Eberos GML2D: A Graphical Domain-Specific Language for Modeling 2D Video Games

Conference Paper · October 2010

DOI: 10.1145/2060329.2060342

2 authors:

Frank Hernandez
Florida International University
11 PUBLICATIONS **59** CITATIONS
SEE PROFILE

Francisco Raul Ortega
Florida International University
27 PUBLICATIONS **185** CITATIONS
SEE PROFILE

Eberos GML2D: A Graphical Domain-Specific Language for Modeling 2D Video Games

Frank E. Hernandez
Rounin Labs
Miami, FL 33175, USA
hernandez.f@rouninlabs.com

Francisco R. Ortega
Rounin Labs
Miami, FL 33175, USA
ortega.f@rouninlabs.com

ABSTRACT

The complexity of game development has increased in the past 30 years, from a task that could almost be entirely handled by a single programmer to an endeavor requiring a large team. To reduce this complexity, we have developed a Domain-Specific Language (DSL) targeting the modeling of two-dimensional (2D) games. We call this language Eberos Game Modeling Language 2D (**Eberos GML2D**[1]). By raising the level of abstraction through modeling, we allowed a simpler specification of the game, and reduced the time and programming efforts. In order to evaluate our approach, we modeled two games and compared the difference between the amount of work required to write the game from scratch and the amount required using our proposed language. These evaluations yielded promising results of 86.4% savings on programming effort, and 82.3% savings on programming time.

Keywords

Graphical Domain-Specific Language, Model-Driven Development, Video Game Modeling Language

1. INTRODUCTION

Over the past 30 years, and especially so the past 10 years, games have become so large and complex that they can no longer be developed by single-man teams [12, 3]. Even though this level of complexity is reduced by the use of game engines, game development still remains a complex, time and resource-consuming task, taking teams from between two to three years to complete a single title [11]. We believe that the game industry could benefit from applying Model-Driven Development (**MDD**) approaches by raising the level of abstraction of the game-development process. We propose a graphical Game Modeling Language (**Eberos GML2D**[2]) that represents the game in an intuitive fashion. The abstraction provided by our graphical language also allows the models to be translated into game engine level code or code to be used by underlying libraries. This means that the same game can be produced for multiple platforms. Finally, we reduce the number of lines of code that must be written by game developers by automating the creation of repetitive code.

[1]A video with the current implementation of the editor for Eberos GML2D can be found at: `www.cs.fiu.edu/~fhern006/Projects/EberosGML2D_Editor.html`

[2]A video on how to model Pong using Eberos GML2D can be found at: `www.cs.fiu.edu/~fhern006/Projects/EberosGML2D_Editor_2.html`

Game development is a multi-disciplinary process, bringing together: graphics, sounds, input, and networking. This means that developers must implement code to interact with the sound card, video card, input devices, and network devices as well as implement the code for the game itself. Most game engines already provide support for interacting with the platform's hardware, but few offer any kind of support for the development of the game code or logic.

In order to validate the expressiveness of our language, we decided to model two completely unrelated games. The first game, Pong, is the minimal game considered. Pong is composed of two paddles, one on each side of the screen, and a ball. The goal of the game is for each player to try and bounce the ball past the opposing player's paddle in order to score. The second game modeled is SpaceKatz, a title currently under development by the members of Florida International University's (**FIU**) SIG-Game, and it is a game of medium complexity consisting of menus, enemies and levels. The goal of the game is for the player to navigate his/her ship through the levels, avoiding obstacles and destroying the enemies along the way to beat a final boss. This game includes multiple menus, enemies, and levels, thus an obvious choice to demonstrate the current capabilities of our proposed language.

The remainder of this paper is organized as follows: Section 2 describes the entire **Eberos GML2D**. Section 3 presents the design of the experiments, our results and what findings we had. Section 4 presents the related work.

