fig. 7b). These experiences consisted of reusing the instructional design and art assets of the videogames corresponding to a *History of Music* course provided by the CNICE. During these experiences we substantially improved the different products in <e-Game>, realizing the benefits of the incremental language-driven approach. We also realized that game writers could easily learn and dominate the operational and presentational aspects of the language to the point of maintaining <e-Game> documents on their own. When they reach the adequate level of proficiency, they can proceed with little support from the other participants (developers and artists).

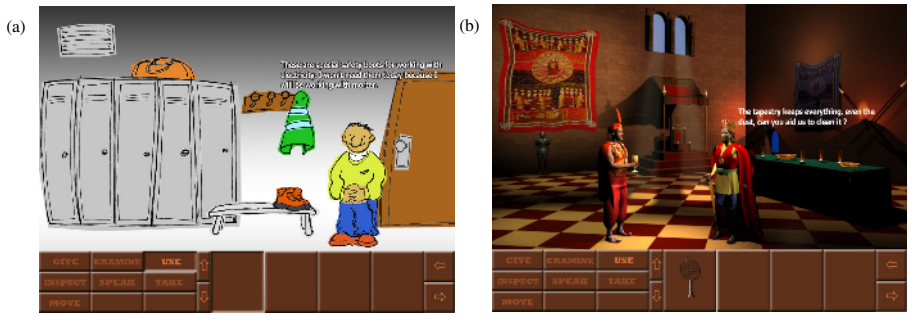

Fig. 7. (a) Snapshot of the safety regulations videogame; (b) Snapshot for *Hall of the Kings*, a little educative game developed by reusing assets provided by CNICE

4 Conclusions and Future Work

In this paper, we have presented a language-driven approach to the development of videogames. This approach promotes the incremental definition of domain-specific languages in order to deal with the particular features of each family of videogames. We have successfully experienced the feasibility of the approach with <e-Game>, a project for the development of graphic adventures with an instructional purpose. On the negative side we must highlight the costs associated with the design, implementation and customization of the languages. We must also indicate as a drawback the highly specialized skills required of developers, who must have rather specialized skills regarding language design and implementation technologies.

We are currently further improving <e-Game> by exploring alternative (visual) concrete syntaxes. We also want to extend the language to explore alternative conversation models. As future work we are planning to apply <e-Game> to alternative domains other than the educational (e.g. advertising and diffusion of ideas). We also want to apply the language-driven approach to other game domains with educational purposes, like interactive fiction or interactive simulation.

References

1. Aho, A., Sethi, R., and Ullman, J.D., Compilers: Principles, Techniques and Tools. Adisson-Wesley (1986).
2. Arango, G., Domain-Analysis: From Art Form to Engineering Discipline. ACM SIGSOFT Notes. Vol. 14(3). (1989).
3. Bray, T., Paoli, J., Sperberg-McQueen, C.M., and Maler, E. Extensible Markup Language (XML) 1.0. W3C Recommendation (2000) Available: www.w3c.org March 27th, 2006.
4. Clark, T.E., Sammut, P., and Willans, J. An eXecutable Metamodelling Facility for Domain Specific Language Design, in The 4th OOPSLA Workshop on Domain-Specific Modeling. Vancouver, Canada (2004).
5. Clément, D., Despeyroux, J., Despeyroux, T., Hascoet, L., and Kahn, G., Natural Semantics on the Computer, in Rapport de Recherche N 416. INRIA Sophia Antipolis: Valbonne, France (1985).

6. Coombs, J.H., Renear, A.H., and DeRose, S.J., Markup Systems and the Future of Scholarly Text Processing. Communications of the ACM. Vol. 30(11). (1987) 933-947.
7. Coplien, D., Hoffman, D., and Weiss, D., Commonality and Variability in Software Engineering. IEEE Software. Vol. 15(6). (1998) 37-45.
8. DIV Community Website. Available from: http://www.divsite.net March 27th, 2006.
9. Friedman, D., Wand, M., and Haynes, C.T., Essentials of Programming Languages Second Edition. MIT Press (2001).
10. Harbour, J. and Smith, J., Beginner's Guide to DarkBasic Game Programming. Premier Press (2003).
11. Ierusalimschy, R., Figueirido, L.H., and Celes Filho, W., LUA-An Extensible Extension Language. Software Practice & Experience. Vol. 26(5). (1996) 635-652.
12. Jenkins, H., Klopfer, E., Squire, K., and Tan, P., Entering the Education Arcade. ACM Computers in Entertainment. Vol. 1(1). (2003).
13. Ju, E. and Wagner, C., Personal computer adventure games: Their structure, principles and applicability for training. The Database for Advances in Information Systems. Vol. 28(2). (1997) 78-92.
14. Marriott, K., Meyer, B., and Wittenburg, K.B.A., Survey of Visual Language Specification and Recognition, in K. Marriot and B. Meyer (eds), Visual Language Theory. Springer-Verlag. (1999).
15. Martinez-Ortiz, I., Moreno-Ger, P., Sierra, J.L., and Fernández-Manjón, B. Production and Maintenance of Content Intensive Videogames: A Document-Oriented Approach, in International Conference on Information Technology: New Generations (ITNG 2006). Las Vegas, NV, USA: IEEE Society Press (2006).
16. Mauw, S., Wiersma, W.T., and Willemse, T.A.C., Language-driven System Design. Int. J. of Software Engineering and Knowledge Engineering. Vol. 14(6). (2004) 625-664.
17. Mills, G. Casual Games. International Game Developers Association White Paper (2005) Available from: http://www.igda.org/casual/IGDA_CasualGames_Whitepaper_2005.pdf March 27th, 2006.
18. Moreno-Ger, P., Martinez-Ortiz, I., and Fernández-Manjón, B. The <e-Game> project: Facilitating the Development of Educational Adventure Games, in Cognition and Exploratory Learning in the Digital age (CELDA 2005). Porto, Portugal (2005).
19. Mosses, P.D., Modular Structural Operational Semantics. Journal of Logic and Algebraic Programming. Vol. 60-61. (2004) 195-228.
20. Ousterhout, J.K., Scripting: Higher Level Programming for the 21st Century. IEEE Computer. Vol. 31(3). (1998) 23-30.
21. Overmars, M., Teaching Computer Science through Game Design. IEEE Computer. Vol. 37(4). (2004) 81-83.
22. Plotkin, G.D., A Structural Approach to Operational Semantics, in Tech. Report DAIMI FN-19. Computer Science Dept. Aarhus University (1981).
23. Sierra, J.L., Fernández-Manjón, B., Fernández-Valmayor, A., and Navarro, A., Document Oriented Development of Content-Intensive Applications. International Journal of Software Engineering and Knowledge Engineering. Vol. 15(6). (2005) 975-993.
24. Sierra, J.L., Navarro, A., Fernández-Manjón, B., and Fernández-Valmayor, A., Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices. Vol. 40(12). (2005) 28-37.
25. Van Deursen, A., Klint, P., and Visser, J., Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices. Vol. 35(6). (2000) 26-36.
26. Water Cooler Games Web Site. 2006; Available from: http://www.watercoolergames.org March 27th, 2006.