Model Driven Game Development - Case Study A MTC for maze game prototyping

Lina Marcela Morales, David Méndez-Acuña, and Willy Montes

Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes, Colombia
{lm.morales70,df.mendez73,wa.montes28}@uniandes.edu.co

Abstract. Serious games are a potential strategy for acquiring learning skills and knowledge. Although games are mainly designed for entertainment, serious games add the learning and research-support value. The inherent complexity of these education-oriented games demands an elaborated design and implementation process, which represents a risk in terms of cost, quality and maintenance. In this paper we present an approach for serious game development, based on the model-driven game development techniques. Our approach aims at reducing the complexity of game design and implementation by separating the conceptual environment of the game and its concrete implementation. We first define the abstract models that characterize the structure and behavior of the game. Then, with the software architecture model and the platform-specific model, we transform the abstract models into concrete implementations. The resulting prototypes are generated completely (100%).

Keywords: Education in engineering, serious games, model-driven, software development, model-driven games development.

1 Introduction

In an educational context, serious games have become a potential strategy for acquiring learning skills and knowledge. Although games are mainly designed for entertainment, serious games add the learning and research-support value.

The term *"serious game"* was primarily defined as a game with "an explicit and carefully thought-out educational purpose [...] not intended to be played primarily for amusement" [1]. Serious games are designed and developed according to an explicit purpose and a specific audience, with the intention of supporting game-based learning. In theory, any video game can be perceived as a serious game depending on its actual use and the player's perception of the game experience [2]. In this sense, any game could be serious depending on its educational utility.

Game-based learning could be an invaluable learning approach for the 21st century, but is currently hindered by the lack of availability of serious games to support this innovative approach [3]. Although many serious games have been developed recently, the use of these games in an educational context is not globally accepted. According to this statement, there is a visible need for developing,

not only serious games, but also tools and techniques for facilitating the design and implementation of these education-oriented games.

The increasing complexity of game development also highlights the need for tools to improve productivity in terms of time, cost, and quality [4]. The development of serious higher education games has proven to be complex, time consuming and costly [5]. These statements not only agree with the need for developing game-support tools and techniques, but also with the need of reducing game development costs and time.

As an opportunity, developments in software engineering that enable automatic generation of software artifacts through modeling promises new hope for game-based learning adopters. Specially, those with little or no technical knowledge to produce their own serious games for use in game-based learning [3]. In game prototyping, abstract models allow designers to specify the particular game concepts and behavior, without addressing platform-specific issues, and then, through transformations and complementary modeling, the prototype could be generated in a specific platform. This approach separates the conceptual level of the game from its concrete implementation; the less changeable part of a software game design is given from a game-centered expert, and then, it is transformed in a specific-platform prototype.

In this paper, we propose a methodology for developing a serious game using MDGD approach. A platform-independent model (PIM) defines the structure and behavior of the game, a second PIM defines the software architecture of the game, and two platform-specific models (PSM) describe the game control mapping. A semi-automatic model transformation chain (MTC) is executed from first to last of these models to generate two software prototypes of the game. We use a maze game scenario, an MVC pattern for defining the software architecture, and the platforms JavaSE and Torque2D for prototyping. We use ATL, QVT, and Acceleo for transformations and code generation.

The reminder of this article is structured as follows: Section 2 describes the state of the art of MDGD and game prototyping. Section 3 explains the methodology we used to generate the two prototypes based on MDGD approach. Finally, section 4 exposes the results and conclusions obtained from our case study.

2 Related work

Model-driven software development (MDSD) is an approach in which models are considered as first-class citizens. Because of its automation cababilities, games-development community has adopted the ideas of MDSD. This combination of approaches is termed as "model-driven game development" (MDGD).

In MDGD models are seen as the basis for game design, development, and maintenance. Within this approach, there are different experts that focus on different aspects of the game. Some of them are experts in the conceptual level of the game, and other ones are responsible for the concrete implementation of the game prototypes. Design and implementation are separated, which implies

a significant reduction in time and costs during game development and maintenance.