2. MODELING A 2D GAME

In this section we describe the requirements for a game modeling language, which was distilled after several discussions from domain experts and five years of game development experience of one of the authors. Then, we present the Eberos Game Modeling Language 2D (**Eberos GML2D**), which is our proposed language to meet our stated requirements. In particular, we have identified the following requirements for the game modeling language:

Simplicity: It must be simple and intuitive.

Platform-independent: It must be independent of any underlying game platforms.

Library/Game Engine-independent: It must be independent of game engines and/or development libraries.

Expressiveness: It must be able to model a large majority of game development scenarios.

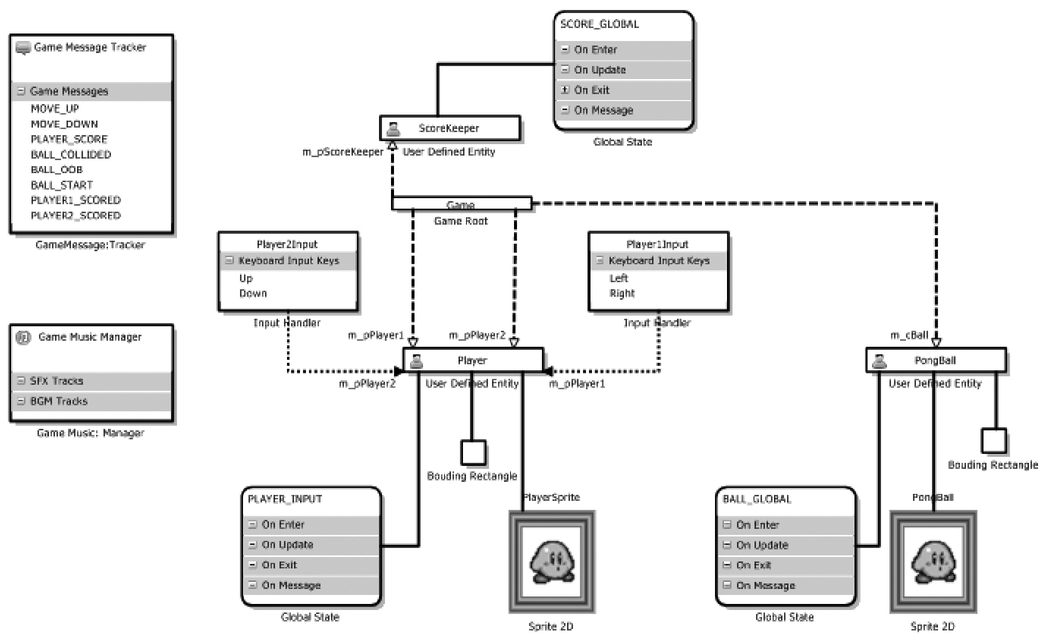

Figure 1: Eberos GML2D Model for the Game of Pong.

2.1 Sprite and Animations

A sprite in the 2D game development domain refers to a two-dimensional image that contains the graphic representation of a single item in the game. Sprite sheets contain multiple consecutive smaller sprites that can be used by the game developer to create the illusion like movement or explosions, among other illusions. A 2D game without any images or movement can be very dull at best, so any game modeling language must provide support for modeling graphical behavior.

In our language, a sprite is represented by the **Sprite2D** construct. This contains information about the initial location in the file system of the image or resource file. A **Sprite2D** also contains information about the height, width, and X and Y positions on the screen of each individual sprite. The size and position information are used when a sprite has no specified animations. If the sprite has an animation, then the size and position information of the current animation replaces the information specified in the Sprite2D.

Animations are used to graphically represent the actions that are performed by game entities on the screen, such as swimming. In **Eberos GML2D**, each **Sprite2D** can have zero or more animations which can be of one of two kinds: **SimpleAnimations** or **CompositeAnimations**. In **Eberos GML2D**, **SimpleAnimations** are composed of uniform frames, ordered from left to right, all together representing a graphical action taken by a game entity. A **SimpleAnimation** contains information about the location on the resource image of the first frame of the animation, the delay of each frame and the total number of frames in the animation. **CompositeAnimations** give more freedom to the modeler when it comes to modeling a sprite's animations by allowing the specification of the animations on a frame-by-frame basis. In our language, animation frames are modeled using the **Frame2D** construct and only **CompositeAnimations** are composed of one or more **Frame2Ds**. **Frame2Ds** contain information about the position on the resource file and the size of a single frame of the animation. **CompositeAnimations** differ from **SimpleAnimations** in that they can be composed of frames of different sizes, represented by **Frame2Ds**.

