Extending Visual Studio .NET as a Software Factory for Computer Games Development in the .NET Platform

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) 2126-8430
{awbf, alms}@cin.ufpe.br

Abstract. This paper presents an extension to the Visual Studio .NET development environment targeted at computer games development industrialization in the .NET Platform. A computer game product line definition and its architecture are specified, as well as implemented by means of software factory assets such as a visual designer based on a domain-specific language, semantic validators and code generators. The proposed approach is then illustrated and validated by the creation of real world case studies. The final intention is to empower game developers and designers to work more productively, with a higher level of abstraction and closer to their application domain.

Keywords: Visual Studio, domain-specific languages, 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 show 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.

According to the Microsoft Software Factories Initiative [3], a software factory can be defined as a product line that configures extensible development tools like Visual Studio [5] or Eclipse [6] with packaged content and guidance, carefully designed for building specific kinds of applications. This paper, therefore, addresses the need for computer games industrialization by proposing the implementation of a game software factory, named **SharpLudus**[1], through the extension and customization of the Visual Studio IDE (Integrated Development Environment). The focus of the approach consists in embedding in the IDE a new visual designer, based on a visual domain-specific language, through which game designers and developers can specify a computer game with a higher level of abstraction and closer to their application domain.

Differently from traditional modeling approaches, however, such a specification is a live development process artifact. It can be used as input to other factory assets which customize the IDE, such as semantic validators and a code generator responsible for outputting computer games in the C# programming language [7] and, therefore, targeted at the .NET Platform [8].

The remaining of this paper is organized as follows. Section 2 presents a specification for the proposed game factory. Section 3 describes how a new modeling designer is embedded in Visual Studio .NET by means of a visual domain-specific language. Section 4 details the code generator and framework (game engine) used by the factory. Chapter 5 presents a case study named Ultimate Berzerk. Chapter 6, finally, concludes about the presented work and points out some future directions.

2 Factory Specification

Before any factory assets (IDE extensions and customizations) can be described, it is important to detail what the software factory will be able to produce as well as how it will be done. This section covers such issues by defining a product line for the factory and a suggested architecture for its products.

2.1 SharpLudus Product Line Definition

The great diversity of games created so far has turned the digital games universe into a very broad domain. Therefore, creating a software factory targeted at computer games development in general, ranging from 2D platform games to 3D flight simulators, constitutes a too broad and ineffective endeavor. In such a scenario, the production process and its tools would not be able to fully exploit factory benefits such as component reuse and assemblage. In other words, a narrower subset of games should be chosen.

For the SharpLudus software factory, the ***adventure*** game genre was chosen for the factory product line. It can be described as a genre encompassing games which are set

[1] Resources and more information can be found in http://www.cin.ufpe.br/~sharpludus.

in a "world" usually made up of multiple, connected rooms or screens, involving a goal which is more complex than simply catching, shooting, capturing, or escaping, although completion of the objective may involve several or all of these. Additional information about the factory domain, contemplating typical computer game features, is presented in Table 1.

Table 1. Additional product line information

Game Feature	SharpLudus Product Line Interpretation
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	PCs running Microsoft Windows 98 or higher.

2.2 SharpLudus Product Line Architecture

The software factory product line architecture is one of the most important assets produced by product line development. It describes the common high-level design features of the products that will be produced by the product line.

For the proposed game software factory, a top-level network topology is presented in Figure 1. It is actually very simple, comprising only two components: the target client where the created game is supposed to be played and a web server that hosts a web service [9] which is responsible for storing and retrieving scores of games created through the factory.

The low-level architecture of games produced by the SharpLudus factory is split in the following three figures, due to space constraints. Figure 2 presents the main `Game` class, which contains collections of sound effect, game state and event objects. Figure 3 details the implementation of game states, their exit conditions and background music. Finally, Figure 4 provides an implementation overview for entities (basic game units such as non-playable characters and items) and sprites (animations).

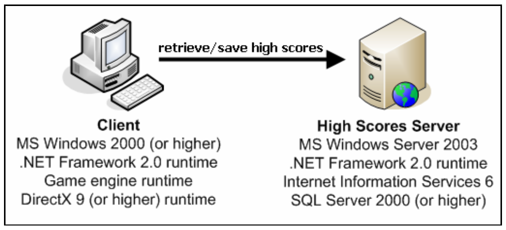

retrieve/save high scores
Client
MS Windows 2000 (or higher)
.NET Framework 2.0 runtime
Game engine runtime
DirectX 9 (or higher) runtime
High Scores Server
MS Windows Server 2003
.NET Framework 2.0 runtime
Internet Information Services 6
SQL Server 2000 (or higher)