MDGD has some commercial advantages to consider. First, the productivity and reusability of software artifacts increases. This fact allows a reduction in terms of game development time and costs. Second, middleware mechanisms that are specific to game development, such as game engines, continue to be encouraged. Finally, the interoperability and the portability between game platforms increase [4].

Research on game prototyping and MDGD have been developed recently [1] [3] [4] [7] [8]; however, there is a long way for continuing researching and exploring on this topic. Table 1 presents some related work within MDGD prototyping.

Table 1. Related work in MDGD prototyping

Related work	**Description**	**Example**
Theoretical-based	Some related work is focused on making theoretical arguments for serious games development.	[1]
Model-driven based (non-technical)	Some related work is focused on developing model-based frameworks for developing serious games without notice of technical issues.	[3]
MDGD-based	Some related work is focused on proposing methodologies for serious game development based on MDGD approach. They are based on a PIM for describing the structure and behavior of the game and a PSM for specifying the platform-specific game control mapping.	[4]
Context-based	Some related work is focused on the development of serious games for context-specific purposes. Games are mainly designed for human-life supporting.	[7]

According to the previous table, the related work identified is mainly focused on the separation between the conceptual meaning of the game and its concrete implementation. In contrast, our approach is not only focused on this separation, but also on the following:

1. **Reducing the scope to increasing the generated-code percentage.** We intend to show that games can be grouped according to its structure and behavior. Specifically, we characterize maze-based prototypes (*e.g.* Packman and Bomberman) and we reduce the scope of our approach to this kind of games. Then, we generate the game-code from abstract models that capture structure and behavior.
2. **Incremental refinement of models.** We focus on a multi-stage model transformation chain. We include different steps during the model transformation chain. In one prototype generation process we take into account the

software architecture of the game, in the other we do not. Steps can be added or removed according to the specific needs of the implementation.

3. **Platform-specific prototyping.** We develop two platform-specific prototypes. We generate one prototype in Java, principally for java project code and descriptors generation, and a second prototype in Torque2D to show that platforms are not always code-based.

3 Our approach

This section presents our model-based game prototyping methodology. We first present an overview, and then, we present our approach step by step.

3.1 Overview

Figure 1 illustrates an overview of our approach. It consists of a MTC that takes a high-level model, that describes the structure and behavior of the game, and, across several refinement steps, generates the game-code in two different platforms.

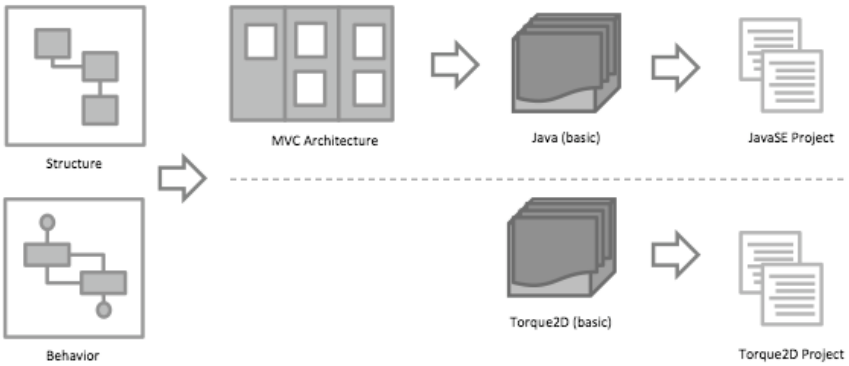

Fig. 1. Overview

Each artifact is considered below:

1. The **structure/behavior** model is a PIM that represents the structure (*i.e.,* elements of he game, players, laberynth, etc) and behavior (*i.e.,* interactions, controls and movements) of the game.
2. The **MVC architecture** model is a PIM for characterizing the elements of the game in terms of models, views and controllers.
3. The **Java** and **Torque2d** models are PSMs that describe the elements of the game in terms of particular platforms. The former corresponding to Java swing applications. The latter to the Torque2D games engine.