2.2 Entities

Every game is composed of entities of one kind or another. While sprites are graphical representations of game components, entities are the program representation of these components. These entities can represent players, items, enemies, and menus, among other components. The same way that games revolve around the concept of entities so does our language.

Currently, **Eberos GML2D** supports the modeling of **UserDefinedEntities** and **EntityPools**. **UserDefinedEntities** allow the modeler to model custom game entities, and are represented using the **UserDefineEntity** construct. In our language, entities can contain zero or one sprite, thus allowing the modeler to represent both graphical and non-graphical entities. While graphical entities may represent players, and enemies, non-graphical entities can be used to model those entities that cannot be seen by the game player, such as triggers or area boundaries. In our language, **UserDefinedEntities** can be composed of zero or more entity references; this allows for the definition of atomic as well as composite entities while modeling the games.

Entities in a game can either be controlled by the player **(PC)** or by the program, as in the case of non-playable characters **(NPC)**. In **Eberos GML2D**, **InputHandlers** are used to model the user's input handling logic. Each entity reference that can be controlled by the player's input will have an **InputHandler** attached to it in the model. The current version of the language supports only the modeling of keyboard input, but the ability to model mouse input and console controller input are currently being explored. Keyboard input is modeled by providing the **InputHandler** with a **Keyboard:Key** for each key to handle. Each **Keyboard:Key** can only map to a single key and is selected from a keyboard key enumeration containing every key in the keyboard. These allow the user to specify which key the **InputHandler** will handle, as well as what **GameMessage** the input handler will send the controlled entity when that key is affected (**GameMessages** will be covered later in this section). **Keyboard:Keys** can be modeled to detect one of four possible key states; **DOWN**, **RELEASED**, **WAS_PRESSED**, and **IS_HOLDING**. This allows the user to model a specific behavior, depending on the state of the key. One might want the player character to move left for as long as the "Left" key is pressed (**DOWN**) but might only want the player's character to attack every time the "Space Bar" is pressed (**WAS_PRESSED**). The means for modeling the behavior of program-controlled entities are described in the following section.

2.3 Logic

Having the ability to model the game logic for game entities was one of the main motivations for developing **Eberos GML2D**. While working on a previous project, we realized that a lot of the code used for modeling our game entities' logic was being constantly reused, and only a small section was modified slightly for each specific entity. This led us to develop a language for modeling this logic, which in turn led us to explore further and see how much of the game could be modeled, thus giving birth to **Eberos GML2D**. The current version of **Eberos GML2D** supports modeling of Moor finite state machines with one concurrent level of execution. This will not remain for long, as we continue to expand our language to allow for modeling of goal-based agents [13], as well as other kinds of artificial intelligence (**AI**) agents.

In our current version of the language, game entities may possess at most one **GlobalState**, and at most one **State** [1]; these represent the initial states of the FSMs for this entity type. In **Eberos GML2D**, finite state machines are modeled by connecting states together via the use of a **TransitionMessage**. A **TransitionMessage** represents the message that triggers a transition from one state to another. Each entity can have at most one global state machine, and at most one current state machine. In our language, transitions can only occur between states of the same type. **States** can only transition to **States**, and **GlobalStates** can only transition to **GlobalStates**. In **Eberos GML2D**, **States** are used to model the logic of a game entity, and can also be used to model the AI or behavior of computer-controlled game entities. **GlobalStates** are similar in behavior to **States**, but instead they model behavior that may occur during the execution of any state of the entity.