Fig. 1. SharpLudus top-level network topology

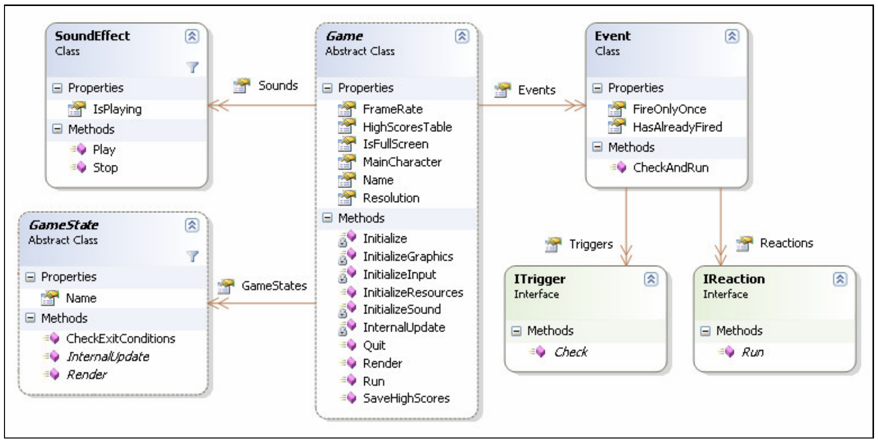


Fig. 2. SharpLudus game architecture (1): the `Game` class and its relations

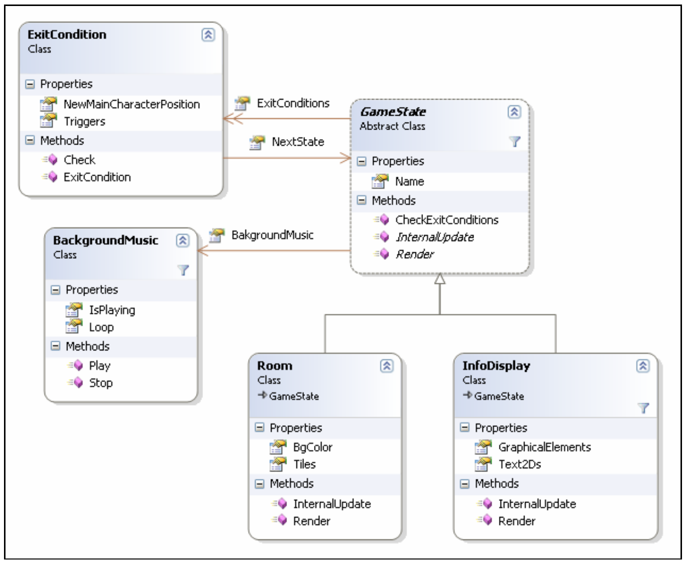


Fig. 3. SharpLudus game architecture (2): the `GameState` class and its relations

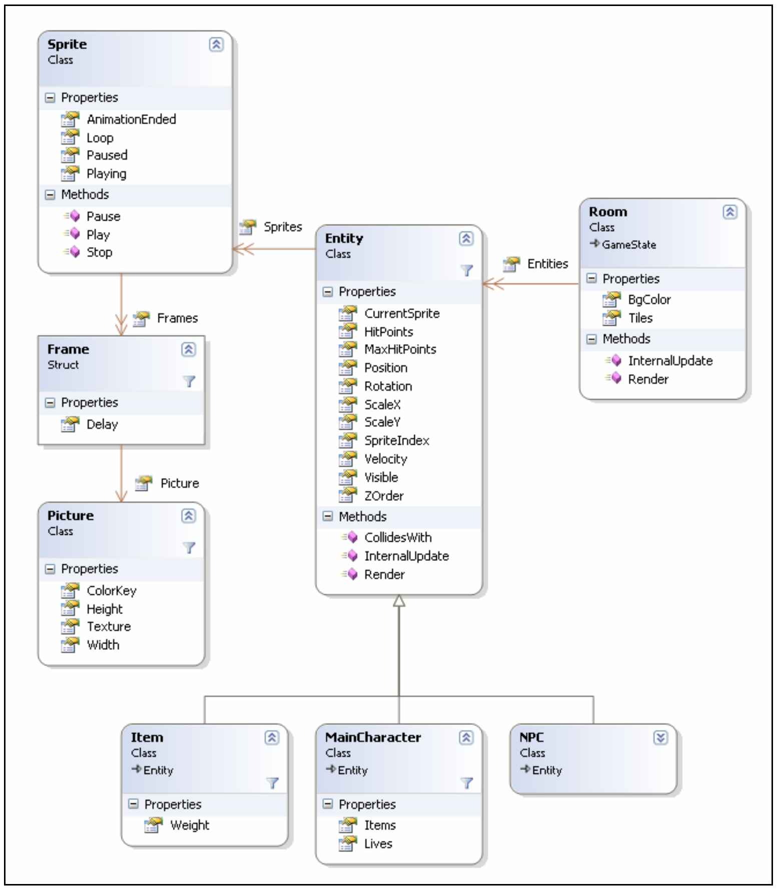


Fig. 4. SharpLudus game architecture (3): the `Entity` class and its relations

3 Embedding a New Designer in Visual Studio .NET

The most important extension that the SharpLudus game factory brings to Visual Studio .NET is the addition of a new visual designer to the IDE, through which the game developer 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 designer includes the creation and manipulation of many game components (events, rooms, sprites, etc.), through enriched Visual Studio property windows.

In order to create such a modeling extension, however, a visual domain-specific language (DSL) [10] was conceived. Such a language, named **SLGML** (SharpLudus Game Modeling Language) is the foundation of the visual designer, which is actually a visual representation of the DSL. Its creation process is presented below.

3.1 Selecting a Language Workbench

Language workbenches contrast the early days of domain-specific modeling, where no tools were available to create domain-specific languages and support modeling with them in a cost effective manner [11]. They make it easy to build tools that match the best of modern IDEs and make language oriented programming much easier to build and support, lowering the barriers that have made language oriented programming so awkward for so many.