4. Finally, **platform-specific templates** are specified for code generation.

The implementation of the MTC is developed over the Eclipse Modeling Framework (EMF). Metamodels are written in Ecore; model to model transformations are specified in ATL and QVTo; and model to text transformations are built using Acceleo templates.

3.2 General description of the game

As we said above, we focus on a generic maze game scenario for presenting our approach. A maze game consists of a matrix that represents the play zone. The matrix determines the different positions where objects can be placed. The game turns around a main character that has to develop a particular purpose during the game. Generally, the main character is controlled manually. Depending on the game, the main character must interact with other elements in order to achieve the goal. These elements can be dynamic or static depending on their behavior. Dynamic elements develop actions (e.g., ghosts in Bomberman), and static elements are providers (e.g., cookies in Pacman) or elements for limiting the play zone (i.e. walls). In general, both dynamic and static elements are controlled automatically.

Figure 2 illustrates the objects we used in our maze game scenario, including the player and the dynamic/static elements.

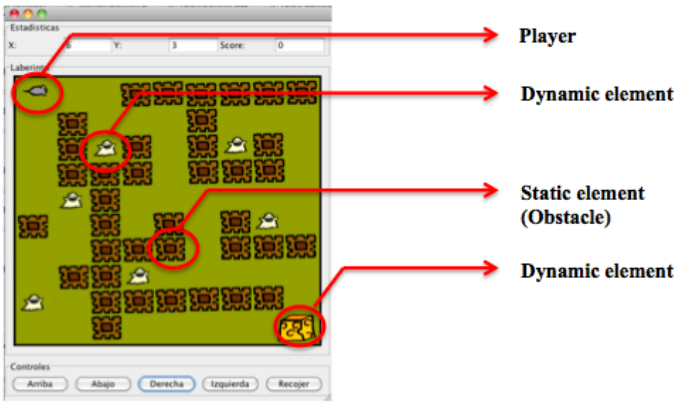

Fig. 2. Maze-game scenario

3.3 Sructure/behavior model

The first step consists of specifying a metamodel for representing the basic structure and behavior of the game. This metamodel contains the conceptual elements that characterize the game, without addressing platform-specific issues.

In order to specify the metamodel, we need to identify and classify the core domain elements, characterize them in terms of properties and behaviors, and determine how they interact with each other. Table 2 presents the core elements identified.

Table 2. Maze game core elements.

Element	**Description**	**Example**
Mazegame	Represents a maze game.	Bomberman
Element	Represents an object that can be placed on the play zone. It can be static or dynamic.	
Static element	Represents a provider or an element used for limiting the play zone.	Wall
Dynamic element	Represents an element that develops specific actions.	Ghost
Property	Represents a property that characterizes an element.	Position
Rule	Represents a guide for assigning element behavior.	
Assignation rule	Represents a rule for assigning property values.	Add 1 life point to the main character
Event rule	Represents a rule for assigning property values when two or more elements interact with each other.	The main character disappears when it crashes with a ghost
Player	Represents the player of the game.	
Keyboard	Represents the control mechanism of the main character behavior.	Press 'Z' to move faster

At this point, we are able to express the conceptual environment of the game in a well-defined notation. We use platform-independent models with the aim of separating the less unchangeable meaning of the game from the concrete, platform-specific implementations.

We specify the structure and behavior metamodel, doing the following:

1. Each element described in Table 2 is specified as a meta-class.
2. Properties are specified as meta-attributes.
3. Behaviors are specified as meta-references.

We first define a root concept called MazeGame. This root concept contains the general properties of the game (name, author, width and height) and is constituted by all existing game elements.

Some other concepts are specified: a player concept that represents the individual who interacts with the game, a set of elements for representing the active

and passive objects, a set of properties for element characterization, and a set of rules for describing element behavior. Keyboard concepts are added for game control mapping. Fig.4 illustrates the final structure and behavior metamodel of the game.

