Using Domain-Specific Modeling towards Computer Games Development Industrialization

André W. B. Furtado, André L. M. Santos
Center of Informatics - Federal University of Pernambuco
Av. Professor Luís Freire, s/n, Cidade Universitária,
CEP 50740-540, Recife/PE/Brazil
+55 (81) 21268430
{awbf,alms}@cin.ufpe.br

ABSTRACT
This paper proposes that computer games development, in spite of its inherently creative and innovative nature, is subject of systematic industrialization targeted at predictability and productivity. The proposed approach encompasses visual domain-specific languages, semantic validators and code generators to make game developers and designers to work more productively, with a higher level of abstraction and closer to their application domain. Such concepts were implemented and deployed into a host development environment, and a real-world scenario was developed to illustrate and validate the proposal.

Categories and Subject Descriptors
D.1.7 [**Programming Techniques**]: Visual Programming.

D.2.2 [**Software Engineering**]: Design Tools and Techniques – Computer-aided software engineering (CASE), Software libraries.

General Terms
Design, Standardization, Languages.

Keywords
Computer games, domain-specific languages, visual modeling, software factories

1. INTRODUCTION

Digital games are one of the most profitable industries in the world. According to the ESA (Entertainment Software Association) [1], digital games (both computer and console games, along with the hardware required to play them) were responsible in 2004 for more than ten billion dollars in sales. These impressive numbers are a match even for the movie industry, while studies reveal that more is spent in digital games than in musical entertainment [2]

The digital game industry, however, is as surrounded by success as it is continuously faced by challenges. Software development industrialization, an upcoming tendency entailed by the exponential growth of the total global demand for software, will present many new challenges to game development.

Studies reveal that there is evidence that the current development paradigm is near its end, and that a new paradigm is needed to support the next leap forward in software development technology [3]. For example, although game engines [4], state-of-the-art tools in game development, brought the benefits of Software Engineering and object-orientation towards game development automation, the abstraction level provided by them could be made less complex to consume by means of language-based tools, the use of visual models as first-class citizens (in the same way as source code) and a better integration with development processes.

This paper explores the integration between game development, an inherently creative discipline, with software factories, which are concerned with turning the current software development paradigm, based on craftsmanship, into a manufacturing process. The focus is on how visual domain-specific languages and related assets (such as semantic validators and code generators) can be used in conjunction within a software factory to make game developers and designers to work more productively, with a higher level of abstraction and closer to their application domain.

The remainder of this paper is organized as follows. Section 2 presents current digital games development tools and techniques, and explains their lack of industrialization. Section 3 introduces a game software factory named SharpLudus, which is targeted at a specific game genre. Section 4 details the SharpLudus Game Modeling Language, the most important factory asset, along with its related assets. Section 5 presents a case study named Ultimate Berzerk. Section 6, finally, concludes about the presented work and points out some future directions.

2. CURRENT TOOLS AND TECHNIQUES

A major evolution in game development technologies has occurred since its early days. Starting from assembly language, many tools and techniques evolved, culminating in game engines. This section describes the most used game development technologies and explains why they do not yet completely fulfill industrialization needs.

2.1 Multimedia APIs

Multimedia APIs (Application Program Interfaces), such as Microsoft DirectX [5] and OpenGL [6], are programming libraries that can be used to directly access the machine hardware (graphics devices, sound cards, input devices). Such APIs are not only useful for providing means to create games with good performance, but also for enabling the portability of computer games among devices manufactured by different vendors. Therefore, by using a Multimedia API, game programmers are provided with a standard device manipulation interface and do not need to worry about low-level peculiarities of each possible target device.

Multimedia APIs set a new stage in game development, by empowering programmers with more abstraction to experience an easier game development process. They are heavily used today and certainly will last for a very long time, being used either directly or indirectly.

Nevertheless, while these libraries handle almost all the desired low-level functions, the game itself still has to be programmed. The APIs provide features that are generic for computer games development and do not offer the abstraction level desired by game programmers. For example, they do not provide features to trigger the transition between game states (phases), entity behavior modeling or artificial intelligence. In other words, the semantic gap between game designers and the final code remains too high if multimedia APIs are the only abstraction mechanism used.

Besides that, interaction with such APIs can only be done programmatically, not visually. This approach may prevent automation and productivity in the execution of some tasks (such as specifying the tiles of a tiled background map), which would have to be executed by exhaustive “copy and paste” commands and through counter-intuitive actions.

2.2 Visual Game Creation Tools

With the intention to simplify game development and make it more accessible to a broader range of communities, visual game creation tools were created and soon became very popular. They aim at creating complete games with no programming at all, sometimes by just clicking with the mouse. The end user is aided with graphical and easy-to-use interfaces for creating game animations, defining entity behavior, the flow of the entire game and to add sound, menus, text screens and other resources to the game.