The SharpLudus software factory uses Visual Studio 2005 Team System (VSTS) [5] language workbench technologies, named DSL Tools [12], to design and implement its visual DSLs. The DSL Tools provides a framework and toolset that enable partners to build custom visual designers and domain-specific language designers using Visual Studio. In other words, it is possible to extend VSTS by creating and plugging into it a new designer, based on a visual domain-specific language. Through the DSL Tools, one can create, edit and visualize metadata that is underpinned by a code framework, which makes it easier to define domain-specific schemas for metadata, and then to construct a custom graphical designer hosted in Visual Studio.

3.2 SLGML Design

A visual domain-specific language is composed by different elements: a graph of concepts (or “classes”), relationships (comprising roles, cardinality, etc.), attributes and so on. A common approach to specify a visual DSL is to use a meta-modeling language, such as the GOPRR (Graph Object Property Relationship and Role) language [13] or the Microsoft DSL Tools meta-modeling language. This last one was used by the SharpLudus software factory to specify its visual DSLs.

A meta-modeling language is used to build a meta-model, which describes a modeling language (such as SLGML), similarly to the way a model describes a system. The root concepts of the SLGML meta-model is presented in Figure 5 (deeper concepts and relationships are not shown due to space constraints).

The *SharpLudusGame* is the root domain concept of the SLGML DSL. It is related to six top-level elements:

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

- *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.
- *GameState:* abstract concept which represents the game flow. It is specialized by *InfoDisplay* and *Room* concepts. *InfoDisplays* contain a *Purpose* attribute, indicating if it is an introduction, game over or ordinary information display screen. Finally, each *GameState* contains an *ExitCondition* collection, which tells when the game should move from one state to another.

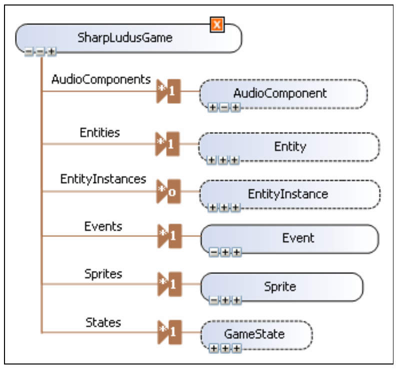


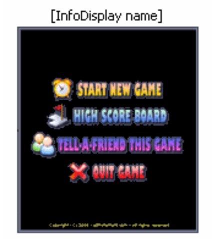
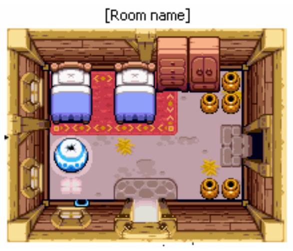
Fig. 5. Top-level SLGML concepts

3.3 SLGML Syntax and Visual Editor

Language syntax defines how the language elements appear in a concrete, human-usable form. In the case of visual languages, the syntax is not only purely textual; it combines graphics, text and conventions by which users may interact with the graphics and the text under the auspices of tools. The visual syntax elements of SLGML are presented in Table 2.

The game designer can add *InfoDisplays*, *Rooms* and *Transitions* to a SLGML model by drag-and-drop operations from the IDE toolbox, which is presented in Figure 6. Other language concepts (such as events, sprites, transition exit conditions, information display purpose or the entities belonging to a room) can be manipulated by the game designer through the IDE Properties window (Figure 7), which is sensitive to the model element in focus. Depending on the element selected, the Properties window can launch custom property editors, such as the SharpLudus software factory designers (room designer, event designer, sprite designer, etc.), which will be better illustrated during the explanation of the case study (Section 5).

Table 2. SLGML Syntax Elements

Name	Graphical Representation	Description
InfoDisplay	 [InfoDisplay name] START NEW GAME HIGH SCORE BOARD TELL-A-FRIEND THIS GAME QUIT GAME	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.

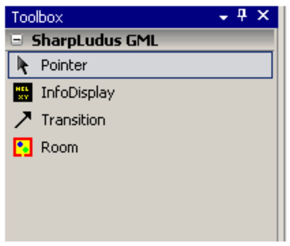
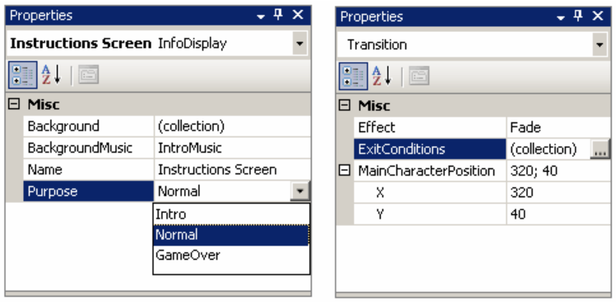


Fig. 6. Toolbox support for the game designer

The game designer is also able to visualize the elements of its current SLGML diagram in a more organized, hierarchical way, by using a tool window named SLGML Explorer (Figure 8). It is possible to add and delete items from this window, as well as select one of its elements and edit its properties through the Properties window.