The following table describes the concepts included in the structure and behavior metamodel in terms of their meta-attributes and meta-references.

In this section, we developed a metamodel for describing the structure and behavior of the game. In view of the fact that game meaning is separated from specific-platform implementation, modeling enforces the reusability and changeability of the conceptual environment of the game, without addressing technical issues. Figures 3 and 4 show the structure/behavior model splited in two for visualization facilities.

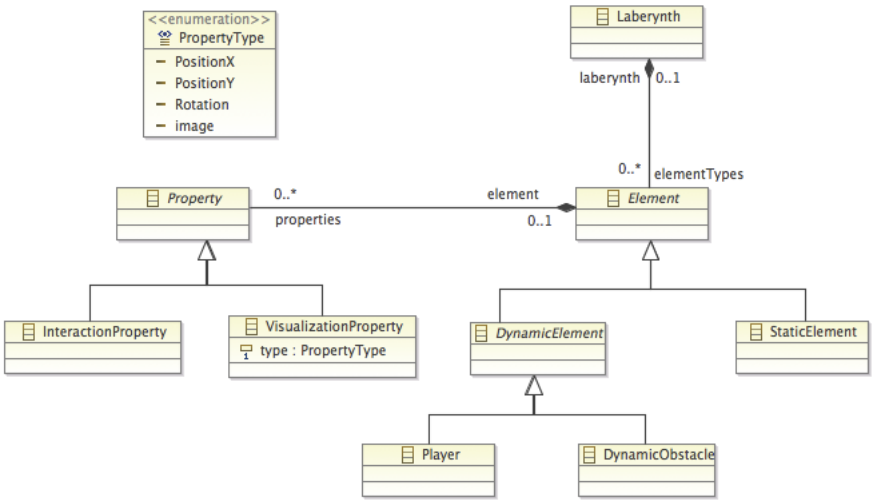

Fig. 3. Maze-games structure metamodel

3.4 JavaSE Prototype

In this section, we describe the process for generating the java prototype. Basically, we focus on the software architecture of the game, and then, selecting a specific-platform, we generate the prototype. The process consists of the following:

1. A PIM is specified for characterizing the software architecture of the game.
2. A transformation is executed from the structure and behavior PIM to (1).
3. A PSM is specified for describing the platform-specific artifacts.
4. A transformation is executed from (1) to (3).

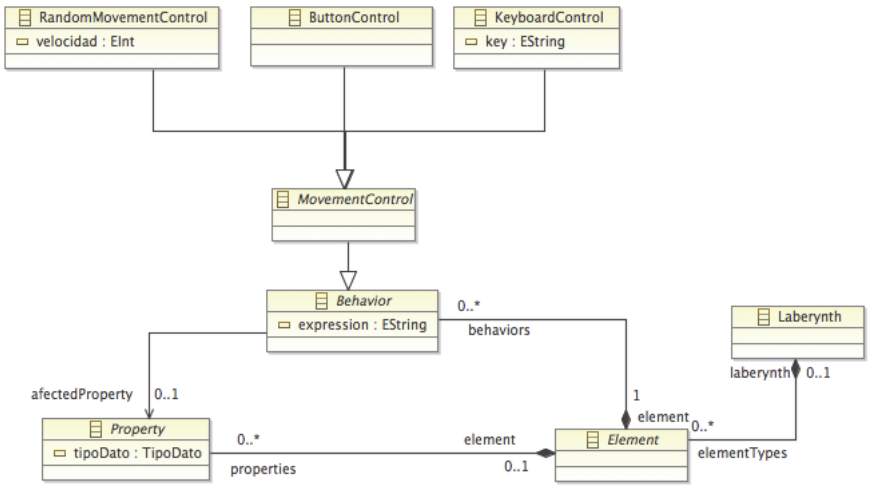

Fig. 4. Maze-games behavior metamodel

5. Platform-specific templates are specified for code generation.
6. A transformation is executed from (3) to (5): the prototype is generated.