A visual game creation tool can be either generic or focused on the creation of games belonging to a specific game genre, such as first-person shooters, role playing (RPG), adventure games and so on. This last category includes one of the most popular visual game creation tools: RPG Maker [7], presented in Figure 1.

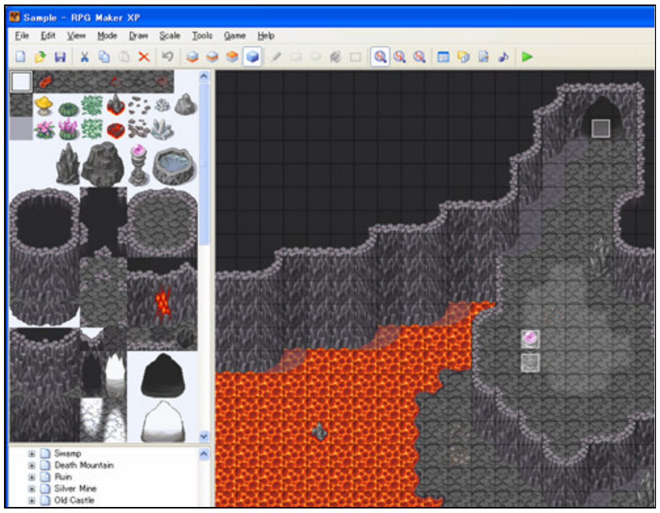


Figure 1. RPG Maker

Being able to finish the creation of a complete game with a few mouse clicks is very impressive indeed. However, although this sounds wonderful at first, the possibilities turn out to be limited. Some types of games can certainly be made, but this approach does not seem adequate for serious games [8]. Visual game creation tools currently do not address the complexity required by the creation of more sophisticated games, and this is reflected by the lack of their adoption by the game industry. Despite being very popular, most users of such tools are beginner and amateur game designers and programmers.

Visual game creation tools try to address such a problem by offering to users script languages, targeted at allowing more complex behaviors to be specified. However, while such languages certainly provide more power to visual game creation tools, they require end-users to learn a new language (perhaps their first language) and to have some programming skills. This may diverge with the original purpose of such tools (to be “visual programming” environments).

Some may say that these built-in languages are not intended to be used by all users, but only by advanced users. But once earning programming expertise, however, users might prefer to have the benefits of true object-oriented programming languages, with the support of robust integrated development environments with full editor and debugging support, instead of working with error-prone scripting languages inside an environment which was not originally conceived for codification.

Besides that, development productivity is much more than having script keywords highlighted. It is composed by a set of complementary concepts, such as refactoring, code and modeling synchronization, test automation, configuration management, quality assurance, real-time project monitoring, domain-specific guidance and organizational process integration, just to mention a few.

2.3 Game Engines

Game engines were conceived as a result of applying Software Engineering concepts to computer games development. An engine can be seen as a reusable API, which gathers common game development foundations (entity rendering, world management, game events handling, etc.) and provides to developers a programmatic interface through which game behavior can be specified. In other words, developers can be more focused on game-specific features, such as its programming logic, intelligence, art and so on.

Differently from the scenario where multimedia APIs are called directly by the computer game code, developers using a game engine are abstracted from low-level game implementation details, while still not being restricted to the limitations of an exclusively visual programming environment. As a matter of fact, the basic game functionalities provided by game engines are built on top of multimedia APIs. Examples of popular game engines are OGRE [9] and Crystal Space [10].

As with visual game creation tools, game engines can be either generic or targeted at a specific game genre. However, in order to be more effective, even generic game engines narrow their target domain by addressing only a subset of all possible computer game genres (for example, a 3D game engine has many specific issues different from a 2D isometric game engine). In fact, the main advantage of using a game engine is that, if it was built in a modular architecture, it can be reused to create a great diversity of games, which consume only the necessary game engine modules [11].

Game engines are the state-of-the-art tools in computer games development. By providing more abstraction, knowledge encapsulation and a reusable game development foundation, they allowed the game industry to reach an unparalleled productivity level. However, as with any technology, some drawbacks can be identified.

First of all, due to the inherent complexity of game engines, it should be noticed that the learning curve for mastering these tools is somewhat high. The demands for understanding the game engine architecture, interaction paradigm and programming peculiarities can turn their use into an unintuitive experience at first. That is the reason why many of today's game engines still present complexity and lack of usability as one of their most cited deficiencies.

Second, using a game engine may involve considerable costs, such as acquisition costs, training costs, customization costs and integration costs [12]. If the discussion is raised from the game developer point-of-view to the game engine developer point-of-view, additional needs for a considerable amount of resources can be identified. Since a diversity of requirements has to be satisfied, creating a game engine is a very complex and expensive task, demanding a substantial infra-structure.