Our language also allows the user to insert previously-generated scripts to further increase the state's logic; these are provided in the form of **ActionScripts**. **ActionScripts** come in four varieties: **ActionScript: OnEnter**, **ActionScript: OnUpdate**, **ActionScript: OnExit**, and **ActionScript: OnMessage**. Each **ActionScript** contains information about the script's ID, the script's import order and the script's location in the file system. The import order of a script represents the location where the script should be inserted in the final code in relation to other inserted scripts. Each of these scripts adds code to the state's respective section **OnEnter**, **OnUpdate**, **OnExit**, and **OnMessage**, and is assumed to have been validated by an external tool. From all of the scripts, **ActionScript: OnMessage** is the only one that is handled in a different fashion; this represent snippets of code that will only be executed when the specified **GameMessage** is received.

ActionScripts allow the modeler to add behavior to the entity logic that can not be modeled with the current version of **Eberos GML2D**. **ActionScripts** are expected to be valid code for the underlaying platform on which the final generated code will be running. The current nature of these scripts both add expressiveness and restricts our language. Since the **ActionScript** are directly copied over during the translation of the model, this means that the game model can only be translated to a specific platform, thus reducing the number of platforms to which the model can be translated. At the same time, this script allows for support of behavior that could not otherwise be expressed with the current version of our language.

2.4 Collision Detection

Collision detection refers to the act of detecting whether or not two or more game entities have come into contact (collide) with one another. There are a few methods for handling collision detection in 2D games: bounding boxes, bounding circles, and pixel-level collision, just to mention a few of the most common ones. **Eberos GML2D** supports modeling of two of these, bounding circles and bounding rectangles (bounding boxes). In our language, entities may contain zero or one bounding object which can either be a **BoundingRectangle** or a **BoundingCircle**.

BoundingRectangles are used to model collision detection between entity boxes. A **BoundingRectangle** represents a bounding box for a game entity, which is used in the game to detect when a collision has occurred. Currently, the user can specify both the width and the height of each entity's bounding rectangle. Similarly, **BoundingCircles** are used to model collision detection between circles. The user specifies the radius of the circle for the entity's collision detection.

2.5 Game Controllers

Game controllers allow the user to model game managers and similar control units, such as music managers and message managers. The **GameRoot** construct represents the entry point into the game. All games must originate at the game root; this represents the Main in many programs. The **GameMessage:Tracker** contains all messages available in the game. Any message handled or triggered by game entities must be registered with the **GameMessage:Tracker**; this ensures that all messages exist in the final generated game code. **GameMessages** represent a single type of message in the game, usually an enum or value. Games can be composed of multiple messages, each representing a kind of information to be sent to or handled by game entities. Last but not least, we have the **GameMusic:Manager**; it allows the **Eberos GML2D** user to model all the music and sound effects in his/her game. This allows the user to specify the sound resource that is needed by allowing him/her to specify the location in the operating system. Upon translation, all the resources are placed in a local directory, and their correct path is written along with the code generation. This automation has been shown to reduce the number of code errors occurring from incorrect resource paths. The user can model sound effect tracks using the **SFXTrack** construct, which allows him/her to set the in-game ID of the track and graphically specify the resource file. Sound effect tracks are used for short sounds used inside the game, a gunshot or an explosion. Similarly, longer music tracks can be modeled with the **BGMTrack** construct. A **BGMTrack** represents a single background music track. These tracks are used for playing songs or similar scenic music. **BGMTracks** are much longer in duration than **SFXTracks**.

3. EXPERIMENTAL EVALUATION

In this section, we describe the experiments we performed to evaluate our language **Eberos GML2D**. We first present the design of our two experiments, then the results, followed by a discussion of our findings. Since our intent is to reduce the amount of both time and work required to develop 2D games, we use two metrics in our experiments: time and lines of code (**LOC**) required to develop each game with and without **Eberos GML2D**. We used Microsoft's XNA game engine for developing the games from scratch. The generated models were also translated into XNA Code as well.