Figure 5 illustrates the java prototype generation process

First, we specify a model conformed to the **structure and behavior metamodel** (See Section 3.3). Afterwards, we execute a first transformation to produce a second model conformed to the software architecture metamodel. Then, we execute a second transformation to produce a third model conformed to the JavaSE (basic) implementation. Finally, we use the platform-specific templates to execute a third transformation and produce the java prototype code and descriptors.

We implement a MVC pattern for (1), we select JavaSE platform for (3) and (5), and we use ATL for specifying model-to-model transformations as in (2) and (4).

Software architecture PIM (MVC-Based) is a metamodel for representing the MVC-based software architecture of the game. Although this metamodel contains some software-related concepts, it is not concerned with platform-specific issues.

In order to specify the metamodel, we need to identify, classify and characterize the core domain elements, as in the structure and behavior PIM. Table 4 presents the core elements identified.

At this point, we are able to express the basic architectural environment of the game in a metamodel. Figure 6 illustrates the MVC-based software architecture metamodel.

The MVC-based software architecture metamodel is formed by a model (the conceptual environment of the game), several views (observers of the model) and the controller (corresponding to the element that manages the actions over the

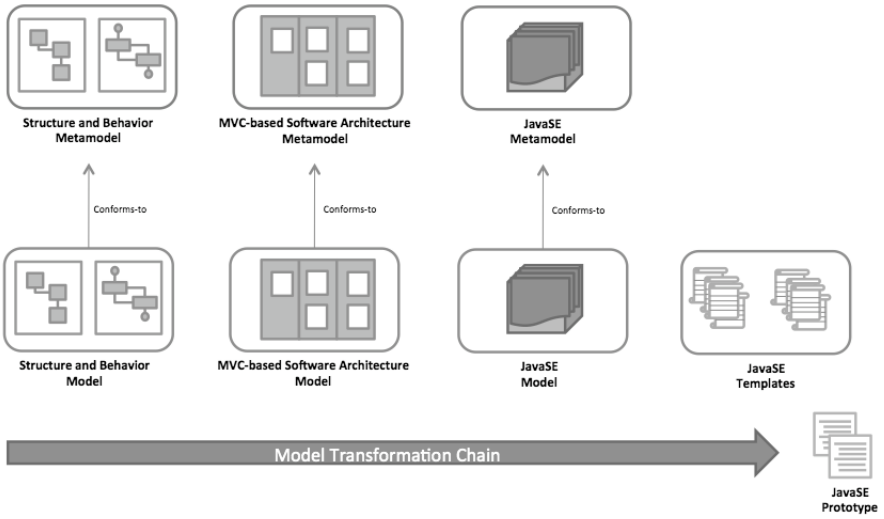

Fig. 5. Java prototype generation process

Table 3. MVC-based software architecture: core elements.

Element	**Description**
Architecture	Represents MVC-based software architecture.
World	Represents the conceptual environment of the game.
GUI Window	Represents the visualization environment of the game.
Controller	Represents the mediation between the conceptual world and its particular visualization.
Panel	Represents a zone in which visualization elements could be located.
Graphic element	Represents a visualization element.

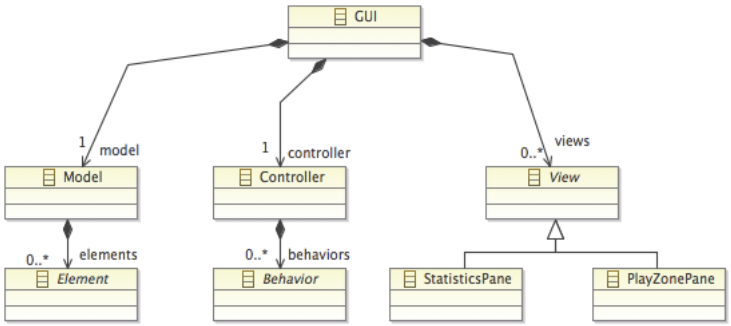

Fig. 6. MVC-Based architecture metamodel

game). The root concept of the metamodel is the GUI, which contains a Model, views, and a Controller. The first and the second refer to the conceptual and visual environment of the game respectively; and the third acts as a mediator between them.