In addition, one of the major difficulties in game engine development is the industrial secrecy. Since such projects involve great investments, many organizations hide their architectures and tools in order to have some advantage over their competitors [13] (for example, it may be difficult to find comprehensive studies about the applicability of design patterns in game engines [11]). Public knowledge regarding the subject, therefore, is only available through open source and academic initiatives. However, it has not been a long time since such initiatives were born, and today's game engine developers are far from having something like "game engine workbenches" to aid the creation of such tools.

2.4 Game Development onto the Next Stage

In general, game development evolution has been compliant with one of the most important software development tendencies: defining a family of software products, whose members vary, while sharing many common features. According to Parnas [14], such a family provides a context in which the problems common to the family members, such as games belonging to a specific genre, can be solved collectively.

If automation in software development is further investigated, it is possible to notice that game engines can still contribute even more to automation in game development. Roberts and Johnson [15], for example, described a recurring pattern that reveals how software development automation, in general, is carried out:

- After developing a number of systems in a given problem domain, a set of reusable abstractions for that domain is identified, and then a set of patterns for using those abstractions is documented.
- Then a runtime is developed, such as a framework or server, to codify the abstractions and patterns. This allows the creation of systems in the domain by instantiating, adapting, configuring, and assembling components defined by the runtime.
- Then languages are defined and tools are built to support the runtime, such as editors, compilers and debuggers, which automate the assembly process. This helps a faster response to changing requirements, since part of the implementation is generated, and can be easily changed.

Game engines are situated in the second of these three "pattern-runtime-language" stages. However, as Roberts and Johnson point out, although a framework (such as a game engine) can reduce the cost of developing an application by an order of magnitude, using one can be difficult. Mapping the requirements of each product variant onto the framework is a non-trivial problem that generally requires the expertise of an architect or senior developer.

Language-based tools (the third stage) automate this step by capturing variations in requirements using language expressions, encapsulating the abstractions defined by a framework, helping users think in terms of the abstractions and generating framework completion code. Language-based tools also promote agility by expressing concepts of the domain (such as the properties or even features of computer games) in a way that customers and users better understand, and by propagating changes to implementations more quickly.

Aligned with the creation of language-based tools, an emerging tendency is to make models first-class citizens for game development, in the same sense that source code already is an essential part of game development. Models can be described by visual domain-specific languages (DSLs) [16], providing a richer medium for describing relationships between abstractions and giving them greater efficiency and power than source code. By using a visual DSL, models can be used not only as documentation but as input that can be processed by tools in other stages of the development process, promoting more automation.

There is evidence, therefore, that game engines can be used together with domain-specific processes, patterns, frameworks, tools and especially languages to create a software factories approach that will situate game development in an industrial stage, by reusing these assets systematically and automating more of the software life-cycle.

3. SHARPLUDUS SOFTWARE FACTORY

In order to illustrate how games development can be turn into a more productive and automated process by means of software industrialization, a software factory named

SharpLudus was conceived. Its product line is focused on the **adventure** game genre, which can be described as a genre encompassing games which are set in a "world" usually made up of multiple, connected rooms or screens, involving an objective which is more complex than simply catching, shooting, capturing, or escaping, although completion of the objective may involve several or all of these. More information regarding the chosen domain is presented in Table 1.

Table 1. SharpLudus Product Line Definition

Feature	Description
Dimensionality	Two-dimensional (2D). World rooms are viewed from above.
User interface	Information display screens containing textual and/or graphical elements are supported. HUDs (heads-up display) can also be configured and displayed.
Game flow	Each game should have, at least, a main character, an introduction screen, one room and a game over screen (this last one is reached when the number of lives of the main character becomes zero).
Sound/Music	Games will be able to reproduce sound effects (wav files) as event reactions. Background music (mp3 files) can be associated with game rooms or information display screens.
Input handling	Keyboard only
Multiplayer	Online multiplayer is not supported by the factory. Event triggers and reactions can be combined, however, to allow two-player mode in a single computer.
Networking	High scores can be uploaded to and retrieved from a web server.
Artificial Intelligence	Enemies can be set to chase the player within a room. More elaborated behaviors can be created visually by combining predefined event triggers and event reactions, or programmatically by developers.
End-user editors	Not supported by the factory. Once created, a game cannot be customized by its players.
Target Platform(s)	PCs running Microsoft Windows 98 or higher

The SharpLudus software factory provides to developers two visual domain-specific languages (DSLs) as assets. The first one is the *Game Modeling DSL*, which together with a room designer and an info display designer allows the specification of the game states flow (info display screens, rooms and their exit conditions).