3.1 Procedure and Scenarios

In order to evaluate **Eberos GML2D**, we developed each of the following game specifications first without using our approach, and then a second time using our DSL. We then timed both entire processes from start to completion. The second metric we used was the lines of code (**LOC**) of each version of the game, DSL vs. Non-DSL. Since SpaceKatz (Game Specification 2) is an attempt to model a game being developed by the students of SIG-Games at Florida International University (**FIU**), we decided to use their existing project as a comparison instead of writing this game from scratch ourselves. We believe this gives us a more accurate set of data by comparing our approach with a real ongoing game-development process.

Game Specification 1 - Pong: Pong is as simple a 2D game as they come; it consists of two paddles (one for each player located at each end of the screen) and a ball. The ball bounces off the walls and the player's paddle. The goal of the game is to get the ball past the opponent's paddle in order to score.

Game Specification 2 - SpaceKatz: SpaceKatz is a 'Shoot 'em Up' style of game, where the player must pilot through asteroid fields while destroying enemies in the process. This game consists of one main screen with the game logo. Following the logo screen, the player is presented with a menu where he/she can select what options to set or directly start the game. After selecting to start the game, the player is presented with a menu to choose the level or area to play. Once an area is selected the game begins, and the player is presented with the spaceship to pilot. A player can move in all four directions by pressing the keyboard keys (Left, Right, Up, Down); he/she can also shoot missiles by pressing the "Space Bar" key on the keyboard. The game ends when the player loses all of his/her lives.

3.2 Experiment Setup

In this section we present the set up for both or our experiments. The results in lines of code represents the number of lines of code in XNA/C# required to complete both the game by hand and the game using **Eberos GML2D**.

Set Up Experiment 1 - Pong: First the game of Pong was implemented by hand, then it was modeled and implemented, both by the main author of this paper.

Set Up Experiment 2 - SpaceKatz: For the experiment of SpaceKatz, we used an existing project from SIG-Games. We gathered the information of lines of code and time taken from this existing project then we preceded to model the current state of this game and program it using our language. In the case of SpaceKatz the game without our proposed language was developed by the members of SIG-Games while the game implemented using **Eberos GML2D** was written by the main author of this paper.

Note: We feel it is important to note that SIG-Games is a group dedicated to training students to become game developers, and that past members have worked on AAA titles such as **Bioshock 2**.

3.3 Results

In order to perform the experiment, we developed a graphical editor for **Eberos GML2D** using the Microsoft DSL Tools [4]. In order to show the efforts saved using our approach compared to implementing directly, we have summarized our results in Table 1 and Table 2. We also implemented a code generator to transform **Eberos GML2D** models into Microsoft XNA code. The numbers given represent the two metrics we used when evaluating our language: time and lines of code (**LOC**).

For each game explored, we show the actual effort of directly implementing the game, and the actual effort of implementing the game using our language. In Pong, we could reduce 39 (131 - 92) lines of code (**LOC**) written by the user and, as a result, 29% (39/131) of the amount of work required by the user was reduced. Table 1 also shows a savings of 3 (34 - 31) minutes, a 8.82% (3/34) savings on the time required to develop the game by using **Eberos GML2D** as opposed to implementing it without it. We also present the results of applying our approach to SpaceKatz, a more complex game than Pong. Using our approach as seen in Table 2, we were able to reduce 3303 (3822 - 519) lines of code written by the user, and save 86.4% (3303/3822) of the amount lines of code as opposed to implementing the game directly. There was also a significant reduction of 24.67 (30 - 5.33) hours required, a savings of 82.3% (24.67/30) of the time spent

Table 1: Effort for creation of Pong using Eberos GML2D vs. manually developed

Game: Pong	Eberos GML2D	Manually Developed
Development Time(Hours)	0.13 Modeling + 0.38 Implementing = 0.51 total	0.56 Implementing
Effort in lines of code(LOC)	92 User Generated + 74 XNA Generated + 1870 Auto Generated = 2036 total	131 User Generated + 74 XNA Generated = 205 total