Properties

Instructions Screen InfoDisplay

Misc

Background	(collection)
BackgroundMusic	IntroMusic
Name	Instructions Screen
Purpose	Normal

Intro
Normal
GameOver

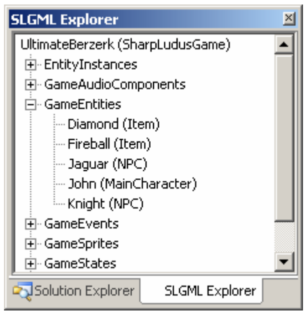
Properties

Transition

Misc

Effect	Fade
ExitConditions	(collection)
MainCharacterPosition	320; 40
X	320
Y	40

Fig. 7. Properties window support for the game designer



SLGML Explorer

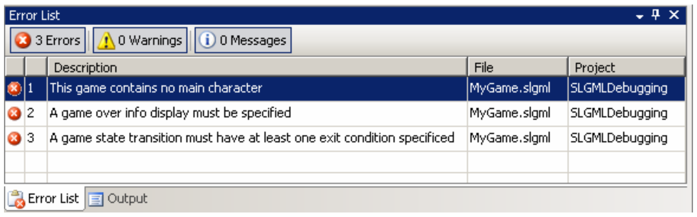
- UltimateBerzerk (SharpLudusGame)
  - EntityInstances
  - GameAudioComponents
  - GameEntities
    - Diamond (Item)
    - Fireball (Item)
    - Jaguar (NPC)
    - John (MainCharacter)
    - Knight (NPC)
  - GameEvents
  - GameSprites
  - GameStates

Solution Explorer | SLGML Explorer

Fig. 8. SLGML Explorer

3.4 Semantic Validators

Besides aiding the game designer with visual edition features, SLGML modeling experience also ensures that the DSL semantics are respected by the game designer. This is done through semantic validators, which are associated to a domain concept and can output validation issues to the Visual Studio Error List, as presented in Figure 9. Double-clicking in an Error List error automatically makes Visual Studio to focus on the model element which is the error source.



Error List

3 Errors | 0 Warnings | 0 Messages

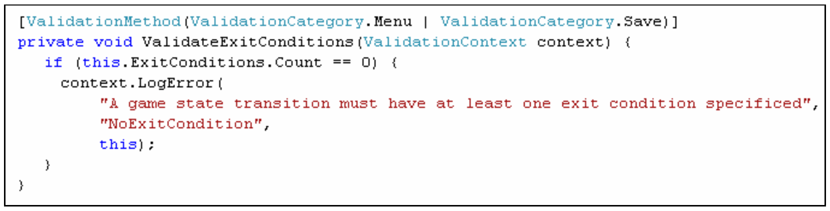
	Description	File	Project
1	This game contains no main character	MyGame.slgml	SLGMLDebugging
2	A game over info display must be specified	MyGame.slgml	SLGMLDebugging
3	A game state transition must have at least one exit condition specified	MyGame.slgml	SLGMLDebugging

Error List | Output

Fig. 9. Semantic errors displayed in the Visual Studio .NET Error List

In the specific case of the Microsoft DSL Tools language workbench, semantic validators can be created programmatically. Since each domain concept generates a

C# partial class, all the language designer needs to do is to implement a method in the desired concept partial class, which checks for special error conditions and logs the error, as shown in Figure 51. C# method attributes [14] can be used to specify when the validators should run: after opening a SLGML model, when saving it or explicitly through a context menu command.



```
[ValidationMethod(ValidationCategory.Menu | ValidationCategory.Save)]
private void ValidateExitConditions(ValidationContext context) {
   if (this.ExitConditions.Count == 0) {
     context.LogError(
         "A game state transition must have at least one exit condition specificed",
         "NoExitCondition",
         this);
   }
}
```

Fig. 20. Implementing a semantic validator in the *Transition* concept class

Some examples of SLGML semantic rules which can be enforced through validators are presented below:

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

Figure 11 gathers all features presented so far and illustrates a complete overview of the SLGML modeling experience, hosted in Visual Studio .NET.

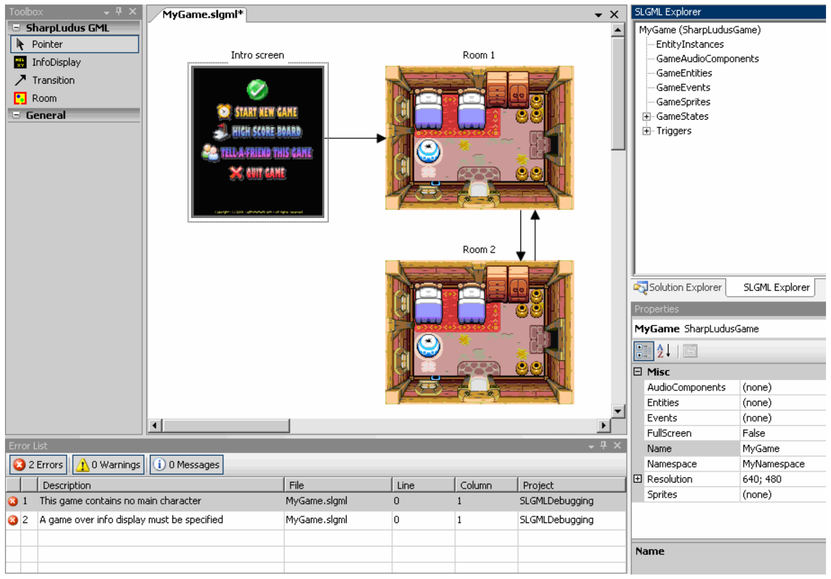
4 Code Generator and Framework

As pointed out by Deursen, Klinten and Visseur [10], a DSL can be made much more useful if it relies on a framework or library which implements the semantic notions and a compiler that translates DSL programs to a sequence of framework calls. In the SharpLudus software factory context, such framework is a game engine, the compiler is a code generator (which outputs code that consumes the game engine) and the DSL programs to be translated are actually SLGML models.