The second domain-specific language is the *HUD Creation DSL*, which allows developers to specify how useful game information (score, remaining lives, hit points, etc.) will be presented to the player by means of a heads-up display. Both DSLs are provided with validators to ensure that semantic errors are caught in design time and shown in the IDE Error List.

By using the factory DSLs, game designers can create a detailed game specification. However, contrary to common game development approaches, such a specification is a set of "live artifacts". This means that they are not only used for documentation, but they can be transformed into other artifacts by means of automation assets. For example, the VSTO [17] technology is used to create a User Manual skeleton with information extracted from the game specification, while code generators associated to the DSLs can be used to automatically create the majority of the game implementation code. Developers, however, can add their own code to the solution since the factory generated code provides extensibility mechanisms such as partial classes[1] and classes which are just ready for customization (for example, special classes for providing custom event triggers and custom event reactions).

Both the factory generated code and the developer added code interacts with a game engine, which consumes the DirectX Multimedia API. Once the solution implementation is compiled, the factory generates the game executable file and a XML configuration file, through which a high scores web server address and custom developer configuration can be specified. Finally, built-in factory organizational assets, such as the runtimes of the game engine and the multimedia API chosen, are automatically made available by the factory.

In order to illustrate how domain-specific modeling is carried out through the SharpLudus factory, Section 4 details the SharpLudus Game Modeling DSL and Section 5 explores some of its designers through the development of a real-world example.

4. GAME MODELING DSL (SLGML)

The SharpLudus Game Modeling Language (SLGML) is a visual DSL through which the game designer can specify the main game configuration (resolution, screen mode, etc.), game states (rooms and information display screens) and their flow, exit conditions and properties. The SLGML underlying concepts are also manipulated by many factory designers (event designer, entity designer, sprite designer, etc.).

According to Deursen, Klint, and Visser [16], the development of a domain-specific language typically involves the following tasks:

- **[Analysis]** (1) Identify the problem domain; (2) Gather all relevant knowledge in this domain; (3) Cluster this knowledge in a handful of semantic notions and operations on them; (4) Design a DSL that concisely describes applications in the domain.
- **[Implementation]** (5) Construct a framework (library) that implements the semantic notions; (6) Design and implement a compiler that translates DSL programs to a sequence of framework calls. Obs: considering language workbenches and visual modeling, Fowler [18] suggests an additional task to this stage: (7) the creation of a visual editor to let developers to graphically manipulate the DSL. Considering a software factory context, this research also suggests an additional step: (8) the creation of *semantic validators* to identify modeling errors in design time.
- **[Use]** (9) Write DSL programs for all desired applications and compile them.

Tasks (1) and (2) are performed as part of the software factory product line definition and product line design. The next subsections detail the other tasks, aside from task (9), which will be explored by means of a case study presented in Section 5.

4.1 Concepts Design

The *SharpLudusGame* is the root domain concepts of the SLGML DSL. As Figure 2 presents, it is related to six top-level elements, which will not be deeply detailed due to space constraints but are explained below:

- *AudioComponent:* an abstract concept representing every sound that can be reproduced in a SharpLudus game. It is specialized by *SoundEffect* and *BackgroundMusic* concepts.

[1] The concept of partial classes makes it possible to split the implementation of a single class in two files.

- *Entity:* an abstract concept which is the base unit of a SharpLudus game design. It is anything that can react with anything else in any way. It is specialized by *MainCharacter*, *NPC* (non-playable character) and *Item* concepts.
- *EntityInstance:* represents an instance of an entity, containing information such as position, speed, number of remaining hit points, etc.
- *Event:* represents a special condition that occurs to a SharpLudus game, fired by one or more *Triggers* (such as "collision between the main character and a specific item"), and that cause one or more *Reactions* (such as "add item to main character inventory"). The *CustomTrigger* and *CustomReaction* concepts, which inherit from *Trigger* and *Reaction* respectively, make it possible to create custom-made events.
- *Sprite:* represents an animation that can be assigned to entities (such as "main character walking", "main character jumping", etc.). It is composed by a *Frame* collection and it may loop after it ends.
- *GameState:* abstract concept which represents the game flow. It is specialized by *InfoDisplay* and *Room* concepts. *InfoDisplays* are used to display information (textual or graphical) on the screen, containing a *Purpose* attribute to indicate if it is an introduction, game over or ordinary information display screen (such as a menu, credits or instructions screen). Finally, each *GameState* contains an *ExitCondition* collection, which tells when the game should move from one state to another (e.g., when a key is pressed).