Table 2: Effort for creation of SpaceKatz using Eberos GML2D vs. manually developed

Game: SpaceKatz	Eberos GML2D	Manually Developed
Development Time(Hours)	0.51 Modeling + 4.81 Implementing = 5.32 total	30.0 Implementing
Effort in lines of code(LOC)	519 User Generated + 74 XNA Generated + 5546 Auto Generated = 6139 total	3822 User Generated + 74 XNA Generated = 3896 total

creating the game by using our approach as opposed to implementing the game without it.

3.4 Discussion

From Table 1 and Table 2, we can see that the amount of savings varies greatly depending on the complexity of the game. We attribute the difference in the effort percentage between Pong 29% and SpaceKatz 86.4% to the expressiveness of our language, and the level of complexity of the games directly. The game of Pong consists mostly of interactions. As a result of this, while we were able to model a large set of the user input game representation, there was also a bit of game behavior that could not be modeled with the current version of **Eberos GML2D**. In comparison, with a more complex game like SpaceKatz, which requires a lot of implementation of user and game logic, as well as enemies' AI, we obtained a better savings of 86.4% less lines of code. With the logic section of **Eberos GML2D**, we were capable of modeling a large section of the agent logic for the enemies, menus, and player, something that had to be otherwise implemented without using our approach. Similarly, we were able to model the entities, bullets, asteroids, and pool of entities using **Eberos GML2D**, which had to be implemented for non-DSL version of the game.

4. RELATED WORK

In this section, we discuss some of the existing works that share our goal: abstracting the complexity of game development through modeling. Prior work on reducing the complexity of the game development process includes tools like: **Unreal Kismet** [5, 2], **Game Maker** [10], and above all **SharpLudus** software factory [6, 8]. **SharpLudus**, the closest of these approaches to ours, is explored in detail at the end of this section.

4.1 Unreal Kismet

Unreal Kismet is a visual scripting system that can be used to create complex scripted sequences quickly and easily, with little programming knowledge [5, 2]. This provides a higher level of abstraction from the **Unreal Engine** [2], and provides game designers with the ability to model their game's interactivity without much programming knowledge.

Unreal Kismet [5, 2] allows users to create objects and entities without making a distinction of graphical representation. Since **Unreal Kismet** is designed to model the interactivity of levels in the **Unreal Engine** [2], it has no means of modeling the graphical representations of game entities. Unlike **Unreal Kismet**, our approach **Eberos GML2D**, makes a clear distinction between sprites and animations, which are the representations of the graphical components. This allows the user of **Eberos GML2D** to model the appearance of the game as well as some of the behavior of the entities.

One aspect that **Eberos GML2D** can improve on is the concept of "combinations" used in **Unreal Kismet** [5, 2]. The approach referred as "combinations," includes gates and delays. The gates allow an impulse to either flow or stop the flow using "open", "close" and "toggle" modes. The delays are impulses, which are held for a set amount of time. One example of delays is for a shooting game, where, for every three seconds, a gun will fire toward the player. Our current version of **Eberos GML2D** provides little support for modeling the actions of the game entities, and thus can benefit from applying some of these concepts used in **Unreal Kismet**.

4.2 SharpLudus

The closest work to **Eberos GML2D** is **SharpLudus**, a domain-specific language approach [6, 8]. It presents a domain-specific language called SharpLudus Game Modeling Language (**SLGML**), and is aimed at modeling video games in the 2D adventure genre. **SharpLudus** can create a game with some restrictions without the need to code while generating .Net 2.0 C# code.

A restriction with **SLGML** is that it only models adventured games. At first glance, this decentralized approach of having multiple modeling languages: one for adventure games, one for racing games, one for shooting games, may seem the proper approach. However, the definition between adventure, racing or shooting games may be a blur, especially if one game designer may consider developing a hybrid design where different aspects from all these games are used to make a new game or even a new genre. In the previous example, using the decentralized approach mentioned, since the adventure DSL knows nothing about the racing DSL, such a game could not be developed. In addition, our belief is that the 2D gaming domain is specific enough to be expressed by a single language [6, 8].