The GUI is the front-end of the game. It contains panels, which are containers of graphical elements. Panels can be classified as statistics panels and visualizacion panels. The first classification is used for those panels that show data about the state of the game (i.e. position of the main player, life points, etc). The second one is used to graphically display the game elements (i.e. matrix, ghosts). The controller refers to the behaviors of the elements. Those behaviors mafe affect some the properties of the elements. The Model is formed by a set of elements, which are described in terms of properties and behaviors. Properties are fields with parameters and return values, and behaviors are queries that allow elements to perform actions.

The above metamodels are related via a model to model transformation called **Structure/behavior to MCV-Based architecture**. The structure/behavior model is transformed to the mvc-based architecture model. Notice that this transformation is a refinement step that takes the elements of a maze-game and put them in terms of mvc-based architecture elements.

Then, the **mvc-based architecture model is transformed to JavaSE model** using a model to model transformation. In general, we mapped each model element to a class, and each field and behavior to an attribute. The abstract concepts are mapped into interfaces, and the references between concepts are mapped into associations or attributes (in case it is a field of an element). The GUI part of the conceptualization is mapped also as classes and associations.

Finally, the **JavaSE code is generated from the JavaSE model** via a model to text transformaiton. To do so, we create templates for defining the different java artifacts (i.e. class files, project descriptors).

3.5 Torque2D Prototype

In this section, we describe briefly the Torque2D prototype generation. The process consists of the following:

1. A PSM is specified for describing the platform-specific artifacts.
2. A transformation is executed from the structure and behavior PIM to (1).
3. Platform-specific templates are specified for descriptors generation.
4. A transformation is executed from (1) to (3): the prototype is generated.

Figure 7 illustrates the torque2D prototype generation process.

First, we specify a model conformed to the structure and behavior metamodel (see section 3.3). Afterwards, we execute a first transformation to produce a second model conformed to the Torque2D (basic) implementation. Finally, we use the platform-specific templates to execute a second transformation and produce the Torque2D prototype files and descriptors. The Torque2D prototype generation process is similar to the JavaSE implementation; however, it differs in some aspects, presented in Table 5.

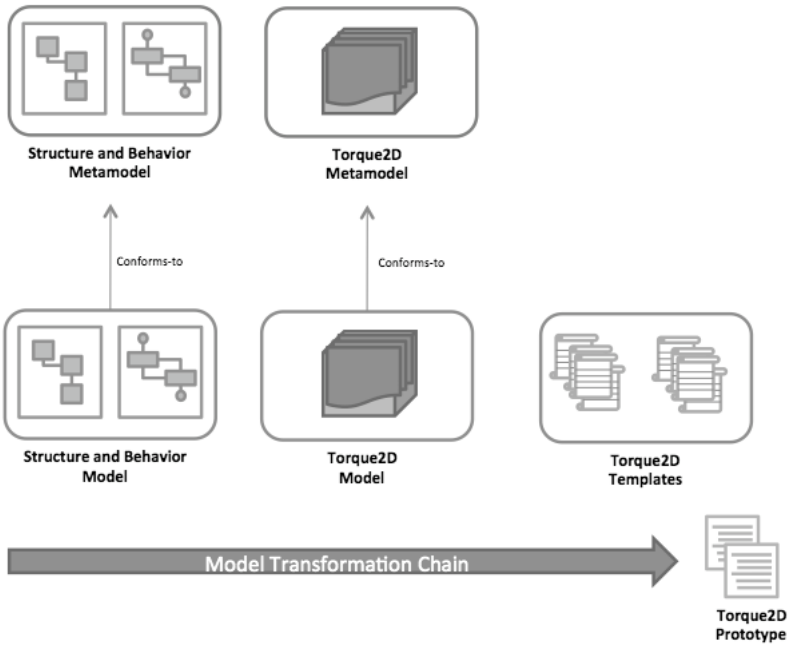

Fig. 7. MVC-Based architecture metamodel

Table 4. Principal differences between Java prototype and Torque2D prototype generation.