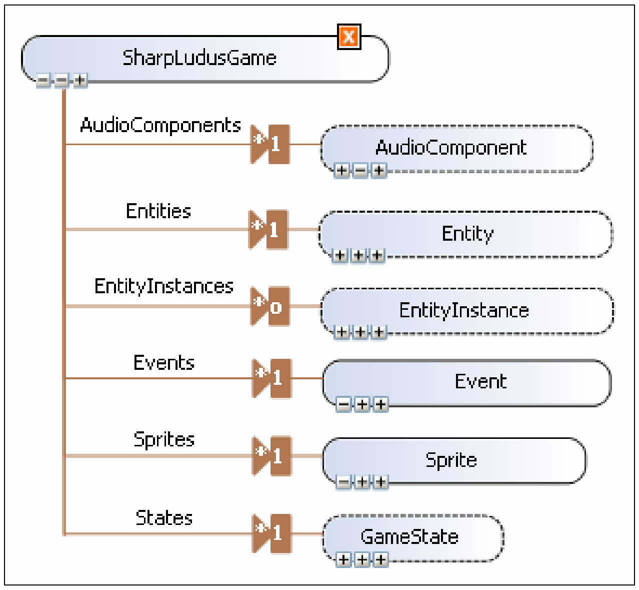
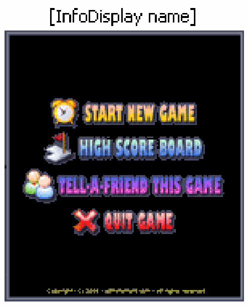


Figure 2. Top-level SLGML concepts

4.2 SLGML Syntax

Language syntax defines how the language elements appear in a concrete, human-usable form. Visual languages syntax is not only purely textual, combining graphics, text and conventions by which users may interact with the graphics and the text under the auspices of tools. Table 2 presents the visual syntax elements of SLGML.

Table 2. SLGML Visual Syntax

Graphical Representation	Description
 [InfoDisplay name] START NEW GAME HIGH SCORE BOARD TELL-A-FRIEND THIS GAME QUIT GAME	**InfoDisplay:** An information display screen is represented by a picture (shown in the left), and contains a textual decorator on its outer top, describing its name.
✓	**Intro Purpose Decorator:** This image decorator is applied to an info display, on its inner top, if the info display purpose is *Intro*.
✗	**Game Over Purpose Decorator:** This image decorator is applied to an info display, on its inner top, if the info display purpose is *GameOver*.
	Room: A game room is represented by a picture (shown in the left) and contains a textual decorator on its outer top, describing its name.
→	**Transition:** State transitions are visually represented as black arrows.

4.3 Semantic Validators

Besides aiding game designers with visual edition features, SLGML modeling experience also ensures that the DSL semantics are respected by them. This is done through semantic validators. The list below shows some examples of semantic rules associated with SLGML and enforced by means of validators:

- A game state transition must have at least one exit condition;
- A SharpLudus game should contain one main character;
- A SharpLudus game should contain only one introduction InfoDisplay;
- A SharpLudus game should contain only one game over InfoDisplay;
- An entity should contain at least one sprite;
- All game states should be reachable.

4.4 Code Generator

A C# [19] code generator was created and associated to SLGML. The generated code consumes a simple game engine developed with DirectX which was specially created for the factory. In other words, the generator receives a SLGML diagram as input and generates the following C# classes as output:

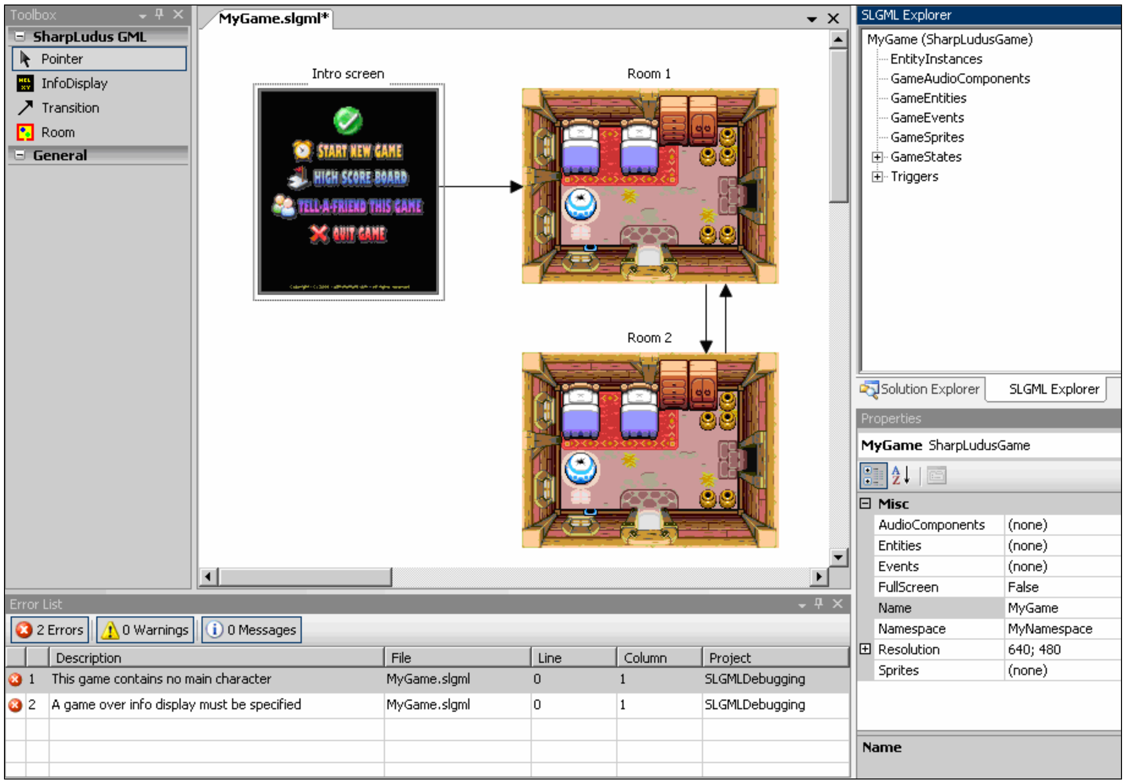
- `AudioComponents`, responsible for providing sound effect and background music objects via C# properties compliant to the Singleton [20] design pattern.
- `Sprites`, responsible for providing sprite objects via C# properties. The Singleton design pattern is not used in this case, since each sprite must be unique due to its own animation information, such as its current frame.
- One class for each *Entity* concept specified by the game designer. Such a class inherits from the `Item`, `MainCharacter` or `NPC` game engine classes.
- `EntityInstances`, responsible for providing entity instance objects via C# properties compliant to the Singleton design pattern.
- `States`, responsible for providing room and information display screen objects via C# properties compliant to the Singleton design pattern.
- The main game class, whose name corresponds to the `Name` property of the SharpLudusGame root concept. Such a class inherits from the `Game` game engine class. The code generator also creates a method in this class named `InitializeResources`, where the game configuration is set and game events are registered.
- `Program`, which contains the `Main` method and is responsible for instantiating and running the game.

Besides the generated classes, the IDE project additionally provides two initial classes which are not re-generated: `CustomTriggers` and `CustomReactions`. Developers should add their own methods to these classes in order to implement custom triggers and custom actions specified by the game designer in the SLGML model.

Figure 3 presents the complete SLGML modeling experience, hosted in the Visual Studio .NET development environment [21]. The Toolbox (at the left) presents some domain concepts that can be dragged and dropped to the SLGML designer (at the middle). The Error List (at the bottom) presents errors risen from semantic validators. The Properties window (at the right bottom) makes it possible to edit properties of the selected item in the diagram, eventually launching factory designers (sprite designer, entity designer, room designer, etc.). By using menu commands, users can launch the code generator as well as create their own code.

5. CASE STUDY: ULTIMATE BERZERK

This section presents the creation of a real-world adventure game named Ultimate Berzerk, which illustrates the use of the SharpLudus software factory. In Ultimate Berzerk, the player controls a main character, using the arrows key, to move around a maze composed by connected rooms. Once the player collects a special item (named *Weapon*), the spacebar can be used to shoot fireballs against enemies. Enemies may have special behaviors (not originally provided by the factory). The goal of the game is to collect the *Diamond* item and find the exit sign. A screenshot of the game is presented in Figure 4.



MyGame SharpLudusGame

Misc

AudioComponents	(none)
Entities	(none)
Events	(none)
FullScreen	False
Name	MyGame
Namespace	MyNamespace
Resolution	640; 480
Sprites	(none)

Name

	Description	File	Line	Column	Project
1	This game contains no main character	MyGame.slgml	0	1	SLGMLDebugging
2	A game over info display must be specified	MyGame.slgml	0	1	SLGMLDebugging