Considering the .NET Platform scenario for the SharpLudus software factory, this research reused a public game engine made available by the DigiPen Institute of Technology [15]. The engine is developed in C# and consumes the DirectX multimedia API [16]. Some of its features include:

- Creation and dynamic manipulation of game entities, including the assignment of sprites and movement;
- Keyboard interaction;
- Sound effects support;
- Text manipulation.

However, the engine was considerably modified and extended to make it compliant with the SharpLudus software factory product line domain. For example, the original version of the engine does not present any features related to game states or game events, and treats all entities equally. Besides that, its architecture was modified to become compliant with the one presented in the SharpLudus software factory product line (Section 2.2).



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

2 Errors | 0 Warnings | 0 Messages

	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

Fig. 31. Complete SLGML modeling experience

The extensions increased the amount of the original game engine code in 40%. Its final version, used by the SharpLudus software factory, contains 45 classes and more than 2340 lines of source code (excluding commented and empty lines), while the original version contains 20 classes and 1700 lines of source code.

The SLGML code generator receives a SLGML model as input and generates the following C# classes as output:

- `AudioComponents`, responsible for providing sound effect and background music objects via C# properties following the Singleton [17] 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 following the Singleton design pattern.
- `States`, responsible for providing room and information display screen objects via C# properties following the Singleton design pattern.
- Program, which contains the `Main` method and is responsible for instantiating and running the game.

- 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.

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. A generated code example for a SharpLudus game is illustrated by the class diagram presented in Figure 12.

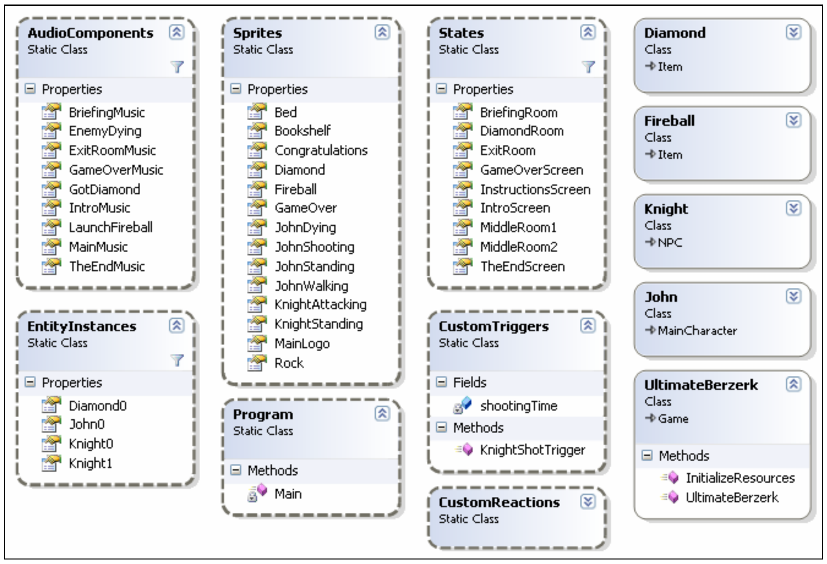
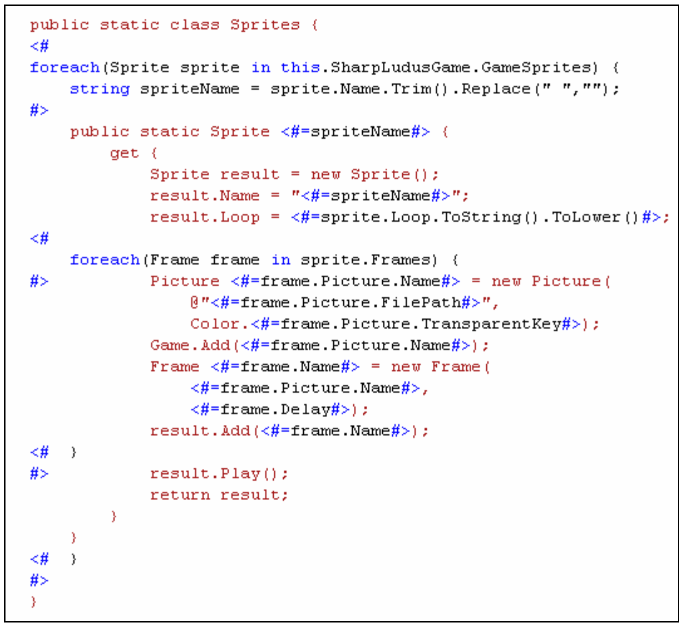


Fig. 12. Class diagram of a generated code example

In the Microsoft DSL Tools language workbench, artifact generators, including code generators, are implemented by a text template transformation toolkit. Such a toolkit provides a script language, whose syntax is similar to C#, to manipulate model concepts.

Figure 13 presents an excerpt of the SLGML code generator which is responsible for generating the `Sprites` class. Text between the **<#** and **#>** tags contains script commands, used to manipulate the visual DSL concepts. For example, a `foreach` loop is used to iterate through all sprites defined in the SLGML model by the game designer in order to create a C# property for it. Text between the tags **<#=** and **#>** are used to evaluate some script expressions. For example, the name of the C# property to be generated is the evaluation result of the expression `spriteName`, which is a string previously declared and assigned in the script. Finally, all text outside tags is just copied to the generated artifact.