Java Prototype	**Torque2D Prototype**
The resulting prototype is a project that contains the classes, folders, and descriptors.	The resulting prototype is a set of files that describe the properties and behavior of the game elements.
We include MVC-based software architecture metamodel.	Software architecture is not taken into account.
We use ATL for the specification of the model-to-model transformations.	We use QVT for the specification of the model-to-model transformations.

The shown differences imply that other models can be added or removed from the model transformation chain, depending on the particular needs. Increasing the level of abstraction and separating the conceptual environment from the concrete implementations where the key for a better understanding of the domain, and a guide for reusability and changeability support.

4 Conclusions

In this paper, we proposed a methodology for developing a serious game based on a generic maze game definition using MDGD approach. We defined a platform-independent model (PIM) for defining the structure and behavior of the game, a second PIM for defining the MVC software architecture of the game, and two platform-specific models (PSM), one for JavaSE and the second one for Torque2D implementation, for describing the game control mapping. We constructed a semi-automatic model transformation chain (MTC) from first to last of these models to generate two software prototypes of the game, JavaSE and Torque2D prototypes respectively.

To our amusement, the methodology proposed for developing the maze game prototypes shows very satisfactory results: The prototypes were generated completely (100%).

Because the conceptualization of the maze game is developed, software designers and platform-specific experts may focus only on the game software development, without notice the conceptual or behavioral determination of the game. This not only separates the two sides: design and implementation, but also facilitates the agile development of the particular games.

The development of more generalized queries for behaviors, and the use of multi-player remain as future work. Furthermore, more complex platform-specific implementations could be added to this maze game prototyping in order to evidence the time and cost reduction of its development.

5 Acknowledgements

We would like to thank all the people who shared their ideas and supported us in any respect during the completion this work. Particularly, we acknowledge the advice and guidance of professor Rubby Casallas.

References

1. Sorensen, b.H., Meyer, B., 2007. “Serious games in language learning and teaching-a theoretical perspective”. In Proceedings of the 2007 Digital Games research Association Conference. pp. 559-566.
2. Egenfeldt-Nielsen, S., Smith, J.H., Tosca, S.P., 2008. “Understanding Video Games: The Essential Introduction, Routledge”.

3. Tang, S. Hanneghan, M., 2010. “A model-driven framework to support development of serious games for game-based learning”. In Proceedings of the 3rd International Conference on Developments in eSystems Engineering, DeSE 2010. Pp. 95-100.
4. Montero, E. Carsí, J., 2009. “Automatic Prototyping in Model-Driven Game Development”. ACM Comput. Entertain. 7, 2, Article 29 (June 2009), 9 pages.
5. Westera W., Nadolski R.J., Hummel H.G.K., I.G.J.H., 2008. Wopereis. “Serious games for higher education: a framework for reducing design complexity”. Educational Technology Expertise Centre, Open University of the Netherlands, Heerlen, The Netherlands.
6. Petrillo, F. Pimiena M., 2009. “What Went Wrong? A Survey of Problems in Game Development”. ACM Comput. Entertain. 7, 1, Article 13 (February 2009), 22 pages.
7. Gobel S., Hardy S., Wendel V., Mehm F., Steinmetz F., 2010. “Serious Games for Health – Personalized Exergames” MM’10, October 25–29, 2010, Firenze, Italy. ACM.
8. Serious Games Summit. Available at ¡http://www.gdconf.com/conference/sgs.html¿