At a superficial level, both languages share some common ideas. For example, they both have entities, **SLGML** being a lot more restricted. There can only exist one main character in **SLGML** restricting the use of simultaneous players. Another shared concept, is the support for sprites, which are common in game development. Sprite support in **SLGML** consists of a list of physical images, each of them having an independent delay and an option to loop the animation. In contrast, in **Eberos GML2D**, the physical link to the image file is created using **Sprite2D**, and animations are modeled with a logical representation using **SimpleAnimations** and **CompositeAnimations**. This means that unlike **SLGML**, where each frame must be loaded into memory when the animation is played [7]; in **Eberos GML2D**, the sprites are modeled with a single resource file, and animations are represented logically within this file.

SLGML is a very restricted domain-specific game modeling language. This restriction can be overcome by allowing the game designer to write additional code for customization. The constraints can explain why the author of **SLGML** needed to modify the game engine, allowing the game engine to adapt to the modeling language. Unlike **SLGML**, the code generated from **Eberos GML2D** models sits on top of current game engine or development libraries, requiring no modifications from these [9, 8].

5. CONCLUSION

We believe that the application of Model-Driven Development [4] techniques can help to reduce the complexity of the game development process. Reducing the complexity of developing games means, among other things, a reduction in the effort required for writing video games. This reduction in effort translates into a shorter time spent programming the game. It is our belief that with the rising complexity of video game development, the methodology currently in use in the industry is near its end. We proposed **Eberos GML2D** in an attempt to reduce the level of complexity of 2D games. While our approach showed some promising results of 84.6%, the question still remains whether or not the domain of 2D games can be fully captured by a Domain Specific Language. Further exploration still remains as we continue to explore the domain and expand our language to model 3D games.

Acknowledgments

We like to thank the members of SIG-Games at FIU for providing us with their data on their development process of SpaceKatz. We also thank Corey Ginsberg for her editorial feedback.

6. REFERENCES

[1] M. Buckland. *Programming Game AI by Example.* Wordware Pub, Plano, Tx, 1st edition, 2005.

[2] Busby, Jason, Z. Parrish, and J. Wilson. *Mastering Unreal Technology Volume 1: the Art of Level Design*, volume 1. Sams Publishing, Indianapolis, 2009.

[3] H. M. Chandler. *The Game Production Handbook.* Infinity Science, Hingham, Mass, 2009.

[4] S. Cook, G. Jones, S. Kent, and A. Cameron Wills. *Domain-specific Development with Visual Studio DSL Tools.* Addison-Wesley, Upper Saddle River, NJ, 2007.

[5] J. Dobbe. A domain-specific language for computer games. Master's thesis, Delft University of Technology, 2007.

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

[7] A. Furtado and A. Santos. Using domain-specific modeling towards computer games development industrialization. *6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*, page 1, 2006.

[8] A. Furtado and A. Santos. Extending visual studio .net as a software factory for computer games development in the .net platform. *2nd International Conference on Innovative Views of .NET Technologies (IVNET06)*, 2007.

[9] A. W. B. Furtado, A. L. M. Santos, and G. L. Ramalho. A computer games software factory and edutainment platform for microsoft .net. *SB Games 2007*, pages 1–29, Jun 2007.

[10] J. Habgood, M. Overmars, and P. Wilson. *The Game Maker's Apprentice: Game Development for Beginners.* Apress, Berkeley, CA, 2006.

[11] J. Novak. *Game Development Essentials.* Delmar/Cengage Learning, New York, 2010.

[12] A. Rollings and E. Adams. *Andrew Rollings and Ernest Adams on Game Design.* New Riders, London, 2003.

[13] S. J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall/Pearson Education, Upper Saddle River, NJ, 2003.