Considering both script instructions and raw text to be copied to the generated classes, the SLGML code generator implemented with the DSL Tools text template transformation toolkit contains 410 non-empty lines of code. It is worth noticing that the generator used other toolkit features, whose explanation is beyond the scope of this paper. More information about the toolkit can be found in the MSDN DSL Tools workbench [12].



```
public static class Sprites {
<#
foreach(Sprite sprite in this.SharpLudusGame.GameSprites) {
    string spriteName = sprite.Name.Trim().Replace(" ","");
#>
    public static Sprite <#=spriteName#> {
        get {
            Sprite result = new Sprite();
            result.Name = "<#=spriteName#>";
            result.Loop = <#=sprite.Loop.ToString().ToLower()#>;
<#
    foreach(Frame frame in sprite.Frames) {
#>          Picture <#=frame.Picture.Name#> = new Picture(
                @"<#=frame.Picture.FilePath#>",
                Color.<#=frame.Picture.TransparentKey#>);
            Game.Add(<#=frame.Picture.Name#>);
            Frame <#=frame.Name#> = new Frame(
                <#=frame.Picture.Name#>,
                <#=frame.Delay#>);
            result.Add(<#=frame.Name#>);
<#  }
#>          result.Play();
            return result;
        }
    }
<#  }
#>
}
```

Fig. 43. Code generator script excerpt: generation of the `Sprites` class

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 shot 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 14.

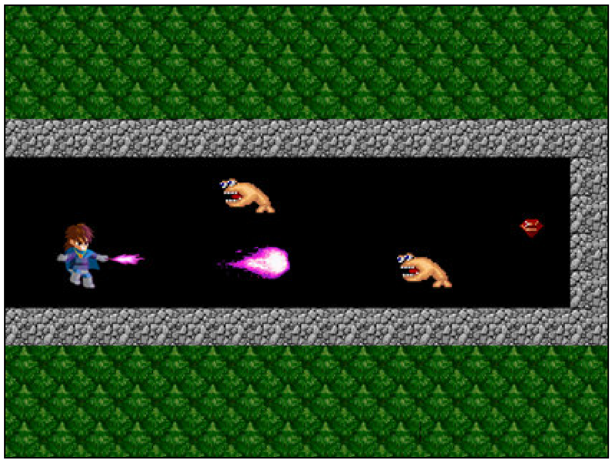
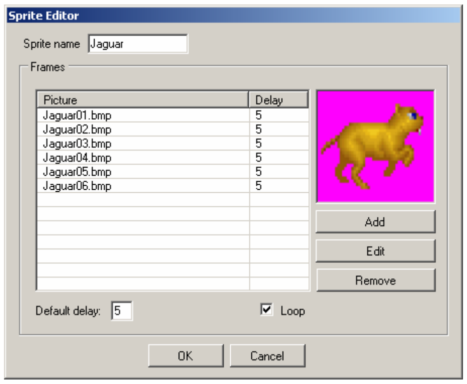


Fig. 54. Ultimate Berzerk screenshot

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 15 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

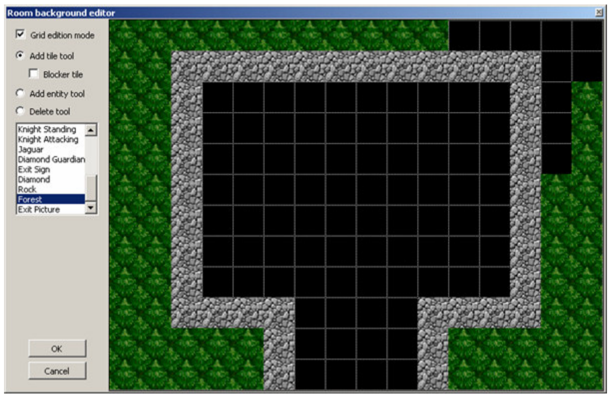
Picture	Delay
Jaguar01.bmp	5
Jaguar02.bmp	5
Jaguar03.bmp	5
Jaguar04.bmp	5
Jaguar05.bmp	5
Jaguar06.bmp	5

Default delay: 5

Loop [✓]

Fig. 65. Sprite Designer

Figure 16, on the other hand, presents the room designer, where previously created sprites can be assigned to room as tiles and entity instances (such as enemies and items) can be added to rooms based on previously created entities.

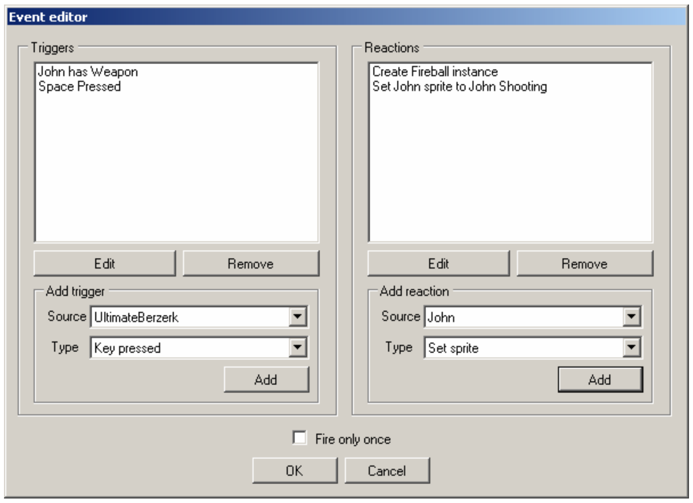


Grid edition mode [✓]
(●) Add tile tool
Blocker tile [ ]
( ) Add entity tool
( ) Delete tool

Knight Standing
Knight Attacking
Jaguar
Diamond Guardian
Exit Sign
Diamond
Rock
Forest
Exit Picture

Fig. 76. Room Designer

Finally, Figure 17 presents the event designer, through which the game designer can specify the rules that govern the Ultimate Berzerk world, by means of event triggers and event reactions.