Figure 3. Complete SLGML modeling experience

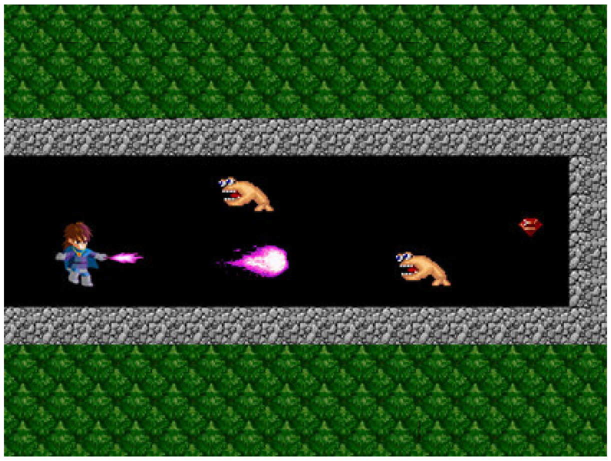
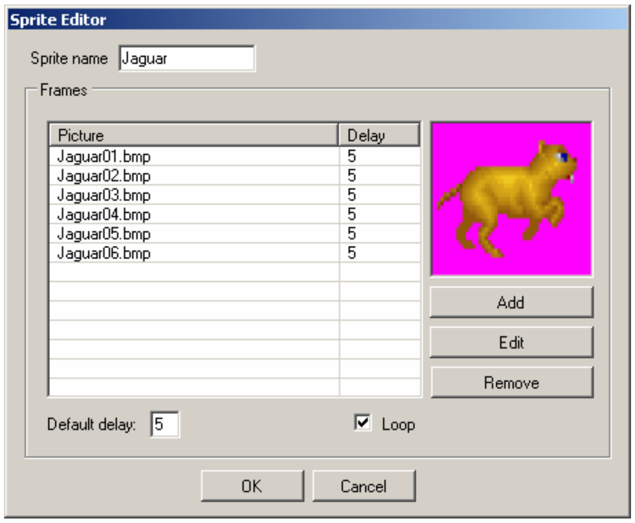


Figure 4. Ultimate Berzerk screenshot

5.1 Designing the Game

By modeling a SLGML diagram and launching factory designers from the Properties window, the game designer is able to visually create the majority of the game: sprites, entities, events, audio components, etc. For example, Figure 5 presents one of the screens of the sprite designer. This designer is launched from the *Sprites* property of a SharpLudus game and makes it possible for the game designer to specify frames and information such as if the animation will loop or not.


Sprite Editor

Sprite name: Jaguar

Frames

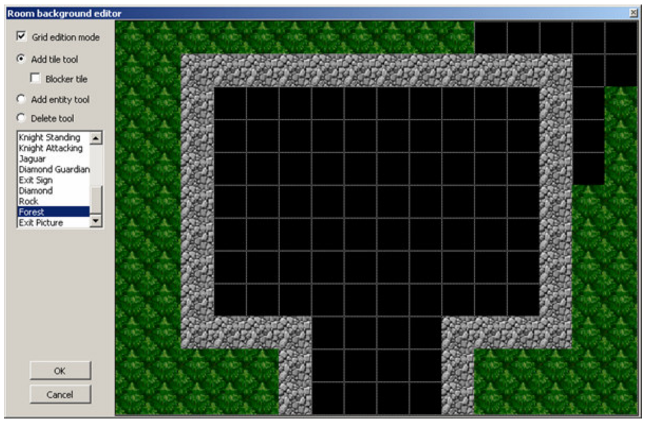
Picture	Delay
Jaguar01.bmp	5
Jaguar02.bmp	5
Jaguar03.bmp	5
Jaguar04.bmp	5
Jaguar05.bmp	5
Jaguar06.bmp	5

Default delay: 5

[✓] Loop

Figure 5. Sprite Designer

Figure 6, on the other hand, presents the room designer, where previously created sprites can be assigned to a room as tiles and entity instances (such as enemies and items) can be added to rooms based on previously created entities.


Room background editor

[✓] Grid edition mode

[● Add tile tool]

[ ] Blocker tile

Add entity tool

Delete tool

Knight Standing
Knight Attacking
Jaguar
Diamond Guardian
Exit Sign
Diamond
Rock
[✓ Forest]
Exit Picture

Figure 6. Room Designer

5.2 Custom Developer Code

Some NPCs (non-playable characters) of Ultimate Berzerk have special behaviors. For example, the enemy known as Diamond Guardian (shown in Figure 4) has a special movement in which it is bounced by room tiles of type "rock". In order to implement such behavior, factory users only need to add a class to the project named `DiamondGuardian`, mark it as "partial" and override the desired methods. This will make the final class to be the combination of the user `DiamondGuardian` class with the factory generated `DiamondGuardian` class.

It is worth noticing that when adding their own code, users will have full IDE editing and debugging support, as well as be able to make complex API calls, such as requesting information from a web service or accessing a database, for example.

5.3 Discussion: Factory Effectiveness

Although Ultimate Berzerk is a relatively simple game, with a few rooms to be investigated by the main character, its development explored many interesting SharpLudus software factories assets and features that illustrate how the factory can be used to create real-world games. Extending Ultimate Berzerk to a game with a better game-play and replay value is just a question of adding more model elements which reflect the creativity of the game designer.

The automation and productivity provided by the SLGML modeling experience, its code generator and consumed game engine is evident: in less than one hour of development effort, 16 classes and almost 3900 lines of source code were automatically generated for the development team. What is most important is that such lines of source code mainly present routine, boring and error-prone tasks, such as assigning pictures to frames, frames to sprites, sprites to entities, entities to rooms, rooms to the game, events to the game and so on.