Triggers
John has Weapon
Space Pressed

Add trigger
Source: UltimateBerzerk
Type: Key pressed

Reactions
Create Fireball instance
Set John sprite to John Shooting

Add reaction
Source: John
Type: Set sprite

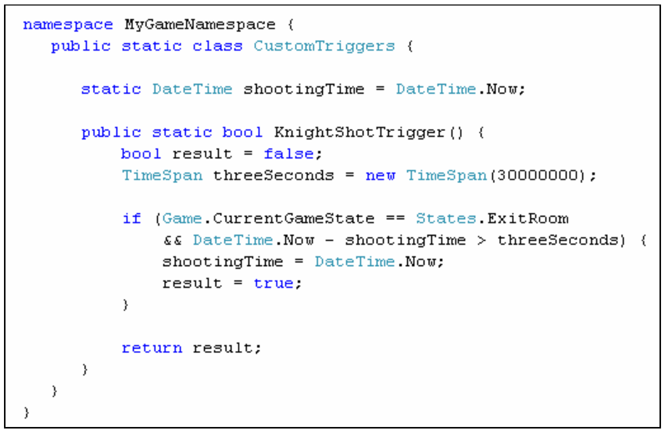
Fire only once [ ]

Fig. 87. Event Designer

5.2 Custom Developer Code

Although the SharpLudus factory provides many interesting predefined event triggers and reactions, there will always be some situations where the game designer needs more complex or non-usual behaviors. For example, suppose the game designer wants the *Knight* entity to shoot a fireball at every 3 seconds. The SharpLudus factory does not provide any built-in trigger for that. However, by adding a custom trigger in the event designer and specifying a method name, the game designer delegates the task of creating such a trigger to a developer.

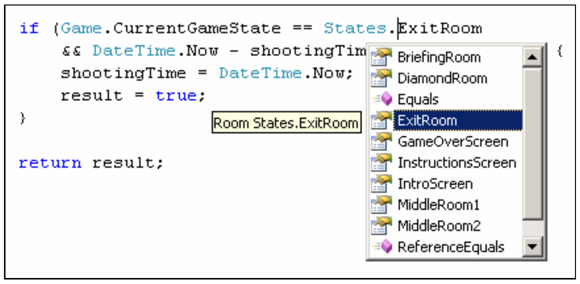
As previously pointed out, the game project in the IDE contains two predefined classes, named `CustromTriggers` and `CustomReactions`. Supposing that the game designer specified in the SLGML model a custom trigger named *KnightShotTrigger* to define when the *Knight* entity should shoot a fireball, a developer would only have to add the code presented in Figure 18 to the `CustromTriggers` class.



```
namespace MyGameNamespace {
   public static class CustomTriggers {

      static DateTime shootingTime = DateTime.Now;

      public static bool KnightShotTrigger() {
          bool result = false;
          TimeSpan threeSeconds = new TimeSpan(30000000);

          if (Game.CurrentGameState == States.ExitRoom
              && DateTime.Now - shootingTime > threeSeconds) {
              shootingTime = DateTime.Now;
              result = true;
          }

          return result;
      }
   }
}
```

Fig. 98. Implementing a custom trigger

While creating custom code, developers are provided with full IDE editor support, as shown in Figure 19. The model elements, once the code generator has run, become accessible in code as strongly-typed C# classes and properties.



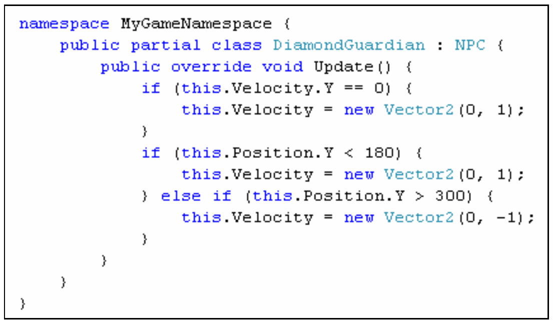
```
if (Game.CurrentGameState == States.ExitRoom
    && DateTime.Now - shootingTim                    {
    shootingTime = DateTime.Now;
    result = true;
}

return result;
```

Fig. 109. IDE support (IntelliSense) and strongly-typed code

The ability to customize the game under development with code is not restricted to custom triggers and custom reactions. The behavior of any entity, state or sprite can be extended or modified by the use of partial classes.

Suppose, for example, that the game designer wants to create a special movement type for *Diamond Guardian* NPC (non-playable character) instances, making them bounce as they touch the tiles of the room where they are placed (Figure 14). A developer can easily accomplish this task by adding to the IDE project a class named `DiamondGuardian` and assign to it the partial modifier. This will make the final `DiamondGuardian` class to be composed by both the factory generated code and the developer added code.