By using the SharpLudus software factory, especially the visual designers, the development team experience was made more intuitive and accurate. At the same time, when more complex behavior was required (such as specifying the Diamond Guardian movement) the factory was flexible to allow developers to add their own code to the solution, using all of the benefits of an object-oriented programming language and being aided by IDE features such as editor support, debug support and so on. This contrasts the development experience of visual-only game development tools, where weak script languages should be used under an environment which was not originally conceived for codification.

Considering the generated code along with the consumed game engine, it can be concluded that the SharpLudus software factory is able to provide, in one hour, a development experience which would require, from scratch, the implementation of 61 classes and more than 6200 lines of source code.

6. CONCLUSIONS AND FUTURE WORK

This paper presented a study, illustrated with a real example, of how digital games development can better exploit an upcoming tendency: software industrialization. Different aspects were encompassed by such a study, being the development of a visual domain-specific language the most appealing subject.

Since the proposed approach and tools are focused on a specific domain, they may not be suitable to other types of game development contexts. Therefore, one interesting future work is the creation, based on previously acquired knowledge, of other factories targeted at other game genres, such as racing games or first-person shooters. Some domain concepts, factory designers, semantic validation rules and excerpts of the code generator may be reused, while others will need to be recreated.

Extending the SharpLudus software factory architecture and code generator to support the creation of games targeted at mobile devices, such as cell phones, seems to be quite appealing, since a recognized issue is that porting the same game to different mobile phone platforms is a burdensome and error-prone task. In such a case, once a code generator is implemented for each platform, all platforms would be able to share a single game model (specified with the SLGML visual domain-specific language) and maintenance would be made much simpler.

While the results obtained so far empirically shows that the SharpLudus factory is indeed an interesting approach, it is important to notice that deploying a complete software factory is also associated with some costs. Return of investment may arise only after a certain amount of games are produced. Besides that, despite being easy to use, software factories are complex to develop. They will certainly require a mindset evolution of the game development industry.

A final remark is that the presented proposal alone will not ensure the success of game development. In fact, no technology is a substitute for creativity and a good game design.

Game industrialization, languages, frameworks and tools are means, not goals, targeted at the final purpose of making people have entertainment, fun and enjoy themselves. Players, not the game or its constituent technologies, should be the final focus of every new game development endeavor.

REFERENCES

[1] Entertainment Software Association, *Essential Facts about the Computer and Video Game Industry*, 2005.

[2] Digital-lifestyles.info, *Men Spend More Money on Video Games Than Music: Nielsen Report*, http://digital-lifestyles.info/display_page.asp?section=cm&id=2091.

[3] Greenfield, J. et. al., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley & Sons, 2004.

[4] Zerbst, S., Duvel O., *3D Game Engine Programming,* Course Technology PTR, 1st edition.

[5] *Microsoft DirectX*, http://www.microsoft.com/directx.

[6] *OpenGL*, http://www.opengl.org.

[7] *RPG Maker XP*, http://www.enterbrain.co.jp/tkool/RPG_XP/eng/index.html.

[8] Wiering, M. *The Clean Game Library*, MSc dissertation, University of Nijmegen, 1999.

[9] Ogre3d.org, *OGRE 3D: Open Source Graphics Engine*, http://www.ogre3d.org.

[10] Sourceforge.net, *Crystal Space 3D*, http://crystal.sourceforge.net.

[11] Rollings, A.; Morris, D.; *Game Architecture and Design*, The Coriolis Group, 2000.

[12] Albuquerque, M. *Revolution Engine: 3D Game Engine Architecture*, BS conclusion paper, Federal University of Pernambuco, 2005.

[13] Rocha, E. *Forge 16V: An Isometric Game Development Framework*, MSc dissertation, Federal University of Pernambuco, 2003.

[14] Parnas, D. *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, March 1976.

[15] Roberts, D.; Johnson, R. *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, Proceedings of Pattern Languages of Programs, 1996.

[16] Deursen, A.; Klint, P.; Visser, J. *Domain-Specific Languages: An Annotated Bibliography*, http://homepages.cwi.nl/~arie/papers/dslbib/.

[17] MSDN.com, *VS Tools for Office Developer Portal*, msdn.microsoft.com/office/understanding/vsto/default.aspx.

[18] Fowler, M. *Language Workbenches: The Killer-App for Domain Specific Languages?*, www.martinfowler.com/articles/languageWorkbench.html.

[19] Microsoft.com, *C# Developer Center*, http://msdn.microsoft.com/vcsharp/.

[20] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, 1998

[21] MSDN.com, *Visual Studio 2005 Team System: Overview*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsent/html/vsts-over.asp.