By overriding the `Update` method of the `DiamondGuardian` class, as shown in Figure 20, it is possible to define a custom movement type for this entity. The code shown in the figure moves the *Diamond Guardian* up, if it is stationary, and inverts its vertical direction as it reaches the desired upper and lower bounds of the room.



```
namespace MyGameNamespace {
    public partial class DiamondGuardian : NPC {
        public override void Update() {
            if (this.Velocity.Y == 0) {
                this.Velocity = new Vector2(0, 1);
            }
            if (this.Position.Y < 180) {
                this.Velocity = new Vector2(0, 1);
            } else if (this.Position.Y > 300) {
                this.Velocity = new Vector2(0, -1);
            }
        }
    }
}
```

Fig. 20. Custom entity behavior with partial classes

5.3 Discussion: Factory Effectiveness for Ultimate Berzerk

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 gameplay 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 or creating custom event triggers) 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.

5.4 Other Case Studies

In order to validate the SharpLudus software factory as capable of creating not only a single product, but a family of products, other case studies were developed as well. Detailing such case studies is out of the scope of this paper, but some screenshots are provided in Figure 21 (Stellar Quest game) and Figure 22 (Tank Brigade game).

In spite of belonging to the same product line definition, both games present unique variabilities (such as world rules, presentation style, influence of collected items and state flow), which made it possible to conclude that the SharpLudus software factory is successful in creating distinct products belonging to the same domain.

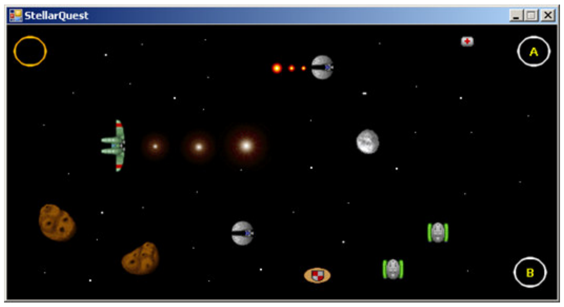


Fig. 21. Stellar Quest

6. Conclusions

This paper presented a study, illustrated with a real example, of how digital games development can better exploit an upcoming tendency: software industrialization. By implementing the factory with extensions and customizations of the Visual Studio .NET integrated development environment, different aspects were encompassed by such a study, being the addition of a new visual designer to the IDE the most appealing subject.

Works related to SharpLudus correspond to currently used game development technologies: multimedia APIs (such as DirectX [16] and OpenGL [18]), visual game creation tools (such as RPG Maker [19]) and game engines (such as OGRE [20] and

Crystal Space [21]). The SharpLudus game software factory, however, does not discard the use of such technologies and tools. On the contrary, it is built on their strengths to provide a higher abstraction level to game designers and developers.

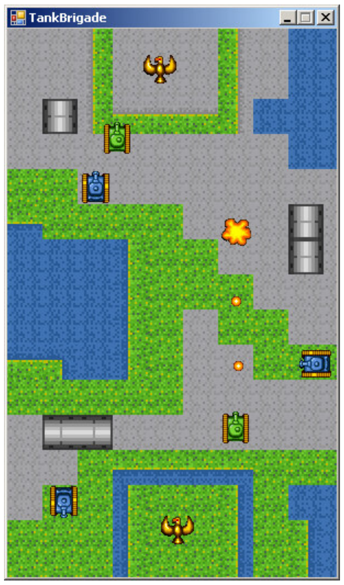


Fig. 22. Tank Brigade

One interesting future work is the creation, following the proposed approach, of other factories targeted at other game genres, such as racing games or first-person shooters. 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 a game development. In fact, no technology is 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. MSDN.com, Visual Studio 2005 Team System: Overview. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsent/html/vsts-over.asp.
6. Eclipse.org, Eclipse Home, http://www.eclipse.org/.
7. Microsoft.com, C# Developer Center, http://msdn.microsoft.com/vcsharp/.
8. Microsoft.com, Microsoft .NET Homepage, http://www.microsoft.com/net.
9. W3C, Web Services, http://www.w3.org/2002/ws/.
10. Deursen, A.; Klint, P.; Visser, J. Domain-Specific Languages: An Annotated Bibliography, http://homepages.cwi.nl/~arie/papers/dslbib/.
11. Tolvanen, J.-P. Domain-specific Modeling: Welcome to the Next Generation of Software Modeling, http://www.devx.com/enterprise/Article/29619.
12. MSDN.com, Visual Studio 2005: Domain-Specific Language Tools, http://msdn.microsoft.com/vstudio/DSLTools/.
13. Lyytinen, K.; Rossi M. METAEDIT+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment, Springer-Verlag, 1999.
14. MSDN.com, C# Programmer's Reference: C# Attributes, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrfcsharpattributes.asp.
15. DigiPen.edu, MSDN Webcast Archive - Video Game Development: Learn to Write C# the Fun Way, http://www.digipen.edu/webcast.
16. Microsoft DirectX, http://www.microsoft.com/directx.
17. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, 1998.
18. OpenGL, http://www.opengl.org.
19. RPG Maker XP, http://www.enterbrain.co.jp/tkool/RPG_XP/eng/index.html.
20. Ogre3d.org, OGRE 3D: Open Source Graphics Engine, http://www.ogre3d.org.
21. Sourceforge.net, Crystal Space 3D, http://crystal.sourceforge.net.